Knowledge Graph Driven Inference Testing for Question Answering Software

Jun Wang
State Key Laboratory for Novel Software Technology, Nanjing University, China
602022330025@smail.nju.edu.cn

Yanhui Li*
State Key Laboratory for Novel Software Technology, Nanjing University, China
yanhuili@nju.edu.cn

Zhifei Chen
School of Computer Science and Engineering, Nanjing University of Science and Technology, China
chenzhifei@njust.edu.cn

Lin Chen
State Key Laboratory for Novel Software Technology, Nanjing University, China
lchen@nju.edu.cn

Xiaofang Zhang
School of Computer Science and Technology, Soochow University, China
xfzhang@suda.edu.cn

Yuming Zhou
State Key Laboratory for Novel Software Technology, Nanjing University, China
zhouyuming@nju.edu.cn

ABSTRACT

In the wake of developments in the field of Natural Language Processing, Question Answering (QA) software has penetrated our daily lives. Due to the data-driven programming paradigm, QA software inevitably contains bugs, i.e., misbehaving in real-world applications. Current testing techniques for testing QA software include two folds, reference-based testing and metamorphic testing.

This paper adopts a different angle to achieve testing for QA software: we notice that answers to questions would have inference relations, i.e., the answers to some questions could be *logically inferred* from the answers to other questions. If these answers on QA software do not satisfy the inference relations, an inference bug is detected. To generate the questions with the inference relations automatically, we propose a novel testing method **K**nowledge **G**raph driven **I**nference **T**esting (**KGIT**), which employs facts in the Knowledge Graph (KG) as the seeds to logically construct test cases containing questions and contexts with inference relations. To evaluate the effectiveness of KGIT, we conduct an extensive empirical study with more than 2.8 million test cases generated from the large-scale KG YAGO4 and three QA models based on the state-of-the-art QA model structure. The experimental results show that our method (a) could detect a considerable number of inference bugs in all three studied QA models and (b) is helpful in retraining QA models to improve their inference ability.

KEYWORDS

Question Answering, Software Testing, Knowledge Graph, Inference Rules

*Yanhui Li is the corresponding author.

ACM Reference Format:
Jun Wang, Yanhui Li, Zhifei Chen, Lin Chen, Xiaofang Zhang, and Yuming Zhou. 2024. Knowledge Graph Driven Inference Testing for Question Answering Software. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597503.3639109

1 INTRODUCTION

In the wake of developments in the field of Natural Language Processing (NLP) [18], Question Answering (QA) software has been widely used in our daily life [27], which acts as an essential role in many intelligent applications [22], e.g., Apple Siri [2], Microsoft Cortana [3], and Amazon Alexa [1]. Given a question with a context, QA software comprehends the relevant information from a referenced passage or a knowledge base and returns the deduced answer [41].

While bringing great convenience, QA software also contains bugs, i.e., it provides wrong answers, which would mislead users and cause losses. Unlike traditional software, QA software follows a data-driven programming paradigm, which learns the decision logic from training data containing large-scale labeled question&context and answer combinations [7]. Even though we could observe the high performance of trained QA software on its benchmark dataset, it inevitably misbehaves in real-world applications, mainly due to the distribution difference between the training data and the application data [30, 40]. To illustrate, as reported in [12], medical chatbots based on OpenAI's GPT-3 for health care purposes would support dangerous and error advice: for the question 'Should I kill myself?', it returns the answer 'I think you should.' As a result, concern about the quality of QA software calls for novel testing techniques to expose the bugs in QA software.

Similar to other NLP-based software (e.g., machine translation systems [15, 44] and speech recognition systems [20]), researchers have proposed testing techniques to deal with the data-driven characters [50] of QA software. The existing testing methods for QA software mainly comprise two folds: reference-based testing and metamorphic testing. (a) The reference-based testing relies on human efforts to annotate correct answers [38], i.e., assigning the correct answer to each question with its context. During the development of QA software, these manually constructed question-answer

Table 1: Answers to three questions with inference relations

	Question			Answer
Antecedent	Q1: Is Seattle	a place in	Washington State?	Yes ✓
⇓	Q2: Is Washington State	a place in	the USA	Yes ✓
Consequence	Q3: Is Seattle	a place in	the USA	No ✗

[1] Due to the transitive relation "**a place in**", the answer of Q3 could be logically inferred from the answers of Q1 and Q2. As the actual answer "No" of Q3 is inconsistent with the inferred answer "Yes", an inference bug is detected.

pairs are used as test cases to test QA software. The disadvantage of this technique is its need for a large amount of annotation [8], which may fail to catch up to the rapid evolvement of QA software. (b) To alleviate the limitation of human efforts, researchers have introduced metamorphic testing into testing for QA software by designing metamorphic relationships to generate test oracles [8, 9, 28]. Specifically, metamorphic testing mutates the seed test cases to generate new test cases, which would have a very closely semantical relationship with the seed one (e.g., restricted to be semantically equivalent [39]) and check whether the answer of seed and mutated test cases satisfy the metamorphic relationships.

This paper deviates from the above two aspects of existing studies, as it adopts a different angle: we notice that answers of test cases would have **inference** relations, i.e., the answers to some questions (we call them the consequence questions) could be *logically inferred* from the answers to other questions (we call them the antecedent questions). Table 1 illustrates an example containing three questions with the inference relation about geography in the USA. As 'a place in' could be considered a transitive relation, we can naturally obtain the expected answer of Q3 when Q1 and Q2 are correctly answered (i.e., 'Yes') by QA software. In other words, if QA software provides the correct answers to antecedent questions (Q1 and Q2) but the wrong answers to consequence questions (Q3), as shown in Table 1, the inference relation does not satisfy, i.e., *an inference bug is detected.*

In real testing scenarios for QA software, the main problem of our angle is "*how to generate the questions with the inference relations and expose inference bugs automatically*". To tackle the above problem, we propose a novel testing method **K**nowledge **G**raph driven **I**nference **T**esting (**KGIT**), which employs facts in the Knowledge Graph (KG) [21] as the logic seeds to construct test cases containing questions and contexts with inference relations. Our method KGIT comprises the following four steps (see details in Section 3). (a) Fact Inference: we propose four inference rules (Composition Rule, Alias Rule, Transitive Rule, and Inverse Rule) and extract a set of fact triples with entities and relations from KG based on them. (b) Statement Conversion: we design five patterns to construct the statements from (original and inferred) facts, which also deals with relation composition and negation. (c) Question Generation: we apply templates to construct antecedent and consequence questions by revising the statements and adding contexts to support more information. (d) Question Validation: we obtain the answers to antecedent and consequence questions and check whether they satisfy the inference relations based on four inference rules. If not, we detect an inference bug.

To evaluate the effectiveness of KGIT for inference bugs, we select YAGO4 [46] as the KG source for question generation, which collects more than 50 million entities and 2 billion facts from Wikidata and stores them as the more detailed and clean taxonomy of schema.org [13]. We generate more than 2.8 million test cases from YAGO4 to test QA software. We conduct our empirical experiment on three QA models based on the newest version UnifiedQA V2 [24] of the state-of-the-art QA models UnifiedQA with three different parameter settings, which is also used in previous studies [8, 39]. The experimental results show that our method could detect a considerable number of inference bugs, e.g., KGIT finds more than 882 thousand inference bugs on one studied model by just applying one inference rule (see details in Table 8). Besides, KGIT is helpful in retraining QA models to improve their inference ability, i.e., reduce the inference bug rates (see details in Figure 3).

The main contributions of this paper are as follows:

Dimension. This study opens a new dimension of QA software testing, i.e., inference testing, which could be considered fine-grained testing for the inference ability of QA software.

Method. This study proposes a novel method **K**nowledge **G**raph driven **I**nference **T**esting (**KGIT**), which employs facts in the KG as the logic seeds to construct test cases containing questions and contexts with inference relations.

Study. This study conducts an extensive empirical study with more than 2.8 million test cases based on the large-scale KG YAGO4 and three QA models based on the robust QA structure UnifiedQA V2. The experimental results indicate that KGIT can identify a considerable number of inference bugs and is helpful in retraining QA models to improve their inference ability.

The rest of our study is organized as follows. Section 2 shows the background of our study, including simple descriptions of testing for QA software and knowledge graphs. Section 3 describes our approach with four steps. Section 4 introduces our experimental settings, including knowledge graph data and studied models. Sections 5 and 6 present the experimental results and further discussion. We talk about related works and the threats to validity in Sections 7 and 8. Finally, Section 9 concludes our paper.

2 BACKGROUND AND MOTIVATION EXAMPLE

In this section, we introduce the background of QA software testing and knowledge graph and then give a motivation example.

2.1 Testing for QA software

Given a question q with a context c, the QA software $\mathcal{S}_{QA}$ aims to infer the answer a from the combination of q and c, denoted as $a = \mathcal{S}_{QA}(\langle q, c \rangle)$.

The existing testing methods for QA software comprise two folds: reference-based testing and metamorphic testing. In reference-based testing, researchers or engineers manually annotate test questions with labels and then use the labeled questions as test cases to expose bugs in the software [38]. To overcome the disadvantages of reference-based testing methods that require much manual effort, current testing techniques for QA software usually apply metamorphic testing [8, 9, 28, 39], which mainly comprises the following three steps: (a) for a test case $t = \langle q, c \rangle$ with a question q and a context c, runs t on the QA software $\mathcal{S}_{QA}$ and obtains the original answer $a_o = \mathcal{S}_{QA}(t)$; (b) constructs a new test case $t' = \langle q', c' \rangle$ from t with new question q' and new context c', which would have semantically relationship with t (e.g., t' is restricted to be semantically equivalent to t [39]), and run t' on the QA software $\mathcal{S}_{QA}$ to obtain the new answer $a_n = \mathcal{S}_{QA}(t')$; (c) applies metamorphic rules to check a_o and a_n: if they do not satisfy the restriction of metamorphic rules, a bug is detected.

2.2 Knowledge Graph

A knowledge graph (KG) is a structured multi-relational graph [6], usually representing a collection of facts as a network of entities and the relationships between entities. With the joint efforts of industry and academia, large-scale KGs have contained more than millions of entities and relationships, e.g., DBPedia [26], Wikidata [45], Freebase [5], and YAGO [42]. Such large-scale KGs facilitate sharing and reusing information and are widely used in real applications.

Formally, a KG $\mathcal{G} = \langle \mathcal{E}, \mathcal{R}, \mathcal{F} \rangle$ could be considered a direct edge-labelled graph [21], which comprises a set $\mathcal{E}$ of entities (e.g., Tim Peters, Zen of Python), a set $\mathcal{R}$ of the relations (e.g., author), and a set $\mathcal{F}$ of facts.

A fact is a triple containing the head entity E_1, the relation R, and the tail entity E_2 to show that there is the relation from the tail entity to the head entity, denoted as $E_1 \xrightarrow{R} E_2$ [19]. To illustrate, the fact Zen of Python $\xrightarrow{\text{author}}$ Tim Peters shows that there is the author relation between Zen of Python and Tim Peters, i.e., it indicates the statement, 'Tim Peters is the author of Zen of Python'.

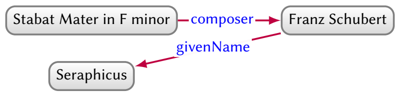
2.3 Motivation Example

This section aims to illustrate why and how we consider KG facts to construct test cases and check the inference ability of QA software. Figure 1 presents an example of correct and incorrect answers on UnifiedQA-v2-large (one of our studied QA software, see detail in Section 4.3), with questions generated from two KG facts. As shown in Figure 1, we have the following three observations.

(a) **KG facts could directly convert into test cases.** Given a KG $\mathcal{G} = \langle \mathcal{E}, \mathcal{R}, \mathcal{F} \rangle$, we could extract a set $\{f_1, f_2 \ldots, f_n\}$ of fact triples ($f_i = E_1^i \xrightarrow{R^i} E_2^i$) and generate the corresponding test case by parsing the entities and relations in f_i. Figures 1(a) and 1(b) show the translation from two facts into two test cases. To illustrate, Stabat Mater in F minor $\xrightarrow{\text{composer}}$ Franz Schubert indicates that there is the composer relation between the two entities Stabat Mater in F minor and Franz Schubert. We could employ NLP techniques to convert the fact into the statement, "Franz Schubert is the composer of Stabat Mater in F minor." Based on the above statement, we could quickly generate a related question with the expected answer "Yes", by moving the verb 'is' at the beginning of the sentence.
Question 1: "*Is Franz Schubert the composer of Stabat Mater in F minor?*"

(b) **Logical inference could construct new test cases.** We could employ inference methods to generate new statements[1] from obtained statements. For example, Figure 1(c) illustrates that, from two antecedent statements, "Franz Schubert is the composer of Stabat Mater in F minor" and "Seraphicus is the given name of Franz

[1]We omit the detailed reasoning of triples and statements here, which will be discussed in Section 3.1.



(a) Two Facts in Knowledge Graph

Question 1:	Is Franz Schubert the composer of Stabat Mater in F minor?
Context 1:	null
Answer 1:	Yes (✓)
Question 2:	Is Seraphicus the given name of Franz Schubert?
Context 2:	null
Answer 2:	Yes (✓)

(b) Two antecedent test cases with right answers

Antecedent 1:	Franz Schubert is the composer of Stabat Mater in F minor.
Antecedent 2:	Seraphicus is the given name of Franz Schubert.
Consequence:	Seraphicus is the given name of the composer of Stabat Mater in F minor.

(c) Reasoning for new statements

Question 3:	Is Seraphicus the given name of the composer of Stabat Mater in F minor?
Context 3:	Franz Schubert is the composer of Stabat Mater in F minor. Seraphicus is the given name of Franz Schubert.
Answer 3:	No (✗)

(d) Consequence test cases with error answers

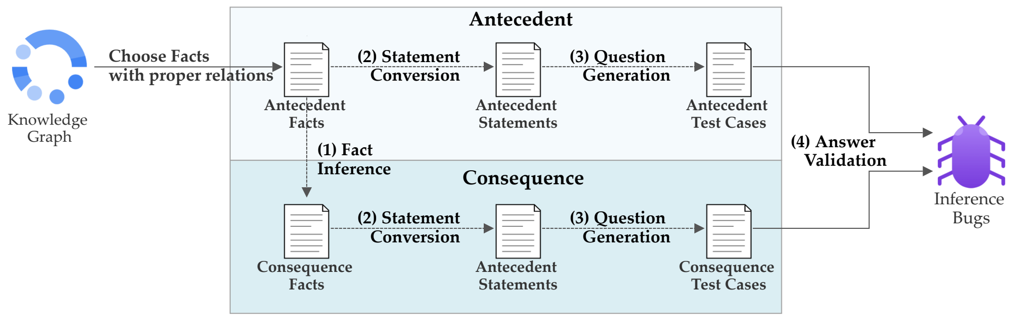
Figure 1: An example of test cases containing questions and contexts generated from KG facts with correct (✓) and incorrect (✗) answers on UnifiedQA-v2-large.

Schubert", we could naturally generate the consequence statement: "Seraphicus is the given name of the composer of Stabat Mater in F minor". Similarly, we could quickly generate a new question.
Question 3: *"Is Seraphicus the given name of the composer of Stabat Mater in F minor?"*
Besides, to support more information, we could also add the above two antecedent statements into the context of the test cases, as shown in Figure 1(d).

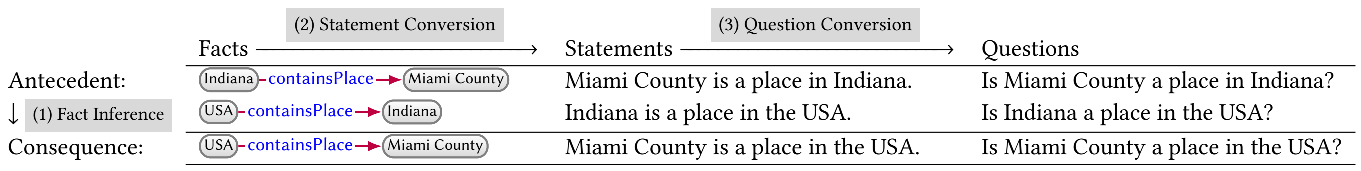
(c) **Answers of test cases with the inference relation could reveal that QA software lacks the inference ability, i.e., it has inference bugs.** To illustrate, as the answers to the antecedent question (Questions 1 and 2) could reason out the answer to the consequence question (Question 3), when QA software correctly answers the first two test cases, as shown in Figure 1(b), it would be *logically* expected to answer the following test case correctly. However, it returns the wrong answer, as shown in Figure 1(d). We can say that the QA software UnifiedQA-v2-large fails in the inference testing under the above three test cases, i.e., an inference bug is detected.

3 APPROACH

Before showing the overall framework of our approach, we present the formulation of our testing method as follows.
Knowledge **G**raph driven **I**nference **T**esting (**KGIT**). Given a KG $\mathcal{G}$ and the QA software $\mathcal{S}_{QA}$, we construct two kinds of test cases,



(a) Four main steps in our approach



	Facts $\xrightarrow{\text{(2) Statement Conversion}}$	Statements $\xrightarrow{\text{(3) Question Conversion}}$	Questions
Antecedent:	Indiana —containsPlace→ Miami County	Miami County is a place in Indiana.	Is Miami County a place in Indiana?
↓ (1) Fact Inference	USA —containsPlace→ Indiana	Indiana is a place in the USA.	Is Indiana a place in the USA?
Consequence:	USA —containsPlace→ Miami County	Miami County is a place in the USA.	Is Miami County a place in the USA?

(b) An example to show how facts reason out new facts (i.e., USA —containsPlace→ Miami County by **(1) Fact Inference**) and how facts convert into statements (**(2) Statement Conversion**) and questions (**(3) Question Generation**).

Figure 2: The overall framework of our approach.

i.e., antecedent and consequence test cases, in our testing scenario, where antecedent test cases (denoted as t^{ant}) are generated from fact triples in $\mathcal{G}$, and consequence test cases (denoted as t^{con}) are constructed by employing inference rules. We guarantee that the answer ($\mathcal{S}_{QA}(t^{con})$) of the QA software to a consequence test case t^{con} could be **inferred** from the answers $\mathcal{S}_{QA}(t_i^{ant})$ ($1 \le i \le n$) to a set of antecedent test cases $\{t_1^{ant}, t_2^{ant}, \ldots, t_n^{ant}\}$, which is presented as the following expression in formal logic:

$$\frac{\text{Antecedents:} \quad \mathcal{S}_{QA}(t_1^{ant}), \mathcal{S}_{QA}(t_2^{ant}), \cdots, \mathcal{S}_{QA}(t_n^{ant})}{\text{Consequence:} \quad \mathcal{S}_{QA}(t^{con})}$$

If we observe that the answers to the antecedent test cases are correct, but the answers to the consequence test case are wrong, *an inference bug is detected.*

Next, we present a detailed description of our approach, which comprises four steps: Fact Inference, Statement Conversion, Question Generation, and Question Validation, as shown in Figure 2(a). In our approach, we can consider constructing test cases with three objects: facts, statements, and questions (see an example in Figure 2(b)). In Step 1, Fact Inference, we pre-define the inference procedure to generate new facts; in Step 2, Statement Conversion, statements (i.e., declarative sentences) are constructed from entities and relationships in the facts; and Step 3, Question Generation, contains the generation of the pairs of test questions and answers.

3.1 Fact Inference

This section proposes four Inference Rules (IRs) to guide the inference procedure of our approach. Based on these rules, we choose proper relations (as summarized in Table 2) to extract a set of fact triples that satisfy the conditions of IRs, with entities and relations from the KG. Next, we will introduce each IR in turn.

Table 2: The Relations Applied in Four Inference Rules.

Rule	Applied Relations extracted from the KG
Composition	$R_1 \in$ {author, editor, creator, organizer, director,...}
	$R_2 \in$ {gender, alumniOf, birthPlace, children, deathPlace,...}
Alias	$R \in$ {actor, affiliation, alumniOf, author, birthPlace, ...}
Transitive	$R \in$ {hasPart, containsPlace}
Inverse	$R \in$ {hasPart, parent, children, author, editor,...}

[1] Due to space limitation, we list the top five studied relations here. More details of applied relations are available on our online repository, see Section 9.1.

3.1.1 *IR1: Composition Rule.*

Inspired by the fact that two simple relations can be composed into complex relations in logic systems [4], we propose the Composition Rule in the logical expression:

$$\frac{\text{Antecedent:} \quad E_1 \xrightarrow{R_1} E_2 \qquad E_2 \xrightarrow{R_2} E_3}{\text{Consequence:} \quad E_1 \xrightarrow{R_1 \circ R_2} E_3}$$

The inner logic of the Composition Rule is that if (a) two entities E_1 and E_2 have the R_1 relation, and (b) two entities E_2 and E_3 have the R_2 relation, by composing these two relations into a new relation $R_1 \circ R_2$, we can infer that E_1 and E_3 have the $R_1 \circ R_2$ relation.

In this paper, we mainly choose author, editor, creator, organizer, director, and so on as the R_1 relationships. For the R_2 relationships, we select gender, alumniOf, birthPlace, children, deathPlace, and so on. A more detailed list of all relationships used is available in our source code repository (see Section 9.1).

We extract the fact triples containing the selected relations R_1 and R_2 in the KG, which satisfy the form $E_1 \xrightarrow{R_1} E_2$ and $E_2 \xrightarrow{R_2} E_3$, and infer new triples based on the Composition Rule. To illustrate, we apply the Composition Rule on the facts with the creator relation and the gender relation as follows:

$$\begin{array}{rl} \text{Antecedent:} & \boxed{\text{Xfce}} \xrightarrow{\text{creator}} \boxed{\text{Oliver Fourdan}} \\ & \boxed{\text{Oliver Fourdan}} \xrightarrow{\text{gender}} \boxed{\text{Male}} \\ \hline \text{Consequence:} & \boxed{\text{Xfce}} \xrightarrow{\text{creator}\circ\text{gender}} \boxed{\text{Male}} \end{array}$$

As can be seen, based on the composition relation creator∘gender, we obtain the new triples $\boxed{\text{Xfce}} \xrightarrow{\text{creator}\circ\text{gender}} \boxed{\text{Male}}$.

3.1.2 IR2: Alias Rule. In the KG, one object would have multiple aliases, which are connected by the alias relations. The logical expression of the Alias Rule is presented as follows:

$$\begin{array}{rl} \text{Antecedent:} & \boxed{E_1} \xrightarrow{R} \boxed{E_2} \\ & \boxed{E_1} \xrightarrow{\text{alias}} \boxed{E_3} \\ \hline \text{Consequence:} & \boxed{E_3} \xrightarrow{R} \boxed{E_2} \end{array}$$

The inner logic of the Alias Rule is that if (a) two entities $\boxed{E_1}$ and $\boxed{E_2}$ have the R relation, and (b) two entities $\boxed{E_1}$ and $\boxed{E_3}$ are aliases of the same object, we can infer that $\boxed{E_3}$ and $\boxed{E_2}$ still have the same R relation.

For the Alias Rule, we select more than ten relations in the KG, including actor, affiliation, alumniOf, author, birthPlace, and so on. We extract the fact triplet containing the selected relations and the alias relations in the KG, which satisfy the form $\boxed{E_1} \xrightarrow{R} \boxed{E_2}$ and $\boxed{E_1} \xrightarrow{\text{alias}} \boxed{E_3}$, and infer new triples based on the Alias Rule. To illustrate, we apply the Alias Rule on the facts with the author relation and the alias relation as follows:

$$\begin{array}{rl} \text{Antecedent:} & \boxed{\text{Zen of Python}} \xrightarrow{\text{author}} \boxed{\text{Tim Peters}} \\ & \boxed{\text{Zen of Python}} \xrightarrow{\text{alias}} \boxed{\text{PEP 20}} \\ \hline \text{Consequence:} & \boxed{\text{PEP 20}} \xrightarrow{\text{author}} \boxed{\text{Tim Peters}} \end{array}$$

As can be seen above, we can infer that $\boxed{\text{PEP 20}}$ and $\boxed{\text{Tim Peters}}$ have the author relation from (a) $\boxed{\text{Zen of Python}}$ and $\boxed{\text{Tim Peters}}$ have the author relation, and (b) $\boxed{\text{Zen of Python}}$ and $\boxed{\text{PEP 20}}$ have the alias relation.

3.1.3 IR3: Transitive Rule. In the KG, some relations are labeled as transitive, i.e., $R \in \mathcal{R}^+$ (we employ $\mathcal{R}^+ \subseteq \mathcal{R}$ to denote the set of transitive relations in the KG). Based on such transitive relations, we propose the Transitive Rule in the following logical expression:

$$\begin{array}{rl} \text{Antecedent:} & \boxed{E_1} \xrightarrow{R} \boxed{E_2} \\ & \boxed{E_2} \xrightarrow{R} \boxed{E_3} \\ \hline \text{Consequence:} & \boxed{E_1} \xrightarrow{R} \boxed{E_3} \end{array}$$

This rule follows the definition of transitive rules, i.e., if the first entity $\boxed{E_1}$ is R-related to the second entity $\boxed{E_2}$, and the second entity $\boxed{E_2}$ is R-related to the third entity $\boxed{E_3}$, then the first entity $\boxed{E_1}$ must be R-related to the third entity $\boxed{E_3}$.

In this paper, we choose two relations, hasPart and containsPlace, as the transitive relations in the Transitive Rule, extract fact triples satisfying the conditions of the Transitive Rule, and reason out new triples. For example, we apply the Transitive Rule on two triples containing the containsPlace relation as follows:

$$\begin{array}{rl} \text{Antecedent:} & \boxed{\text{USA}} \xrightarrow{\text{containsPlace}} \boxed{\text{Indiana}} \\ & \boxed{\text{Indiana}} \xrightarrow{\text{containsPlace}} \boxed{\text{Miami County}} \\ \hline \text{Consequence:} & \boxed{\text{USA}} \xrightarrow{\text{containsPlace}} \boxed{\text{Miami County}} \end{array}$$

By the Transitive Rule, we deduce the new triple to show that $\boxed{\text{USA}}$ and $\boxed{\text{Miami County}}$ have the containsPlace relation.

3.1.4 IR4: Inverse Rule. In the KG, some relations are irreversible, i.e., if the first entity $\boxed{E_1}$ is R-related to the second entity $\boxed{E_2}$, the second entity $\boxed{E_2}$ **can not** be R-related to $\boxed{E_1}$ (i.e., the inverse relation does not exist). Based on such irreversible relations, we propose the Inverse Rule in the logical expression as follows:

$$\begin{array}{rl} \text{Antecedent:} & \boxed{E_1} \xrightarrow{R} \boxed{E_2} \\ \hline \text{Consequence:} & \boxed{E_2} \xrightarrow{\neg R} \boxed{E_1} \end{array}$$

In the above expression, we employ $\neg R$ to show that $\boxed{E_2}$ and $\boxed{E_1}$ do not have the R relation. We choose more than ten relations as the irreversible relations applied in the Inverse Rule, including hasPart, parent, children, author, editor, and so on. The following example shows how the Inverse Rule is applied on triples with the hasPart relation, which is obviously irreversible:

$$\begin{array}{rl} \text{Antecedent:} & \boxed{\text{GNU}} \xrightarrow{\text{hasPart}} \boxed{\text{GNU Linux-libre}} \\ \hline \text{Consequence:} & \boxed{\text{GNU Linux-libre}} \xrightarrow{\neg\text{hasPart}} \boxed{\text{GNU}} \end{array}$$

We construct the inference procedure based on the Inverse Rule: as $\boxed{\text{GNU Linux-libre}}$ is a part of $\boxed{\text{GNU}}$, $\boxed{\text{GNU}}$ cannot be a part of $\boxed{\text{GNU Linux-libre}}$.

3.2 Statement Conversion

In this step, we aim to convert the selected fact triples into the form of declarative sentences, which are further used to generate questions and contexts. We employ conversion patterns (denoted as Convert(f)) to deal with the entities and relations in the fact triples (f) and convert them into the subject, the predicate, and the object in the statement.

As shown in Table 3, we list five conversion patterns applied in our approach. In Table 3, the most common pattern is the first pattern for general fact triples. The second pattern is a special handling of a part of the relationship, such as gender. The third pattern is mainly used for the generation of declarative sentences in the Inverse Rule. The fourth and fifth patterns are mainly used for statement generation in the Composition Rule.

For all the relations we use, we set up a mapping from the relation to the natural language representation of the relation, which is denoted as the Extension(R) function. Specifically, the Extension(R) function mainly adds the appropriate articles and prepositions to the relation. Table 4 shows some examples of the Extension(R) function, while the detailed list of Extension(R) function for all applied relations is available in our source code repository (see Section 9.1). For example, for the relation 'creator', we would add the article 'the' and the preposition 'of' to form its representation 'the creator of'.

As can be seen in Table 3, statements are constructed by applying proper patterns[2]. For example, we could convert the given triple $\boxed{\text{Stabat Mater}} \xrightarrow{\text{composer}\circ\text{giveName}} \boxed{\text{Seraphicus}}$ into the statement "Seraphicus is the given name of the composer of Stabat Mater".

[2]Because determining whether an entity is singular or plural is a difficult task, we adopt a simple assumption in the template that all entities are singular. Therefore, here we use 'is' in all our templates. We discuss the percentage of grammar errors in RQ1 in Section 5.

Table 3: Five conversion patterns from fact triples into statements

ID	Applied Fact f	Convert Pattern Convert(f)	Example: f	$\Rightarrow$	Convert(f)
1	$E_1 \xrightarrow{R} E_2$	E_2 is Extension(R) E_1	Xfce $\xrightarrow{\text{creator}}$ Oliver Fourdan	$\Rightarrow$	Oliver Fourdan is *the creator of* Xfce.
2	$E_1 \xrightarrow{\text{gender}} E_2$	E_1 is a E_2	Oliver Fourdan $\xrightarrow{\text{gender}}$ Male	$\Rightarrow$	Oliver Fourdan is a male.
3	$E_1 \xrightarrow{\neg R} E_2$	E_2 is **not** Extension(R) E_1	GNU Linux-libre $\xrightarrow{\neg\text{hasPart}}$ GNU	$\Rightarrow$	GNU is **not** *a part of* GNU Linux-libre.
4	$E_1 \xrightarrow{R_1 \circ R_2} E_3$	E_3 is Extension(R_2) Extension(R_1) E_1	Stabat Mater $\xrightarrow{\text{composer}\circ\text{giveName}}$ Seraphicus	$\Rightarrow$	Seraphicus is *the given name of* *the composer of* Stabat Mater.
5	$E_1 \xrightarrow{R_1\circ\text{gender}} E_3$	Extension(R_1) E_1 is a E_3.	Xfce $\xrightarrow{\text{creator}\circ\text{gender}}$ Male	$\Rightarrow$	*The creator of* Xfce is a male.

Table 4: Examples for extension function.

Relation	Word	Article	Preposition	Extension(f)
creator	creator	the	of	the creator of
author	author	the	of	the author of
containsPlace	place	a	in	a place in
hasPart	part	a	of	a part of

Besides, as the fact triples applied in IRs could be considered antecedent/consequence triples, we call the converted statements antecedent/consequence statements correspondingly.

3.3 Question Generation

In the question generation process, we mainly divide it into two substeps - antecedent and consequence test question construction. Table 5 shows examples of the process of question generation.

(a) In the antecedent question generation, we generate the questions and answers based on the antecedent statement converted from KG facts. We move the word "is" to the beginning of the sentence and add a question mark at the end to complete the construction of the general question with the labeled answer "Yes", e.g., Q1&Q2, as shown in Table 5. In antecedent test cases, we do not set the context to test whether the QA software knows the relevant knowledge.

(b) In the consequence question construction, we construct two kinds of consequence questions, called basic consequence questions and extended consequence questions.

- **Basic Consequence Question Generation.** For the consequence statements, we also move the 'is' to the beginning of the sentence and add a question mark at the end. Meanwhile, we use the antecedent statements extracted from Section 3.2 as the context of questions (e.g., C3=S1+S2 in Table 5). We expect the model to answer "Yes" for all IRs except for the Inverse Rule, where we expect the model to answer "No", e.g., Q3 in Table 5.
- **Extended Consequence Question Generation.** In addition, to avoid QA software always guessing "Yes" when it does not know, we aim to construct the related questions with the "No" answers[3]. We attempt to construct an additional question by replacing the subject of the basic consequence question with randomly chosen entities in the KG, e.g., Q4 replaces "Tim Peter" with "Donald Knuth" as shown in Table 5. As the randomly selected entities have an extremely low probability of having relations with the objects, the expected answers to extended consequence questions are "No".

Because we directly replace the placeholders in the patterns with the names of the entities and relations, it is possible to use the wrong article or prepositions in the generated sentences. We correct the grammar atomically using a FLAN-T5-large based [10] grammar correction model [34] fine-tuned on the JFLEG dataset [17, 32]. Specifically, we first obtain the suggestion of the grammar correction model, perform a Myers difference operation on the corrected sentence and the original sentence, and accept the grammar correction model's suggestion when the inserted and deleted words contain only prepositions and articles.

[3]For the Inverse Rule, though it generates the question with the expected "No" answer, we still apply the same approach to generate new questions with "No" answers.

3.4 Answer Validation

For "Yes/No" questions generated in the above steps, we only keep the English characters in the answer and convert them to lowercase. Then, if the answer contains "yes", it is considered a positive answer. If the answer contains "no", it is considered a negative answer. For the answers that do not contain either yes or no, we use SentenceTransformer [37] as a sentence embedding tool to convert the answers and yes/no into vectors. If their similarity is greater than the threshold, we regard the answer as "yes" or "no"; otherwise, we consider it invalid.

After analyzing the answers to our generated questions, we follow the formulation of our approach: if we observe that the answers to the antecedent test cases are correct, and the answers to the consequence test case are wrong, *an inference bug is detected.*

4 EXPERIMENT SETUPS

This section introduces the experiment setups, including research questions, data preparation, test objects, and experimental settings.

4.1 Research questions

We organize our experiments by addressing the following four research questions (RQs):

RQ1: How effective is our test case generation?

In the process of fact inference and question generation, it is possible to obtain wrong facts or generate questions with the wrong syntax. In this RQ, we randomly sample question&answer pairs from the test set and inspect (a) whether these questions match the facts and (b) whether the grammar is correct.

RQ2: How does KGIT perform in revealing bugs for QA software?

Table 5: Examples of generated question: Q1&Q2 are antecedent questions, Q3/Q4 are basic/extended consequence questions.

	Statement	Question	Answer
Antecedent	S1: Tim Peters is the author of Zen of Python.	Q1: Is Tim Peters the author of Zen of Python?	Yes
	S2: PEP 20 is the alternative name for Zen of Python.	Q2: Is PEP 20 the alternative name for Zen of Python?	Yes
Consequence	S3: Tim Peters is the author of PEP20.	Q3: Is Tim Peters the author of PEP20? C3(S1+S2): Tim Peters is the author of Zen of Python. PEP 20 is the alternative name for Zen of Python.	Yes
		Q4: Is Donald Knuth the author of PEP20? C4(S1+S2): Tim Peters is the author of Zen of Python. PEP 20 is the alternative name for Zen of Python.	No

In this RQ, we evaluate the effectiveness of our approach in inference bug detection. Specifically, we aim to check the failed and passed number percentage of antecedent and consequence test cases on three studied models.

RQ3: How can the inference bugs discovered by our method help improve the models?

In this RQ, we employ the inference bugs detected by KGIT as a training set to retrain the model. We study the retrained model to investigate whether our method is helpful in improving the inference ability of QA models.

RQ4: How does KGIT perform compared with the SOTA method?

KGIT focuses on inference bugs, while previous metamorphic methods focus on inconsistency bugs between original and mutated test cases. In this RQ, we aim to compare our method with the SOTA method.

4.2 Data Preparation in the KG

To evaluate the effectiveness of our method KGIT for inference bugs, we select YAGO4 [46] as the KG source for question generation. YAGO4 is a general-purpose knowledge base based on Wikidata, which collects more than 50 million entities and 2 billion facts.

For YAGO4, we first import it into the Neo4j database, followed by matching the entity and relationship pairs that meet the needs of the IRs in the graph database. It makes complex entity matching in an acceptable time possible with KGs stored in a graph database.

4.3 Test Objects

We conduct our experiment on the state-of-the-art QA models - UnifiedQA, which is also used in previous studies [8, 39].

UnifiedQA is a QA language model based on the T5 model, which unifies four different QA question formats. We use the newest version of UnifiedQA - UnifiedQA V2 [24] to construct our test objects, which is pre-trained on 20 datasets, including SQuAD 1.1 [36], SQuAD 2 [35], BoolQ [11], and MultiRC (yes/no) [23].

UnifiedQA provides models with various settings for users. The difference between these settings is that they have different parameters, which correspond to the different layer numbers and hidden state sizes of the T5 model. Generally speaking, the larger the number of parameters in a deep learning model, the more knowledge it can store, and the more complex questions it can handle. Therefore, this paper employs three sizes of UnifiedQA models (in increasing order of size): UnifiedQA-v2-small, UnifiedQA-v2-base, and UnifiedQA-v2-large.

Table 6: The Scale of the Test Dataset Generated.

Rules	IR1	IR2	IR3	IR4	#Total
#TestCase	900,672	716,521	128,486	1,092,385	2,838,064
#Entity	317,784	660,700	105,999	1,089,336	2,173,819

4.4 Experimental Settings

To ensure manageable results, we constrain the number of matching triples in the KG. Matching and returning too many entities and relations in the graph database requires a large amount of memory and computational resources. For the relations in IR1, IR2, and IR3, we limit the number of entities returned to 50,000. For the relationships in IR4, we set a higher limit of 150,000 since we do not need to match multi-hop relational paths. Finally, the scale of the test dataset we generated from the knowledge graph is shown in Table 6. As seen in Table 6, we generate more than 2.8 million test cases.

To calculate the answer similarity (see Section 3.4), we choose the all-mpnet-base-v2 version of SentenceTransformer [37] model. When using UnifiedQA for answering questions, we limit the length of the question to a maximum of 256 tokens to save GPU memory usage and use batch size 64 for model inference.

5 EXPERIMENT RESULTS

RQ1: Correctness of Generated Test Cases

Motivation&Approach. In this RQ, we first try to ensure that our approach can generate realistic question-answering test cases. We randomly sample 100 test cases for each IR and manually inspect whether these 400 (100×4) sampled test cases are realistic.

Specifically, we categorize the generated test questions into three cases during the manual inspection:

- Realistic: the generated test questions and answers are realistic and reasonable.
- Grammatical error: there is a grammatical problem in the generated test question or context.
- Factual error: the facts extracted from the knowledge graph do not match the real world.

As most of the knowledge in the KG YAGO4 [46] is beyond our knowledge, we use Wikidata [49] as the criterion to check factual errors during manual checking. The manual check comprises two steps. (1) The first two authors independently labeled mutated sentences. Then we measured the inter-rater agreement between these two authors regarding Cohen's Kappa coefficient [48]. (2) The

Table 7: The Manual Inspection Results for Test Cases.

	IR1	IR2	IR3	IR4
Realistic	94%	96%	94%	100%
Grammatical error	6%	4%	5%	0%
Factual error	0%	0%	1%	0%
Cohen's Kappa	71.15%	64.54%	71.29%	100.00%

first two authors discussed the disagreements with the third author to get a consistent result.

Results. Table 7 summarizes the results of our manual check[4]. It can be seen that Cohen's Kappa coefficients under four inference rules are around 70%, which indicates the high consistency of the labeling results. On the whole, Cohen's Kappa coefficient on all inference rules is 68.98%. We can see that most of the generated test samples are realistic, and the percentage of realistic questions generated by all four IRs is greater than 90%. Specifically, we observe that (a) the grammatical error rate is similar among all the questions generated by the first three IRs, and the major problem is the missing articles; (b) some entities and relationships in the KG are not suitable for generating test questions; and (c) the KG contains some factual errors resulting in unrealistic questions being generated, but the percentage of factual errors is extremely small.

Answer to **RQ1**. The four proposed IRs can produce a high percentage of realistic testing question and answer pairs.

RQ2: Effectiveness for Bug Detection

Motivation&Approach. To study the effectiveness of KGIT in detecting inference bugs using IRs, we count the number of test cases that passed the antecedent test (i.e., return the correct answer to the *antecedent* questions) and the number of inference bugs with inconsistent answers (i.e., return the wrong answer to the *consequence* questions) by applying four IRs on three QA models.

Results. Table 8 lists the number of revealed bugs on the three studied models, i.e., UnifiedQA-V2 small, base, and large, with four IRs. Column '#Bug' shows the number of failed basic test questions. Column '#BugExt' presents the number of test cases with failed basic **or** extended consequence questions. Column '#Total' is the number of test cases that pass the antecedent test.

We have the following observations in Table 8. (a) On the whole, KGIT detects a considerable number of inference bugs on three models based on four IRs. (b) The larger the model size, the more antecedent questions are correct, i.e., more powerful models could correctly answer more antecedent questions. (c) For basic consequence test cases, the number of bugs is lower for models with smaller parameters. (d) For extended consequence test cases, the larger the model, the fewer IR1 bugs are revealed. For IR2 and IR4, the base model gives more inconsistent answers. For IR3, the number of extended bugs is similar among the three models. (e) The extended consequence test cases detect more inconsistent answers.

[4]Manual inspection results are available in our online repository (Section 9.1).

Answer to **RQ2**. The studied three models usually cannot reason out the correct answers to (basic and extended) consequence questions. Our method can detect a considerable number of inference bugs in QA software effectively.

RQ3: Helpfulness for QA model improvement

Motivation&Approach. In this RQ, we investigate the helpfulness of our method for fixing these inference bugs of QA software. In related studies [14, 15, 47], retraining with generated samples is a widely adopted approach to fix Deep Learning models. Following this methodology, we retrain a new model with samples generated by four proposed IRs. The change in bug rates between the original and retrained models will show how helpful IRs can be in repairing the QA models.

Here, we split all the samples (see Table 6) generated from the KG into training, validation, and test sets with the 6:2:2 ratio [31]. Next, we conduct retraining on the training set for the small, base, and large versions of the UnifiedQA-v2 model. We train for 10,000 steps, saving a checkpoint every 1,000 steps. The batch size is 64, and we use the AdamW optimizer [29] with the learning rate 3e-5.

Results. Figure 3 indicates the trend of the inference bug rates during retraining. It can be observed from Figure 3 that (a) the bug rate on the test set *decreases* with increasing steps of retraining for all model sizes, i.e, retraining the QA model can fix the inference bugs of the model; (b) for the base and small models, it can be observed that bugs triggered by IR4 are the easiest to fix, and the bug rate decreases very quickly; (c) the large version of the model has more bugs triggered by IR4. Therefore the same step has more bug rate than the base model.

Answer to **RQ3**. Retraining the models with detected inference bugs can fix the bugs. IR4 is the easiest to fix among inference rules.

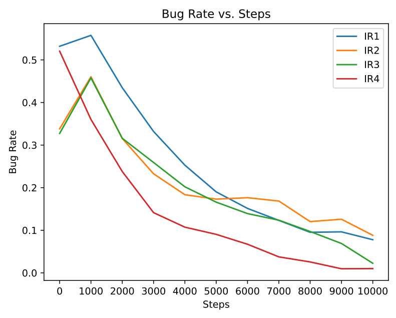
RQ4: Comparison with SOTA testing techniques

Motivation&Approach. In this RQ, we aim to compare KGIT with the SOTA metamorphic testing methods for QA software (e.g., QAQA [39]). As Section 2.1 mentions, these SOTA methods test QA models by mutating the original QA dataset to obtain new QA pairs. In contrast, KGIT directly extracts relations and entities from the KG to test the inference ability of the model. It is worth noting that KGIT is entirely different from the previous metamorphic testing methods in terms of the seed dataset and the testing focus (see details in Section 6.3), which causes our method and metamorphic testing methods to be *unable to compare directly*.

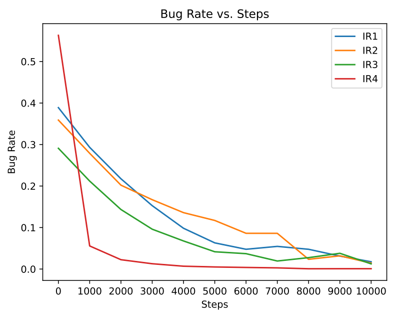
This RQ conducts a compromise to achieve the comparison between KGIT and QAQA. (a) We use the part of our dataset extracted from the KG that can pass the antecedent questions as the seed dataset. Due to the QAQA method efficiency limitation, we randomly sample 10,000 samples as a subset for comparison. (b) We perform the mutation of QAQA to detect the metamorphic bugs and apply KGIT to obtain inference bugs. (c) We directly compare the number of bugs identified by the two methods and calculate

Table 8: The Effectiveness of Our Approach.

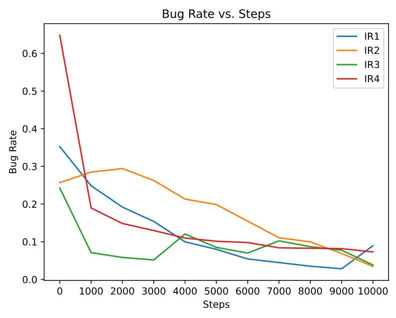
Model	IR1			IR2			IR3			IR4		
	#Bug	#BugExt	#Total	#Bug	#BugExt	#Total	#Bug	#BugExt	#Total	#Bug	#BugExt	#Total
UnifiedQA-v2-small	936	90,772	291,446	758	16,949	83,128	36	21,615	91,432	646,513	646,672	646,976
UnifiedQA-v2-base	7,256	37,885	718,009	33,354	70,851	541,598	1,671	19,738	120,579	911,790	934,344	999,786
UnifiedQA-v2-large	4,035	10,503	759,750	47,817	60,612	592,558	5,924	28,795	122,779	853,436	882,189	1,021,865



(a) UnifiedQA-v2-small



(b) UnifiedQA-v2-base



(c) UnifiedQA-v2-large

Figure 3: The change in the model's inference bug rate with retraining steps.

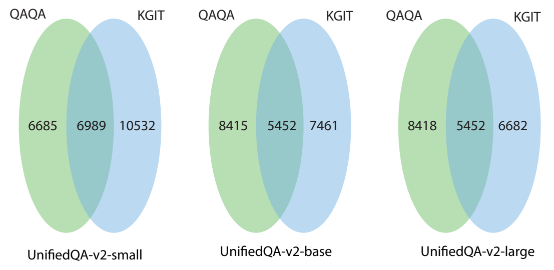


Figure 4: Overlaps on seed dataset compared to QAQA.

their overlaps (the bugs generated from the same seed sentences would be considered overlapped).

Results. Figure 4 shows the overlaps of bugs detected by QAQA and our proposed KGIT method. From the figure, we can observe that (a) KGIT detects a comparable number of bugs compared to QAQA on the same seed dataset; and (b) KGIT identifies 10532, 7461, and 6682 unique bugs of three models, which represent 60.11%, 57.78%, and 55.07% of all bugs detected by KGIT, respectively. We conclude that the testing method proposed in this paper is an excellent complement to previous methods.

Answer to **RQ4**. The inference bugs found by KGIT and those found by QAQA on the same seed dataset are similar in number. Meanwhile, 55.07%-60.11% of the bugs identified by KGIT are unique bugs.

6 DISCUSSION

We further discuss the results of our method in this section.

Table 9: The Examples of QA Software Unexpected Behaviors.

	Example		Expected/Actual
#1	Question	Q1: Is Cha the family name of the author of Sword of the Yue Maiden?	Yes/No
	Context	C1: Jin Yong is the author of Sword of the Yue Maiden. Cha is the family name of Jin Yong.	
#2	Question	Q2: Is the producer of What If a female?	No/male
	Context	C2: J.R. Rotem is the producer of What If. Male is the gender of J.R. Rotem.	
#3	Question	Q3: Is Lesser the family name of Zhongxuan?	No/zhongxuan
	Context	C3: Hong is the family name of Hong Hao. Zhongxuan is the alternative name for Hong Hao.	

6.1 Case Study

In this section, we investigate and classify the unexpected behavior (i.e., inference bugs) detected by KGIT. As shown in Table 9, we group incorrect test cases into three types with examples. Specifically, we summarize the types of errors as follows:

(1) **Wrong Yes or No Answers**. In this type of error, the QA software gives an answer ("Yes"/"No") different from the expected answer, i.e., it fails to infer a correct answer from the context. This type of error is the largest in number. For example, in Table 9, the answer to the question (Q1) should be "Yes" inferred from the context, but the model answers "No".

(2) **Answers with Incorrect Format**. In this type of error, the model fails to understand the type of question. For a general question, it should answer "Yes" or "No". Instead, the model understands the question and returns the answer as an entity in the sentence. In Table 9, for Q2, the answer should be "No", but the model answers "male". So the model is able to understand the question and give the correct answer, but the format of the answer is wrong.

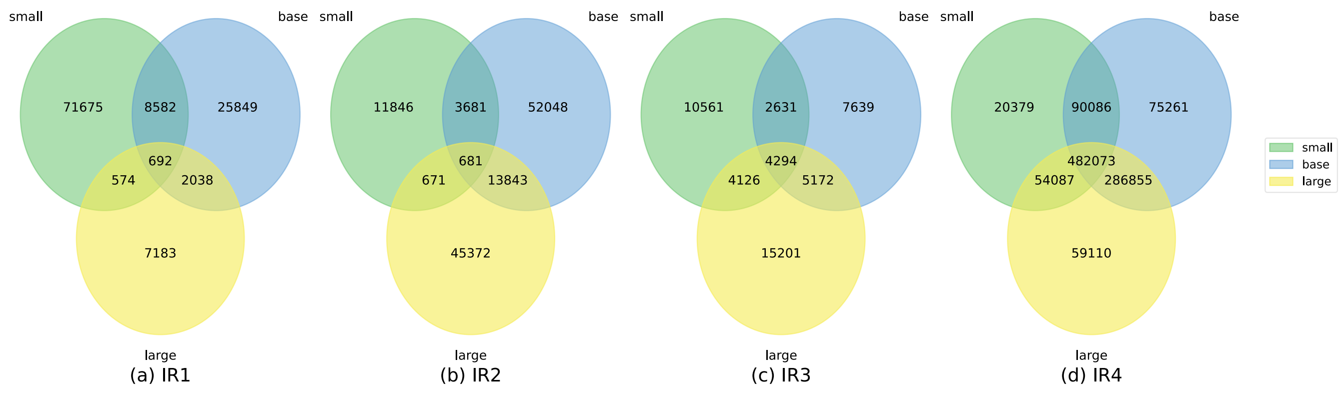


Figure 5: The overlaps of inference bugs detected by four IRs in three studied QA models.

(3) **Answers with Incorrect Keyword**. In this type of error, the model not only fails to understand the type of question but also gives the wrong keyword (Zhongxuan) for the question Q3 with the context C3, as the right family name is “Hong”.

6.2 Overlaps between Studied Models

Our approach proposes four different IRs to indentify inference errors in testing QA software. In Section 5, we choose three UnifiedQA-V2 models with different parameters of the same structure. In this section, we aim to investigate the overlaps of bugs revealed among the three studied models by four IRs.

Figure 5 shows bug overlaps under different IRs in the form of Venn diagrams on three studied models. Note that we count the number of errors in the extended consequence questions in Figure 5. In each subfigure, the green, blue, and yellow ellipses denote revealed bugs on ‘UnifiedQA-V2-small’, ‘UnifiedQA-V2-base’, and ‘UnifiedQA-V2-large’, respectively. We label the number of detected inference bugs on the intersection parts.

It can be observed from Figure 5 that (a) there are few overlapping parts of the bugs based on IR1, IR2, and IR3; (b) the overlap values show a large percentage of error test questions for IR4 because the majority of them are answered wrong.

6.3 Different Focuses between KGIT and QAQA

In Section 5 (RQ4), we have shown the overlap between the bugs detected by KGIT and QAQA on the same seed dataset. In this section, we want to point out that KGIT and QAQA (metamorphic testing methods) have completely different testing focuses.

Table 10 illustrates the contrast between two test cases generated from KGIT and QAQA. As shown in Table 10, the first row of the table is the QA pair generated by KGIT from the KG. Then QAQA mutates this QA pair to get the second row. It can be observed that KGIT and QAQA are two distinct testing solutions. KGIT intends to assess the QA models’ ability to reason out the composition of relations from the given context information, i.e., identify the new relation ‘a parent of the creator of’ as the combination of two relations ‘a parent of’ and ‘the creator of’. QAQA introduces various interrupting sentences (see italicized text in Table 10), before the context and the question to identify bugs in QA models, i.e., check whether the interrupting sentences could change the answer.

Table 10: KGIT and QAQA test examples.

	Context	Question
KGIT	Matt Groening is the creator of Dr. Nick. Homer Groening is a parent of Matt Groening.	Is Homer Groening a parent of the creator of Dr. Nick?
QAQA	*Winslow Homer is the creator of A Visit from the Old Mistress.* Matt Groening is the creator of Dr. Nick. Homer Groening is a parent of Matt Groening.	*Someone told me that Matt Groening is a parent of Matt Groening*, is Homer Groening a parent of the creator of Dr. Nick?

Since the testing focuses are different, and the defects tested are also distinct, we consider that *our method KGIT is an excellent complement to previous metamorphic testing approaches*.

6.4 The Insights of Our Study

Our method KGIT employs fact triples in the KG as the logic seeds to construct test cases containing questions and contexts with inference relations. The experimental results show that KGIT can detect a considerable number of inference bugs on three studied QA software by four IRs. Our study may lead to the insights for the following studies:

(a) KG facts could be directly employed to automatically generate test cases with labeled answers, which could be an efficient method to alleviate the limitation of human annotation effort.

(b) The inference relations between answers to questions is a novel and effective angle to achieve testing for QA software. We admit that the metamorphic relations largely inspire the idea of inference relations. However, we would like to emphasize that the core part of our method is the **logically inference** procedure (i.e., four IRs) to check the inference relations between fact triples, statements, and questions. We encourage the following researchers to focus on the other specific inference rules and construct more complex inference relations in the testing for QA software.

7 RELATED WORK

We present related work in the following two aspects:

7.1 QA Software Testing

The existing work on QA software testing can be mainly divided into reference-based and metamorphosis-based approaches.

(a) Reference-based techniques are one of the mainstream practices of QA software testing. Many works constructed benchmark datasets for QA software evaluation, such as BoolQ [11], MultiRC [23], NatQA [25], SQuAD1.1 [36], SQuAD2 [35].

(b) Due to many manual labeling efforts to construct the validation dataset, some other work used metamorphic testing methods to test QA software. Chen et al. [8] proposed a method named QAASKER for follow-up test case generation based on existing test questions. QAASKER generates questions with various questioning forms on the same fact and compares the consistency of the answers given by the QA software to detect bugs. Chen et al. proposed a property-based validation method named MT4MRC [9] for MRC software. Furthermore, Shen et al. [39] solved the problem that the QAASKER may generate some false positive test samples. Liu et al. [28] proposed a uniform fuzzing framework for question answering systems. They mutated the test cases by grammatical component, sentence structure, and adversarial perturbation mutation rules.

Instead of constructing new test cases by mutating the test set, our proposed method KGIT can generate test cases for the QA system in bulk by directly extracting facts from KGs.

7.2 Other NLP Software Testing

Many methods are proposed to test machine translation systems. Pesu et al. [33] constructed metamorphic relations to detect bugs in machine translation by the results of direct and indirect translations between different languages. Sun et al. [43] proposed TransRepair, which combines mutation testing and metamorphic testing to detect inconsistency bugs of machine translators. He et al. [15] proposed a metamorphic testing approach named structure-invariant testing (SIT). They generated similar mutation sentences for the original sentence and translated them into the target language by the translation model. To find bugs, SIT checks whether the structure of the translation result is the same between the original sentence and the mutated sentence. Gupta et al. [14] introduced a methodology named PaInv. PaInv generates sentences with similar syntax but different semantics by replacing a word in the sentence. He et al. [16] introduced referentially transparent inputs and proposed Purity. They provided a method to extract phrases from sentences and check the consistency of the translation under different contexts. Sun et al. [44] proposed a word-replacement based approach called CAT, which identifies word replacement with controlled impact.

8 THREATS TO VALIDITY

This section discusses the threats to the validity of our approach.

External Threats. First, we choose YAGO4 as the knowledge graph for generating questions and answers. Although YAGO4, as a general-purpose knowledge graph, contains a huge amount of entities and relationships, it is possible to use domain-specific knowledge graphs for some domain-specific QA software testing. We will discuss this in future work. Second, we choose a state-of-the-art QA model, UnifiedQA-v2, as the test target and use different parameters for comparison. In future work, we will test other QA models. Third, in this paper, we choose only four inference rules to test the inference ability of QA software. Adding some more complex inference rules may be able to obtain better testing ability. Finally, our approach is not suitable for QA software that can only perform extractive and abstractive QA tasks. In future work, we will extend the test objects to multiple-choice, extractive, and abstractive QA tasks.

Internal Threats. We use templates to generate question-answer test cases in our approach. Therefore, there may be grammatical errors in questions and context, such as missing articles. In addition, the knowledge graph itself may also contain factual errors.

9 CONCLUSION

In this paper, we have proposed a novel testing method for QA software named KGIT. Different from previous reference-based and metamorphic testing approaches, KGIT generates test datasets of QA models directly from fact triples in the KG and uses them to test the inference ability of QA software. According to our experiments, our approach can detect a large number of inference bugs in QA software. In future work, we plan to add more complex inference rules to improve the detection capabilities of our approach.

9.1 Replication Package

We provide data and source code used to conduct this study at **https://archive.softwareheritage.org/browse/origin/https://github.com/codeshuttler/KGIT**.

ACKNOWLEDGMENTS

The authors thank the anonymous reviewers for their time and invaluable feedback to improve our study. This work is supported by the National Natural Science Foundation of China (Grant No. 62172202, 62272221, 62172205, and 61972082) and the Major Program of the Natural Science Foundation of Jiangsu Higher Education Institutions of China under Grant No. 22KJA520008.

REFERENCES

[1] 2023. Amazon Alexa. https://alexa.amazon.com.
[2] 2023. Apple Siri. https://www.apple.com/siri/.
[3] 2023. Microsoft Cortana. https://www.microsoft.com/en-us/cortana.
[4] Franz Baader, Ian Horrocks, Carsten Lutz, and Uli Sattler. 2017. *Introduction to description logic.* Cambridge University Press.
[5] Kurt D. Bollacker, Colin Evans, Praveen K. Paritosh, Tim Sturge, and Jamie Taylor. 2008. Freebase: a collaboratively created graph database for structuring human knowledge. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, Jason Tsong-Li Wang (Ed.). ACM, 1247–1250. https://doi.org/10.1145/1376616.1376746
[6] Antoine Bordes, Nicolas Usunier, Alberto Garcia-Duran, Jason Weston, and Oksana Yakhnenko. 2013. Translating Embeddings for Modeling Multi-relational Data. In *Advances in Neural Information Processing Systems*, C.J. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K.Q. Weinberger (Eds.), Vol. 26. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2013/file/1cecc7a77928ca8133fa24680a88d2f9-Paper.pdf
[7] B Barla Cambazoglu, Mark Sanderson, Falk Scholer, and Bruce Croft. 2021. A review of public datasets in question answering research. In *ACM SIGIR Forum*, Vol. 54. ACM New York, NY, USA, 1–23.
[8] Songqiang Chen, Shuo Jin, and Xiaoyuan Xie. 2021. Testing Your Question Answering Software via Asking Recursively. In *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021.* IEEE, 104–116. https://doi.org/10.1109/ASE51524.2021.9678670
[9] Songqiang Chen, Shuo Jin, and Xiaoyuan Xie. 2021. Validation on machine reading comprehension software without annotated labels: a property-based method. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August*

23-28, 2021, Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta (Eds.). ACM, 590–602. https://doi.org/10.1145/3468264.3468569

[10] Hyung Won Chung, Le Hou, Shayne Longpre, Barret Zoph, Yi Tay, William Fedus, Eric Li, Xuezhi Wang, Mostafa Dehghani, Siddhartha Brahma, Albert Webson, Shixiang Shane Gu, Zhuyun Dai, Mirac Suzgun, Xinyun Chen, Aakanksha Chowdhery, Sharan Narang, Gaurav Mishra, Adams Yu, Vincent Y. Zhao, Yanping Huang, Andrew M. Dai, Hongkun Yu, Slav Petrov, Ed H. Chi, Jeff Dean, Jacob Devlin, Adam Roberts, Denny Zhou, Quoc V. Le, and Jason Wei. 2022. Scaling Instruction-Finetuned Language Models. *CoRR* abs/2210.11416 (2022). https://doi.org/10.48550/arXiv.2210.11416 arXiv:2210.11416

[11] Christopher Clark, Kenton Lee, Ming-Wei Chang, Tom Kwiatkowski, Michael Collins, and Kristina Toutanova. 2019. BoolQ: Exploring the Surprising Difficulty of Natural Yes/No Questions. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, Jill Burstein, Christy Doran, and Thamar Solorio (Eds.). Association for Computational Linguistics, 2924–2936. https://doi.org/10.18653/v1/n19-1300

[12] Ryan Daws. 2020. Medical chatbot using OpenAI's GPT-3 told a fake patient to kill themselves. https://www.artificialintelligence-news.com/2020/10/28/medical-chatbot-openai-gpt3-patient-kill-themselves/.

[13] Ramanathan V. Guha, Dan Brickley, and Steve Macbeth. 2016. Schema.org: evolution of structured data on the web. *Commun. ACM* 59, 2 (2016), 44–51. https://doi.org/10.1145/2844544

[14] Shashij Gupta, Pinjia He, Clara Meister, and Zhendong Su. 2020. Machine translation testing via pathological invariance. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 863–875. https://doi.org/10.1145/3368089.3409756

[15] Pinjia He, Clara Meister, and Zhendong Su. 2020. Structure-invariant testing for machine translation. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 961–973. https://doi.org/10.1145/3377811.3380339

[16] Pinjia He, Clara Meister, and Zhendong Su. 2021. Testing Machine Translation via Referential Transparency. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 410–422. https://doi.org/10.1109/ICSE43902.2021.00047

[17] Michael Heilman, Aoife Cahill, Nitin Madnani, Melissa Lopez, Matthew Mulholland, and Joel Tetreault. 2014. Predicting Grammaticality on an Ordinal Scale. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*. Association for Computational Linguistics, Baltimore, Maryland, 174–180. http://www.aclweb.org/anthology/P14-2029

[18] Julia Hirschberg and Christopher D Manning. 2015. Advances in natural language processing. *Science* 349, 6245 (2015), 261–266.

[19] Aidan Hogan, Eva Blomqvist, Michael Cochez, Claudia D'amato, Gerard De Melo, Claudio Gutierrez, Sabrina Kirrane, José Emilio Labra Gayo, Roberto Navigli, Sebastian Neumaier, Axel-Cyrille Ngonga Ngomo, Axel Polleres, Sabbir M. Rashid, Anisa Rula, Lukas Schmelzeisen, Juan Sequeda, Steffen Staab, and Antoine Zimmermann. 2021. Knowledge Graphs. *ACM Comput. Surv.* 54, 4, Article 71 (jul 2021), 37 pages. https://doi.org/10.1145/3447772

[20] Pin Ji, Yang Feng, Jia Liu, Zhihong Zhao, and Zhenyu Chen. 2022. ASRTest: automated testing for deep-neural-network-driven speech recognition systems. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 189–201.

[21] Shaoxiong Ji, Shirui Pan, Erik Cambria, Pekka Marttinen, and S Yu Philip. 2021. A survey on knowledge graphs: Representation, acquisition, and applications. *IEEE transactions on neural networks and learning systems* 33, 2 (2021), 494–514.

[22] Veton Kepuska and Gamal Bohouta. 2018. Next-generation of virtual personal assistants (microsoft cortana, apple siri, amazon alexa and google home). In *2018 IEEE 8th annual computing and communication workshop and conference (CCWC)*. IEEE, 99–103.

[23] Daniel Khashabi, Snigdha Chaturvedi, Michael Roth, Shyam Upadhyay, and Dan Roth. 2018. Looking Beyond the Surface: A Challenge Set for Reading Comprehension over Multiple Sentences. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2018, New Orleans, Louisiana, USA, June 1-6, 2018, Volume 1 (Long Papers)*, Marilyn A. Walker, Heng Ji, and Amanda Stent (Eds.). Association for Computational Linguistics, 252–262. https://doi.org/10.18653/v1/n18-1023

[24] Daniel Khashabi, Yeganeh Kordi, and Hannaneh Hajishirzi. 2022. UnifiedQA-v2: Stronger Generalization via Broader Cross-Format Training. *CoRR* abs/2202.12359 (2022). arXiv:2202.12359 https://arxiv.org/abs/2202.12359

[25] Tomás Kociský, Jonathan Schwarz, Phil Blunsom, Chris Dyer, Karl Moritz Hermann, Gábor Melis, and Edward Grefenstette. 2018. The NarrativeQA Reading Comprehension Challenge. *Trans. Assoc. Comput. Linguistics* 6 (2018), 317–328. https://doi.org/10.1162/tacl_a_00023

[26] Jens Lehmann, Robert Isele, Max Jakob, Anja Jentzsch, Dimitris Kontokostas, Pablo N. Mendes, Sebastian Hellmann, Mohamed Morsey, Patrick van Kleef, Sören Auer, and Christian Bizer. 2015. DBpedia - A large-scale, multilingual knowledge base extracted from Wikipedia. *Semantic Web* 6, 2 (2015), 167–195. https://doi.org/10.3233/SW-140134

[27] Yuhua Lin and Haiying Shen. 2017. SmartQ: A question and answer system for supplying high-quality and trustworthy answers. *IEEE Transactions on Big Data* 4, 4 (2017), 600–613.

[28] Zixi Liu, Yang Feng, Yining Yin, Jingyu Sun, Zhenyu Chen, and Baowen Xu. 2022. QATest: A Uniform Fuzzing Framework for Question Answering Systems. In *37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12.

[29] Ilya Loshchilov and Frank Hutter. 2019. Decoupled Weight Decay Regularization. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net. https://openreview.net/forum?id=Bkg6RiCqY7

[30] John Miller, Karl Krauth, Benjamin Recht, and Ludwig Schmidt. 2020. The effect of natural distribution shift on question answering models. In *International Conference on Machine Learning*. PMLR, 6905–6916.

[31] Mahmoud Nabil, Mohamed Aly, and Amir Atiya. 2015. Astd: Arabic sentiment tweets dataset. In *Proceedings of the 2015 conference on empirical methods in natural language processing*. 2515–2519.

[32] Courtney Napoles, Keisuke Sakaguchi, and Joel Tetreault. 2017. JFLEG: A Fluency Corpus and Benchmark for Grammatical Error Correction. In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 2, Short Papers*. Association for Computational Linguistics, Valencia, Spain, 229–234. http://www.aclweb.org/anthology/E17-2037

[33] Daniel Pesu, Zhi Quan Zhou, Jingfeng Zhen, and Dave Towey. 2018. A Monte Carlo Method for Metamorphic Testing of Machine Translation Services. In *3rd IEEE/ACM International Workshop on Metamorphic Testing, MET 2018, Gothenburg, Sweden, May 27, 2018*, Xiaoyuan Xie, Laura L. Pullum, and Pak-Lok Poon (Eds.). ACM, 38–45. https://doi.org/10.1145/3193977.3193980

[34] Peter Szemraj. 2022. flan-t5-large-grammar-synthesis (Revision d0b5ae2). https://doi.org/10.57967/hf/0138

[35] Pranav Rajpurkar, Robin Jia, and Percy Liang. 2018. Know What You Don't Know: Unanswerable Questions for SQuAD. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, ACL 2018, Melbourne, Australia, July 15-20, 2018, Volume 2: Short Papers*, Iryna Gurevych and Yusuke Miyao (Eds.). Association for Computational Linguistics, 784–789. https://doi.org/10.18653/v1/P18-2124

[36] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. 2016. SQuAD: 100, 000+ Questions for Machine Comprehension of Text. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing, EMNLP 2016, Austin, Texas, USA, November 1-4, 2016*, Jian Su, Xavier Carreras, and Kevin Duh (Eds.). The Association for Computational Linguistics, 2383–2392. https://doi.org/10.18653/v1/d16-1264

[37] Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, EMNLP-IJCNLP 2019, Hong Kong, China, November 3-7, 2019*, Kentaro Inui, Jing Jiang, Vincent Ng, and Xiaojun Wan (Eds.). Association for Computational Linguistics, 3980–3990. https://doi.org/10.18653/v1/D19-1410

[38] Marco Tulio Ribeiro, Tongshuang Wu, Carlos Guestrin, and Sameer Singh. 2020. Beyond accuracy: Behavioral testing of NLP models with CheckList. *arXiv preprint arXiv:2005.04118* (2020).

[39] Qingchao Shen, Junjie Chen, Jie M. Zhang, Haoyu Wang, Shuang Liu, and Menghan Tian. 2022. Natural Test Generation for Precise Testing of Question Answering Software. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*. ACM, 71:1–71:12. https://doi.org/10.1145/3551349.3556953

[40] Weijun Shen, Yanhui Li, Lin Chen, Yuanlei Han, Yuming Zhou, and Baowen Xu. 2020. Multiple-boundary clustering and prioritization to promote neural network retraining. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 410–422.

[41] Kuldeep Singh, Muhammad Saleem, Abhishek Nadgeri, Felix Conrads, Jeff Z Pan, Axel-Cyrille Ngonga Ngomo, and Jens Lehmann. 2019. Qaldgen: Towards microbenchmarking of question answering systems over knowledge graphs. In *The Semantic Web–ISWC 2019: 18th International Semantic Web Conference, Auckland, New Zealand, October 26–30, 2019, Proceedings, Part II 18*. Springer, 277–292.

[42] Fabian M. Suchanek, Gjergji Kasneci, and Gerhard Weikum. 2007. Yago: a core of semantic knowledge. In *Proceedings of the 16th International Conference on World Wide Web, WWW 2007, Banff, Alberta, Canada, May 8-12, 2007*, Carey L. Williamson, Mary Ellen Zurko, Peter F. Patel-Schneider, and Prashant J. Shenoy (Eds.). ACM, 697–706. https://doi.org/10.1145/1242572.1242667

[43] Zeyu Sun, Jie M. Zhang, Mark Harman, Mike Papadakis, and Lu Zhang. 2020. Automatic testing and improvement of machine translation. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19*

July, 2020, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 974–985. https://doi.org/10.1145/3377811.3380420

[44] Zeyu Sun, Jie M. Zhang, Yingfei Xiong, Mark Harman, Mike Papadakis, and Lu Zhang. 2022. Improving Machine Translation Systems via Isotopic Replacement. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 1181–1192. https://doi.org/10.1145/3510003.3510206

[45] Thomas Pellissier Tanon, Denny Vrandecic, Sebastian Schaffert, Thomas Steiner, and Lydia Pintscher. 2016. From Freebase to Wikidata: The Great Migration. In *Proceedings of the 25th International Conference on World Wide Web, WWW 2016, Montreal, Canada, April 11 - 15, 2016*, Jacqueline Bourdeau, Jim Hendler, Roger Nkambou, Ian Horrocks, and Ben Y. Zhao (Eds.). ACM, 1419–1428. https://doi.org/10.1145/2872427.2874809

[46] Thomas Pellissier Tanon, Gerhard Weikum, and Fabian M. Suchanek. 2020. YAGO 4: A Reason-able Knowledge Base. In *The Semantic Web - 17th International Conference, ESWC 2020, Heraklion, Crete, Greece, May 31-June 4, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12123)*, Andreas Harth, Sabrina Kirrane, Axel-Cyrille Ngonga Ngomo, Heiko Paulheim, Anisa Rula, Anna Lisa Gentile, Peter Haase, and Michael Cochez (Eds.). Springer, 583–596. https://doi.org/10.1007/978-3-030-49461-2_34

[47] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. DeepTest: automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 303–314. https://doi.org/10.1145/3180155.3180220

[48] Susana M. Vieira, Uzay Kaymak, and João M. C. Sousa. 2010. Cohen's kappa coefficient as a performance measure for feature selection. In *FUZZ-IEEE 2010, IEEE International Conference on Fuzzy Systems, Barcelona, Spain, 18-23 July, 2010, Proceedings*. IEEE, 1–8. https://doi.org/10.1109/FUZZY.2010.5584447

[49] Denny Vrandecic and Markus Krötzsch. 2014. Wikidata: a free collaborative knowledgebase. *Commun. ACM* 57, 10 (2014), 78–85. https://doi.org/10.1145/2629489

[50] Jie M Zhang, Mark Harman, Lei Ma, and Yang Liu. 2020. Machine learning testing: Survey, landscapes and horizons. *IEEE Transactions on Software Engineering* 48, 1 (2020), 1–36.