Towards Synthesis of Code for Calculations Using Their Specifications

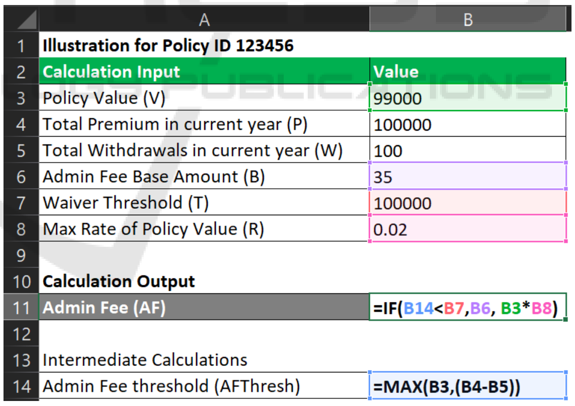
Advaita Datar, Amey Zare, Venkatesh R and Asia A
TCS Research, Tata Research Development and Design Centre (TRDDC), Pune, India

Keywords: Program Synthesis, Programming by Examples, Formal Specifications.

Abstract: Banking, Financial Services, and Insurance (BFSI) software are calculation intensive. In general, these calculations are formally specified in spreadsheets, known as Calculation Specification (CS) sheets. CS sheets describe the calculation inputs and the business logic applied on these inputs to compute calculation output(s). Additionally, an illustration of the calculation is provided with at least one valid value for each calculation input. However, manual implementation of code corresponding to such CS sheets remains to be effort-intensive and tedious. This includes writing database queries to retrieve values for calculation inputs from the enterprise database and converting these queries and corresponding business logic to code. We propose a novel idea to synthesize code corresponding to CS sheets that will i) automatically identify the calculation inputs ii) formulate a *Programming By Example (PBE) specification* for each calculation input where, *PBE input* is the textual description of the calculation input, *PBE output* is the valid value provided in the calculation's illustration, iii) then for each *PBE specification* a) synthesize a set of possible database queries, b) manually review them and mark the intended query, and finally iv) generate code, in desired target language, for all intended queries and the business logic specified in CS sheets.

1 INTRODUCTION

Program synthesis is the task of automatically finding a program, in the underlying programming language, that satisfies the user intent expressed in the form of specifications (Gulwani et al., 2017). Various formats for expressing user intent include, but are not limited to, formal logical specifications, informal natural language descriptions and illustrative input-output examples. Writing formal specifications is a tedious and time-consuming task, and in some cases, it may be as complex as writing the program itself. On the other hand, informal specifications are often criticized for being ambiguous and verbose, even for simple programs. As an alternative, illustrative input-output examples, in the form of *Programming By Example (PBE)* specifications (Gulwani, 2016), are used to express user intent. While PBE specifications work well for string transformation examples (Gulwani, 2011), they aren't suitable for real-world applications. This is because it is extremely difficult to specify all the requirements of such large and complex applications using a limited number of input-output examples. Moreover, sometimes only a subset of the requirements of such applications need to be implemented as the code for business rules for such



	A	B
1	**Illustration for Policy ID 123456**	
2	**Calculation Input**	**Value**
3	Policy Value (V)	99000
4	Total Premium in current year (P)	100000
5	Total Withdrawals in current year (W)	100
6	Admin Fee Base Amount (B)	35
7	Waiver Threshold (T)	100000
8	Max Rate of Policy Value (R)	0.02
9		
10	**Calculation Output**	
11	**Admin Fee (AF)**	**=IF(B14<B7,B6, B3*B8)**
12		
13	Intermediate Calculations	
14	Admin Fee threshold (AFThresh)	**=MAX(B3,(B4-B5))**

Figure 1: A sample Calculation Specifications (CS) sheet.

applications is already in place. Hence, accurately expressing and interpreting user intent in the form of specifications continues to be a major challenge for program synthesis approaches.

Large enterprise applications, especially in the Banking Financial Services and Insurance (BFSI) industry, rely heavily on calculations for day-to-day operations. BFSI software, the largest market for Information Technology (IT) services (Arc, 2022), contains numerous types and instances of such calculations. These calculations represent the business

Datar, A., Zare, A., R, V. and A, A.
Towards Synthesis of Code for Calculations Using Their Specifications.
DOI: 10.5220/0011940900003464
In *Proceedings of the 18th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2023)*, pages 497-504
ISBN: 978-989-758-647-7; ISSN: 2184-4895

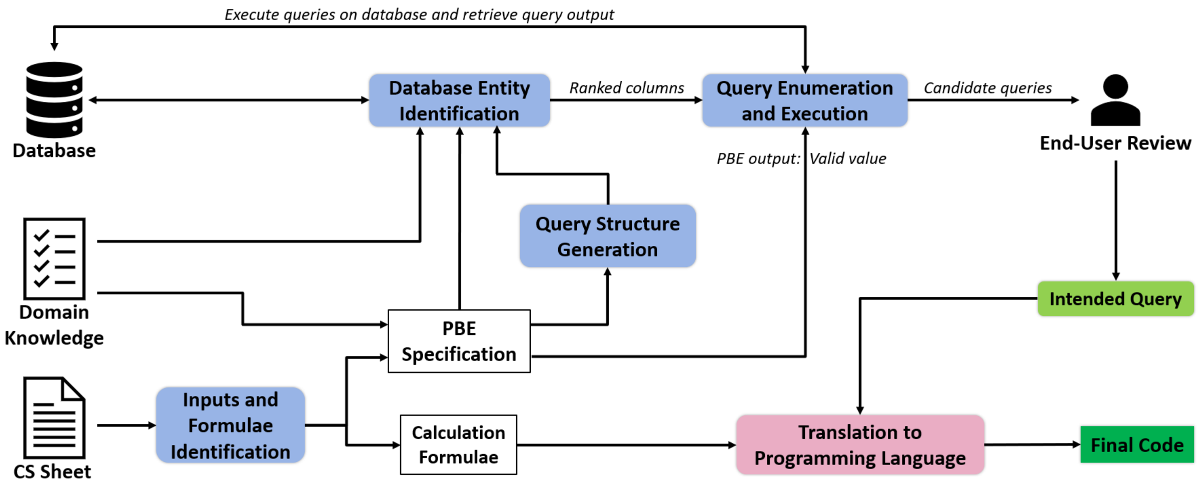

Execute queries on database and retrieve query output
Database
Database Entity Identification
Ranked columns
Query Enumeration and Execution
Candidate queries
End-User Review
PBE output: Valid value
Query Structure Generation
Intended Query
Domain Knowledge
PBE Specification
CS Sheet
Inputs and Formulae Identification
Calculation Formulae
Translation to Programming Language
Final Code

Figure 2: Overview of the proposed approach for code generation from Calculation Specifications (CS) sheet.

rules that are adhered to by BFSI organizations, and they are developed by domain experts following rigorous review. Such calculations are typically specified in spreadsheets, known as Calculation Specification (CS) sheets. Figure 1 shows a CS sheet from insurance domain that specifies the calculation for computing admin fee for a policy holder who may have exceeded the limit for premium payments in a year.

As evident from the sample CS sheet, calculations are formally specified in terms of calculation inputs (cells A3-A8) and the business rules for computing the value of calculation output *Admin Fee* (cell A11) using the values of calculation inputs (cells B3-B8). The business rules (cells B11, B14) are implemented by domain experts in the form of mathematical and logical formulae. For ease of understanding, an illustration of the calculation is provided that contains at least one valid value for each calculation input. For example, the given CS sheet (Figure 1) illustrates how the admin fee is calculated for policy ID 123456 by considering the valid values (cell B3-B8) for each calculation input. Here, valid values are the output of queries corresponding to the calculation input text (A3-A11) for the policy ID 123456. As soon as values for calculation inputs (cells B3-B8) are entered in the CS sheet, all formulae (cell B11, B14) are triggered and the respective calculation output is computed (cell B11).

In real-world scenarios from the BFSI domain, manually implementing code corresponding to given calculation specifications, such as CS sheets, is an effort-intensive and tedious task, because:

- To trigger/execute such calculations in code, values of calculation inputs (cells B3-B8 in Figure 1) need to be retrieved from the enterprise database. This requires inferring database queries that incorporate the table relationships (primary/foreign keys for joining tables) in the database schema. Real-world enterprise databases contain hundreds of tables that are part of complicated database schema and hence creating the *intended query* manually is extremely difficult.
- These intended queries need to be written in a parameterized way to ensure data for multiple policies/users can be retrieved on demand.
- The business rules (cell B11, B14 in Figure 1), referred henceforth as 'business formulae', in the BFSI domain are complex and may contain long sequence of mathematical and logical expressions that refer to multiple calculation inputs. Hence, in real-world scenarios, it is very difficult to manually translate such business formulae into the desired programming language.

Incorporating all these factors while writing the code manually is a challenge. Hence, there is a clear need for automating code generation for such calculations as per their formal specifications i.e., CS sheets.

In this paper, we propose an idea to synthesize the code corresponding to the calculation specifications in CS sheets. Figure 2 shows the overview of the approach implementing our proposed idea. We believe CS sheets are the perfect combination of text and business formulae, and hence CS sheets offer a novel and simple means of specifying user intent for synthesizing code for calculations. Unlike ambiguous natural language specifications, the contents of CS sheets are precise and machine-interpretable. Additionally, the illustration of the underlying calculations, provided along-with CS sheets, contains a valid value for each calculation input. These valid values play a key role in our proposed approach that is described as follows.

As a first step of our approach, we plan to parse the input CS sheet to automatically identify all cells containing calculation inputs and business formulae (more details in Section 2.1). Next, we plan to formulate a custom *PBE specification* for each calculation input. The *PBE input* will contain all available textual information corresponding to the calculation input and the *PBE output* will be its valid value mentioned in the illustration. We refer to PBE specification as *PBE spec* henceforth.

As a next step, the *PBE spec* will be used to synthesize database queries corresponding to each calculation input. Firstly, we will guess the query structure and database entities needed in the various clauses (SELECT, WHERE, etc.) of the *intended query*. This will be done by applying Natural Language Processing (NLP) techniques on the formulated *PBE input*. Next, the guessed query structure and entities will be traversed, as per the grammar and algorithm shown in Section 2.3.3, to *enumerate* all possible queries and these queries will then be executed on the underlying database. The queries that return same value as *PBE output* will then be presented for the end-user review for marking the *intended query*. Once the intended queries for all calculation inputs are synthesized, these queries, along-with the specified business formulae, will be translated to the programming language desired by the end-user. Initially, we plan to generate the final code in Python using available open-source code translation tools (Richter, 2022; C.W., 2022). However, our approach can be easily extended to generate code in other programming languages through corresponding state-of-the-art code translation techniques.

The key contributions of our paper are:

- A user-friendly format for specifying calculations i.e. CS sheets.
- An end-to-end approach for synthesis of code corresponding to CS sheets.
- A novel PBE-based query synthesis technique.

2 PROPOSED APPROACH

Figure 2 shows the overview of our proposed approach. The key steps of the same are explained as follows, using the example CS sheet in Figure 1.

2.1 Identification of Calculation Inputs and Business Formulae

As a first step, we identify all calculation inputs and business formulae by parsing the input CS sheet. For this, the contents of all non-empty cells will be read to identify *referred cells*. These are the cells that are used or referred in at least one other cell in the CS sheet. For example, in Figure 1 cell B14 has a formula, "=MAX(B3,(B4-B5))", and hence B3, B4 and B5 will be identified as *referred cells* for B14. The relationship between a cell and it's *referred cell(s)* will be maintained in the form of a directed graph, termed as Cell References Graph (CRG). Figure 3 shows the CRG for the example CS sheet (Figure 1). In CRG, each cell with references to other cells (e.g. B14 in Figure 1) will be tagged as 'destination node' and an edge will be drawn from each of it's *referred cells* (B3, B4, B5 in Figure 1) by tagging them as 'source nodes'. Once all non-empty cells are parsed, the following three artifacts will be identified though CRG:

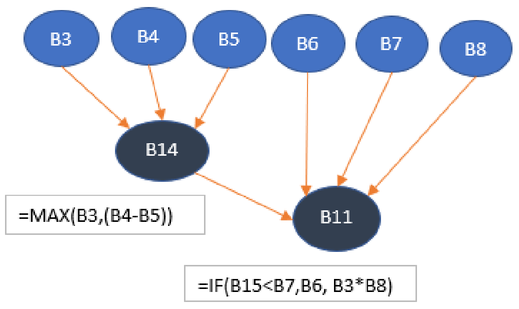


Figure 3: Cell References Graph (CRG) for the example CS sheet shown in Figure 1.

- Cells that are never tagged as a 'destination node' but are tagged as a 'source node' at least once will be marked as *input cells*. (Example: cells B3-B8 in Figure 1)
- Each cell immediately to the left of each *input cell* will be tagged as calculation input. (Example: cells A3-A8 in Figure 1)
- All non-empty cells, apart from *input cells*, that are part of CRG will be marked as *business formulae cells*. (Example: cells B11-B14 in Figure 1)

2.2 Formulation of Programming by Example (PBE) Specification

As a next step, a *PBE spec* will be formulated for each identified calculation input. We formally define *PBE spec* as:

Input_Text ***[Supporting_Information]*** {***Domain_Keywords***} = ***Value***, where

- *Input_Text* contains the name of the calculation input. In the example CS sheet (Figure 1), contents of calculation input cells A3 to A8 i.e. *Policy value*, *Total premium in current year* etc. will be considered as *Input_Text* for respective *PBE spec*.
- *[Supporting_Information]* captures additional information, apart from the *Input_Text*, that can be

used in synthesizing queries for calculation inputs. In the BFSI domain, this typically includes information that is unique to each illustration, such as policy ID, customer ID, etc. In the example CS sheet (Figure 1), the illustrated values for all calculation inputs are taken from the policy 123456. Hence, *Policy ID = 123456* will be considered as *Supporting_Information* in the *PBE spec* of each calculation input.

- {*Domain_Keywords*} is an optional field of *PBE spec* that contains all domain-specific information that denotes the context and/or the calculation type in which the calculation input is being used. These keywords are included in order to assist query synthesis by narrowing down the search for relevant database entities that are needed in query clauses. The example CS sheet (Figure 1) belongs to insurance domain and the calculation name, i.e. *Admin Fee*, will be considered as *Domain_Keywords* in the *PBE spec* of each calculation input.
- *Value* contains the valid value(s) of the calculation input that is(are) provided in the illustration. It is determined by extracting the contents inside *input cells* that are identified in Section 2.1. In the example CS sheet (Figure 1), the *input cells* are B3 to B8 and hence, their contents i.e. *99000*, *90000* etc. will be considered as *Value* for *PBE spec* formulation. Please note that we use valid value i.e. *Value* and *PBE output* interchangeably throughout the paper.

Using the formal definition described above, the *PBE spec* for each calculation input in the example CS sheet (Figure 1) will be as follows.

1. Policy value [Policy ID = 123456] {Admin Fee} = 99000
2. Total premium in current year [Policy ID = 123456] {Admin Fee} = 90000

   ....

Formulation of such a *PBE spec* from calculation specifications (CS sheet) and using it to synthesize queries is one of the novel contributions of our approach. To the best of our knowledge, none of the existing techniques create such a specification for synthesis and/or make use of valid values i.e. *PBE output* to synthesize the query.

2.3 Query Synthesis Based on PBE Spec

Once the *PBE spec* is formulated, a query synthesis apparatus will be used to synthesize the desired database query. The approach for the same is described as follows.

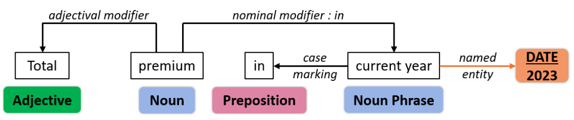


Figure 4: Phrases and relationships for 'Total premium in current year' from the CS sheet in Figure 1.

2.3.1 Generation of Query Structure

The structure of the desired query can be guessed by parsing the *Input_Text* using standard natural language parser. Figure 4 shows the parsed phrases and their relationships, identified through CoreNLP parser(Manning et al., 2014), for *Total premium in current year*. The query structure for this calculation input will be constructed by a rule-based apparatus on the basis of identified phrases and their relationships. This will be accomplished by mapping *related* phrases to respective query clauses using natural language semantics, as explained below.

In *Total premium in current year*, the subject noun is *premium* and hence it is considered in SELECT clause. As per natural language semantics, the subject noun's adjectival modifier is *Total* which indicates addition or cumulative sum, and hence the SELECT clause of the desired query should be updated to include aggregate function i.e. *SUM(premium)*. Next, the main noun's nominal modifier *current year* (linked via preposition *in*) will be considered in WHERE clause as it indicates the scope or condition of the desired query. As per natural language semantics, the phrase *current year* is a named-entity and intuitively indicates the ongoing year i.e. 2023, and hence the WHERE clause should contain *year = 2023*. However, date-related information is generally mentioned in columns with date data type, and hence the phrase *current year* can also relate to a date on or between 1-Jan-2023 and 31-Dec-2023. Therefore, the WHERE clause will also contain *1-Jan-2023 <= date <= 31-Dec-2023*, leading to two query structures. Lastly, the *Supporting_Information* in *PBE spec* i.e., *Policy ID = 123456* becomes part of the WHERE clause by default as the given illustration contains values from policy ID 123456. Hence, based on the aforementioned information for the calculation input *Total premium in current year*, the identified query structures are shown in Table 1.

2.3.2 Identification of Database Entities Needed in the Query

Once the query structure is ready, the database entities i.e., table and column names corresponding to each query clause need to be identified. As shown in Table 1, every query clause will contain either text

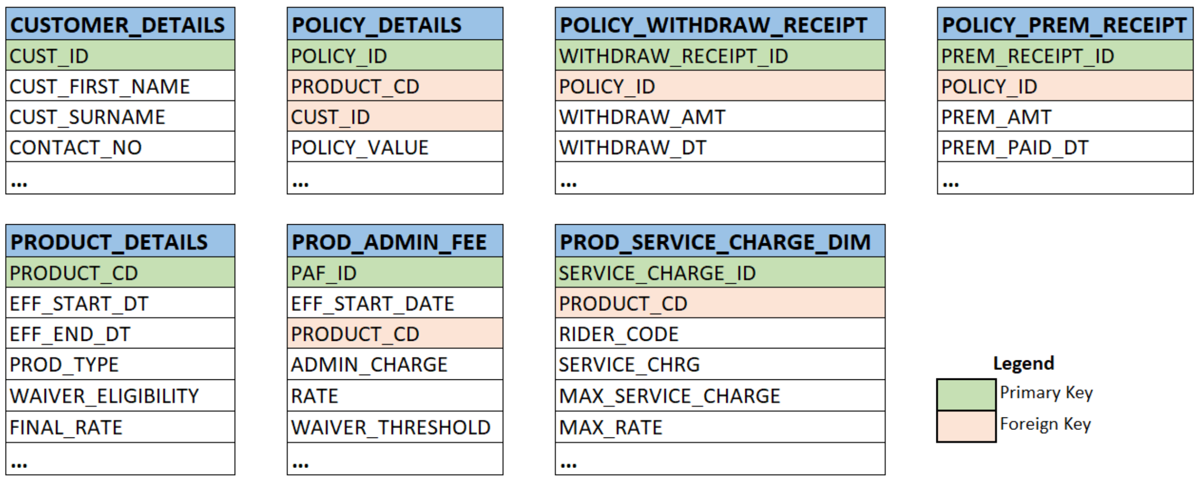


Figure 5: Database details corresponding to the example CS sheet in Figure 1.

Table 1: Query structures for 'Total premium in current year' from the CS sheet in Figure 1.

Stru-cture ID	Clause	LHS		Oper-ator	RHS	
		Aggr-egate	Variable		Aggr-egate	Variable
1	SELECT	SUM	premium			
	WHERE_1		Policy ID	=		123456
	WHERE_2		Date	>=		1-Jan-2023
	WHERE_3		Date	<=		31-Dec-2023
2	SELECT	SUM	premium			
	WHERE_1		Policy ID	=		123456
	WHERE_2		year	=		2023

or a text-operator-value triplet. To infer the intended column(s) for the query clause text, we will compare it with all table and column names, and their description provided in the database schema. The identified columns will then be assigned a *text matching score* by using standard text matching utilities (Foundation, 2022) to quantify how similar these columns are to the query clause text. In case a query clause also contains a value and operator, a *clause predicate* will be created.

For example, the *clause predicate* for clause WHERE_2 belonging to query structure ID 2 in Table 1 will be 'year = 2023'. Columns that contain at least one value mentioned in such a clause predicate will be considered as *candidate columns* for the respective query clause. All such *candidate columns* will then be given a representative *value matching score*, and later they will be sorted based on their combined *value and text matching scores* to get a ranked list of candidate columns. Table 2 shows the ranked list of candidate columns, as per database schema in Figure 5, for the *PBE spec* shown in Table 1. We have observed that the accuracy of identifying database columns improves if *value matching* is done along with conventional text matching.

Apart from the *Input_Text*, the *Domain_Keywords*

Table 2: Ranked columns for the query structures of 'Total premium in current year' from the CS sheet in Figure 1.

Clause	Rank	Table	Column	Value Score	Text Score
SUM (premium)	1	POLICY_PREM_RECEIPT	PREM_AMT	0	20
	2		PREM_PAID_DT	0	15
	...	...	...	...	...
Policy ID = 123456	1	POLICY_DETAILS	POLICY_ID	25	25
	2	POLICY_PREM_RECEIPT	POLICY_ID	25	20
	...	...	...	...	...

included in *PBE spec* also play a vital role in the identification of database entities for query clauses. We explain this through the example of the calculation input 'Max Rate of Policy Value' (cell A8 in Figure 1) for the database schema in Figure 5. The SELECT clause for this calculation input, as per the described query structure generation approach (refer Step 2.3.1), will be 'MAX(Rate)'. As a result, the columns 'FINAL_RATE' from PRODUCT_DETAILS, 'RATE' from PROD_ADMIN_FEE, and 'MAX_RATE' from PROD_SERVICE_CHARGE_DIM will get significantly higher *text matching score* due to the sub-string 'rate' in their name. Assuming that all three columns contain the value '0.02' or '2%', the column 'RATE' from PROD_ADMIN_FEE will be ranked highest as it's table name is very similar to the provided *Domain_Keywords* 'Admin Fee'.

2.3.3 Query Enumeration and Execution

To generate possible queries, all identified information i.e., ranked columns, query clauses and query structures, will be traversed as per the algorithm described in Algorithm 1. Firstly, *high-scoring tables* will be identified by considering the sum of text and

Algorithm 1: Generation of Candidate Queries.

Inputs: Database (*DB*), list of query structures (*Structs*), ordered list of ranked columns (*RC*), expected query output (*Value*).
Output: Ordered list of candidate queries (*CandQueries*).

```
procedure GENCANDQUERIES($RC$,$Value$)
    $CandQueries \leftarrow [..]$
    $RT \leftarrow getRankedTables(RC)$
    for $s \in Structs$ do
        for $i \leftarrow 1..MaxJoins$ do
            // MaxJoins will be configurable
            $SelTabs \leftarrow P(RT[0..i-1])$
            // P(A) indicates power set of set A
            for $\{t\} \in SelTabs$ do
                $q \leftarrow queryInSyntax(s,\{t\},RC)$
                $result \leftarrow executeQuery(q,DB)$
                if $result == Value$ then
                    Add q to CandQueries
                end if
            end for
        end for
    end for
    return $CandQueries$
end procedure

procedure GETRANKEDTABLES($RankedCols$)
    $TabMap \leftarrow \{..\}$
    for $c \in RankedCols$ do
        $tab \leftarrow table\ of\ c$
        $score \leftarrow text + value\ matching\ score\ of\ c$
        $TabMap[tab] \leftarrow TabMap[tab] + score$
    end for
    $sorted \leftarrow sort(TabMap, TabMap.values())$
    return $sorted.keys()$
end procedure
```

value matching scores (computed in Step 2.3.2) of ranked columns from each table. Next, queries will be *enumerated* as per the grammar shown in Figure 6. As evident from this grammar, we plan to focus on the key constructs without loss of generality, and hence we restrict our query synthesis approach to consider only a subset of query language constructs. To form a query through enumeration, ranked columns will be picked, as per the order of aforementioned high-scoring tables, and inserted in identified query clauses as per the desired query language syntax. For explanation purpose we have used SQL as the desired query language (Figure 6), but our idea can easily be extended to work for other query languages by simply using corresponding grammar for query enumeration.

Once an enumerated query is ready, it will be executed on the input database and the value returned

```
Q → SELECT SelExpr FROM TabExpr JoinExpr WhereExpr
SelExpr → ColExpr | SelExpr, SelExpr
ColExpr → column | Agg(column)
Agg → SUM | AVG | COUNT | MIN | MAX | DISTINCT |
      COUNT(DISTINCT) | ∅
TabExpr → table | TabExpr, TabExpr
JoinExpr → JOIN TabExpr ON EqExpr | ∅
EqExpr → column = column | EqExpr AND EqExpr
WhereExpr → WHERE BaseCond GroupExpr OrderExpr | ∅
BaseCond → OperExpr | BaseCond AND BaseCond |
           BaseCond OR BaseCond | NOT BaseCond
OperExpr → ColExpr OP column | ColExpr OP constant |
           ColExpr BETWEEN constant AND constant | ∅
OP → > | < | = | <= | >= | <>
GroupExpr → GROUP BY column | ∅
OrderExpr → ORDER BY ColExpr ASC LimExpr |
            ORDER BY ColExpr DESC LimExpr | ∅
LimExpr → LIMIT constant | ∅
```

Figure 6: Grammar used for enumeration of queries.

by the query will be compared with the *PBE output* identified in *Step 2.2*. If the values match, the query will be marked as a *candidate query* and added to an ordered list of candidate queries *CandQueries* (as shown in Algorithm 1). The order in which the candidate queries are added in *CandQueries* represents their *rank*. Once all enumerated queries are executed on the database, *CandQueries* will be presented to the end-user for review and verdict of *intended query*.

2.3.4 Query Review by the End-User

We aim to provide only a few queries for end-user review, hence we execute all enumerated queries on the database to get a reduced set of candidate queries. In case all enumerated queries are presented to the end-user, then he/she may prefer spending time manually writing the *intended query*, instead of reviewing a large number of queries that may not even result in the expected query output. Hence, only top N candidate queries will be given to the end-user to mark the *intended query*. The value of N will be configurable and will get fine-tuned after the end-user reviews.

There is a possibility that none of the candidate queries is the end-user's intended query, or our approach doesn't generate even a single candidate query. In such scenarios, we plan to ask the end-user to create the intended query manually and use the query for online learning (Hoi et al., 2021) to improve our approach to enable generation of queries for such scenarios in the future.

2.4 Translation to Target Programming Language

Once the database queries are synthesized and finalized for all calculation inputs in the CS sheet, the code corresponding to these queries and the business formulae (identified in Section 2.1) will be generated in the programming language desired by the end-user. In the initial prototype of our approach, we generate the *final code* in Python using well-tested openly available techniques (Richter, 2022; C.W., 2022). These techniques automatically identify the participating formulae in the input CS sheet and convert them into Python. Using these techniques, the final code for the input CS sheet in Figure 1, as per the CRG in Figure 3, will be as shown below:

```
import os, sys
# Function synthesized for Total premium in
current year
def getTotalPremium(policyID):
    cursor = database.cursor()
    cursor.execute("SELECT SUM(PREM_AMT)
    FROM POLICY_PREM_RECEIPT WHERE ...")
    result = cursor.fetchall()
    return result

# Functions for V, W, B, T, R will be
synthesized similar to getTotalPremium

# Function synthesized for Admin Fee
def getAdminFee(policyID):
    V = getPolicyValue(policyID)
    P = getTotalPremium(policyID)
    W = getTotalWithdrawals(policyID)
    AFThresh = MAX(V,(P-W))
    B = getAdminFeeBase(policyID)
    T = getWaiverThreshold(policyID)
    R = getMaxRatePolicyValue(policyID)
    AF = B if AFThresh < T else V * R
    return AF
```

The constructed CRG (Figure 3) plays a vital role in generation of final code. The example CS sheet (Figure 1) contains two business formulae in cells B11, B14, but as evident in the CRG, the formula in cell B14 needs to be executed before the formula in cell B11. This sequential dependency in business formulae is maintained by the directed edge from cell B14 to B11 in CRG, and hence the code for the formula in cell B14 appears before the same for cell B11.

3 RELATED WORK

The key challenge for program synthesis is the diversity of user intent and the inability to express the intent precisely. Natural language descriptions (Li and Jagadish, 2014; Desai et al., 2016) are the most common way of stating user intent, but they are ambiguous and result in incorrect programs. Specifying user intent through input-output examples (Gulwani, 2016) works for a certain class of string manipulation programs, but it hasn't been used effectively in query synthesis paradigm. State-of-the-art query synthesis techniques either require the user to provide partial queries (Bastani et al., 2019) or create large input-output tables (Wang et al., 2017; Takenouchi et al., 2020). Creating such complicated artifacts is as complex and effort-intensive as writing the query manually. To address these drawbacks, we have proposed a novel concise and precise way of specifying user intent in the form of Calculation Specification (CS) sheets. A CS sheet represents the mathematical and logical relationship between the inputs and outputs of a program. Furthermore, the end-users, who are not programming experts, may find providing valid value(s) of variables participating in the calculations to be more approachable and natural as compared to creating complicated input-output tables and partial queries.

Another challenge for program synthesis is enumerating and searching programs in a large and intractable program space. In query synthesis paradigm, existing techniques (Desai et al., 2016; Yaghmazadeh et al., 2017) haven't overcome this challenge of program space explosion. A few techniques (Li and Jagadish, 2014; Wang et al., 2017; Takenouchi et al., 2020) indeed reduce the search space but require the end-user to provide additional hints like the selection of aggregate function, constants, etc., which can not be provided by the end-users who typically don't have technical knowledge of query language and syntax. On the contrary, in our proposed approach we plan to reduce the search space by filtering the enumerated queries based on expected query output and other heuristics described in detail in Section 2.3. Hence, the end-user doesn't need to give technical guidance, like the selection of aggregate functions, constants, conditions, etc., for reducing search space for queries.

4 CONCLUSION AND FUTURE WORK

In this paper we have presented a novel idea to synthesize code corresponding to CS sheets. As described in the paper, business rules can be easily expressed in CS sheets through a combination of text and formulae. This makes CS sheet a precise and machine-interpretable way to specify calculations. We have also proposed a novel query synthesis approach that firstly formulates a custom *PBE spec* for each input in CS sheet, and later uses text and value inference to synthesize database query corresponding to each *PBE spec*. Lastly, the synthesized database queries and the business formulae specified in the CS sheet are translated into the desired programming language.

Going ahead, we plan to improve our approach by incorporating machine learning to ensure that the user-intended query is inferred with a high rank. Our goal is to develop a technique that is self-adaptive and capable of continuously improving the underlying steps, such as identifying query structures, database entities, query enumeration etc., based on feedback from the end-user.

REFERENCES

Arc, I. (2022). It bfsi market overview. https://www.industryarc.com/Research/It-Bfsi-Market-Research-500664. Accessed: 2023-01-26.

Bastani, O., Zhang, X., and Solar-Lezama, A. (2019). Synthesizing queries via interactive sketching. *arXiv preprint arXiv:1912.12659*.

C.W. (2022). pyexcel - lets you focus on data, instead of file formats. https://docs.pyexcel.org/en/latest/. Accessed: 2023-01-26.

Desai, A., Gulwani, S., Hingorani, V., Jain, N., Karkare, A., Marron, M., and Roy, S. (2016). Program synthesis using natural language. In *Proceedings of the 38th International Conference on Software Engineering*, pages 345–356.

Foundation, P. S. (2022). difflib - helpers for computing deltas. https://docs.python.org/3/library/difflib.html. Accessed: 2023-01-26.

Gulwani, S. (2011). Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices*, 46(1):317–330.

Gulwani, S. (2016). Programming by examples. *Dependable Software Systems Engineering*, 45(137):3–15.

Gulwani, S., Polozov, O., Singh, R., et al. (2017). Program synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119.

Hoi, S. C., Sahoo, D., Lu, J., and Zhao, P. (2021). Online learning: A comprehensive survey. *Neurocomputing*, 459:249–289.

Li, F. and Jagadish, H. V. (2014). Nalir: an interactive natural language interface for querying relational databases. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 709–712.

Manning, C. D., Surdeanu, M., Bauer, J., Finkel, J. R., Bethard, S., and McClosky, D. (2014). The stanford corenlp natural language processing toolkit. In *Proceedings of 52nd annual meeting of the association for computational linguistics: system demonstrations*, pages 55–60.

Richter, S. (2022). Xlcalculator: Ms excel formulas to python. https://github.com/bradbase/xlcalculator. Accessed: 2023-01-26.

Takenouchi, K., Ishio, T., Okada, J., and Sakata, Y. (2020). Patsql: efficient synthesis of sql queries from example tables with quick inference of projected columns. *arXiv preprint arXiv:2010.05807*.

Wang, C., Cheung, A., and Bodik, R. (2017). Synthesizing highly expressive sql queries from input-output examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 452–466.

Yaghmazadeh, N., Wang, Y., Dillig, I., and Dillig, T. (2017). Sqlizer: query synthesis from natural language. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–26.