An Approach and a Software Tool for Automatic Source Code Generation driven by Business Rules

Andrii Kopp and Dmytro Orlovskyi

National Technical University "Kharkiv Polytechnic Institute", Kyrpychova str. 2, Kharkiv, 61002, Ukraine

Abstract
This paper proposes an approach to automatic source code generation driven by business rules. This approach is inspired by low-code and automatic programming to improve the software development process and accelerate product delivery through the source code generation from natural language statements. The proposed approach considers business rules as input, uses the triplestore model for knowledge representation based on business rules, utilizes association rules to suggest attribute data types, and produces an abstract data model. This abstract data model is a framework for software components generation of various purposes and syntax, such as SQL scripts for database tables creation and Java Beans for server-side implementation. A software solution based on the proposed approach translates the data model into the source code of software components: MySQL database and Java classes. But it can be extended to generate various software components based on different syntax rules. Performed experiments demonstrate that generated software components are verified and valid since they were checked using static code analysis and dynamic testing. Conclusion formulates research outcomes, obtained results, and limitations. Future work outlines the next research steps in this field.

Keywords
Business Rules, Low-Code, Source Code Generation, Data Model, Software Development

1. Introduction: Related Work and Problem Statement

The time between specifications capturing and the product delivery is critical for the software development process and its stakeholders. The source code generation could significantly increase the software development process by shortening the time between requirements gathering and delivery using automatic programming and low-code solutions. The main idea of automatic programming is to "tell the computer what to do rather than how to do the task" [1]. Hence, automatic programming should be supported by some definitive high-level language that is closer to a natural language than a programming language [1]. Low-code platforms provide information technologies that automate the creation and deployment of business applications that encourage business transformations [2]. We can say that low-code platforms encapsulate automatic programming tools since they usually provide user-friendly visual environments to create ready software solutions for business needs by users with minimal programming skills [2]. The encapsulation mentioned above means that knowledge models that automatic programming toolkits use to generate the source code can be created in a drag-and-drop manner via low-code platforms. However, in this study, we address the "low-code" term as the general concept rather than some software development platform. However, in this study, we address the "low-code" term as a general concept rather than some software development platform and focus mostly on the automatic programming aspect of the low-code approach. This paper aims at automatic source code generation from natural language statements given as SBVR business rules to facilitate the software development process by bridging a gap between business analysis and engineering.

COLINS-2022: 6th International Conference on Computational Linguistics and Intelligent Systems, May 12–13, 2022, Gliwice, Poland
EMAIL: kopp93@gmail.com (A. Kopp); orlovskyi.dm@gmail.com (D. Orlovskyi)
ORCID: 0000-0002-3189-5623 (A. Kopp); 0000-0002-8261-2988 (D. Orlovskyi)

1.1. Related Work

In this study, we propose to use the Semantic of Business Vocabulary and Rules (SBVR) standard for specifications representation comprehensible by humans and computers [3]. SBVR is an Object Management Group (OMG) standard that facilitates the software development process.

Extraction of SBVR-based business rules from natural language business rules for computer processing was recently proposed by authors of papers [3] and [4]. Moreover, authors of [5] consider the RDF-based (Resource Description Framework) knowledge representation models generation from SBVR business rules to provide a formal description of natural language as a graph or set of subject-predicate-object triples.

Authors of [6] state that SBVR guarantees traceability between models and quick applications development. However, [6] only proposes the transformation of SBVR statements to UML (Unified Modeling Language) use case diagrams. Earlier studies consider the generation of SQL queries from SBVR business rules. For example, papers [7] and [8] consider SELECT SQL statements generation from pre-defined business vocabularies and rules. A later study [9] also considers translation from SBVR specifications into SQL queries to provide an interface for the database. Authors of one of the recent papers [10] also consider the translation of natural language descriptions into SQL queries for data retrieval. Earlier we also proposed an approach and software prototype for the translation of natural language business rules into SQL database creation scripts [11].

1.2. Problem Statement

Our previous study is based on the translation of fact business rules (statements that define entities and relationships within data models [12]) structured according to the Wiegers taxonomy [12] into SQL DDL (Data Definition Language) scripts. Now we attempt to use the SBVR OMG standard to provide a unified solution that can be integrated with other software development utilities. Also, we want to improve the previously proposed approach by using the Model-Driven Development (MDD) paradigm when business rules are given as subject domain descriptions to build an abstract model with multiple possible implementations [13]. In MDD, abstract models are major artifacts that serve as sources to generate the source code or other software engineering artifacts (see Fig. 1 below).

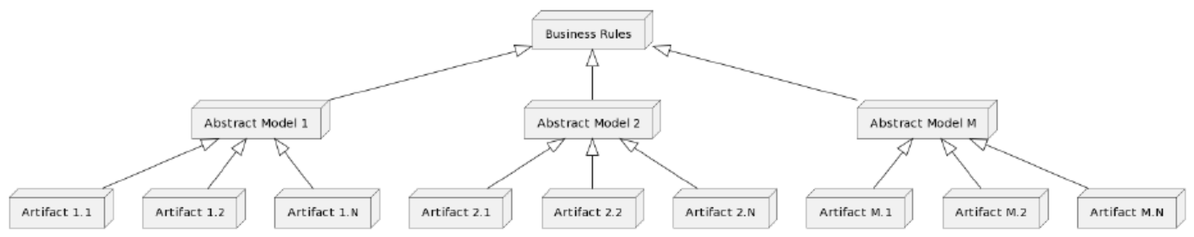


Figure 1: Abstract models serving as sources to generate the source code or other artifacts

The abstract model implementation includes automatic programming activities such as source code and other artifacts generation (forward engineering).

Therefore, in this study we need to solve the following tasks.

1. Propose a procedure for translation of business rules given as SBVR statements into the RDF-based triplestore model that operates subject-predicate-object triples.
2. Propose a procedure for translation of the triplestore model that describes a subject domain on the conceptual level into the data model that describes an abstract software artifact.
3. Propose a procedure to suggest attribute data types that will be added to the abstract model, considering their translation into language-specific data types, e.g. primitive data types in Java or MySQL data types.
4. Develop a software solution based on previously mentioned procedures (1–3) to implement the proposed approach.
5. Perform experiments with a sample set of business rules by translating them into the software components. Verify and validate generated source code.

2. An Approach to Automatic Source Code Generation driven by Business Rules

2.1. Translation of Business Rules given as Structured English Statements into the Triplestore Model

According to [14], the SBVR Structured English (SSE) is a syntax that serves for textual mapping of SBVR concepts and rules. In general, SSE defines four formatting styles, including terms, names, verbs, and keywords [14]. The verbs-based style seems most suitable for the representation of fact business rules. This SSE style defines relationships between concepts. For example, the business rule “order includes items” contains two nouns and a verb. These two nouns describe concepts “order” and “item”, and the “includes” verb describes the relationship between these two concepts.

The Resource Description Framework (RDF) is a modern knowledge representation framework. It solves the problem of knowledge representation since irregularities and exceptions make it difficult to process natural languages. It statements contain three elements (triples): subject, predicate, and object. Such knowledge representation is easily processed by machines and readable for humans [15].

Considering the SSE syntax, we propose the following algorithm for translating fact business rules given as the SSE statements into the triplestore model as it is done in [5] (see Fig. 1 below).

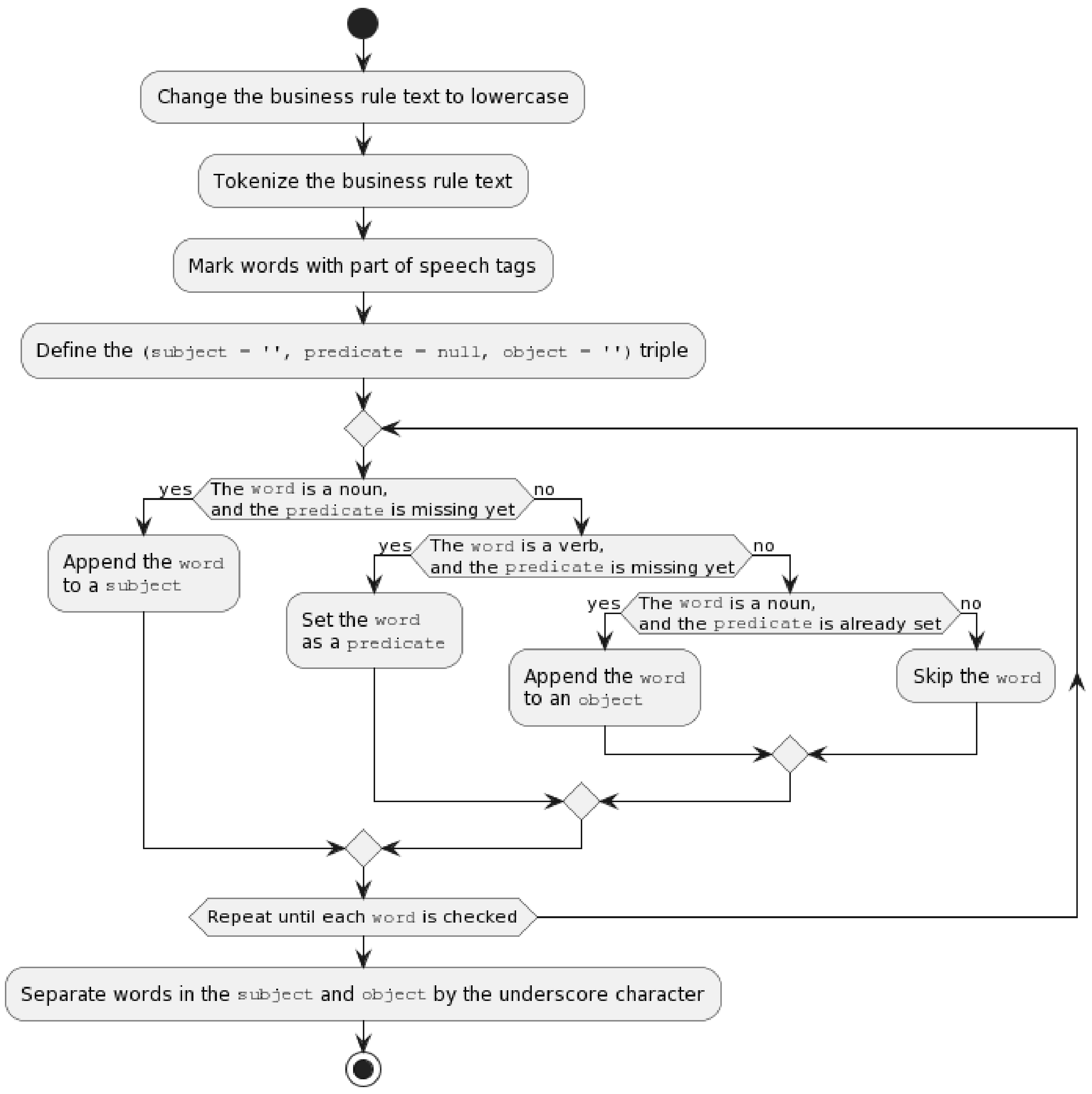

Change the business rule text to lowercase
Tokenize the business rule text
Mark words with part of speech tags
Define the (subject = '', predicate = null, object = '') triple
yes
The word is a noun, and the predicate is missing yet
no
Append the word to a subject
yes
The word is a verb, and the predicate is missing yet
no
Set the word as a predicate
yes
The word is a noun, and the predicate is already set
no
Append the word to an object
Skip the word
Repeat until each word is checked
Separate words in the subject and object by the underscore character

Figure 2: The algorithm for translating fact business rules into the triplestore model

Therefore, having the n business rules, we can obtain a set T of n triples using the algorithm for translating fact business rules into the triplestore model (Fig. 1):

$$T = \{t_i = \langle s_i, p_i, o_i \rangle \mid i = \overline{1,n}\}, \tag{1}$$

where:

- t_i is the i-th triple, $i = \overline{1,n}$;
- s_i is the subject within the i-th triple t_i, $i = \overline{1,n}$;
- p_i is the predicate within the i-th triple t_i, $i = \overline{1,n}$;
- o_i is the object within the i-th triple t_i, $i = \overline{1,n}$;
- n is the number of business rules and corresponding triples.

Using the triplestore (1), now we can store triples, obtained using the input business rules, and retrieve triples to build object-oriented, entity-relationship, and other data models. Furthermore, we can use these data models to generate information system components, such as application classes and structures, database tables, etc.

2.2. Translation of the Triplestore Model into the Data Model

Now we can use the triplestore formulated in the previous sub-section to build a data model. First of all, it is necessary to identify entities and their attributes based on fact business rules given as SSE statements and then translated into triples.

At the data modeling step, we need to define which attributes can have null or missing values and which attributes cannot. In the future, these attributes will be implemented as class properties or database table columns. Therefore, certain business logic, even such primitive, should be already introduced.

Moreover, entities should be interrelated to achieve data consistency. Hence, we should treat business rules not only as sources of entities and the respective attributes but also as sources of relationships among entities.

Thus, to restrict the syntax of fact business rules, we propose to consider only several verbs according to their purpose:

6. "has" and "owns" verbs should be used to detect entities and their attributes (see Fig. 2), e.g. "product has description" or "product owns title", where:

- "has" means unnecessary attributes that can hold null or missing values;
- "owns" means mandatory attributes that cannot hold null or missing values;

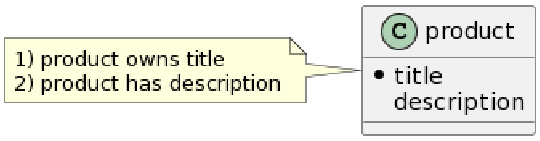


Figure 3: Sample business rules and the expected entity with attributes

7. "includes" verb should be used to detect relationships between entities (see Fig. 3), i.e. how instances of one entity reference instances of another entity, e.g. "product includes category".

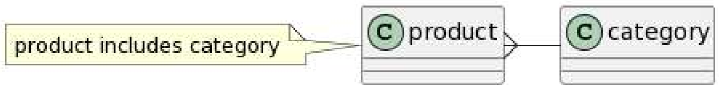


Figure 4: Sample business rules and the expected related entities

According to Fig. 3, the "includes" verb allows creating such relationships between the entities that are called one-to-many in the relational database theory. Based on the SSE syntax considered earlier and the given restrictions of the verbs used in fact business rules, we propose the following algorithm for data model design based on the triplestore model (see Fig. 4 below).

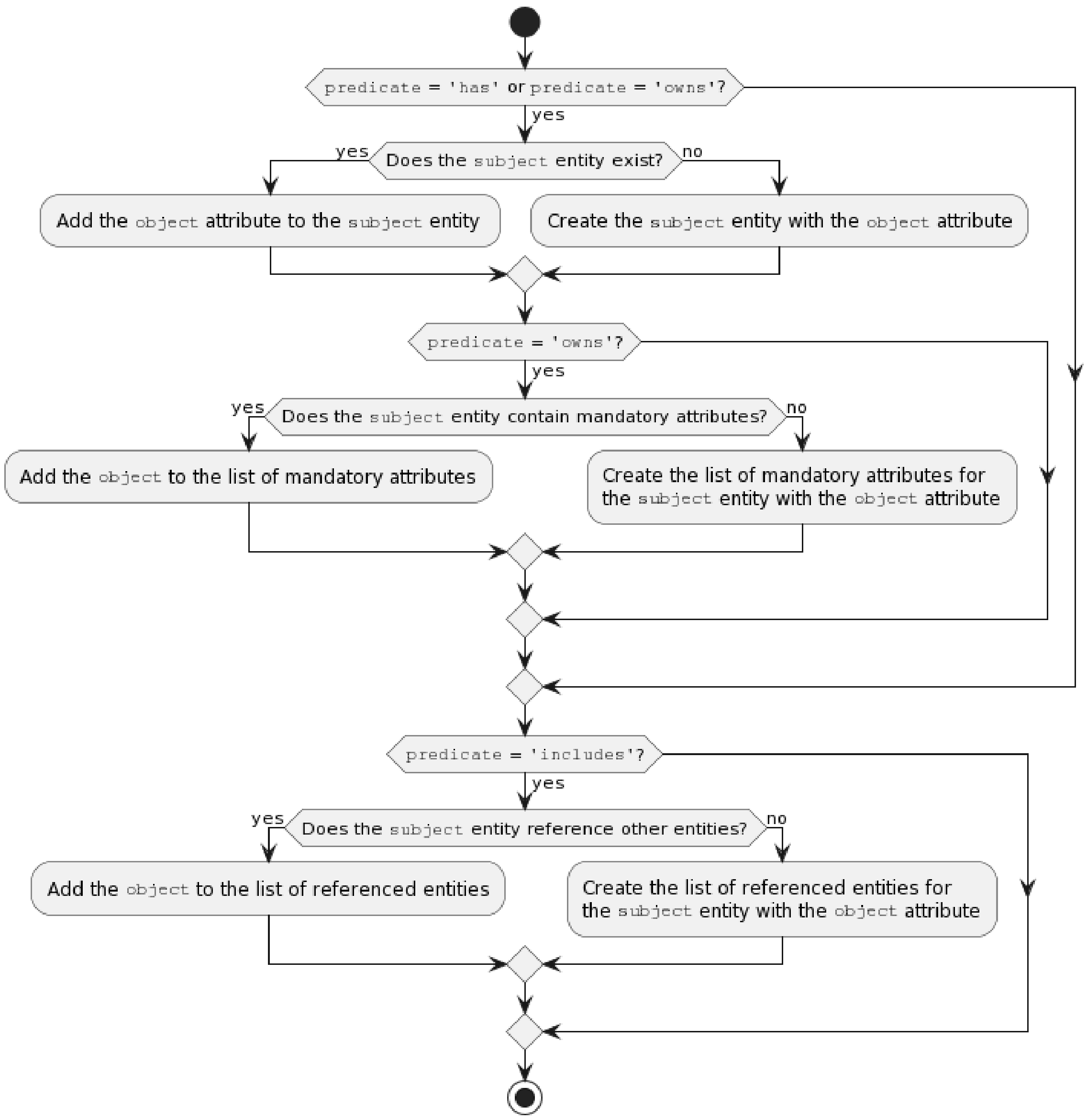

predicate = 'has' or predicate = 'owns'?
yes
Does the subject entity exist?
yes
no
Add the object attribute to the subject entity
Create the subject entity with the object attribute
predicate = 'owns'?
yes
Does the subject entity contain mandatory attributes?
yes
no
Add the object to the list of mandatory attributes
Create the list of mandatory attributes for the subject entity with the object attribute
predicate = 'includes'?
yes
Does the subject entity reference other entities?
yes
no
Add the object to the list of referenced entities
Create the list of referenced entities for the subject entity with the object attribute

Figure 5: The algorithm for data model design based on the triplestore model

Therefore, having the set T we can obtain data model elements DM, such as entities E, attributes A, and relationships R using the algorithm (Fig. 4) for data model design based on the triplestore model:

$$DM = \langle E, A, R, \theta, \varphi, \mu \rangle, \quad (2)$$

where:

- E is the set of entities, $E = \{e_j | j = \overline{1, m}\}$;
- A is the set of attributes, $A = \{a_k | k = \overline{1, p}\}$;
- R is the set of relationships, $R = \{r_l | l = \overline{1, q}\}$;
- θ is the mapping between attributes and entities these attributes belong to: $\theta: a_k \to e_j$, $k = \overline{1, p}$, $j = \overline{1, m}$;
- φ is the mapping between relationships and entities they link: $\varphi: r_l \to \langle e^c, e^p \rangle$, $l = \overline{1, q}$;
- $e^c \in E$ is the so-called "child entity" that depends on the so-called "parent entity" $e^p \in E$;
- μ is the mapping between attributes and their status to define whether they are mandatory, $\mu(a_k) = 1$, or not, $\mu(a_k) = 0$: $\mu: a_k \to \{0,1\}$, $k = \overline{1, p}$.

Now having the data model elements (2) we can build different software implementations of these models. For example, we can use DM (2) to build SQL scripts for relational database creation. Or we can use DM (2) to create Java Beans for enterprise information system development. We can even use DM (2) to produce smart contract code for blockchain-driven decentralized applications development.

2.3. Suggestion of Attribute Data Types based on Association Rules

In this sub-section, we propose to use the approach based on association rules [16] to suggest attribute data types. According to this approach, the sets of items are called antecedent or left-hand-side LHS and consequent or right-hand-side or RHS of the rule. Hence, the rule is formed as the implication $LHS \Rightarrow RHS$, where $LHS \cap RHS = \emptyset$.

According to our idea, as the left-hand side, we can use column names of existing databases and as the right-hand side, we can use data types of the respective columns. To implement the proof-of-concept and conduct experiments, we propose to use the "Spider" dataset [17] maintained by Yale students. This dataset includes 200 databases with multiple tables covering 138 different domains.

Each database in the "Spider" dataset is given as the set of SQL scripts and as the SQLite file. Hence, such databases could be easily processed table by table to build association rules $LHS \Rightarrow RHS$, where LHS is the column (attribute) name and RHS is the data type respectively.

According to the approach based on association rules, we propose the following algorithm to obtain attribute data types suggestions based on the association rules (see Fig. 5 below).

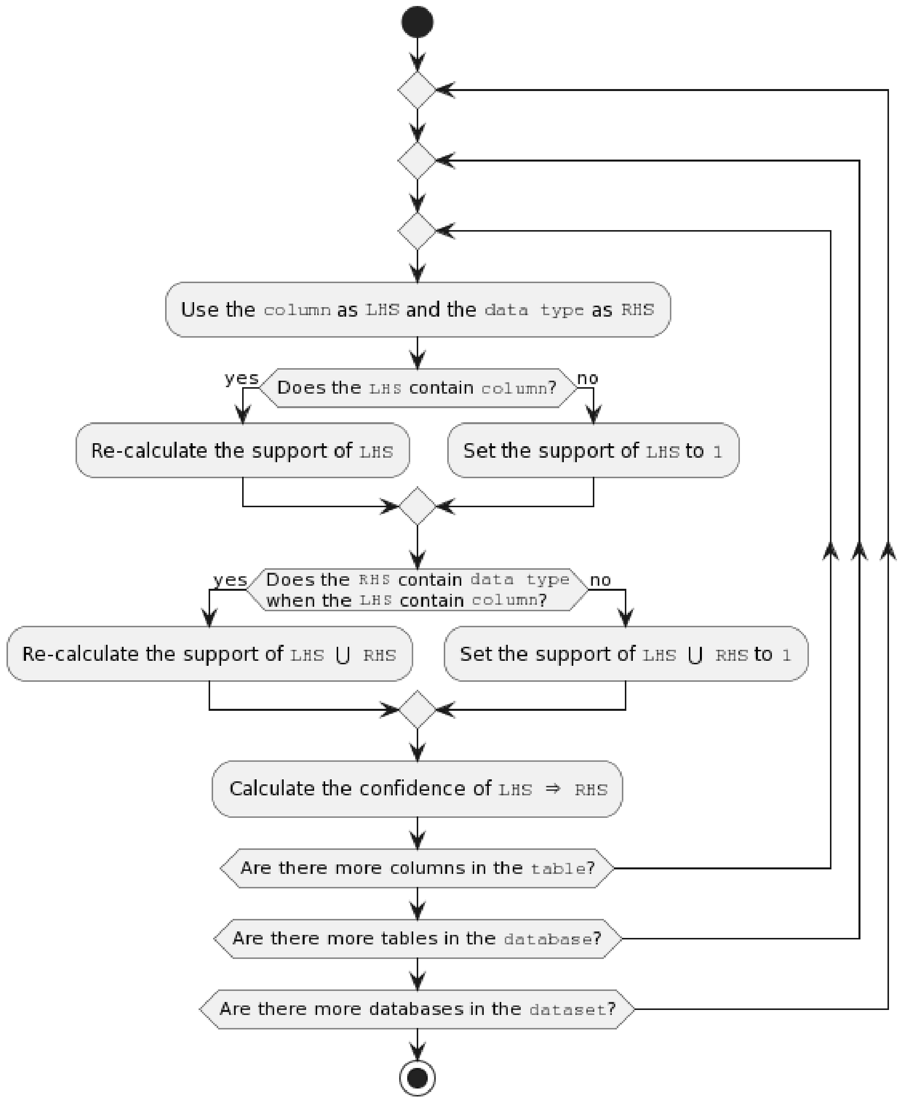
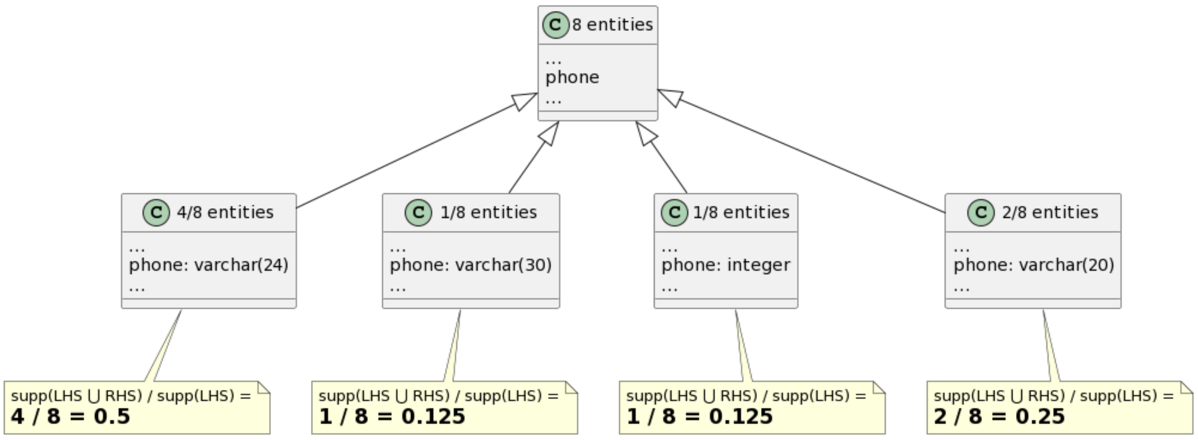

Use the column as LHS and the data type as RHS
yes
Does the LHS contain column?
no
Re-calculate the support of LHS
Set the support of LHS to 1
yes
Does the RHS contain data type when the LHS contain column?
no
Re-calculate the support of LHS ⋃ RHS
Set the support of LHS ⋃ RHS to 1
Calculate the confidence of LHS ⇒ RHS
Are there more columns in the table?
Are there more tables in the database?
Are there more databases in the dataset?

Figure 6: The algorithm to obtain attribute data types suggestions based on the association rules

Let us make the proposed idea clearer by introducing the following example. We assume that dealing with the dataset of multiple databases each of which contains an entity (table) with the "phone" attribute (column). Let us consider 8 cases of tables with the "phone" column (see Fig. 6 below). Data types used for the "phone" column vary from one to another database (see Fig. 6 below). Nevertheless, after processing the dataset, we discover that the "phone" column has the following data types (see Fig. 6 below):

- "varchar(24)" in 4 out of 8 cases;
- "varchar(30)" in 1 out of 8 cases;
- "integer" in 1 out of 8 cases;
- "varchar(20)" in 2 out of 8 cases.



8 entities
...
phone
...
4/8 entities
phone: varchar(24)
1/8 entities
phone: varchar(30)
1/8 entities
phone: integer
2/8 entities
phone: varchar(20)
supp(LHS ⋃ RHS) / supp(LHS) = 4 / 8 = 0.5
supp(LHS ⋃ RHS) / supp(LHS) = 1 / 8 = 0.125
supp(LHS ⋃ RHS) / supp(LHS) = 1 / 8 = 0.125
supp(LHS ⋃ RHS) / supp(LHS) = 2 / 8 = 0.25

Figure 7: Sample dataset for association rules calculation and data types suggestion

Hence, we can calculate the confidence values (see Fig. 6 above) for each rule:

1. $"phone" \Rightarrow "varchar(24)"$, conf = $4/8 = 0.5$;
2. $"phone" \Rightarrow "varchar(30)"$, conf = $1/8 = 0.125$;
3. $"phone" \Rightarrow "integer"$, conf = $1/8 = 0.125$;
4. $"phone" \Rightarrow "varchar(20)"$, conf = $2/8 = 0.25$.

Since the maximum confidence (conf = 0.5) was reached for the first association rule $"phone" \Rightarrow "varchar(24)"$, the "varchar(24)" data type could be suggested for the "phone" attribute of the data model DB (2) created in the previous stage.

Therefore, having business rules of p attributes each of which is associated with w data types, we can use the following equation to calculate confidence values of the association rules:

$$\mathrm{conf}_k^v(LHS_k \Rightarrow RHS_k^v) = \frac{\mathrm{supp}(LHS_k \cup RHS_k^v)}{\mathrm{supp}(LHS_k)}, k = \overline{1,p}, v = \overline{1,w}, \tag{3}$$

where:

- LHS_k is the k-th attribute placed as the left-hand-side of the rule;
- RHS_k^v is the v-th data type associated with the k-th attribute placed as the right-hand-side of the rule;
- supp is the number of rules that contain a given set of items.

Let us introduce the set of data types $D = \{d_k | k = \overline{1,p}\}$ as part of the data model DM (2) that correspond to each attribute a_k, $k = \overline{1,p}$, so it will look like: $DM = \langle E, A, R, \theta, \varphi, D \rangle$.

The task includes a search of the most suitable data types d_k, $k = \overline{1,p}$ for given attributes a_k, $k = \overline{1,p}$ mentioned in the data model DM (2) created in the previous stage.

Therefore, using calculated confidence values (3) of considered association rules we can find the most suitable data types for data model DM (2) attributes:

$$d_k = \arg\max_{RHS_k^v} \{\mathrm{conf}_k^v(LHS_k \Rightarrow RHS_k^v) | v = \overline{1,w}\}, k = \overline{1,p}. \tag{4}$$

Since we use the "Spider" dataset, the expected data types suggested for data model attributes will be specific for the SQLite databases this dataset includes.

2.4. Adjustment of Suggested Attribute Data Types for usage in Different Software Development Technologies

Therefore, we should introduce the bottom-up mapping from SQLite-specific to generic data types. Moreover, we also need to introduce a set of top-down mappings from the generic data types from generic data types to technology-specific ones used in chosen programming languages, platforms, database management systems, etc.

Hence, taking into account the generic data types, we can derive them from SQLite-specific data types [18] d_k, $k = \overline{1,p}$ suggested for corresponding attributes a_k, $k = \overline{1,p}$ (see Fig. 7 below).

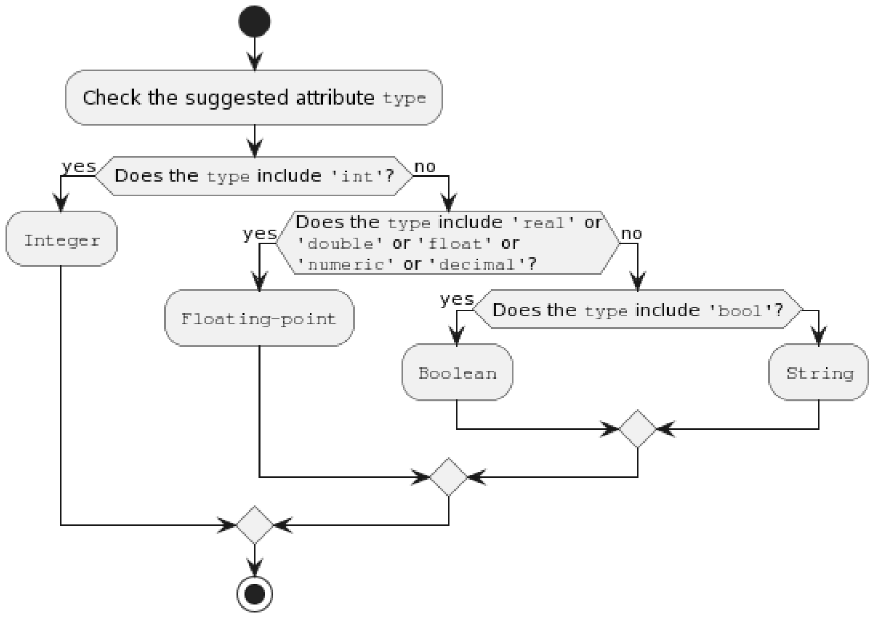

Check the suggested attribute type
Does the type include 'int'?
yes
no
Integer
Does the type include 'real' or 'double' or 'float' or 'numeric' or 'decimal'?
yes
no
Floating-point
Does the type include 'bool'?
yes
Boolean
String

Figure 8: The algorithm to obtain generic data types based on the SQLite-specific data types

According to the PYPL (PopularitY of Programming Language) rating for February 2022 [19], the most popular statically-typed programming languages used in enterprise software development are Java and C#. As for the database management systems, the most popular for February 2022 are SQL-based relational database management systems Oracle, MySQL, and Microsoft SQL Server according to the DB-Engines Ranking [20]. According to the “Top 5 Programming Languages To Build Smart Contracts” rating by 101 Blockchains research platform [21], Solidity is the first among popular smart contract programming languages.

Corresponding data types used in the most popular enterprise programming languages and SQL-based database management systems are demonstrated in Table 1 below.

Table 1
Data types used in the most popular software development technologies

Generic type Technology	Integer	Floating-point	Boolean	String
Java	int	double	boolean	String
C#	int	double	bool	string
SQL	int	real	smallint(1)	varchar(255)
Solidity	int	int	bool	string

Using the algorithm (Fig. 7) and the mapping between languages and generic data types (Table 1), various software development components can be generated based on the data model DM (2): classes or structures, database scripts, smart contracts, or other source code that declare data structures.

2.5. Implementation of Software Components using Different Technologies based on the Data Model

The proposed approach assumes building the abstract data model DM (2) from the SSE business rules to define entities (or concepts), their attributes, and relationships among them. Also, the proposed approach assumes the extension of the data model DM (2) with the attribute data types based on the association rules build from the “Spider” dataset of SQLite databases of different industries. SQLite-based data types then generalized up to the generic data types, such as Integer, Floating-point, Boolean, and String. Then, we can use the data model DM (2) to generate software components having only the SSE business rules on input, according to the specific technology, platform, or language that has its syntax and data types.

Therefore, the object-oriented model of the proposed approach is represented in Fig. 8 below.

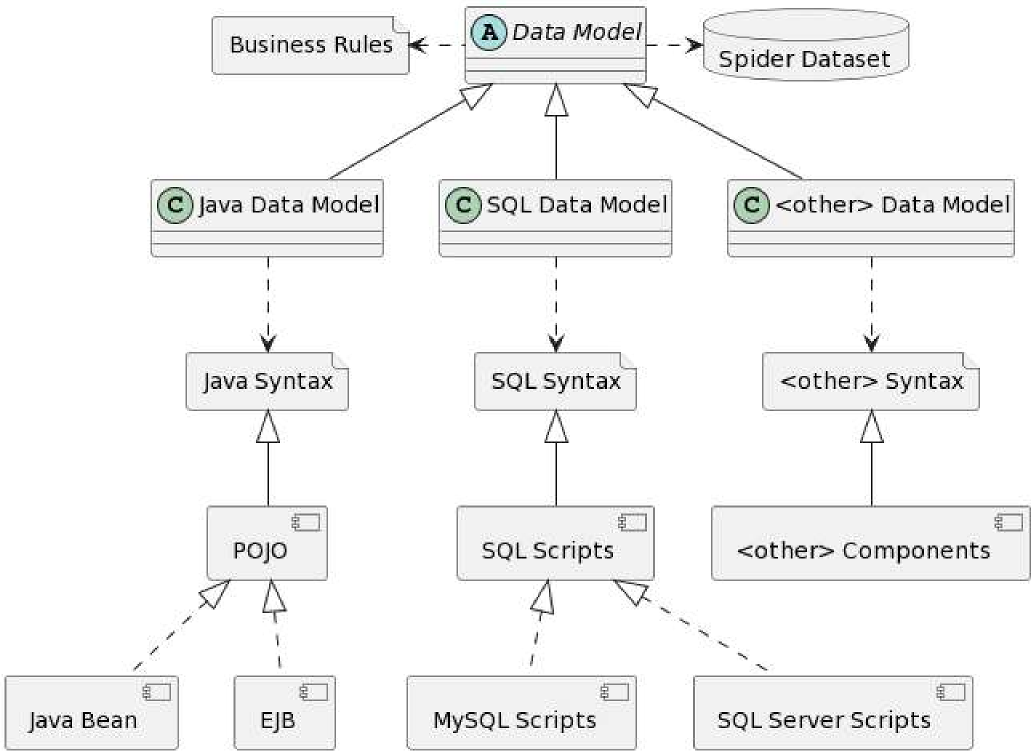


Figure 9: The object-oriented model of the proposed approach

As it is demonstrated in the model above (see Fig. 8), the data model DM (2) should be extended with the syntax rules and specific data types to generate:

- Java-based software components, such as POJO (Plain Old Java Objects, i.e. simple Java objects), Java Beans, Enterprise Java Beans (EJB), and others;
- SQL-based software components, such as MySQL, Microsoft SQL Server, and other database creation scripts;
- other software components, such as source code or configuration files according to the given syntactic rules.

Therefore, the data model DM (2) based on the SSE business rules and data type association rules can be used to automatically generate almost any software component for which are only necessary:

- rules on entity representation according to a given syntax;
- rules on attribute representation, including mandatory ones, according to a given syntax;
- rules on relationship representation according to a given syntax;
- rules on attribute data type representation according to a given syntax.

Such rules could be programmed when the object-oriented model (see Fig. 8) is transformed into a software solution to implement the proposed approach and perform experiments.

3. Results and Discussion

3.1. Software Implementation of the Proposed Approach

We have implemented the software solution using the Python programming language because of its relative simplicity, flexibility, and rich collection of packages, including the packages for natural language processing and database operations.

The software solution uses the following external dependencies demonstrated in Fig. 9 below.

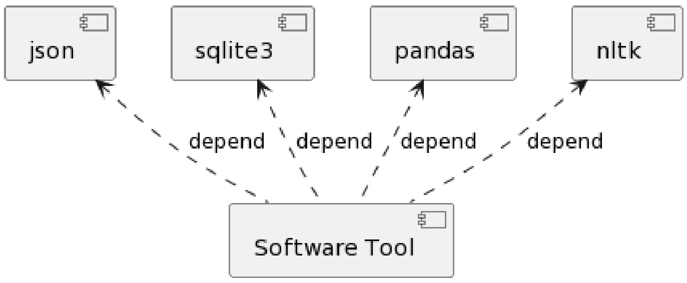


Figure 10: External dependencies of the software tool

According to Fig. 9, which shows external dependencies, there are the following modules:

- json – the JSON (JavaScript Object Notation) format encoder and decoder;
- sqlite3 – the API (Application Programming Interface) for SQLite databases;
- pandas – the open-source data analysis and manipulation tool;
- nltk – computational linguistics library known as the Natural Language Toolkit (NLTK).

The component diagram (see Fig. 10 below) shows the detailed structure of the software tool that implements the proposed approach outlined in section 2.

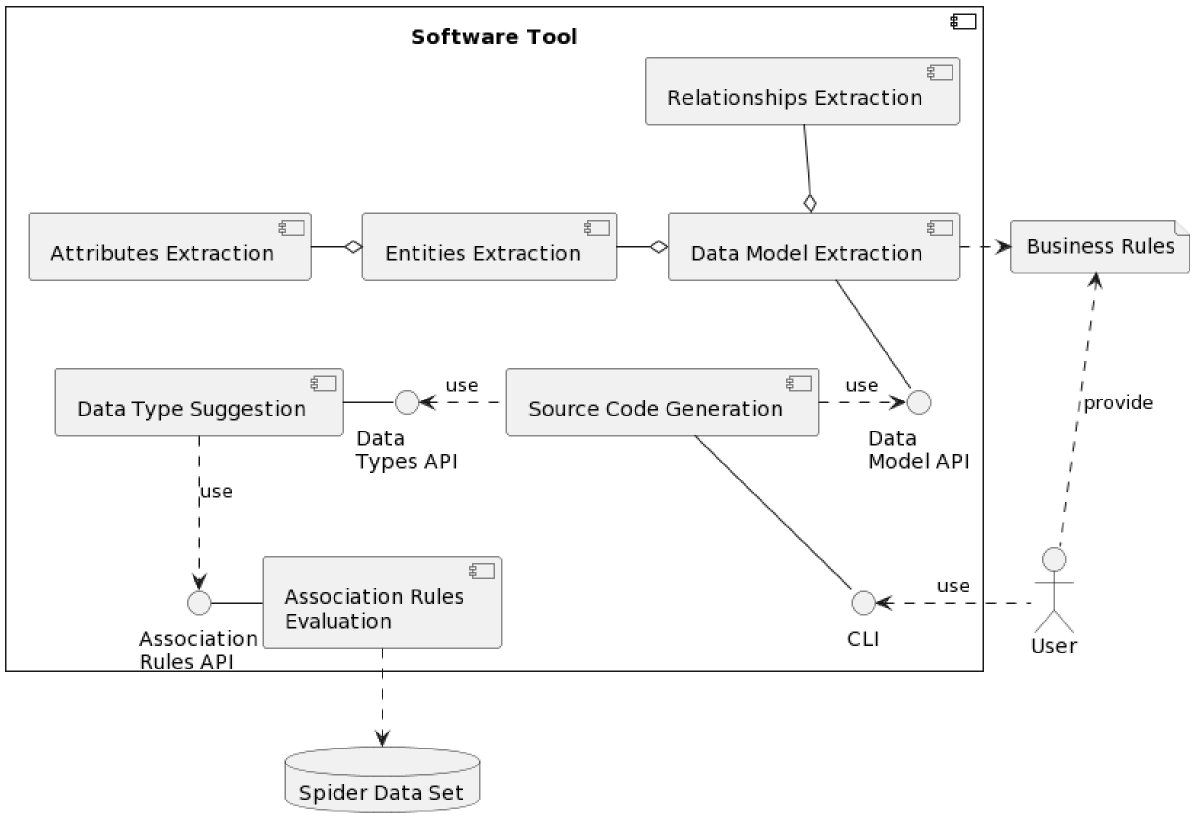


Figure 11: The detailed structure of implemented software tool

According to Fig. 10 demonstrated above, the software solution includes several components:

- the "Data Model Extractor" component is responsible for processing SSE business rules given as input by translating them into triples and then formulating entities, their attributes, and relationships from these triples;
- the "Data Type Suggestion" component is responsible for utilizing association rules and suggesting the data types for entity attributes according to the selected programming language;
- the "Association Rules Evaluation" component is responsible for processing the "Spider" dataset and building association rules that can be used then by the previously mentioned component to suggest the entity attribute data types;
- the "Source Code Generation" component uses all of the previously mentioned components to produce the source code using the selected programming language (e.g., Java Beans, SQL scripts, etc.) according to the created data model and suggested data types.

The source code generation scheme (see Fig. 11 below) describes translation from the data model DM (2) (that includes entities, their attributes (mandatory or not-null and optional), and relationships) into the source code of software components described only by SSE business rules on input.

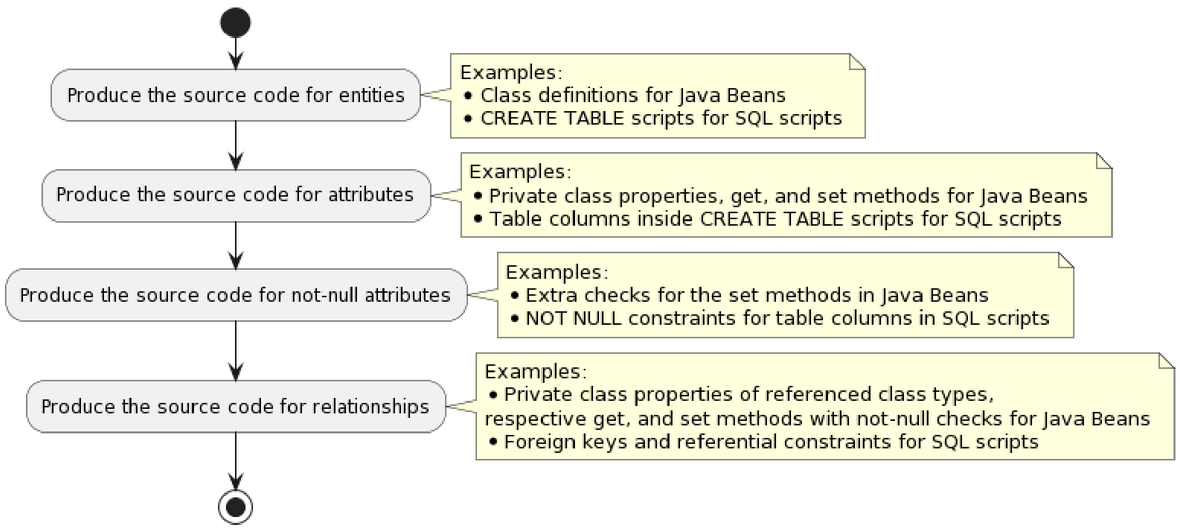

Produce the source code for entities
Examples:
• Class definitions for Java Beans
• CREATE TABLE scripts for SQL scripts
Produce the source code for attributes
Examples:
• Private class properties, get, and set methods for Java Beans
• Table columns inside CREATE TABLE scripts for SQL scripts
Produce the source code for not-null attributes
Examples:
• Extra checks for the set methods in Java Beans
• NOT NULL constraints for table columns in SQL scripts
Produce the source code for relationships
Examples:
• Private class properties of referenced class types, respective get, and set methods with not-null checks for Java Beans
• Foreign keys and referential constraints for SQL scripts

Figure 12: The source code generation procedure

The developed software tool can use the data model DB (2) to produce Java Beans, SQL scripts, or other software components according to pre-configured syntactic rules. In the following sub-section, we demonstrate sample business rules, corresponding data models with suggested attribute data types, and produced software components created using the Java and SQL languages syntax.

3.2. Demonstration of Source Code Generation from Business Rules

Let us consider the example of business rules that describe brands, products, and categories as part of the e-commerce platform. It is well known that nowadays online stores are large websites with complex frontend but even more complex backend software components used as their parts.

Development and maintenance of medium and large e-commerce websites require considerable software development and database design effort. However, this effort could be reduced using the proposed approach and developed software tool. Therefore, in this sub-section, we will demonstrate the automatic generation of Java Beans for the backend business logic and SQL scripts to create the database tables of an enterprise e-commerce platform.

According to the proposed approach, the business rules of intended software components should be given in the SSE format. Hence, sample business rules are following:

- product owns title;
- product includes brand;

- product has description;
- product has image;
- product owns price;
- product owns amount;
- product includes category;
- product has votes;
- product has rating;
- category owns title;
- category has description;
- category has image;
- brand owns title;
- brand has image;
- brand has description;
- brand owns origin country.

Therefore, we can pass these business rules as the input data of developed software and obtain the following triplestore model (see Fig. 12 below).

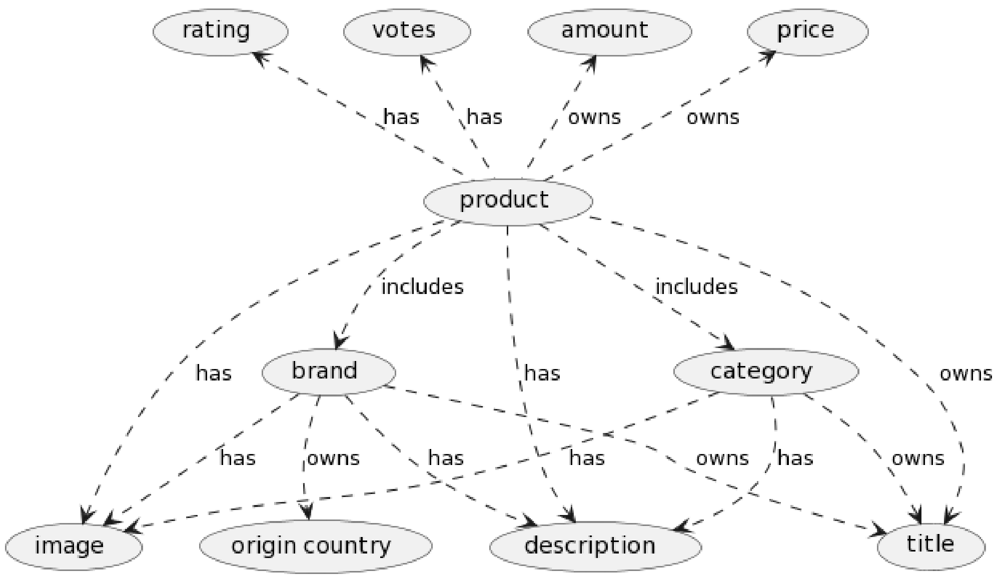


Figure 13: The triplestore model based on business rules

Obtained triplestore model represents on the conceptual level the subject domain that business rules describe. This triplestore model was then transformed into the data model (see Fig. 13 below).

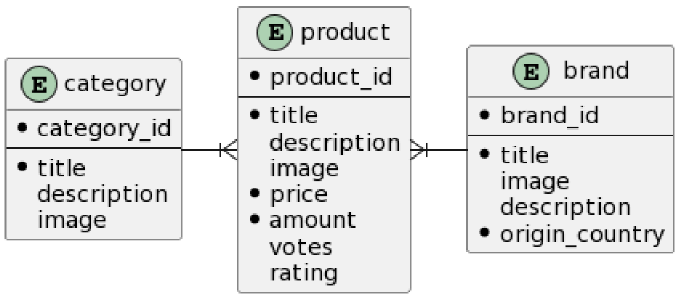


Figure 14: The data model based on triplestore model

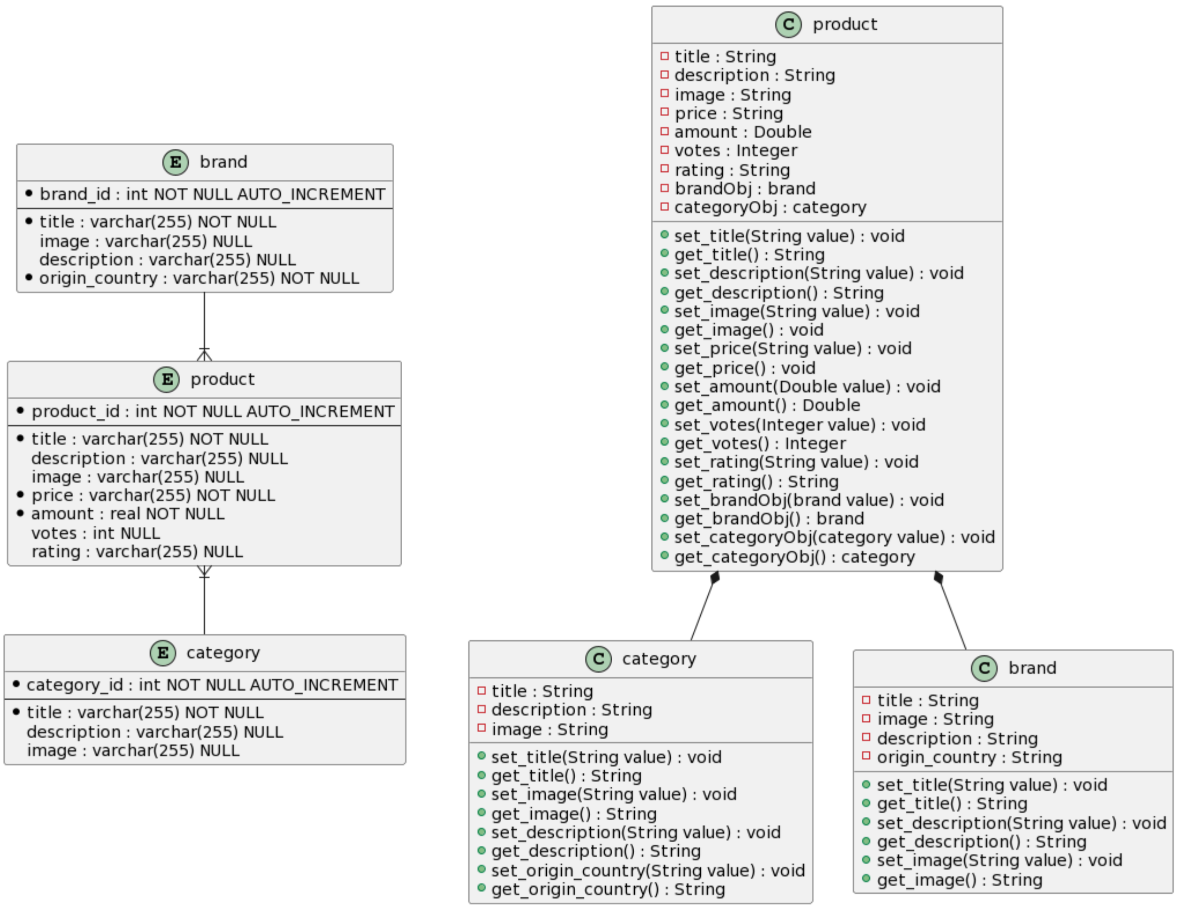
The data model then was extended by attribute data types (see Table 2) based on association rules built using the “Spider” dataset that contains 200 databases of 138 different domains.

Table 2
The suggested data types based on association rules

Attribute	Data type	Confidence	Attribute	Data type	Confidence	Attribute	Data type	Confidence
title	String	0.60	price	String	0.29	rating	String	0.38
description	String	0.25	amount	Float	0.33	origin country	String	–
image	String	–	votes	Integer	0.50			

Obtained results (see Table 2) show the highest confidence of suggested data types for “title” (0.60) and “votes” (0.50) attributes, intermediate confidence for “rating” (0.38) and “amount” (0.33) attributes, and low confidence for “price” (0.29) and “description” (0.25) attributes.

Finally, SQL scripts and Java Beans were generated based on the data model (see Fig. 14 below). The data model also demonstrates specific data types of database table columns in SQL scripts and class fields in Java Beans respectively.



E brand
brand_id : int NOT NULL AUTO_INCREMENT
title : varchar(255) NOT NULL
image : varchar(255) NULL
description : varchar(255) NULL
origin_country : varchar(255) NOT NULL
E product
product_id : int NOT NULL AUTO_INCREMENT
title : varchar(255) NOT NULL
description : varchar(255) NULL
image : varchar(255) NULL
price : varchar(255) NOT NULL
amount : real NOT NULL
votes : int NULL
rating : varchar(255) NULL
E category
category_id : int NOT NULL AUTO_INCREMENT
title : varchar(255) NOT NULL
description : varchar(255) NULL
image : varchar(255) NULL
C product
title : String
description : String
image : String
price : String
amount : Double
votes : Integer
rating : String
brandObj : brand
categoryObj : category
set_title(String value) : void
get_title() : String
set_description(String value) : void
get_description() : String
set_image(String value) : void
get_image() : void
set_price(String value) : void
get_price() : void
set_amount(Double value) : void
get_amount() : Double
set_votes(Integer value) : void
get_votes() : Integer
set_rating(String value) : void
get_rating() : String
set_brandObj(brand value) : void
get_brandObj() : brand
set_categoryObj(category value) : void
get_categoryObj() : category
C category
title : String
description : String
image : String
set_title(String value) : void
get_title() : String
set_image(String value) : void
get_image() : String
set_description(String value) : void
get_description() : String
set_origin_country(String value) : void
get_origin_country() : String
C brand
title : String
image : String
description : String
origin_country : String
set_title(String value) : void
get_title() : String
set_description(String value) : void
get_description() : String
set_image(String value) : void
get_image() : String

Figure 15: The source code of Java and SQL software components generated from business rules

The source code of generated software components, including SQL database tables creation scripts and Java Beans, is available in the project’s GitHub repository [22].

3.3. Verification and Validation of the Generated Source Code

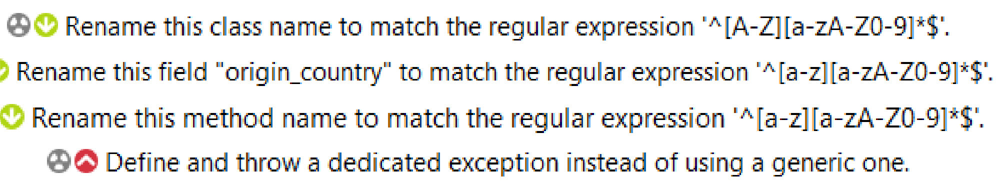
Now we need to verify and validate generated software components. Whereas verification answers the question “are we building the product right?” and corresponds to the static code checking, validation answers the question “are we building the right product?” and corresponds to the dynamic code testing [23].

We used the following tools to verify and validate generated SQL database creation scripts and Java Beans:

- MySQL – one of the most popular free relational databases;
- phpMyAdmin – a free web software tool for handling the administration of MySQL;
- JDK (Java Development Kit) and Eclipse IDE (Integrated Development Environment);
- SonarLint – an IDE extension for identification of quality and security issues in the code;
- JUnit – a unit testing framework for the Java programming language.

Using the SonarLint extension for Eclipse IDE we completed the static analysis of SQL database creation scripts. Verification results show no issues identified in the code.

However, the static analysis of Java Beans code shows four types of identified issues. These issues include invalid class names, field names, and method names, as well as usage of generic exceptions instead of dedicated ones (see Fig. 15 below).



Rename this class name to match the regular expression '^[A-Z][a-zA-Z0-9]*$'.

Rename this field "origin_country" to match the regular expression '^[a-z][a-zA-Z0-9]*$'.

Rename this method name to match the regular expression '^[a-z][a-zA-Z0-9]*$'.

Define and throw a dedicated exception instead of using a generic one.

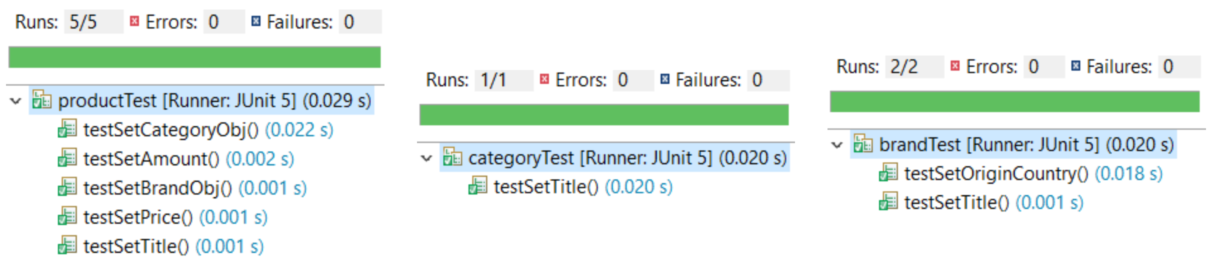
Figure 16: Types of identified issues in generated Java Beans

Execution of generated SQL database creation scripts demonstrates successfully created tables on the MySQL server. Data manipulation SQL statements (see Fig. 16 below) demonstrate the integrity and consistency of created database tables.

```
MariaDB [product_test]> --- test foreign key constraints
MariaDB [product_test]> INSERT INTO product (title, description, image,
    -> price, amount, votes, rating, brand_id, category_id)
    -> VALUES ('Xiaomi Mi 11', 'Smartphones and accessories', 'mi11.png',
    -> 89.99, 999, 4321, 4.3, 0, 0);
ERROR 1452 (23000): Cannot add or update a child row: a foreign key constraint fails (`product_test`.
`product`, CONSTRAINT `product_ibfk_1` FOREIGN KEY (`brand_id`) REFERENCES `brand` (`brand_id`))
MariaDB [product_test]>
MariaDB [product_test]> --- test null columns
MariaDB [product_test]> INSERT INTO product (title, description, image,
    -> price, amount, votes, rating, brand_id, category_id)
    -> VALUES (NULL, NULL, NULL,
    -> NULL, NULL, NULL, NULL, NULL, NULL);
ERROR 1048 (23000): Column 'title' cannot be null
```

Figure 17: Scripts that demonstrate integrity and consistency of created database tables

Execution of generated Java Beans demonstrates successfully compiled classes. JUnit tests (see Fig. 17 below) demonstrate successfully passed cases that check setters for mandatory class fields.



```
Runs: 5/5   Errors: 0   Failures: 0
productTest [Runner: JUnit 5] (0.029 s)
    testSetCategoryObj() (0.022 s)
    testSetAmount() (0.002 s)
    testSetBrandObj() (0.001 s)
    testSetPrice() (0.001 s)
    testSetTitle() (0.001 s)

Runs: 1/1   Errors: 0   Failures: 0
categoryTest [Runner: JUnit 5] (0.020 s)
    testSetTitle() (0.020 s)

Runs: 2/2   Errors: 0   Failures: 0
brandTest [Runner: JUnit 5] (0.020 s)
    testSetOriginCountry() (0.018 s)
    testSetTitle() (0.001 s)
```

Figure 18: Successfully passed JUnit test cases

Obtained results (see Fig. 16) demonstrate that created database tables fully support data integrity and consistency, including enforcement of foreign key constraints and not null columns. Passed JUnit tests (see Fig. 17) show the validness of created Java Beans that contain setters throwing exceptions if null values are passed as input parameters.

As for the limitations, static code analysis of created Java Beans (see Fig. 15) shows that generated classes, fields, and methods do not match Java naming conventions. Therefore, we should elaborate on the proposed approach and software solution in future work to prevent such issues and introduce an automatic generation of dedicated exceptions.

3.4. Contribution to Intelligent Source Code Generation Systems

Despite the detected limitations (missing dedicated exceptions and naming violations in Java code) of the proposed approach and the software tools, the generated SQL database creation scripts and the Java Bean classes are valid and correspond to the given business rules (see Section 3.3). Generated artifacts can be used in an information system software development project after minor tuning – to customize data types and meet coding conventions. Therefore, this study encourages projects to move toward Intelligent Software Engineering practices that assume the usage of intelligent techniques in Software Engineering [24]. The proposed approach and the software tool based on Natural Language Processing techniques assume the implementation of an intelligent source code generation system that is supposed to bridge the gap between software requirements (given as business rules) and the design of the information system's data layer. Furthermore, the elaborated intelligent system will result in an automatic source code generation environment that can augment traditional IDEs.

4. Conclusion and Future Work

In this paper, we proposed the approach to automatic source code generation driven by business rules. This approach includes several steps: translation of business rules given as SBVR Structured English statements into the triplestore model, translation of the triplestore model into the data model, suggestion of attribute data types based on association rules, adjustment of suggested attribute data types for usage in different software development technologies. The object-oriented model of the proposed approach describes its general idea and area of usage.

The Python-based software solution uses the "Spider" dataset of about two hundred databases with multiple tables covering different domains to suggest attribute data types. The software tool uses the Natural Language Toolkit package for business rules tokenization and part of speech tagging.

Currently, the software tool has a command-line user interface and allows to generate software components using SQL and Java syntax to produce SQL tables creation scripts for database layer implementation and Java Beans for backend layer implementation.

We used the developed software solution to perform experiments with the sample set of business rules given in the SSE format. These business rules were translated into triples to build the conceptual model of the subject domain. The conceptual model based on the triplestore model was built, attribute data types were suggested and adjusted according to SQL and Java syntax rules. Finally, generated software components were verified using the static code analysis performed by SonarLint for Eclipse IDE and validated using the dynamic code testing performed by SQL data manipulation statements and JUnit test cases.

Produced SQL scripts and Java classes are completely operable, tables with foreign key constraints were created and Java Beans were compiled. However, obtained verification and validation results show naming issues and generic exceptions misuse in generated Java Beans.

Future work in this field includes the elaboration of the proposed approach and created software solution to avoid current limitations and extend types of software components that can be generated based on business rules.

5. References

[1] M. O'Neill, L. Spector, Automatic programming: The open issue?, Genetic Programming and Evolvable Machines 21 (2020) 251-262. doi:10.1007/s10710-019-09364-2.

[2] K. Talesra, G. S. Nagaraja, Low-Code Platform for Application Development, International Journal of Applied Engineering Research 16(5) (2021) 346-351. doi:10.37622/IJAER/16.5.2021.346-351.
[3] A. Haj et al, The semantic of business vocabulary and business rules: an automatic generation from textual statements, IEEE Access 9 (2021) 56506-56522. doi:10.1109/ACCESS.2021.3071623.
[4] A. Haj, Y. Balouki, T. Gadi, Automatic extraction of SBVR based business vocabulary from natural language business rules, in International Conference on Advanced Information Technology, Services and Systems, Springer, Cham, 2018, pp. 170-182. doi:10.1007/978-3-030-11914-0_19.
[5] B. Akhtar et al., Generating RDFS Based Knowledge Graph from SBVR, in International Conference on Intelligent Technologies and Applications, Springer, Singapore, 2018, pp. 618-629. doi:10.1007/978-981-13-6052-7_53.
[6] I. Essebaa, S. Chantit, Tool Support to Automate Transformations from SBVR to UML Use Case Diagram, in ENASE, 2018, pp. 525-532. doi:10.5220/0006817705250532.
[7] S. Moschoyiannis, A. Marinos, P. Krause, Generating SQL queries from SBVR rules, in International Workshop on Rules and Rule Markup Languages for the Semantic Web, Springer, Berlin, Heidelberg, 2010, pp. 128-143. doi:10.1007/978-3-642-16289-3_12.
[8] A. Marinos, S. Moschoyiannis, P. J. Krause, An SBVR to SQL Compiler, RuleML Challenge 649 (2010).
[9] T. Nadeem, Automated translation of SBVR to SQL queries, Master in Science Thesis, The Islamia University of Bahawalpur, 2013.
[10] A. Kate et al., Conversion of Natural Language Query to SQL Query, 2018 Second International Conference on Electronics, Communication and Aerospace Technology (ICECA), IEEE, 2018, pp. 488–491. doi:10.1109/ICECA.2018.8474639.
[11] A. Kopp, D. Orlovskyi, S. Orekhov, An approach and software prototype for translation of natural language business rules into database structure, in Proceedings of the 5th International Conference on Computational Linguistics and Intelligent Systems (COLINS 2021), 2021, pp. 1274-1291.
[12] K. E. Wiegers, J. Beatty, Software Requirements. Best practices. Developer Best Practices, Microsoft Press, 2013.
[13] P. André, M. E. Amin Tebib, More Automation in Model Driven Development, in International Conference on Model and Data Engineering, Springer, Cham, 2021, pp. 75-83. doi:10.1007/978-3-030-78428-7_7.
[14] W. el Abed, Semantic Business Vocabulary and Rules (SBVR), 2017. URL: https://tdan.com/semantic-business-vocabulary-and-rules-sbvr.
[15] D. Tomaszuk, D. Hyland-Wood, RDF 1.1: Knowledge representation and data integration language for the Web, Symmetry 12 (2020) 84. doi:10.3390/sym12010084.
[16] K. Poonsirivong, C. Jittawiriaynukoon, Big data analytics using association rules in eLearning, in 2018 IEEE 3rd International Conference on Big Data Analysis (ICBDA), IEEE, 2018, pp. 14-18. doi:10.1109/ICBDA.2018.8367643.
[17] Yale University’s Spider 1.0 NLP Dataset, 2020. URL: https://www.kaggle.com/jeromeblanchet/yale-universitys-spider-10-nlp-dataset.
[18] Datatypes In SQLite, 2021. URL: https://www.sqlite.org/datatype3.html.
[19] PYPL PopularitY of Programming Language, 2022. URL: https://pypl.github.io/PYPL.html.
[20] DB-Engines Ranking, 2022. URL: https://db-engines.com/en/ranking
[21] G. Iredale, Top 5 Programming Languages To Build Smart Contracts, 2021. URL: https://101blockchains.com/smart-contract-programming-languages/.
[22] The project’s GitHub repository, 2022. URL: https://github.com/andriikopp/SBVR-to-source-code-generation-study.
[23] T. Hamilton, Static Testing vs Dynamic Testing: What’s the Difference?, 2022. URL: https://www.guru99.com/static-dynamic-testing.html.
[24] M. Perkusich et al., Intelligent software engineering in the context of agile software development: A systematic literature review, Information and Software Technology 119 (2020) 106241. doi:10.1016/j.infsof.2019.106241.