TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Pjotr Volostnõh 211922IAPM

Design and Implementation of a Tool for Generating PostgreSQL Functions Based on Contracts of Database Operations

Master's Thesis

Supervisor: Erki Eessaar
PhD

Tallinn 2023

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Pjotr Volostnõh 211922IAPM

ANDMEBAASIOPERATSIOONIDE LEPINGUTE PÕHJAL POSTGRESQL FUNKTSIOONIDE GENEREERIMISE TÖÖVAHENDI KAVANDAMINE JA REALISEEMINE

Magistritöö

Juhendaja: Erki Eessaar
PhD

Tallinn 2023

Author’s Declaration of Originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Pjotr Volostnõh

09.05.2023

Abstract

A good way to design a piece of software is to create models and produce a working system based on these. Operations, including the operations that modify data in a database, can be modelled by using contracts, where one specifies conditions that must be satisfied before the operation takes place (i.e., preconditions), the expected input from the users of the operation (i.e, parameters), and the expected situation afterwards (i.e, postconditions). If the invoker of an operation can ensure the fulfilment of preconditions, then they can expect that the operation achieves its postconditions. This approach is known as design by contract and nowadays it is mainly used to specify software elements of applications.

This work explores a way to formalise analysis-level contracts of database operations in such a way that allows one to use the contracts to generate a conforming implementation for these operations. Analysis-level contracts of database operations refer to the elements of conceptual data model (i.e, entity types, attributes, and relationship types) not to the actual data structures of a database (i.e., tables and their columns). More specifically, domain specific languages for presenting analysis-level contracts of database operations and the technical context (i.e., tables and columns) of the operations are proposed. The syntax of the languages is presented by using Extended Backus–Naur form. Moreover, the abstract syntax of the languages is also presented as UML class diagrams. Based on these a program in Java is created that takes a contract and the context as an input and generates an implementation of the operation as a PostgreSQL SQL-language function. The program has graphical user interface and must be placed on the computer of its user. The tool is validated by creating contracts of various database operations and generating implementation of these.

Currently known problems of the proposed solution and proposals for improvement and further development are also presented in the work. The resulting software is open source. The tool is licensed under GPL version 3 license. Its source code as well as packaged version for the immediate usage is available at `https://github.com/ghpv/dbcsql`.

The thesis is written in English and is 106 pages long, including 8 chapters, 23 figures and 0 tables.

Annotatsioon
Andmebaasioperatsioonide lepingute põhjal PostgreSQL funktsioonide genereerimise töövahendi kavandamine ja realiseemine

Hea viis tarkvara loomiseks on kirjeldada tarkvara struktuuri ja käitumist mudelite abil ning luua mudelite põhjal toimiv süsteem. Operatsioone, sealhulgas andmebaasis andmeid muutvaid operatsioone, saab modelleerida lepingute abil, kus määratakse kindlaks tingimused, mis peavad olema täidetud enne operatsiooni läbiviimist (st eeltingimused), kasutajalt oodatav sisend operatsioonile (st parameetrid) ja oodatav olukord pärast operatsiooni lõpetamist (st järeltingimused). Kui operatsiooni käivitaja suudab tagada eeltingimuste täidetuse, siis võib ta eeldada, et operatsioon saavutab oma järeltingimused. Seda lähenemisviisi nimetatakse lepingprojekteerimiseks ja tänapäeval kasutatakse seda peamiselt rakenduste osaks olevate tarkvaralementide kirjeldamiseks.

Käesolevas töös uuritakse, kuidas esitada analüüsitaseme andmebaasioperatsioonide lepinguid nii, et oleks võimalik nendest genereerida nende operatsioonide realisatsioon. Analüüsitaseme andmebaasioperatsioonide lepingutes ei viidata mitte andmebaasi andmestruktuuridele (nt tabelid ja nende veerud), vaid kontseptuaalses andmemudelis kirjeldatud olemitüüpidele, atribuutidele ja seosetüüpidele. Täpsemalt pakutakse välja valdkonnaspetsiifilised keeled andmebaasioperatsioonide analüüsitaseme lepingute ja nende operatsioonide tehnilise konteksti (st tabelite ja veergude) esitamiseks. Nende keelte süntaksi esitamiseks kasutatakse laiendatud Backus-Naur notatsiooni. Lisaks esitatakse abstraktne süntaks ka UML klassidiagrammina. Nende keelte põhjal luuakse Java programmeerimiskeele abil kasutaja arvutis olev graafilise kasutajaliidesega programm, mis võtab sisendiks lepingu ja konteksti ning genereerib operatsiooni realisatsiooni PostgreSQL SQL-keele funktsioonina. Tulemuste valideerimiseks kasutati valminud vahendit erinevate lepingute kirjeldamiseks ja nendest koodi genereerimiseks.

Töös esitatakse ka loodud lahenduse hetkel teadaolevad probleemid ja ettepanekud edasiarenduseks. Tarkvara on avatud lähtekoodiga. See on litsenseeritud GPL versioon 3 litsentsiga. Selle lähtekood ja kasutamiseks pakendatud versioon on kättesaadav `https://github.com/ghpv/dbcsql`.

Lõputöö on kirjutatud inglise keeles keeles ning sisaldab teksti 106 leheküljel, 8 peatükki, 23 joonist, 0 tabelit.

List of Abbreviations and Terms

API	Application Programming Interface
CASE	Computer Aided System Engineering
CLI	Command Line Interface
CRUD	Create-Read-Update-Delete
DBMS	Database Management System
DDL	Data Definition Language
DSL	Domain Specific Language
EBNF	Extended Backus–Naur form
GPL	General Public License
GUI	Graphical User Interface
JDBC	Java Database Connectivity
MBD	Model Based Development
MDD	Model Driven Development
MOF	Meta-Object Facility
ORM	Object-Relational Mapping
RAM	Random Access Memory
RUP	Rational Unified Process
SQL	Structured Query Language
UML	Unified Modeling Language
UTF	Unicode Transformation Format
URI	Uniform Resource Identifiers

Table of Contents

List of Figures

1 Introduction

This chapter gives a short background, states the problem, and gives an overview of the structure of the work.

1.1 Background and Problem

Database routines (functions and procedures) can be used to implement virtual data layer. [1] The layer provides the public interface through which database applications can access the data. Having such layer makes it possible to continuously evolve the application view of the data without having to change the underlying structure of base tables all the time. [1] Other advantages include execution speed (routine is compiled and executed in the database itself), security (direct access to base tables can be restricted to admins and only the interface kept public), and tracking dependencies between routines and base tables in the system catalog of the database. [2, 3] There are also disadvantages to using these routines, which include requiring specialised developers (most developers can handle SQL of up to moderate complexity according to the experience of the author), portability problems (different database management systems - DBMSs - support different languages for creating the routines), and spread of code (it is unclear whether business rules should be implemented within application or database). [2, 4] Although the procedural language for writing SQL routines has already been standardised in 1996 [5] it is still the case that in different DBMSs the syntax and semantics of the language for creating the routines are different. Many developers consider the use of database routines an outdated approach and advocate for using alternatives such as Object-Relational Mapping (ORM). However, the fact is that developers still use these (e.g., due to legacy code, or they evaluate that there are more advantages than disadvantages). Additionally, languages for creating routines in MS SQL Server (TransactSQL) and Oracle (PL/SQL) databases are still quite popular among programming languages. In May 2023 TransactSQL and PL/SQL were at #31 and #44 place in a popularity index of programming languages, respectively. [6] Furthermore, some of the most popular NewSQL and NoSQL systems also support database routines, i.e., their creators deem these necessary. Some examples include Cassandra [7], Redis [8], MongoDB [9], neo4j [10], voltDB [11], and influxDB [12].

During system analysis it is possible to identify database operations and explain the intention of these with the help of contracts. Each contract of an operation describes the responsibilities of the invoker of the operation (in terms of preconditions) and the responsibilities of the operation (in terms of postconditions). If the invoker ensures fulfilment of the preconditions, then they can expect fulfilment of the postconditions by the operation. As it is discussed in chapters 3.6 and 3.7 there is a clear lack of tooling to support smooth transition from system analysis to implementation of database operations. Thus, analysts would create contracts for database operations following some free-form notation that the implementer must read and fully understand before being able to implement the operations.

This process has the following problems:

- It takes time for the implementer to understand the intention of the contract, as well as its context (i.e., the specific DBMS that it must be implemented into).
- If the contracts are insufficiently detailed, then the implementer has more leeway in interpretation as well as more difficulty in translating these into a precise implementation.
- Since the translation from contract to implementation must be done manually, there is room for human error, including failure to check for some preconditions, implementation of side effects, and failure to satisfy all the postconditions.
- Each SQL DBMS has its own dialect of SQL. These dialects are not fully conformant with each other. Thus, this manual process has to be repeated for all the target platforms.
- If the contract has to be changed, then this change has to be manually applied to all target platforms.

1.2 Description of the Work

This work aims to propose a formal framework for defining and generating implementation of analysis-level operation contracts. In particular, a Domain Specific Language (DSL) is proposed that allows expressing operation contracts in a formal way. Moreover, another DSL that allows defining the technical context (i.e., tables and columns) of these contracts is proposed. Finally, a tool is implemented that takes a contract as well as its context information (i.e., description of relevant database tables) and generates a fully conforming implementation for SQL databases that are implemented with the help of PostgreSQL DBMS.

Chapter 2 presents the high-level goals of the work and discusses the overall process that this work follows. All theoretical background is given in chapter 3, including discussions as to why these analysis-level contracts are used at all, what are the benefits from formalising these, and introducing code generation tools around these. Moreover, the chapter gives an overview of current tooling supporting contracts and SQL-code generation. Chapter 4 discusses the format for database contracts and how it was formalised. We analyse contracts that have the old free-form format. In this way, it is possible to collect all the expected usages of the tool and design a solution that satisfies all of these. The tool itself is discussed in chapter 5. In this chapter functional requirements to it are expressed and the iteration plan for the development is presented. The inner workings of the tool are represented with the help of UML class diagrams, which are a standard way to represent and design object-oriented software. Finally, validation is presented in chapter 7 and summary is presented in chapter 8.

2 Methodology

This chapter states the goals of the work, brings out research questions that the work tries to answer, gives an overview of the work process as well as lists the used tools.

2.1 Goals of the Work

The high level goal of the work is to design and implement a tool that takes description of database tables and analysis-level contracts of database operations as an input and produces SQL statements for creating database routines that implement the contracts as routines in a particular DBMS.

To be more specific, here are all the high-level goals presented to the tool by the supervisor:

- The tool must be implemented in a way that allows extension:
  - Possible to add DBMS support.
  - Possible to add language support (in case a DBMS supports multiple languages for creating routines).
  - Possible to add other generators (e.g., Object-relational mapping (ORM) code).
  - Possible to add support of other data models (e.g., document data model, property graph model, object-oriented data model). It could be that the database and its operations would be implemented in a DBMS that has other data model than the underlying data model of SQL.
- The tool must be open source.
- The tool must be published to the world (e.g., GitHub).
- The tool must be easy enough to be used by non-developers:
  - Contracts must be easy to use (read and modify) by non-programmers.
  - The tool must be easy to use by non-programmers (e.g., have a graphical user interface).
  - Installation should be easy and self-contained, i.e., should not require installation of additional software as a precondition.
- The tool must support at least generating code to PostgreSQL.
- The tool must be able to deal with operations that modify data in the database, i.e., operations that only read data are out of the scope of this work.

2.2 Research Questions

This work (as well as the implementation) aim to explore the following questions:

1. Is it possible to express analysis-level database operations by using contracts?
2. Is it possible to express analysis-level contracts of database operations in a form that allows code generation based on the contracts?
3. Does the description of contracts in a more formalised form and generation of database routines based on these improve the development process compared with writing contracts less formally and creating database routines manually?

2.3 Overview of the Process

In general the work is design science[13] work where one designs and implements a novel artifact (in this case a DSL for representing analysis-level contracts, a DSL for representing the technical context of the contracts, and a tool that transform the contracts in the DSL to database routines) and then evaluates the result.

The implementation part of the thesis used some of the elements of Scrum methodology[14]. In particular, the implementation was done in one-week iterations, where both the author and the supervisor had meetings to review progress done during the last iteration, and agree on the scope and goals of the next iteration. While Scrum methodology mandates providing testable deliveries to the client (in this case the supervisor) at the end of each iteration, this was not done by the intent of the supervisor for most iterations (though there was a live demo during every review). The iteration plan is presented in section 5.3.

2.4 Description of Used Tools

Below is a list of all used tools:

- LaTeX- typesetting of this paper.
- overleaf - environment to share the paper text between the author and the supervisor.
- java 18 - programming language that the tool was implemented in. While latest long-term support release of java at the time of writing is 17, this version was chosen due to problems with not having UTF-8 as a standard encoding.
- maven - build tool.
- ANTLR4 - library to parse the DSL.
- StringTemplate4 - library for string formatting.

- javafx - GUI library.
- JCommander - CLI argument parsing library.
- PostgreSQL 15.1 - target database management system that was used to test the generated code.
- testng - testing library.
- neovim - code editor.
- jdtls - java language server.
- plantuml - UML modelling tool.
- staruml - UML modelling tool.
- jacoco - code coverage library.
- dbeaver - universal database management tool that was used to create a diagram that depicts database tables.

3 Theoretical Background

This section discusses why modelling is used at all, what are the benefits of formalising models and using these as the basis for code generation, gives some introduction to Domain-Specific Languages and discusses tooling that supports using contracts, and tooling that generates SQL code.

3.1 System Analysis and Modelling

According to the software engineering triptych, one has to collect and understand the requirements of software before the software can be implemented. One has to understand the application domain to be able to present the requirements.[15] Thus, software development has to include requirements analysis in some form. During this process the data that the system needs in the database should be analysed, and the result can be represented as a conceptual data model. Moreover, one could specify requirements to the operations with data by using the design-by-contract approach, which is discussed further in the subsection 3.2. From the contracts of system operations, one could derive methods of classes in an object-oriented program. From the contracts of database operations, one could derive database routines.

3.2 Design By Contract

A "design-by-contract" approach [16] proposes a way to express operations by describing their "contracts". In terms of database operations, these contracts define what is the expected state of the database before the operation so that the operation could succeed and what is the expected state after the successful completion of the operation. Operations may also have invariants, which are logical assertions that should always be held to be true during the execution of the operation. [17] In other words, the invariants of database operations are rules of the data that the operations can never violate. To avoid duplication of the invariants in different operations and facilitate later implementation, these invariants can be defined as constraints in the conceptual data model. Later these constraints can be implemented as database constraints by creating declarative constraints or triggers. The contracts declare pre- and postconditions but do not specify internal steps to achieve the postconditions. Conceptually, the contract is between the operation and the invoker of the operation. If the invoker ensures fulfilment of preconditions, then the operation ensures achievement of postconditions, assuming, that these do not violate invariants. An example of preconditions of an operation could be that an order entity must be registered and must be in a specific state. The postcondition of the operation could be that the order has been deleted from the database. The idea of contracts of database operations is similar to the Larman's system operation contracts [18, 19] that describe requirements to the system behaviour in terms of domain model. Just as system operation can be described using contracts that refer to the elements of domain model, so too can the database contracts be described using contracts that refer to the elements of conceptual data model.

3.3 Model Driven Development

If the models and operations described in the previous sections are done in free form and must be translated manually by developers into an actual implementation, then such development is called "model-*based* development" (MBD). [20] One of the biggest problems with such approach is that at best these models have an *intentional* relationship with the resulting software, meaning that they *should* describe the same system, but may not in practice. In other words, the development is *based on* (hence the name) these models. However, as the software by its very nature has to constantly evolve, so too must the models be adapted to these changes in order to ensure their adequacy. Similarly, from the developer's point of view these models provide a good overview of what the system should be and how it should behave as well as a starting point for the implementation. However, ultimately it is the code that dictates the truth. Thus, if there are any discrepancies between the model and the implementation, then obviously the latter is correct. Because of these issues, developers typically see MBD models as an overhead and intermediate results at best. These are good to get started and work out the general idea at the start. However, later these will become a source of a constant debt and must be continuously *manually* updated. [20]

There is, however, another approach to model usage called Model-*Driven* Software Development (MDD) where the models have the central stage in the development process. The reason why models are seen as an overhead in MBD is because model and code are two disconnected views of the same system, which are not connected in a *formal* way. Therefore, to facilitate Model-*Driven* software development, there must be a formal way to express these models, as well as the tooling that is able to convert a model into its implementation. [20]

More formally, in the book "Model-Driven Software Development" [20] the authors define the following requirements, that must *all* be satisfied:

- Models must be formulated in Domain-Specific Languages (DSLs).

- There must be language(s) that can express the necessary transformations to get code from the model.
- There must tooling (compilers, generators, transformers, etc.) that can generate the implementation on supported platforms from the models expressed in DSL.

Such approach allows automating software production as it is necessary to only formulate the model and the rules of how the model is transformed to the code. Then, by following the rules, the tooling is able to convert from formal model into supported implementations. This immediately leads to an increase in development speed as it is no longer necessary to *manually* transform the model into a conforming implementation. Additionally, this facilitates earlier and better testing as it is possible to generate testable implementation from the model. In a way it allows us to test the model itself. As the generation of code is automated there are no discrepancies of the resulting implementation on the supported systems, which leads to an enhanced software quality. Another benefit that this approach provides is that it is easier to manage the complexity of the problem as models allow expressing the solution on a more abstract level. Additionally, once the rules and the tooling have been defined, these can be used as a software factory based on the conforming models, allowing for better reusability and providing expert knowledge in the problem domain in the form of a formal modelling ecosystem. Furthermore, these models can also be understood by domain experts who may provide insights to the problem without necessarily being familiar with the software. [20] Finally, this allows for all parties to communicate in precise terms on formal models, leaving little room for varying interpretation.

3.4 Domain Specific Software Languages

All software languages can be broadly split into two categories - general purpose and domain specific. A general purpose language, such as C, C++, and Java, can be used for various purposes - indeed it is possible to implement all sorts of software systems in such languages. In contrast, a domain specific language belongs to some very specific domain, and has specific goals. For example, Structured Query Language (SQL) is used to communicate with DBMSs to manage data, database objects, and data access. Another example is LaTeX, which is used to typeset documents, such as this document.

Further, DSLs can be split into external and internal categories. Internal DSLs are defined within some already existing programming language - one can think of these as a high-level API. Since one of the requirements is for the contracts to be usable by non-programmers, it can not be assumed that they know a programming language. Thus, this work can not use this category of DSLs.

External DSLs are defined in a standalone fashion, using its own metamodel and/or grammar[21, p. 25]. This category suits the requirements of this work better because it is possible to define a standalone grammar that deals strictly with defining these contracts. In this way defining contracts does not depend on knowledge of programming languages. It is possible for external DSLs to be textual or graphical.

3.4.1 Ways to Describe Languages

Each software language has an abstract syntax and a concrete syntax. The abstract syntax describes language concepts and rules how to combine the concepts in the statements written in the language. The concrete syntax describes the human-readable notation that is used to present the contracts. To put into simple terms:

- *Abstract* syntax defines what is *possible for the language to express*. For example, `literal plus literal` would imply that the language supports summing two literals.
- *Concerete* syntax defines some *precise syntax that follows the abstract rules*. To continue from previous example, notations that express the abstract summation rule include `2 + 2`, `(+ 2 2)`, `(2 2 +)`, `the sum of 2 and 2`.

Human users write contracts by using the concrete syntax. The contracts have to use the concepts and follow the composition rules that are defined by the abstract syntax. The abstract syntax can be defined textually by creating a context-free grammar or graphically by creating a metamodel. The graphical representation could be done in Meta-Object Facility (MOF) language that is essentially a subset of UML for modelling class diagrams. MOF is a language for defining modelling languages. Graphical representation as a metamodel facilitates better representation of complex relationships between language concepts and allows modelling of attributes of language concepts. All this contributes to more concise and comprehensible representation compared to the representation as a context-free grammar. [22] An example of language, which abstract syntax has been defined visually by using MOF, is UML [23]. Contracts of database operations, both free-form and formal in some DSL are also models - textual models.

To make the abstract syntax of the DSLs understandable to as wide audience as possible, the metamodel of the languages will be created. To facilitate implementation of the languages in the translator as well as formalise the concrete syntax of the languages, the context-free grammar will be created as well.

To connect the work with the existing standards - according to the metalevels of MOF language the description of a DSL (either graphically as a metamodel or textually as a context-free grammar) belongs to the M2 level and actual contracts that will be defined by using the language belong to the M1 level.

3.4.2 Best Practices of DSL Design

As this work includes creating DSLs, the author tried finding guidelines and best practices[24] to follow:

- **Identify language uses early** - in this way it is possible to consider all requirements and identify concepts the language will offer.
- **Ask questions** - Language design is an iterative process, so it is important to keep asking questions to achieve the best possible syntax.
- **Be consistent** - A DSL exists to contribute towards a specific goal. Thus, all its features too should contribute to it. If a feature does not contribute towards a goal, then it should be omitted.
- **Decide carefully between graphical and textual representations** - There are advantages and disadvantes to both. Thus, these should be weighted in the context of the intended use cases.
- **Use existing languages if possible** - Design and implementation of a standalone DSL is a complicated process. Thus, it should be avoided if there are already existing tools that apply.
- **Use existing language definitions if possible** - If there is no applicable language, then it is still advisable to reuse at least parts of existing languages to save on development time and have a more familiar interface to the end users.
- **Use existing type systems if possible** - Type system is one of the hardest parts to implement. Therefore, it would benefit the most from reuse.
- **Use only necessary domain concepts** - DSL should be able to express all the *necessary* domain elements, everything else is not needed.
- **Keep it simple** - A goal of a DSL is to make it easier to work within some domain. Hence, the language itself must be as simple as possible.
- **Be precise** - As a DSL must be simple, it does not need to include features that *might* get used, only those that definitely *will* be used. Similarly, it is not necessary to have multiple redundant ways to express the same meaning.
- **Limit the number of language elements** - A language with more elements is likely harder to understand than the one with fewer elements.
- **Use efficient elements** - While the DSL typically raises the level of abstraction, one should not neglect that it will have to be implemented in some way. Therefore, if there are some ways of expression that allow for more efficient implementations,

then these should be preferred.

- **Reuse existing notations** - Learning and mastering a notation is hard, using notation that domain experts already understand gives a greater benefit.
- **Use descriptive notation** - Notation should clearly express what it will do.
- **Language elements must be distinguishable** - Every language element should be distinct from others so that it is easy to identify it and understand what it does.
- **Use syntactic sugar sparingly** - Syntactic sugar does not contribute to the expressiveness of the language. Therefore, it should only be used in places where it does not limit readability and comprehensibility of the language.
- **Use comments** - Comments allow the expression of various information that may not be relevant to the model, but are still important to the users (e.g., design decisions and explanations)
- **Allow organizational structures** - DSL projects can grow large. Thus, users should be permitted to split it between files and connect these by using mechanisms like importing.
- **Balance between being comprehensive and compact** - Typically, a formal body of information is read much more often than it is modified. Thus, it should be as comprehensive as possible. On the other hand, it should not be too verbose as that makes it both harder to write and read.
- **Use consistent style** - If there are multiple languages, then these should all match as to not surprise the user.
- **Reuse conventions** - If there are some existing conventions for notation, then these should be employed to reduce learning cost.
- **Align the abstract and the concrete syntax** - This makes it easier to process and reformat the DSL.
- **Use a layout that allows for easy conversion between the concrete and the abstract syntax** - Layout of the DSL should not impede understanding or translation of the model it expresses. Ideally, the layout of a program or a model should not affect the semantics at all.
- **Use interfaces** - Interfaces hide implementation levels and allow communication between systems (or modules) on a more abstract level.

As with software, there are also design patterns for DSLs [25] that the author managed to find. Below is an explicit list of all of these, as well as a short description:

1. **Piggyback** - a DSL can be designed based on another DSL to get the features of the parent "for free". Examples of this approach include GNU Flex [26] that allows defining a language specification and get a lexer implementation (in C or C++) as output.

2. **Pipeline** - use multiple DSLs to allow better expression, like C pre-processor macros that allow defining and controlling C source code.
3. **Lexical Processing** - DSLs are, by their nature, restricted languages. Thus, it is possible to restrict it further so that it can be processed by simple string processing (e.g., by employing various lexical hints, such as prefixes and suffixes), *without* having to deal with syntax tree based analysis.
4. **Language Extension** - a DSL can be extended to create an "enhanced" version of the original. This is different from *Piggyback* (described at the start of this list) in that this does *not* create a new language. Instead, it extends the original one with new data types, semantics, and syntactic sugar.
5. **Language Specialisation** - remove unnecessary features from an otherwise applicable DSL to make it more applicable in a domain.
6. **Source-to-Source Transformation** - have the DSL translate into some other code that deals with execution.
7. **Data Structure Representation** - design the DSL in such a way that allows the best expression of the relevant data.
8. **System Front-End** - allow a complex system to be configured and operated by the use of a DSL, rather than configuration options.

3.5 Current Approaches to Implementing Operations

It is possible to split current approaches to implementing database-backed software into three broad categories that can be called code-first, database-first, and data-first.

In the *code*-first approach, the database layer is usually generated by the applications that use the data through technologies like Object-Relational Mapping (ORM). In case of SQL databases all the base tables are generated from their respective source code data classes, and usually, operations are just implemented in the application, with some exceptions where, for example, performance is critical. Similarly, in case of using MDD, database design model that describes base tables is created based on design level class model of object-oriented software. Based on database design model SQL code will be generated. For instance, in case of Rational Unified Process (RUP) the design level data model is created in the Elaboration phase, based on the architecturally significant persistent classes. [27] In case of using RUP, one could define stored procedures as a part of the design level data model. [28]

Opposite to that is *database*-first approach that prioritises the database - it gets a formal definition of tables, usually very verbose access rules, and operations that allow interacting with the stored data.

Finally, one can choose to use external tooling to simply define the data that application uses, maybe even some (more simple) operations that are needed, and generate all the relevant code - source code data classes, database definition statements, database operations, etc. Here it's also possible to use tools like AL [29] and X++ [30] that try to abstract the data layer, allowing to write operations for it on a higher level of abstraction.

This work, by itself, falls into the second category - it is expected that one would have a fully working database in terms of tables and uses this tool to generate implementation of database operations in terms of database routines. With other tools it is possible to extend it to be in the third category, where, admittedly, it makes more sense. The tool's DSL allows expressing contracts at the analysis level, i.e., as requirements. If that is combined with another tool (or if this tool is extended) that allows expressing the data model at the same level, then one could have tooling that generates both the database and its operations from the analysis-level models. Furthermore, with enough extensions it is also possible to have it generate the data classes to be used in the application, too. For this purpose a CASE (Computer Aided System Engineering) tool could be created or an existing CASE tool modified. In such CASE tool, one can define *both* conceptual data model and analysis-level contracts, and generate tables and routines from these, respectively.

3.6 Tools that Support Design by Contract

The author was unsuccessful in finding any tools that allow us to give a contract of a database operation as an input and get an implementation for it for some DBMS as an output. However, design by contract is an established idea that has been tried in programming languages.

One example where design-by-contract has native support is the Eiffel language[31], that allows expressing both pre- and postconditions as part of a function's implementation. This allows the programmer to clearly express what inputs they expect and what the invoker of the function can expect after its completion.

Since general-purpose programming language are, by definition, general-purpose, there also exist third-party libraries that add support for design-by-contract to (usually their own) language. One example is the Object Validation (OVal) library for java. With that library it is possible to place all sorts of restrictions on data fields and methods, allowing the programmer to express all their expected limitations.

Finally, if it is okay to have these checks done at runtime, then most general purpose languages can be said to have support for design-by-contract as one can introduce both pre- and postconditions into the body of a function, throwing exceptions or raising errors if those are violated.

A standalone tool called RM2PT [32] allows giving a requirements model, including contracts of system operations, as the input and get a conformant Model-View-Controller prototype in Java as the output. It includes support for manipulation of database objects, including retrieval, creation and deletion of instances, retrieval and modification of instance attributes, and creation and removal of links between instances. The operations must be defined in Object Constraint Language. [33]

3.7 Other Tools that Support Generating Database Code

There are many tools that try to ease the creation of SQL *tables* – for example UML [23] CASE tools as well as dedicated data modelling tools typically provide a way to generate SQL data definition language (DDL) statements for creating tables and indexes based on the database design model either natively or through a plugin. Sometimes they provide limited support for generating routines. [34] There are also standalone tools like quick-sql [35] that allow a user to describe tables in a simplified form and generate SQL DDL statements based on that. RuleGen framework, written in PL/SQL, was able to generate triggers for Oracle databases that implement complex integrity constraints. In the spring 2023 the website of the tool is not available. [36] However, these tools typically do not generate routines, let alone ones defined by contracts. There is a tool called quick-plsql [37] that in addition to tables can generate routines as well. However, it does not use contracts in terms of pre- and postconditions as an input. Instead, only a select number of templates is supported, such as sample implementations of Create-Read-Update-Delete (CRUD)-operations as well as logging and debugging templates. It is also defined entirely in PL/SQL (the language used mainly in Oracle DBMSs), making it hard to adapt and extend it. There are also some tools that concentrate on generating only CRUD type routines that are based purely on the schema (i.e., without any contracts), such as Visual Studio [38], SSMS tools pack [39], and genfuncpostgres [40].

Anchor modeling is a database development methodology that is use mainly for creating data warehouses. The resulting SQL database has highly normalized (sixth normal form) tables and has good performance characteristics as well as has other advantages like efficient dealing with missing information, schema changes, and preserving historical data. The methodology provides online modeler for creating anchor models. The online modeler has attached SQL code generators for various DBMSs (MS SQL Server, Oracle, PostgreSQL, and Vertica) for generating code (base tables as well as triggers, routines, and views) based on the anchor models. Anchor modeling language is a DSL and the online modeler is an example of tool that allows generating complete database implementation based on an analysis model (anchor model), which presents requirements. [41]

3.8 Summary of the Theoretical Analysis

There are clear benefits to have models drive development as is mandated by MDD, rather than being a mere base for development. At the same time there is a clear lack of tooling to facilitate such approach. The best that the current tooling provides is to provide templates for implementation that do not take contracts into account - in other words, these tools aim to ease the manual work the implementer has to do, rather than alleviating it alltogether.

Therefore, this work aims to provide a framework to facilitate MDD for database routines by providing DSL for the expression of database routines, a DSL for expression of technical context information, and tooling to translate these DSLs into conforming SQL implementation. In this way, the effort of designers can be focused on whether the operation does what it is intended to do, rather than implementing these operations.

4 A Language for Representing Contracts

As was discussed in section 3.3, MDD requires a DSL in order to express models formally. In scope of this work only analysis model operations are considered, and in particular, DSL is trying to make possible to represent *contracts* of these operations.

During the course of this chapter some examples of free-form contracts will be analysed and transformed into formal DSL syntax.

4.1 Free Form Contracts

During the course of this work the supervisor (who teaches database courses in TalTech) provided the author with some sample student projects that were tasked with designing and implementing a simple database-backed system. These sample projects were used to extract examples of how these contracts are represented and used, to use it as a foundation for building the formal DSL syntax. In this chapter some simple examples are shown from a single sample project that deals with a car rental system. The context (both DSL and SQL form) as well as all contracts (in formal notation) are available at `https://github.com/ghpv/dbcsql/tree/master/process/postgres/src/test/resources/sdt` The provided contracts were written in Estonian. For the sake of clarity these are translated into English in this work. Since the contracts in the sample projects follow an MBD approach there is no requirement for the contracts to be formal. Indeed, while the projects followed the general style provided by their teacher, they all ended up having slight differences, and at times, mistakes.

4.1.1 Sample Contracts

Figure 1 represents an entity-relationship diagram (created as UML class diagram) that depicts a relevant part of conceptual data model for sample operations:

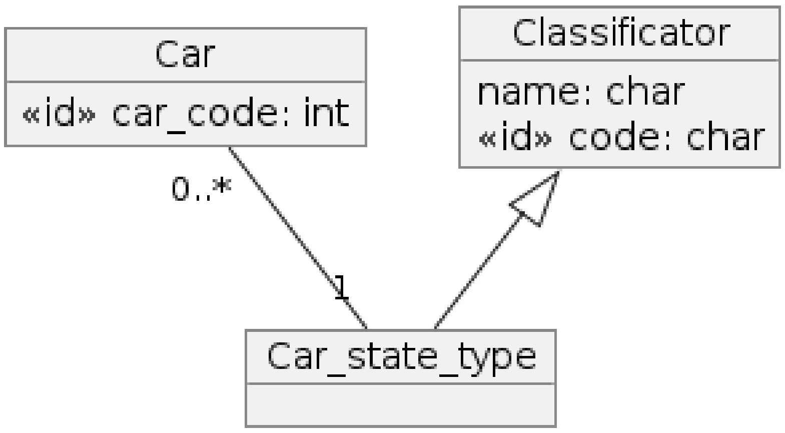


Figure 1. Context for sample operations.

There is a car entity (that has a lot of data omitted as it is not used by the examples shown below) that is connected with a car state classifier entity, which itself extends a generic classifier entity that has numerical and textual IDs.

Figure 2 is an example of a delete operation (note: lines are purposefully broken to fit into page).

```
OP2 Forget a car(
    p_car_code
)
Preconditions:
* Car instance c (has car_code=p_car_code)
    was registered
* c was connected with Car_state_type instance st
    (has name="Waiting")
Postconditions:
-- Delete instances and relationships
* c and all of its relationships were deleted
    from the database
Usage by use cases: Forget a car
```

Figure 2. Sample operation: "Forget a car", free-form contract.

The operation aims to delete a car with a given identifier (car code), provided it is in the state "Waiting". The contract mentions that `c and all of its relationships were deleted`. From the implementer's point of view this postcondition is a bit vague in terms of who is responsible for deleting all the linked data. Is it correct to assume that all the relevant foreign key constraints are defined with the `ON DELETE CASCADE` compensating action? Or is it the job of the implementer to find all linked tables and ensure everything is deleted?

To give another example, figure 3 contains an operation that permanently inactivates a car.

```
OP5 Permanently inactivate a car(
    p_car_code
)
Preconditions:
* Car instance c (has car_code=p_car_code)
    was registered
* c was connected with Car_state_type instance cst_old
    (has name="Active") or
    (has name="Temporariliy inactive")
* Car_state_type instance cst_new
    (has name="Permanently inactive") was registered
Postconditions:
-- Delete relationships
* c connection with cst_old was deleted
-- Form relationships
* c was connected with cst_new
Usage by use cases: Finish a car
```

Figure 3. Sample operation: "Permanently inactivate a car", free-form contract.

The operation aims to change the state of a car with given ID to "permanently inactive", provided it has the state "active" or "temporarily inactive". It should be noted that the contract is somewhat excessively verbose as any given car may be associated with precisely one state at any given time. Therefore, a "connect with the new state" is sufficient here as it would overwrite the old link rendering the "delete connection with the old state" redundant.

4.1.2 Sample Implementations

As a part of the course, students are also tasked with implementing their system in a database of the DBMS of their choice (they can choose between PostgreSQL and Oracle). Since in the scope of the current work only support for PostgreSQL is desired, samples were chosen that implement operations into PostgreSQL. Below are verbatim implementations (reformatted for readability) as done by the authoring students for the contracts given before.

```
CREATE OR REPLACE FUNCTION public.f_forget_car
(
    p_car_code INTEGER
)
RETURNS void
LANGUAGE SQL SECURITY DEFINER
SET search_path TO 'public', 'pg_temp'
BEGIN ATOMIC
    DELETE FROM
        car
    WHERE
        car_code = p_car_code
    RETURNING
        car_code
    ;
END;
COMMENT ON FUNCTION public.f_unusta_auto
(
    p_auto_kood INTEGER
) IS 'Function for car handler
    to delete cars (OP2)'
;
```

Figure 4. Sample operation: "Forget a car", implementation.

As one would expect, this operation deletes a car from the database. Even in such a simple example however, one can already spot some problems. Firstly, the operation returns `void`, i.e., no data despite there being a `RETURNING` clause in the body of the function. This is valid as far as PostgreSQL is concerned. However, it does seem like it was the implementer's intention for the function to return an `INTEGER` value instead. Secondly, despite contract saying that only cars in "Waiting" state must be deleted no attempts are made by the function to check that precondition - the operation simply deletes a car that has a matching identifier, *regardless* of the state it is in. That is not to say that the precondition is not checked at all - when the author tried using this function to delete a car in an incorrect state, it was discovered that this check was offloaded into a separate database trigger that prevents deletion of *any* cars that are not in the "Waiting" state. It is debatable whether this precondition should be checked by the function or by the trigger - the author's personal opinion here is that it should have been checked in the function as well, as it is precisely the precondition of the function. Should the requirement to the database change so that it would be permissible to delete cars in non-"Waiting" state (for some other reason, in some other operation), the trigger would be removed and one would have to remember to implement this check, instead of already having it.

Second contract is implemented like depicted in figure 5.

```
CREATE OR REPLACE FUNCTION
    public.f_permanently_inactivate_car
(
    p_car_code integer
)
RETURNS integer LANGUAGE sql SECURITY DEFINER
SET search_path TO 'public', 'pg_temp'
BEGIN ATOMIC
    UPDATE
        car
    SET
        car_state_type_code = 4::smallint
    WHERE
        car_code = p_car_code
    ;
    RETURN p_car_code;
END;
COMMENT ON FUNCTION public.f_lopeta_auto
(
    p_auto_kood integer
) IS 'Function for manager to finish cars (OP5)';
```

Figure 5. Sample operation: "Permanently inactivate a car", implementation.

As was pointed out after the contract - a car may be in precisely one state. Thus, the implementation takes advantage of that and simply overwrites it. Once again the function makes no attempt to check the state, instead offloading that check to a trigger. It should also be noted that the function uses numerical identifiers instead of the textual ones as the contract specifies. It is very likely that this is done for the sake of easing the implementation - checking against textual identifiers would require querying the state classifier table as well, whereas by using numerical identifiers that check can be avoided by using the correct value. Since this is a technical state classifier table where data changes rarely, it is indeed highly unlikely that the mapping between the numerical and textual identifiers of a state ever changes, making this a reasonable compromise.

4.2 Considerations for Formalisation

As it is mandated by MDD, contracts of operations must be expressed using a formal DSL. Here it should also be noted that since the tool must parse contracts in order to translate these into a conforming implementation, this DSL must also be machine-parseable. In this section, the author documents higher-level decisions and process taken to formalise the provided examples.

4.2.1 Contracts of Database Operations

Students are provided a template for their operations by their teacher. None of the sample projects the author had access to attempted to change it much. The template is indeed very logical. Postconditions refer to instances defined in preconditions and may additionally refer to the parameters of the operation. Preconditions may refer to the parameters. Thus, there is a very simple dependency chain making the order parameters, pre- and postconditions natural. This format simplifies reading and writing of the operations - the *signature* of an operation (i.e., its name and parameters) is defined first, and it defines the high level intent of the operation. Additionally, in order to understand on what instances the postconditions operate, one would firstly have to understand the preconditions. As such, it makes sense for the DSL to mimic this pattern. Since the example contracts are in a textual form, it makes sense for the DSL to be also textual. While it could have been graphical, the author did not see much reason to make it so - textual representation seems sufficient here as there is no information that requires, or benefits from, graphical representation. Additionally, the tooling for textual languages is much easier to implement.

The only part from sample operations that was not kept are short identifiers (e.g., `OP2`) - these seemed like information specific to the database courses where the contracts originated from. Since the sample contracts contain comments, it was deemed acceptable that these short identifiers be moved there and treated as free-form user data. Therefore, the operation signature in DSL is composed only of its name and parameters. Both of these do not present any special challenges as they are just identifiers.

During the course of this work it was identified that only two types of preconditions are sufficient:

- `exists` precondition that introduces an instance, as well as restrictions that befall upon it. This keyword is already evident from the sample operations as all of them introduce instances of entities and applicable restrictions (e.g., a car must have some specific identifier or a state must have a specific name).
- `connection between` precondition that denotes a connection between two instances. It is impossible to define a connection between instances using restrictions because in a conceptual data model the entities do not have attributes that correspond to "columns used in the foreign key" but rather have only connections. Thus, there is nothing for `exists` precondition to restrict in order to signify a connection.

During the course of this work it was identified that there are four types of postconditions. These correspond to four basic operations that can be done with the data - Create, Read, Update, and Delete, commonly referred to as CRUD [42, p. 381-382]. None of the sample contracts contained operations for Read-type operations. In the context of databases, these can be implemented as `SELECT` statements. It would be possible to define contracts of only read operations and generate views, snapshots, or read-only routines based on these. However, it was decided that it falls out of the scope of the current work. Thus, the DSL will also have no support for Read-type operations, concentrating on the remaining types of operations.

4.2.2 Context

As with free-form operations, formal operations also must have context. Furthermore, this context must also be expressed by using a machine-parseable formal language. The biggest problem that prevents the direct usage of entity-relationship diagrams that are depicted as UML class diagrams is that there is no one-to-one mapping between conceptual data model elements (entities, attributes, and relationships) and SQL database elements (tables, columns, and foreign keys). However, this mapping is required since operation contracts refer to the conceptual data model elements, while the generated implementation must use the database ones. It is not immediately obvious how such a mapping can be deduced automatically from a conceptual data model. Because the scope of this work is to explore formalizing contracts of operations and automatic generation of their implementations, rather than the *automatic* creation of this mapping, a standalone DSL was created to specify this mapping. This context DSL could benefit from a graphical representation. However, since the operation contracts' DSL is textual, and it is much easier to implement tooling for textual DSLs, a textual DSL was chosen. That is not to say that it is impossible to have context parsed from UML models. In fact, this feature is proposed in section 7.2.2.

4.3 Formal contracts

Before defining any syntax for the contracts, the author tried finding libraries that facilitate parsing DSLs. In the end, the author had a choice between two libraries - `flex`[26] & `bison`[43] combo or `ANTLR4`[44]. `ANTLR4` seemed like the easiest choice, especially since it allows defining languages in pure Extended Backnus-Naur Form (EBNF) [21, p. 166]. Therefore, `ANTLR4` was selected and EBNF was used to represent the concrete syntax of the DSL. By using parsers generated by `ANTLR4`, it is possible to walk the syntax tree and define callbacks for every node. Full EBNF will not be provided in this document, but is available in github at `https://github.com/ghpv/dbcsql/blob/master/core/src/main/antlr4/ee/taltech/dbcsql/core/phase/input/Contract.g4` (more generic rules are available in parent `antlr4` directory). In this chapter only relevant high-level parts of the language will be provided in order to give the reader an idea of how the language looks like.

4.3.1 Abstract Syntax of the Context DSL in UML

Figure 6 shows the abstract syntax of the context DSL expressed by using a UML class diagram.

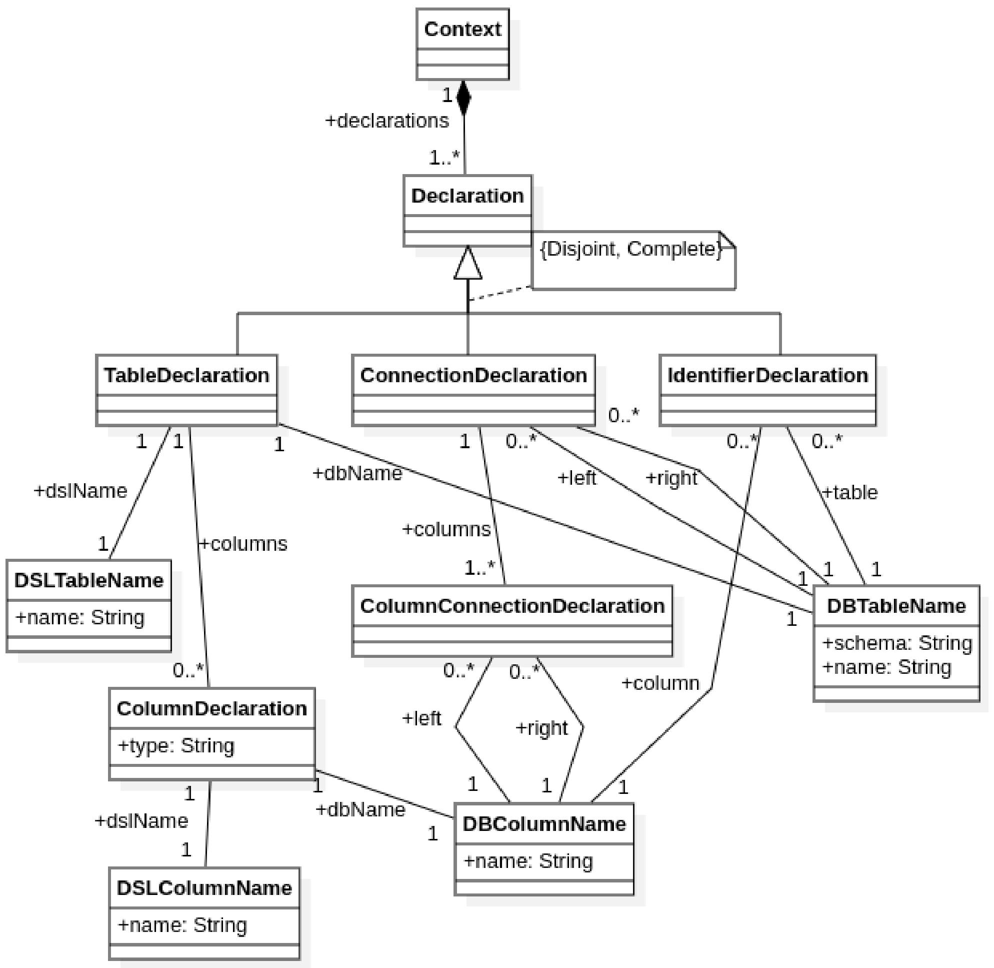


Figure 6. Abstract syntax of the context DSL.

A context is simply a collection of declarations that explain the structure of the database to the tool. There are three types of declarations:

- `TableDeclaration` - This declares a table and all its columns. Both columns and table names have a notion of "DSL" and "DB" names, where the former is used in the conceptual data model/contracts and the latter is used in the database. It is possible to supply only one name, in which case "DB" and "DSL" names are assumed to be equal. In this case the table/column in the database has the same name as the corresponding entity/attribute in the conceptual data model. Column definitions must also include their type. This is used to resolve parameter types. Type definitions must be simplified (e.g., use `VARCHAR` instead of `CHARACTER VARYING (50)`).

- `ConnectionDeclaration` - This maps to foreign keys in the database and relationships in the conceptual data model. "DB" names must be used for tables and columns.
- `IdentifierDeclaration` - This declares an "identifier" for a table. This somewhat maps to candidate keys in databases. However, it does not support defining composite keys. This limitation is discussed in section 7.2.1.

4.3.2 Abstract Syntax of the Contract DSL in UML

Figure 7 shows the abstract syntax of the contract DSL expressed by using a UML class diagram.

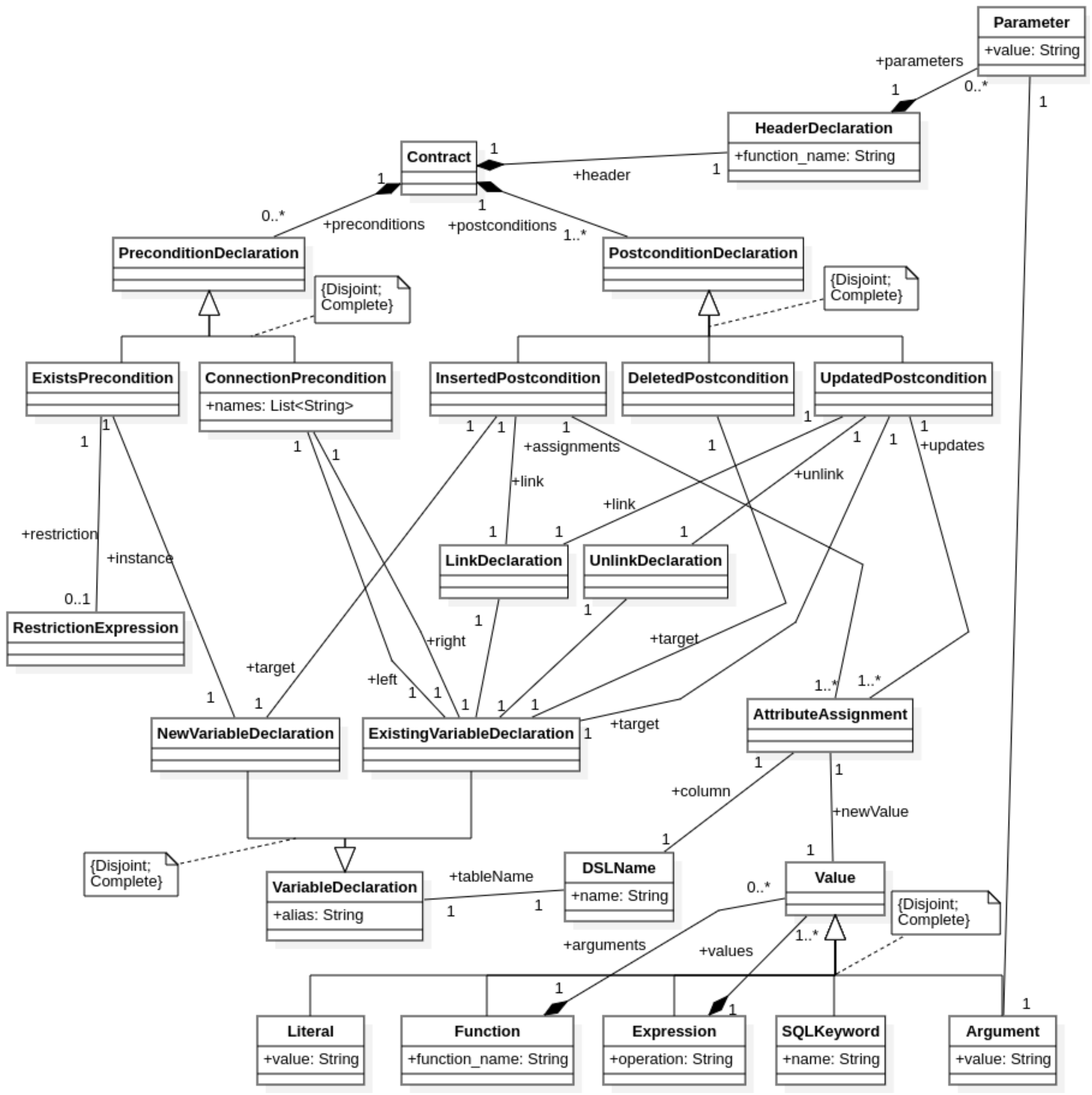


Figure 7. Abstract syntax of contract DSL.

A `Contract` is a set of its signature (`HeaderDeclaration`), pre- and postconditions. In total five types of conditions are supported - two types of preconditions and three types of postconditions:

- `ExistsPrecondition` - this precondition declares that an instance of an entity must exist. It can also be optionally restricted by specifying an expression (e.g., that the value of an attribute must be greater than some value). These instances are referred to as "variables" by the language as that is how they are treated.
- `ConnectionPrecondition` - this precondition can be used to denote that instances must be connected. Since it is possible for two entities to have multiple relationships (e.g., a product can have a worker who initially registered the product and a worker who has last modified the product values) the DSL allows us to specify names of connections if necessary.

- `DeletedPostcondition` - this postcondition declares that the targeted instance must be deleted. There is no need for any further information, so this simply takes a target variable.
- `InsertedPostcondition` - this postcondition specifies that a new instance must be created and unlike other postconditions it creates an instance, instead of using an existing one. It is possible to supply attribute values as literal values (such as `5`, `'a string'`, `5 + 6`), parameters, or target's attributes. This should also include specifying linkage with other instances (since in conceptual data model there are only relationships, there is no attribute to update directly).
- `Updated` - this postcondition allows specifying update of a targeted instance, and, as with `InsertedPostcondtion`, specification of new values. In addition to allowing specifying a linkage with another instance, update also needs to be able to specify unlinkage (i.e., setting to null).

For clarity, `VariableDeclaration` was split into `ExistingVariableDeclaration` and `NewVariableDeclaration`, as the names imply, `NewVariableDeclaration` must declare a new instance (i.e., new alias), and `ExistingVariableDeclaration` must refer to an already existing variable.

4.3.3 Syntax of the Context DSL in EBNF

Figure 8 shows EBNF rules for contexts.

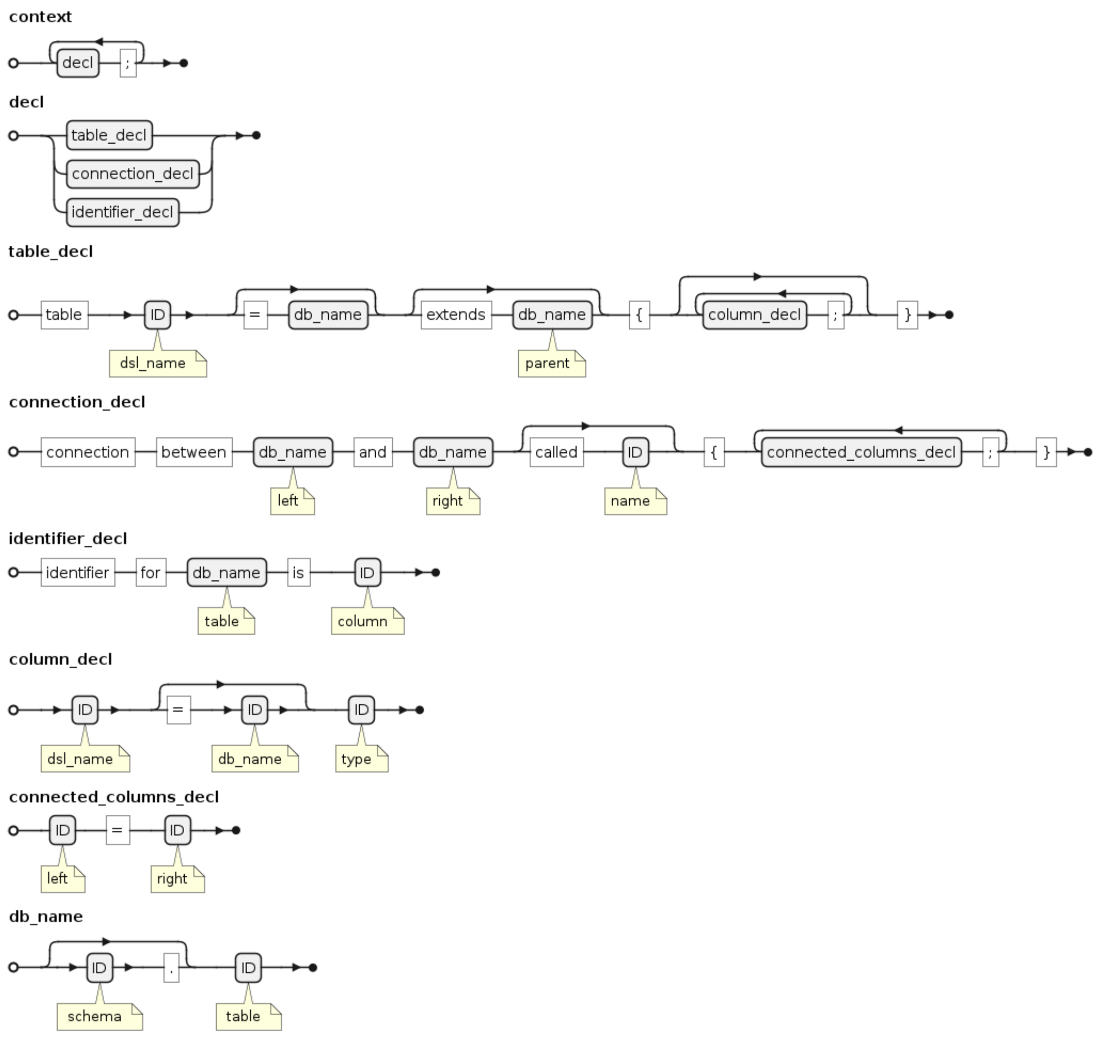


Figure 8. Higher-level EBNF notation for context.

As was shown in case of abstract syntax, a context file is a collection of (semicolon-separated) declarations (`decl`). In this notation hollow boxes denote literal values (e.g., `;` represents a literal semicolon), and grey ones refer to other rules. EBNF allows skipping specifying "DB" names of tables and columns, in this case "DSL" and "DB" names are assumed to be equal. Formal rule for `ID` is not included here as it is somewhat detailed - to summarise it is a "variable like" identifier (digits, UTF-8 letters, underscore (_)), that also allows us to use spaces if the identifier is wrapped in double quotes (`"`).

4.3.4 Syntax of the Contract DSL in EBNF

Since the EBNF for contracts is somewhat large, only relevant higher-level EBNF rules are expressed in figure 9.

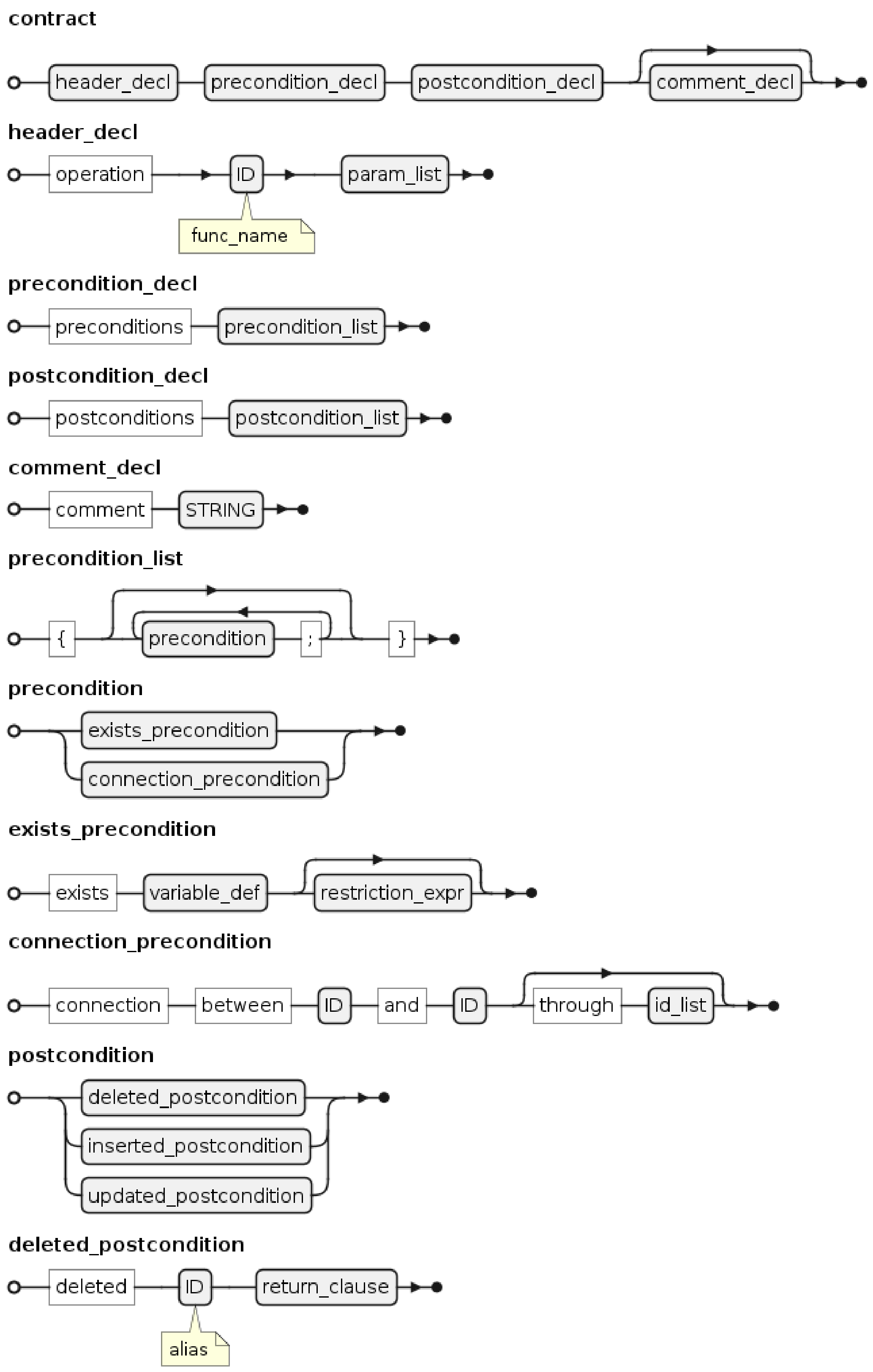


Figure 9. Higher-level EBNF notation for contracts.

`postcondition_decl` is very similar to `precondition_decl`, so it was not included.

4.3.5 Example with the New Language

Now that the syntax has been explained, it is possible to show how the same example contracts look in the new DSL. Firstly, as with free-form examples, figure 10 contains context for the operations. Note that it does not have to necessarily describe the entire database - it is enough to specify only the information that the implementations of operations will use.

```
table car
{
    car_code INTEGER;
    car_state_type_code SMALLINT;
};

table car_state_type
{
    car_state_type_code SMALLINT;
    name VARCHAR;
};

connection between car and car_state_type
{
    car_state_type_code = car_state_type_code;
};

identifier for car is car_code;
identifier for car_state_type is car_state_type_code;
```

Figure 10. Context for formal operations.

This context snippet defines two tables and connection between these that maps 1:1 to the conceptual data model that was given in case of the free-form contracts. Every column definition includes definition of a simplified type that will be used by the tool to resolve parameter types. For both tables an identifier is defined.

The operation for the deletion of a car looks like shown in figure 11.

```
operation "forget a car"
{
    p_car_code;
}
preconditions
{
    exists car c (car_code=p_car_code);
    exists car_state_type state (name = 'Waiting');
    connection between c and state;
}
postconditions
{
    deleted c;
}
comment 'OP2';
```

Figure 11. Sample operation: "Forget a car", DSL representation.

There are very few differences from the free-form definition. The first most obvious one is that the existence of a variable and its linkage to other variables had to be separated. Secondly, the DSL does not have a "delete with all links" operation, as all connections that must be deleted must either be explicitly defined or have the assumption that `ON DELETE CASCADE` takes care of that. This allows the implementer and the analyst to communicate in clear terms - if the analyst does not specify a precise deletion order/target list, then it is correct for the implementer to assume that `ON DELETE CASCADE` can be used. Moreover, the name of the operation is between quotation marks because the name contains spaces. Finally, the short identifier of the operation, which is used to refer to the operation from other analysis models, is presented as a comment.

The operation for the permanent inactivation of a car looks like shown in figure 12.

```
operation "permanently inactivate a car"
{
    p_car_code;
}
preconditions
{
    exists car_state_type cst_old
        (name in ('Active', 'Temporarily inactive'));
    exists car_state_type cst_new
        (name in ('Permanently inactive'));
    exists car c (car_code=p_car_code);
    connection between c and cst_old;
}
postconditions
{
    updated c
    {
        link with cst_new;
    };
}
comment 'OP5';
```

Figure 12. Sample operation: "Permanently inactivate a car", DSL representation.

Here, the only new kind of difference is that the contract does not have an `unlink from cst_old` clause as it is redundant - a `car` can have only a singular link to `car_state_type`. Thus, introducing a new link automatically overwrites the old one.

The context DSL allows us to specify different names for related conceptual data model entities and database tables. For instance, if the contracts are in Estonian but the names of tables/columns are in English, then one could use still use Estonian names in contracts, and the implementation would use the database names that are in English.

The context that provides translation between Estonian and English is shown in figure 13, contract in figure 14. Using different natural languages in the names of different database objects within the same database or even mixing languages within one database object is not a good practice. However, importantly, the same mechanism could be used in case the names of entities/attributes and tables/columns are in the same natural language but are different from each other (e.g., the names of entities contain spaces or letters õäöü but the names of tables do not contain such symbols).

```
table auto=car
{
    auto_kood = car_code INTEGER;
    auto_seisundi_liik_kood=car_state_type_code SMALLINT;
};

table auto_seisundi_liik=car_state_type
{
    auto_seisundi_liik_kood=car_state_type_code SMALLINT;
    nimi=name VARCHAR;
};

connection between car and car_state_type
{
    car_state_type_code = car_state_type_code;
};

identifier for car is car_code;
identifier for car_state_type is car_state_type_code;
```

Figure 13. Context for formal operations with a translation.

```
operation "lopeta auto"
{
    p_auto_kood;
}
preconditions
{
    exists auto_seisundi_liik asl_vana
        (nimi in ('Aktiivne', 'Mitteaktiivne'));
    exists auto_seisundi_liik asl_uus
        (nimi in ('Lõpetatud'));
    exists auto a (auto_kood=p_auto_kood);
    connection between a and asl_vana;
}
postconditions
{
    updated a
    {
        link with asl_uus;
    };
}
comment 'OP2';
```

Figure 14. Sample operation: "Forget a car", DSL representation with Estonian names.

5 A Tool for Generating SQL code Based on Contracts

This chapter explains the resulting tool starting from requirements. The description of requirements elaborates the listing of the goals presented in section 2.1. Next, it explains some of the most important aspects of design, implementation, and testing. Finally, installation and licensing is discussed.

5.1 Functional Requirements

Functional requirements explain the functionality of the system that it must provide in order to allow its users to fulfil their tasks and thus achieve their goals.

5.1.1 Generate Conforming Implementation

Actor: Analyst

Description: An actor has a contract and its relevant context expressed in DSLs. The actor provides these declarations to the tool and after selecting the desired options requests a conforming implementation. The tool processes the request and provides the user with implementation for the given contract in the given context.

5.1.2 Set Function Name Prefix

Actor: Analyst

Description: An actor has a prefix that he/she wants to see in the name of the implementing function. The actor supplies the prefix to the tool (without the following underscore) and the name of the generated function has the prefix. For instance, operation name is *add_car*, prefix is *f* and the name of the generated function is *f_add_car*. It is possible to not have a prefix in the name. In this case the field should be empty.

5.1.3 Set Parameter Name Prefix

Actor: Analyst

Description: An actor has a prefix that he/she wants to see in the names of parameters of the implementing function. The actor supplies the prefix to the tool (without the following underscore) and the tool generates a function where each parameter name has the prefix. For instance, operation has parameter *pin*, prefix is *p* and the name of the parameter of the generated function is *p_pin*. It is possible to not have a prefix in the name. In this case the field should be empty.

5.1.4 Choose Between Security Define and Invoker

Actor: Analyst

Description: An actor has a specific need that requires the generated function to be `SECURITY INVOKER`. This option determines that the function has to be executed with the privileges of the user who invokes it. Thus, in addition to the execution privilege, the user has to have all the privileges that are needed to conduct the operations in the function. The actor lets the tool know whether they need `SECURITY INVOKER` and the tool generates the function with this option. By default, the generated function is `SECURITY DEFINER` meaning that the function has to be executed with the privileges of the user who created it.

5.1.5 Use Schema Tables

Actor: Analyst

Description: An actor has tables that require specifying a schema. The actor specifies the schema in the "DB" name of the tables/columns in the context file and gives them a desired label. The tool uses tables from the specified schemas and in the generated function extends the search path accordingly.

5.1.6 Implicitly Return Identifier of Last Postcondition

Actor: Analyst, Implementer

Description: An actor wants the generated function to return the identifier value from a row that was inserted/updated/deleted based on the last postcondition in order to get feedback that the function execution achieved its expected results. The actor notifies the tool that it should generate return for the last postcondition implicitly and the tool treats the contract as if the last postcondition has a `return _identifier` clause. Otherwise, the tool treats the input as usual.

5.2 Non-functional Requirements

As was mentioned in section 2.1, there are also non-functional requirements to the expected properties of the DSLs and the tool:

- **The tool and the DSLs must be open source, and publicly available.** The first and immediate reason is that the implementation is a big part of this thesis. Because all research work must be reproducible, one would need to be able to see and test the author's implementation. Additionally, it is the belief of the author that any software tooling that people are to rely on must be "free software" as is defined by Free Software Foundation[45]. It means that the tool should wholly and unconditionally belong to the user, with full freedom to study, modify, and extend it as required. In addition, it should not be up to any third-party how the tool is used by the user.
- **The tool must be easy to extend.** In particular, it should be possible to extend the tool to support generation of code for other DBMSs, multiple syntaxes of the same DBMS, and even something completely different like the generation of ORM code.
- **The tool must be easy to use for non-programmers.** The tool aims to formalise an aspect of system design which is typically done by analysts who do not necessarily have to be programmers. The tool therefore should make the assumption that the user may not have programming knowledge, or, at extreme, the user has little technical knowledge (e.g., some domain expert that does not work in IT). To this end the tool must provide a GUI and the installation must be as simple as possible too.
- **The tool must support at least generating code to PostgreSQL.** In order to have some code to generate and implementations to check, the tool must support at least one target platform. PostgreSQL was chosen as it is free and open source as well. The tool should generate PostgreSQL functions in SQL language. Thus, it does not have to generate procedures (supported in PostgreSQL from version 11) and the generated function is in a declarative language rather than a procedural language like PL/pgSQL. In case of typical use cases of the generated functions the use of PL/pgSQL will not give anything useful and only increases complexity and decrease performance. The generated SQL functions must have a function body that conforms to the SQL standard and is more easily portable to other DBMSs. Support of such functions was added to PostgreSQL version 14. This new syntax allows PostgreSQL to track dependencies of the function. [46]

5.3 Iteration Planning

In total there were seven iterations (+1 pre-iteration) required for the implementation. Below is a list of all of these, as well as the high-level milestones they achieved.

1. Pre-iteration
   (a) The author parsed through the sample data provided by the supervisor.
   (b) Draft of contracts in future DSL was created.
2. Iteration 1
   (a) Overall project and testing structure was established.
   (b) Basic internal model was created.
   (c) Generation of simple delete statements was implemented.
   (d) Parsing of context from the database was implemented.
   (e) Parsing of DSL for current model was implemented.
3. Iteration 2
   (a) Generation of `deleted` postconditions was extended to support more complicated cases (e.g., when there are dependencies and restrictions).
   (b) Support for relationships between entities was added.
4. Iteration 3
   (a) Support for context files and renaming of entities/tables was added.
   (b) Restrictions were extended into proper trees (thus, it is possible to nest these).
   (c) More formal model-to-model translation was implemented (before that generators simply parsed the input object, without really translating it to internal model).
   (d) Sample data operation 8 was added to the test suite.
5. Iteration 4
   (a) Handling of depending tables was decided - `ON DELETE CASCADE` is assumed, or the contract must specify everything that must be deleted in correct order.
   (b) Parameter resolution code was added. Option to specify parameter types was removed because precise parameter types are an "implementation detail".
   (c) Support for various literals was added: decimals (e.g., `5.3`), `nulls` (as well as comparisons like `is null` and `is not null`), explicit `in` and `not in`, positive and negative numbers, proper strings (before that could only have ASCII symbols without spaces), and function invocations.
   (d) Sample data operations 1 and 2 were added to the test suite.
   (e) Support for simple `inserted` postconditions was added.
6. Iteration 5
   (a) Support for `inserted` was extended.

    (b) Insert allows linking to other entities.
    (c) Sample data operation 7 was added to the test suite.
    (d) Support for simple `updated` postconditions was added.
7. Iteration 6
    (a) Support for `updated` was extended.
    (b) Sample data operations 3, 4, 5, and 6 were added to the test suite.
    (c) GUI was added.
    (d) Initial packaging and first testable delivery was done.
8. Iteration 7
    (a) `updated` allows unlinking from entities (i.e., setting table's column that is used in a relationship to `null`).
    (b) Support for returning values, including special keyword for identifier, was added.
    (c) Support for "unrelated" preconditions (before that postconditions checked preconditions that were somehow related to them).
    (d) Framework for handling errors was added. It is very rudimentary, just throws an exception that can be caught in case there is a problem.
    (e) Tables in context file can extend other tables, allowing for a very simple inheritance support.
    (f) Simple CLI was added.
    (g) Support of less strict identifier names was added. Thus, it is possible to have operation like "spaced name".
    (h) Code reorganisation was done.

During the review of the last iteration it was agreed that no more iterations will be held so that the author can concentrate on writing this document. A few more problems were identified, and fixed, but outside the "iterative" approach.

5.4 Program Design

This section explores the design of the tool.

5.4.1 Implementation Language

The author had a choice between three programming languages that he could implement the tool in - `C++`, `Java`, and `Python`.

Python was immediately discarded due to it being dynamically typed and unsuitable for long-term tooling. In particular, author's experience with Python is such, that it seems like it is prone to introduce changes that may not be compatible between versions, libraries written in it may not work on latest versions of the language, and libraries may introduce changes that break their users. The problem with dynamic typing is that type-related errors that are caught by compilers in statically-typed languages become runtime errors in dynamically typed languages. It is therefore possible to ship a tool that will raise an exception due to having a wrong type somewhere, especially when library or language versions are mismatched. Since the author knows and feels confident in statically-typed Java as well, there was no clear benefit of using Python over Java.

C++ seemed like a very good choice for this project - it compiles into a standalone executable that would be very easy for end users to use. Moreover, it is also quite fast. The main reason against C++ was that multiple systems had to be supported. The author exclusively uses Linux systems. Therefore, there would be no easy way to check that the code compiles and works on Windows as well. As the author has no experience in cross-compiling or supporting multiple platforms, this would have been a struggle of its own that is not present in case of using Java.

Thus, only Java remained as a candidate and was chosen as the language for implementation. A problem with Java is that it somewhat goes against the ease of installation by requiring a java runtime. This was solved by packaging a Java runtime along with the tool and providing simple scripts that run the tool with that runtime. In order to avoid dealing with dependencies the tool is packaged as a so-called "fat jar", meaning that it includes all the dependencies in itself, rather than requiring them to be present in the classpath. A downside is that the unpacked folder of the tool is about 660 MB in size. At first, java 17 was chosen as it is the latest long-term support version available at the time of writing. The author ran into some problems related to UTF-8 support on Windows. Thus, the version was changed to 18 as it is the first version to introduce UTF-8 as the default encoding.

5.4.2 Dependencies

In total, there are five libraries that the tool depends on - `ANTLR4`, `StringTemplate4`, `javafx`, `JCommander`, and `PostgreSQL JDBC Driver`.

`ANTLR4` is a library that allows generation of DSL parsers from their EBNF notation. In this way it was possible to just write the ENBF notation for the language and get information from it by writing callbacks to each node.

`StringTemplate4` is used internally by `ANTLR4`. As there was a need for string formatting, it was chosen to not introduce extra dependencies and use libraries that the tool already has to use.

`javafx` is a GUI library that is provided by maintainers of Java. Because the author has little experience or preference in Java GUI libraries it seemed like a good choice.

`JCommander` is a CLI argument parser that eases the creation of the command line interface for the tool. The author is familiar with it from his place of employment. Thus, it was chosen out of familiarity.

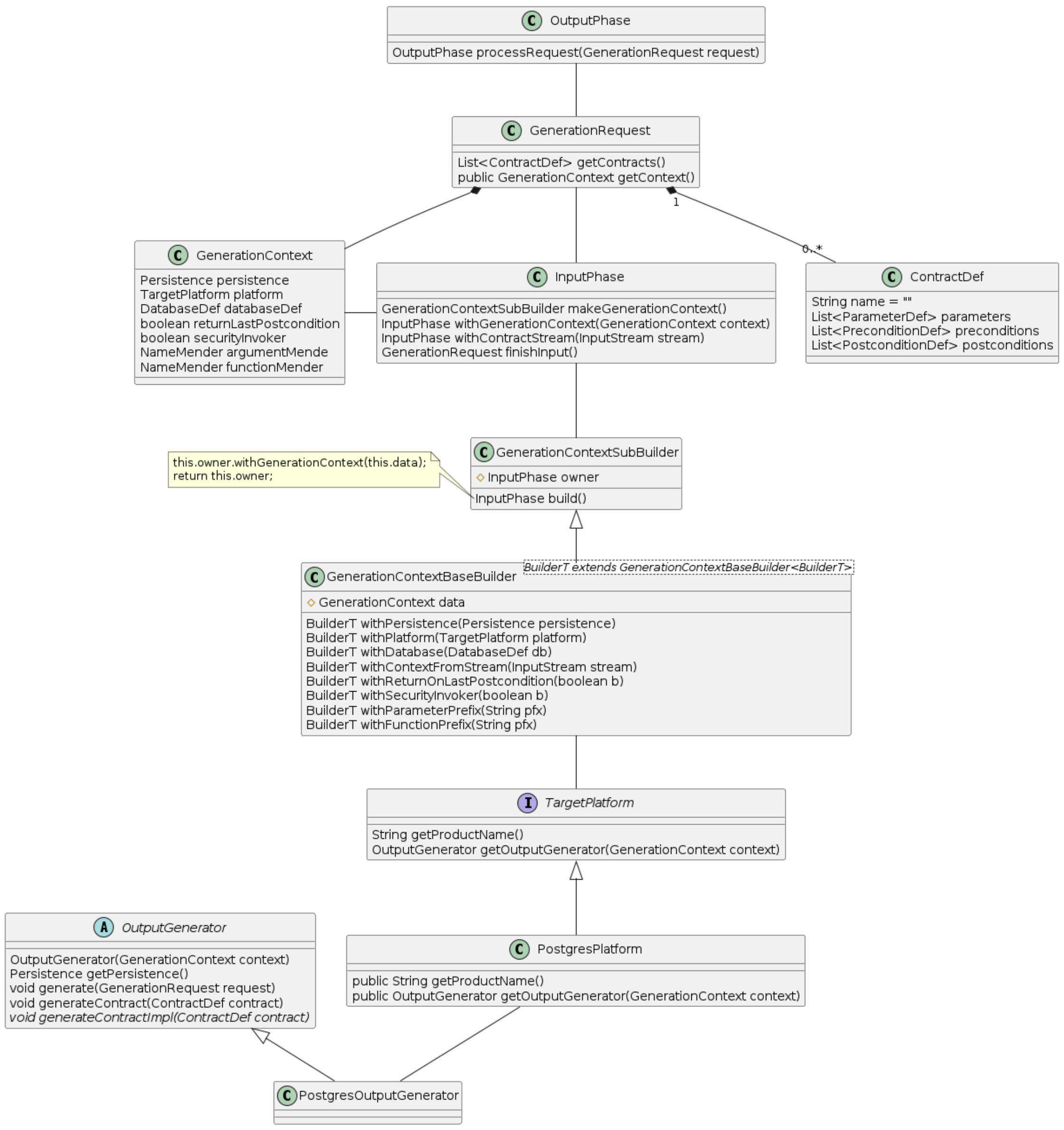
`PostgreSQL JDBC Driver` is a driver for PostgreSQL DBMS that is compliant with java's standard Java Database Connectivity (JDBC) API. It is used primarily for testing in case of this work. However, its use could be extended as proposed in 7.2.3.

5.4.3 Architecture

The whole project is about 300 java files and ~28k lines of code. Thus, there is no way to talk about all its details. It is split into five modules:

- `core` - The main core of the tool that defines the API as well as the request model. Extenders (i.e., generators) are given an interface to extend.
- `process/postgres` - PostgreSQL generator.
- `iface/gui` - Graphical User Interface.
- `iface/cli` - Command Line Interface.
- `package` - Release package that gathers all artifacts into a single zip, which can be given to an end user.

Below is a figure representing a rough high-level architecture view of the core interface:



OutputPhase
OutputPhase processRequest(GenerationRequest request)
GenerationRequest
List<ContractDef> getContracts()
public GenerationContext getContext()
1
0..*
GenerationContext
Persistence persistence
TargetPlatform platform
DatabaseDef databaseDef
boolean returnLastPostcondition
boolean securityInvoker
NameMender argumentMende
NameMender functionMender
InputPhase
GenerationContextSubBuilder makeGenerationContext()
InputPhase withGenerationContext(GenerationContext context)
InputPhase withContractStream(InputStream stream)
GenerationRequest finishInput()
ContractDef
String name = ""
List<ParameterDef> parameters
List<PreconditionDef> preconditions
List<PostconditionDef> postconditions
this.owner.withGenerationContext(this.data);
return this.owner;
GenerationContextSubBuilder
InputPhase owner
InputPhase build()
GenerationContextBaseBuilder
BuilderT extends GenerationContextBaseBuilder<BuilderT>
GenerationContext data
BuilderT withPersistence(Persistence persistence)
BuilderT withPlatform(TargetPlatform platform)
BuilderT withDatabase(DatabaseDef db)
BuilderT withContextFromStream(InputStream stream)
BuilderT withReturnOnLastPostcondition(boolean b)
BuilderT withSecurityInvoker(boolean b)
BuilderT withParameterPrefix(String pfx)
BuilderT withFunctionPrefix(String pfx)
TargetPlatform
String getProductName()
OutputGenerator getOutputGenerator(GenerationContext context)
OutputGenerator
OutputGenerator(GenerationContext context)
Persistence getPersistence()
void generate(GenerationRequest request)
void generateContract(ContractDef contract)
void generateContractImpl(ContractDef contract)
PostgresPlatform
public String getProductName()
public OutputGenerator getOutputGenerator(GenerationContext context)
PostgresOutputGenerator

Figure 15. High-level core API.

There are two highest level classes that are intended for users of the `core` package - `InputPhase` and `OutputPhase`. At the end of `InputPhase`, one gets a `GenerationRequest` that, when supplied to `OutputPhase` produces SQL code into `Persistence` class, according to the `TargetPlatform` chosen (note: `Postgres-OutputGenerator` is from the `process/postgres` module and is added for clarity, it is *not* a part of the `core`). There is, admittedly, little modification to be done on the `GenerationRequest` between the two phases, and they could be united into a single class. However, the author decided that it is best to split these for clarity. Both phase classes, and the project in general, make heavy use of the *Builder Pattern*. This allows for very easy configuration by using code auto completion tools - one simply makes an instance of the class and consults the methods. As there are many places where the builders are used (e.g., directly, or as sub builders for other builders), the main configuration is typically offloaded into an `XBaseBuilder` generic class that uses the *Curiously Recurring Template Pattern* so that it can return to the precise derivative builder, rather than the base builder. `GenerationRequest` is split into two main data points - a list of contracts and a `GenerationContext`. The latter is used to collect all information that does not apply to contracts, such as context, various generation options, and the output method. Output is done with a very loose contract - one has to derive `OutputGenerator`, implement the only abstract method that transforms given contract into a target representation, and write it into persistence. How that transformation is exactly done is not enforced in any capacity - one can make an internal model and translate to it, or simply parse through the input.

5.4.4 PostgreSQL Generation

PostgreSQL generation was offloaded into its own module as it is one of the requirements for this tool to be extensible.

The generator does a formal model-to-model translation (from `GenerationRequest`'s list of contracts to a list of `Function` classes) that it represents using a `String-Template4` library, below is a high level overview:

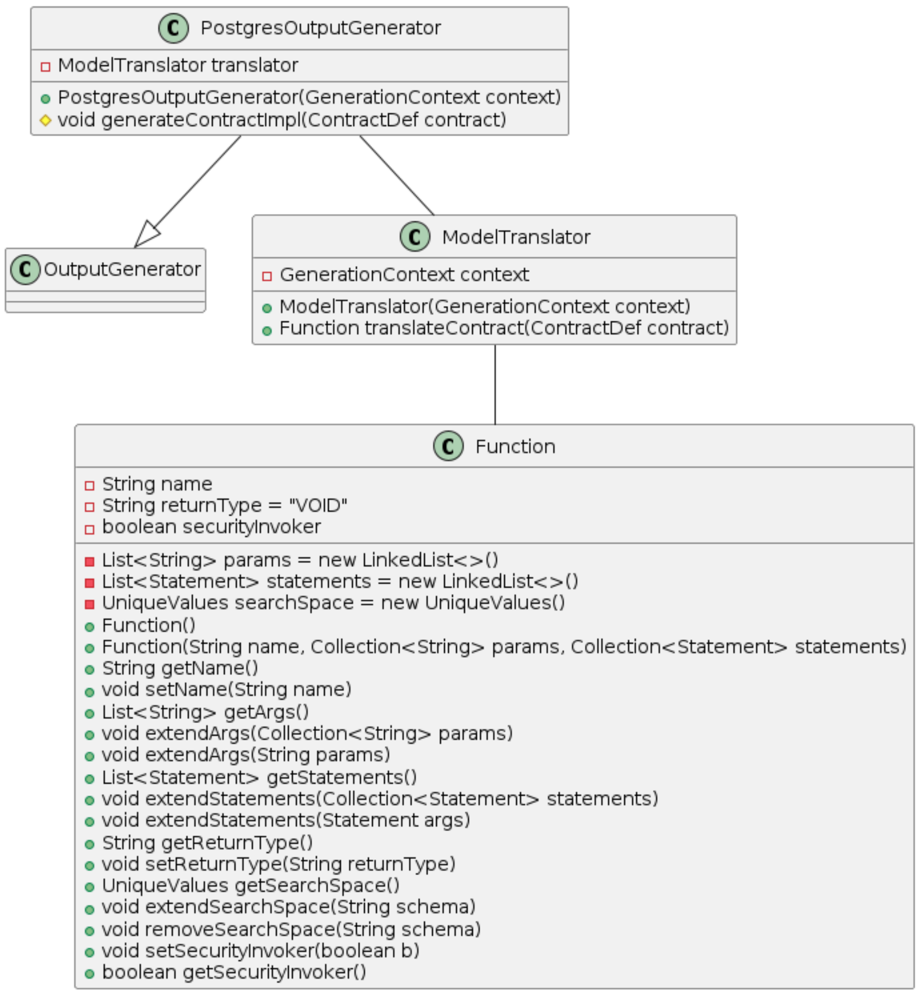

PostgresOutputGenerator
ModelTranslator translator
PostgresOutputGenerator(GenerationContext context)
void generateContractImpl(ContractDef contract)
OutputGenerator
ModelTranslator
GenerationContext context
ModelTranslator(GenerationContext context)
Function translateContract(ContractDef contract)
Function
String name
String returnType = "VOID"
boolean securityInvoker
List<String> params = new LinkedList<>()
List<Statement> statements = new LinkedList<>()
UniqueValues searchSpace = new UniqueValues()
Function()
Function(String name, Collection<String> params, Collection<Statement> statements)
String getName()
void setName(String name)
List<String> getArgs()
void extendArgs(Collection<String> params)
void extendArgs(String params)
List<Statement> getStatements()
void extendStatements(Collection<Statement> statements)
void extendStatements(Statement args)
String getReturnType()
void setReturnType(String returnType)
UniqueValues getSearchSpace()
void extendSearchSpace(String schema)
void removeSearchSpace(String schema)
void setSecurityInvoker(boolean b)
boolean getSecurityInvoker()

Figure 16. High-level view of PostgreSQL generation.

5.4.5 Extension

Admittedly, the extension of this tool is not as simple as it could be. Ideally, all the rules on how to translate analysis-level contracts into a SQL function would be inside the core. However, they are not. Instead, they are a part of PostgreSQL generator module. This was done so as it is otherwise very hard to tell what parts are generic and what are PostgreSQL-specific. Additionally, it is not clear how should these rules translate to other models that are not based on SQL statements. Therefore, the author decided that instead of making any assumptions which may or may not hold, he would concentrate on keeping PostgreSQL-specifics outside the `core`. This way, if one wants to implement another generator, then it is possible to simply copy the entire PostgreSQL processing module and modify it for the platform in question, *without* having any effect on other generators. Afterwards, when there are working implementations for different DBMSs it will be much easier to see which parts are the same and unify these into the `core`. Additionally, one is not *required* to use the provided PostgreSQL generating module - `core`'s API has two interfaces that must be implemented. Hence, one could simply implement these however they see fit. This is precisely the approach one would take in case one wants to implement something that cannot be based on the PostgreSQL module, like ORM code or support for DBMSs with another data model (e.g., document or property graph) It should also be noted that all tests are currently included in the PostgreSQL module as well (as this is precisely the module that generates a testable implementation). Thus, these would have to be reintroduced, and if copied, adapted to the new module.

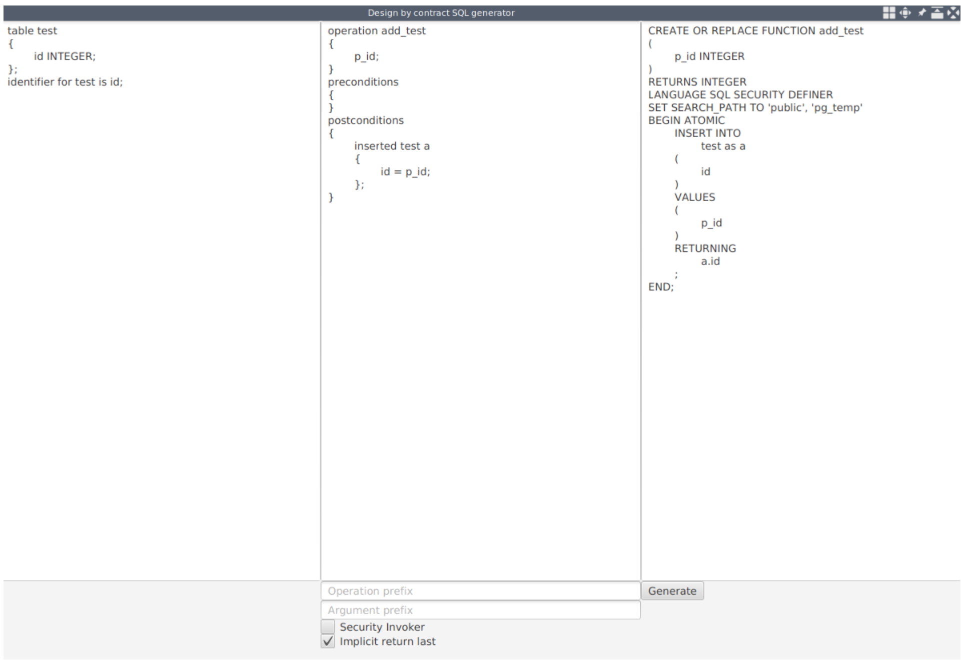
5.4.6 Testing

As with any software, this tool is only as good as its tests. To this end, testing has been a major concern throughout the whole implementation. The author used a very rigorous test driven development approach, where no feature would be written without the test first. As such, before any feature would be implemented, its test would be written first, and only then would the feature be implemented. Most of the tests are very high level, testing the whole system integration rather than single units of code (i.e., classes). This was done purposefully, as at the end of the day, what matters the most is that *the whole system* works as intended rather than its parts.

While the author makes no claims that these tests cover all possible expected usages and guard against all bugs, these tests represent a very strong base for acceptance testing. Additionally, according to the code coverage tool jacoco, they achieve 97% instruction and 88% branch coverage. The tests are available at `https://github.com/ghpv/dbcsql/tree/master/process/postgres/src/test/java/ee/taltech/dbcsql/uc` (this does not include tests based on sample data, only ones the author wrote himself). Other implementations should make use of this test suite and adapt these. However, as there is only one module currently, there is currently no framework in place to accommodate for that. As with implementation of the module - it will be much easier to understand how testing should be shared between modules when there are more modules available.

5.5 Graphical User Interface

To make the tool as easy to use as possible it comes with a GUI shown in figure 17.



```
table test
{
	id INTEGER;
};
identifier for test is id;
```

```
operation add_test
{
	p_id;
}
preconditions
{
}
postconditions
{
	inserted test a
	{
		id = p_id;
	};
}
```

```
CREATE OR REPLACE FUNCTION add_test
(
	p_id INTEGER
)
RETURNS INTEGER
LANGUAGE SQL SECURITY DEFINER
SET SEARCH_PATH TO 'public', 'pg_temp'
BEGIN ATOMIC
	INSERT INTO
		test as a
	(
		id
	)
	VALUES
	(
		p_id
	)
	RETURNING
		a.id
	;
END;
```

☐ Security Invoker
☑ Implicit return last

Figure 17. GUI.

It is very simple. It has three big boxes that take majority of the screen space. These are: context input, contract input, and generated output. Below that there are a few options and a button to initiate generation. Currently, as only PostgreSQL is supported there is no option to select other platforms. Context and contract input have a simple example provided by default. When no input is supplied text boxes show hints as to what input they expect.

5.6 Installation

The tool is packaged as a standalone portable package. This means that a user is not required to formally install it on their system. It is enough to download a release, unzip it, and run the corresponding `run` script to get into tool's GUI. CLI (command line interface) with the same functionality is also available. Because it is only useful to more advanced users, it is expected that they understand how to run it. It displays usage instructions if ran without parameters, so more precise instructions are available there.

5.7 Licensing and Distribution

The tool (and by extension the DSLs) are licensed under General Public License (GPL), version 3. As the author already mentioned, it is his belief that software tools that people are to rely on should be free software. Thus, the author does *not* want to use a more permissive license that would allow third parties to fork the tool, extend it, and make it into a commercial product. The *output* of any software (i.e., the generated SQL in this case) is in general *not* considered a part of licensed work, and as such, is free to be used in proprietary systems.

6 Validation of Results

In this chapter the author reviews the work. Firstly, the generated implementation is reviewed - the author demonstrates the code that the tool generates based on the examples given in section 4.3.5 and discusses various points of interest in it. In addition, the author measures the time performance of the tool. Finally, the author compares the results against DSL best practices and other tools.

6.1 Expressing Contracts of Course Projects

As the author had access to example projects, one of these was selected and completely implemented by using this tool. The tool's output was checked by both the author and the supervisor of this work to check that it is all correct. Furthermore, these adapted contracts are in fact in the testing suite of the tool. Thus, it is always tracked that the tool is indeed able to generate implementations of the contracts. As the contracts in the projects the author had access to are very similar it could be said with some certainty that the tool supports generating contracts as required by the database course. As there are examples of formal contracts given in section 4.3.5 their implementation is discussed here as well.

The first example contract is about forgetting, i.e., deleting a car if it is in an applicable state. The implementation that the tool generates is shown in figure 18 (lines wrapped manually to fit into page).

```
CREATE OR REPLACE FUNCTION forget_a_car
(
    p_car_code INTEGER
)
RETURNS INTEGER
LANGUAGE SQL SECURITY DEFINER
SET SEARCH_PATH TO 'public', 'pg_temp'
BEGIN ATOMIC
    DELETE FROM
        car AS c
    USING
        car_state_type AS state
    WHERE
        (
            c.car_code = p_car_code
            AND state.name = 'Waiting'
            AND c.car_state_type_code
                = state.car_state_type_code
        )
    RETURNING
        c.car_code
    ;
END;
COMMENT ON FUNCTION forget_a_car
(
    p_car_code INTEGER
)
IS 'OP2';
```

Figure 18. Generated implementation of the operation "Forget a car".

The contract conforms to the implementation very literally - as the state is specified in the contract by using a textual identifier, so too does the function make use of the textual identifier. Since the instances of the state and the car must be connected, the tool resolves their connection columns and links these. This implementation was generated with the implicit return option. Therefore, the function also has a `RETURNING` clause, despite it not being requested in the contract. Also, the parameter type is resolved by the tool by using the connection `car_id = p_car_id` from the contract.

The second example was about permanently inactivating a car if it is in an applicable state. The implementation that the tool generates is shown in figure 19 (lines wrapped to fit into page):

```
CREATE OR REPLACE FUNCTION
    permanently_inactivate_a_car
(
    p_car_code INTEGER
)
RETURNS INTEGER
LANGUAGE SQL SECURITY DEFINER
SET SEARCH_PATH TO 'public', 'pg_temp'
BEGIN ATOMIC
    UPDATE
        car AS c
    SET
        car_state_type_code
            = cst_new.car_state_type_code
    FROM
        car_state_type AS cst_old,
        car_state_type AS cst_new
    WHERE
        (
            cst_old.name in
                ('Active', 'Temporarily inactive')
            AND cst_new.name in
                ('Permanently inactive')
            AND c.car_code =
                p_car_code
            AND c.car_state_type_code =
                cst_old.car_state_type_code
        )
    RETURNING
        c.car_code
    ;
END;
COMMENT ON FUNCTION permanently_inactivate_a_car
(
    p_car_code INTEGER
)
IS 'OP5';
```

Figure 19. Generated implementation of the operation "Permanently inactivate a car".

Similarly to the previous case, the implementing function uses textual identifiers of states as those are precisely the ones requested. Since there are two states (before and after the update), the table is queried twice, using the aliases (cst_old, cst_new) provided by the contract.

Finally, the output for the translated case is shown in figure 20.

```
CREATE OR REPLACE FUNCTION lopeta_auto
(
    p_auto_kood INTEGER
)
RETURNS INTEGER
LANGUAGE SQL SECURITY DEFINER
SET SEARCH_PATH TO 'public', 'pg_temp'
BEGIN ATOMIC
    UPDATE
        car AS a
    SET
        car_state_type_code
            = asl_uus.car_state_type_code
    FROM
        car_state_type AS asl_vana,
        car_state_type AS asl_uus
    WHERE
        (
            asl_vana.name
                in ('Aktiivne', 'Mitteaktiivne')
            AND asl_uus.name in ('Lõpetatud')
            AND a.car_code = p_auto_kood
            AND a.car_state_type_code
                = asl_vana.car_state_type_code
        )
    RETURNING
        a.car_code
    ;
END;
COMMENT ON FUNCTION lopeta_auto
(
    p_auto_kood INTEGER
)
IS 'OP2';
```

Figure 20. Generated implementation of the operation "Permanently inactivate a car" in case of the translated context.

6.2 Expressing a More Challenging Database

As a challenge, the client of the work (i.e. the supervisor) presented the author with a database schema that he came up with, and five contracts in free-form notation that express desired operations on this schema. The challenge was to see if it is possible to express these contracts in the new DSL and generate working code based on it given these contracts. The first version of the challenge the author was unable to fulfil because it referenced other instances during inserts and updates, as well as during restrictions, which was never discussed during implementation. The supervisor adjusted the requirement, and the second version is possible to express. The full context is shown in appendix 2 and its DSL representation in appendix 11, formal contracts of operations (with the free-form conditions in comments) are shown in appendices 3, 5, 7, 9, 12 and their respective implementations in 4, 6, 8, 10 and 13.

The author checked the generated code by running it in a PostgreSQL instance. The author executed some tests against it. All the produced code is valid SQL. For the operations 1, 2, 3, and 5 everything works as one would expect. The operation 2 had a small caveat in that the author had to reorder the postconditions so that the update happens first and delete second. This is because otherwise, the code does not have enough values to check against and results in more updates than expected.

The operation 4, however, while expressible, does not result in an implementation that works as expected. There are multiple problems that the author found. Firstly, the operation has a notion of "optional" update, where it is desired that all orders that *might* be connected to the product in question get updated. This does not work with the code generator because it must check all the preconditions. Therefore, it does not allow for a precondition to *optionally* hold. In the way the author has translated the contact a product *must* be connected to some orders before it can be deleted. This is discussed further in section 7.2.12. The second problem that the author found is about the comment field - the contract intends to change the comment to a value that depends on the old one by appending it to the already existing comment. This is supported, but all the postconditions afterwards would also have to check for this comment, meaning that the tool would somehow have to calculate the new comment. This is documented in section 7.2.11. In this specific contract, one could work around this by supplying the postcondition last, so that it is not necessary to check its fulfilment later. Finally, this operation also ran into a bug where the tool does not keep updates correctly and the implementation of the last postcondition is trying to check that the product is in both "Active" and "Dropped" states at the same time. This is documented in 7.2.10.

The operation 5 also demonstrates that the tool is able to treat "unrelated" entries correctly. That is, the entity `is_open` is not associated with the entity `product_event`. However, in the generated code the table `is_open` is still checked correctly and the generated code only works if the value is indeed `TRUE` in the `is_open` table.

6.3 Performance Measurements

The author ran all samples included with the tool on an `AMD Ryzen 5 3600X` processor with `256 MB` of RAM using a simple shell for-loop to iterate over contracts and feed these into the CLI version. The total run time of the tool for eight included contracts is about three seconds. Considering that these samples constitute all analysis-level contracts for one course project, this is a very reasonable result. This metric can be improved with some effort by allowing the tool to parse multiple contracts from a single file, as currently, it has to parse through the same context for every single contract.

6.4 Conformance of the New DSL to the Best Practices of DSL Design

The tool uses a *Source-to-source transformation* where contracts expressed in the DSL are translated into the desired SQL implementation. It could have been done with *Lexical processing*, i.e., where the language would have been defined in such a way that it is manually parseable without a syntax tree. However, to save on implementation effort the author found and used a library that allowed for an easy way to use a syntax tree. There is a high-level API that allows interfacing with the core from user interfaces. This is illustrated by having both GUI and CLI available, with very little code in these that actually deals with SQL generation. Most of the code in the user interface modules actually deals with the user interface related activities like taking input and presenting output. The DSL ignores most of the whitespace (except where it is necessary like in strings), allowing the code to be formatted in whatever way desired - it can be formatted like the author did in this work, a more "java-like" style can be used, or it can be expressed in a single line too (though comments might have to be adjusted to use a different notation). There is no support for importing other files, as this work treats the tool and the DSL as more of a prototype, showing that it is possible to have such transformations. Comments are allowed within the DSL. It supports comment clause that allows specifying a text that should be added to the database with the COMMENT statement and comments that are ignored by the tool. The DSL has precisely enough keywords to express everything required, and it does not allow expression of anything that does not somehow impact the generated code (except for comments that are ignored by the parser, of course).

6.5 Comparison with the Existing Tools

As was mentioned in section 3.6 the author searched tools that take analysis-level database contracts as an input and generate their implementation as the output, and was unable to find any. Therefore, to his knowledge, this is the only tool (as of spring 2023) that can take a contract of database operation and generate a conforming implementation based on it. All the other tools discussed in section 3.7 support only generating some very specific templates that can only be used as a *base* for the implementation. With this tool, it is possible to get a working implementation for a contract without having to write any SQL code manually.

7 Discussion

In this chapter the author discusses answers to the posed research questions and outlines some possible ways to further develop the produced tool.

7.1 Answers to the Research Questions

Section 2.2 posed three questions, that the author will now answer.

1. Is it possible to express analysis-level database operations by using contracts?
   (a) Yes, it is indeed possible to express simple database Create-Update-Delete operations with the help of contracts by using both formal and informal notation.
2. Is it possible to express analysis-level contracts of database operations in a form that allows code generation based on the contracts?
   (a) Yes, by using the formal notation of this tool, it is possible to have conforming code generated from these contracts. It is, however, currently necessary to also manually provide the context for these, but it is not much more than a conceptual data model would be anyway - the only new information is the mapping between conceptual data model elements and database elements.
3. Does the description of contracts in a more formalised form and generation of database routines based on these improve the development process compared with writing contracts less formally and creating database routines manually?
   (a) The formal syntax very closely resembles the informal one. Because it helps us to automatically generate a conforming implementation without a room for human errors, the author would argue that yes, this improves the development process.
   (b) Moreover, once the conceptual data model (and its DSL representation as well) is more stabilised, this tool will allow users to rapidly test new operations without having to incur development time.

7.2 Limitations of the Work and Further Development

In this section the author lists all the known limitations that the implemented solution has, as well as proposes areas for further improvements.

7.2.1 Composite Identifier

The syntax of the context DSL allows specifying only simple identifiers, i.e., identifiers that involve one column. The author is fully aware that this is not true in neither conceptual data models (as it is possible for an entity to have an identifier that involves more than one attribute), nor in databases (as it is possible for a table to have a composite key, which involves more than one column). If composite identifiers were to be supported, then the tool would have to support generating routines that return multiple values (i.e., `RETURNS TABLE` functions), which would require extra implementation effort. Since the intended use case for returns by functions is to provide information to the invoker whether the operation succeeded (i.e., that the function returned *at all*), it was deemed that having only simple return types is acceptable. Hence, not supporting multi-value returns (and therefore also composite identifiers) is an acceptable sacrifice. While it would be trivial to have at least the DSL support specifying composite identifiers for an entity (and, for example, only the value of one attribute is returned), it was decided that the tool should be explicit about this lack of support and not allow this declaration at all. If the only use case does not support using multiple values, then the tool should not pretend that it needs (or understands what to do with) all the values.

The support for composite identifiers (and multi-valued return) could be added without big refactoring. It could be considered as a future feature.

7.2.2 Context from UML Analysis Models

As it stands, one of the main roadblocks that the current solution has from analysis point of view is that UML-based conceptual data models must be manually translated into a standalone context DSL. Furthermore, this must be done for every target platform. Similarly, one would need to create SQL DDL statements (e.g., `CREATE TABLE`) that create the context database in which the operations can be created and tested, though there are a lot of tools (such as UML CASE tools discussed in 3.7) that allow automatic conversion. This is a lot of, frankly needless, manual work that can, and should, be automated.

Most of the context DSL maps directly to what a conceptual data model expresses - `tables` along with their columns map almost directly to entities and attributes (at least "DSL" names), `connections between` map directly to relationships, and `identifier` could have some special stereotype that denotes an identifier attribute. In fact, sample projects already tend to use «`id`» stereotype for that. As was discussed in section 4.2.2, the main hurdle for this feature is that a mapping to database elements must be produced. In particular, the tool has to know the following information that is not explicitly defined in the conceptual data model:

- How conceptual entities map to database tables.
- How attributes of conceptual entities map to columns of database tables.
- DBMS supported data type that each parameter must have (i.e., not `string`, but rather `CHAR`, `VARCHAR`, or `TEXT`).

The easiest way to solve the problem is to extend the tool so that it also generates code (DDL statements) for creating all the database objects, rather than only the routines. In this way, since the tool generates the entire database definition it knows exactly the mapping between entities/attributes/relationships and database objects (i.e, tables/columns/constraints). Similarly, the last point would be solved by enforcing a strict enumeration of supported types and introducing a mapping of what they are to be translated into. This can be achieved in multiple ways. If one wants to reuse existing work, then one could try to integrate the translator with already existing tools, though, of course, this would limit the application to that specific tool. Alternatively, one could create their own generation engine that generates statements as required. In this way, the tool would be able to transform analysis-level models into a fully functional database. Moreover, one could even allow specifying a target database that the tool can connect to and execute all the generated code directly in the database, creating a fully testable database from analysis-level models (conceptual data model and contracts of database operations).

7.2.3 More Database Integration

The tool already has code for working with PostgreSQL, and this could further be extended to be a part of a typical workflow. For example, one could extend the tool to allow supplying a database URI (uniform resource identifier), so that the generated operations would be automatically created there. Similarly, there is a way to parse the database for the context. If it uses the same names as are used in a conceptual data model, then it would be possible to get a fully qualified context, rendering the need for context description unnecessary. In fact, this is how most of the tests are written, so most of that functionality exists already.

7.2.4 More Lenient Name Requirements

Currently, the tool forbids the usage of SQL keywords as parameter names, even though it could work around it by, e.g., prepending a prefix to it. This limitation exists because otherwise the names become a bit too vague. For example, if one gives a contract as depicted in figure 21, then it is unclear what the `column` should be assigned to. Is it actual `null` that denotes missing data? Is it a parameter `null` (that would have been changed to `_null` by the tool)?

```
operation op
{
    null;
}
preconditions
{
}
postconditions
{
    inserted into test a
    {
        col = null;
    };
};
```

Figure 21. A vague contract.

`null` is not the only SQL keyword that runs into this problem - all "current X" type of keywords (such as `current_timestamp`) are also affected. To avoid such problems it was decided that the most simple solution is used, and all SQL keywords are forbidden from being used as parameter names. Thus, if the writer of the contract intends to use a SQL keyword (such as `null`) they explicitly have to prefix it themselves (for example, `_null`).

7.2.5 Context File Statements Must be Ordered

Currently, the statements of context description must be ordered - connections and identifiers require that tables they refer to exist. This exists purely because this is the most simple way to implement reading of context description - data classes can be found and used immediately. With some effort, one could extend the tool to allow these statements to be given in any order. However, as this feature was deemed unnecessary for the goals of this work, it was not done by the author.

7.2.6 Array Support

Arrays are not supported by this tool, or at least not explicitly. Support for arrays was deemed as not mandatory for the scope of this work, as it is expected that arrays are not used much. The support for arrays was not tested much, but as they can be given as strings, it might be that they are, in fact, supported.

7.2.7 Support for Tables with Zero Columns

In PostgreSQL, it is possible to create a base table that has no columns. Such table can be used as a global Boolean switch, i.e., if the table has a row, then it represents the truth value TRUE, otherwise FALSE. To check the truth value, one would check if the table contains a row or not.

It is possible to express this table in both context and contract DSLs. However, the code generator does not check it correctly because it does not impose any "restrictions" in how the tool understands that. Simply supplying an exists clause does not constitute a restriction in tool's understanding because the tool needs a column to check against.

Appendixes 12 and 13 present a contract and its implementation where a global Boolean switch is used (table `is_open`). However, this table has one column instead of zero columns.

7.2.8 Error Handling

Error handling is very rudimentary in this tool - the tool simply has an exception that interfaces are expected to catch and process if they want to treat errors gracefully. The tool does not provide any framework for validation of input. Instead, it only reacts to errors as they appear. This is suboptimal from user experience point of view because it is much better if a user is given a list of all problems in their input immediately, rather than having to fix them one by one.

7.2.9 Referencing Other Instances

It is currently impossible for an instance in a contract to reference other instances during update and delete, and in restrictions. For example, this is not permitted: `exists` `table_a(col_a = table_b.col_b)`. This makes expressing contracts that have to use values stored in other tables impossible.

7.2.10 Multiple Postconditions

The tool supports generating implementation of multiple postconditions. However, their validity has not been checked too much. Indeed, in section 6.2, the author ran into a problem where subsequent posconditions would check for impossible cases, such as a product being in two states simultaneously. Most likely there are more cases that the author is not aware of because for most of the development only a single postcondition in a contract was used.

7.2.11 Difficult Update Cases

After a tool processes an `updated` statement, its result must apply to all subsequent postconditions. For example, if a contract updates values of some instance, then all subsequent postconditions must check that the update was successful. There are cases that the tool does not handle correctly. One such case is an update that depends on current value of the column it is updating, such as `updated entity e(atr = atr || ' suffix')`. The tool does not handle this case correctly, as in all subsequent postconditions it checks that `e.atr = e.atr || ' suffix'`, which is incorrect. It is unclear how such case should be checked. In some cases, it is possible to work around this issue by reordering the postconditions. However, this is not a proper solution because one might need the postconditions to be executed in the order given. Additionally, there might be more than one postcondition with such an update, making it necessary to somehow verify the success of such updates.

7.2.12 More Control Over Preconditions

All preconditions must be valid at all times. This assumption holds most of the time. The only case where this does not hold in tool's implementation is if an instance was deleted. This is because the data it would verify against is no longer available. However, there are cases where it might be desirable to disable some preconditions to achieve "optional" behaviour.

For example, assume there is a contract as shown in figure 22. The goal is to delete products with the price 100 and the orders of these products. The sequence of postconditions in the contract must exactly be `deleted order` and `deleted product`. Otherwise, products with the price 100 that have an order will be deleted and after that all the orders (regardless of their products) will be deleted.

```
operation op
{
}
preconditions
{
    exists product product(price = 100);
    exists order order;
    connection between product and order;
}
postconditions
{
    deleted order;
    deleted product;
}
```

Figure 22. Contract with an optional instance dependence.

Assume that for this contract exists a simple database that contains records of products and orders. Each order is associated with a product and each product is associated with zero or more orders. For this contract, when the tool generates implementation for the `deleted order` (SQL DELETE statement), it can check all the preconditions in it. However, when it reaches `deleted product`, the `order` has already been deleted. Thus, it is impossible to check this precondition in the corresponding `DELETE` statement. To this end, the tool adds the `order` symbol to an internal list of disabled symbols, and does not check its preconditions. This results in *optional* dependency between orders and products - firstly, all the related orders are deleted. After that, *regardless of whether the tool has managed to find such orders*, the product is deleted.

Now, assume that instead of deleting, one would want to keep the data, but change it to some special state instead, as depicted in figure 23. The goal is to update the state of orders with products that have the price 100 to the state "in review" and update the state of products with the price 100 to the state "deleted".

```
operation op
{
}
preconditions
{
    exists product product(price = 100);
    exists order order;
    connection between product and order;
}
postconditions
{
    updated order
    {
        state = 'In review';
    };
    updated product
    {
        state = 'Deleted';
    };
}
```

Figure 23. Contract with desired optional instance dependence.

Here, while the author of the contract would want to *optionally* set relevant orders to a new state the tool will be unable to generate a desired implementation. This is because again, all preconditions must hold at all times. Therefore, when the tool gets to `updated product` in this contract, it would have to check for now-'in review' state orders. Thus, for a product to be updated, there must be at least one order it relates to that it has updated. In other words, the implementation does not update products with the price 100 that do not have an associated order. One solution to this situation is to add a postcondition that would mark symbols as "deleted", much like the `deleted` statement does. In this way, the tool would be able to treat updated orders as if they were deleted and delete products regardless whether it has managed to update related orders or not.

7.2.13 SQL Keyword Values

During `updated` and `inserted` postconditions of the contract DSL one is allowed to assign attributes to SQL keywords. This is because there are valid keywords that the columns can be assigned to, such as `null` and `current_timestamp`. However, the tool does not verify if the keyword given is a valid value. Thus, it is possible to "assign" attributes to any SQL keyword, such as `select`. The tool does not verify such cases in any way. Furthermore, it generates code that treats these keywords as if they are assignable (such as `null`). This results in incorrect SQL code.

7.2.14 Further Development

Some further additions to the GUI could be:

- Providing additional options to influence the generation result. For instance, one could influence the generation of comments or choose whether to generate a function or a procedure as a result.
- Allowing writing multiple contracts to the window to generate code for all of these.
- Allowing reading a set of contracts from a file and writing the resulting functions to another file.
- Providing functionality of a typical text editor for programmers like coloring different language elements differently, showing the line numbers, checking the balance of brackets or automatically formatting the context and operation text.

Of course the tool could be extended to support other SQL DBMSs or DBMSs that are based on some other data model than the underlying data model of SQL. Another possible extension would be supporting generating implementations of operations that only read data. Such operations could be implemented as views, materialized views, or functions (including table functions).

8 Summary

The purpose of this work was to design and implement an open-source tool that is able to take an analysis-level contract of a database operation that manipulates data in the database as an input and generate its implementation as a PostgreSQL database SQL function, which has the body conforming to the SQL standard, as an output. In order to create such tool a formal language to express these contracts had to be defined as well.

External domain-specific languages were created to specify contracts and their contexts (i.e., tables and columns in the database) and a tool was created to take a contract and its context as an input and produce SQL implementation as an output. The tool is fully functional although with some limitations as it is discussed in the work. It could be said that the DSLs and the tool prove the concept that the proposed approach is possible and useful. The tool has a graphical user interface. The source code of the tool is open source. The code is available at `https://github.com/ghpv/dbcsql`. The tool (and by extension the DSLs) is licensed under GPL version 3 license. The thesis started with a theoretical background about the need for contracts in the first place and why it is better to be formal and use code generation tools, instead of implementing these contracts manually. Following chapters discussed both the formal languages for these contracts and their and context, and the translator tool. Finally, the author presented validation of the achieved results and discussed further areas of improvement.

References

[1] L. Burns. *Building the Agile Database: How to Build a Successful Application Using Agile Without Sacrificing Data Management*. Technics Publications, 2011. ISBN: 9781634620239. URL: https://books.google.ee/books?id=Fx%5C_XBgAAQBAJ.

[2] Segue Technologies. *Advantages and Drawbacks of Using Stored Procedures for Processing Data*. Accessed on 2022-10-25. June 2013. URL: https://www.seguetech.com/advantages-and-drawbacks-of-using-stored-procedures-for-processing-data/.

[3] David Baxter. *ORM for Database Abstraction or Stored Procedures for Data Structure Abstraction?* Accessed on 2022-11-11. Oct. 2013. URL: https://davidwaynebaxter.com/2013/10/28/orm-or-sprocs/.

[4] Joe Emison. *Stored Procedures as a Litmus Test*. Accessed on 2022-10-25. Sept. 2017. URL: https://medium.com/@JoeEmison/stored-procedures-as-a-litmus-test-c7ec772b985e.

[5] ISO/IEC JTC 1/SC 32. *ISO/IEC 9075-4:1996 - Part 4: Persistent Stored Modules (SQL/PSM)*. Accessed on 2022-10-25. Oct. 1996. URL: https://www.iso.org/standard/24358.html.

[6] TIOBE. *TIOBE Index for October 2022*. Accessed on 2022-10-25. Oct. 2022. URL: https://www.tiobe.com/tiobe-index/.

[7] Apache Cassandra. *Cassandra Documentation / Functions*. Accessed on 2022-10-25. URL: https://cassandra.apache.org/doc/latest/cassandra/cql/functions.html.

[8] Redis. *Redis functions*. Accessed on 2022-10-25. URL: https://redis.io/docs/manual/programmability/functions-intro/.

[9] MongoDB. *Atlas Functions*. Accessed on 2022-10-25. URL: https://www.mongodb.com/docs/atlas/app-services/functions/.

[10] neo4j. *User Defined Procedures and Functions*. Accessed on 2022-10-25. URL: https://neo4j.com/developer/cypher/procedures-functions/.

[11] VoltDB. *Using VoltDB / Create Procedure Reference*. Accessed on 2022-10-25. URL: https://docs.voltactivedata.com/UsingVoltDB/ddlref_createprocsql.php.

[12] InfluxDB. *Get started with InfluxDB tasks*. Accessed on 2022-10-25. URL: https://docs.influxdata.com/influxdb/v2.4/process-data/get-started/.

[13] Alan R. Hevner et al. “Design Science in Information Systems Research”. In: *Management Information Systems Quarterly* 28 (Mar. 2004), pp. 75–.

[14] Ken Schwaber and Jeff Sutherland. *Scrum Guide*. Accessed on 2023-02-03. URL: https://www.scrum.org/resources/scrum-guide.

[15] Dines Bjørner. “Domain Theory: Practice and Theories A Discussion of Possible Research Topics”. In: *Theoretical Aspects of Computing – ICTAC 2007*. Ed. by Cliff B. Jones, Zhiming Liu, and Jim Woodcock. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 1–17. ISBN: 978-3-540-75292-9.

[16] B. Meyer. “Applying ’design by contract’”. In: *Computer* 25.10 (1992), pp. 40–51. DOI: 10.1109/2.161279.

[17] Eric W. Weisstein. *Invariant*. Accessed on 2023-01-30. URL: https://mathworld.wolfram.com/Invariant.html.

[18] C. Larman and P. Kruchten. *Applying UML and Patterns: An Introduction to Object-oriented Analysis and Design and Iterative Development*. Prentice Hall PTR, 2005, pp. 287–289. ISBN: 9780131489066. URL: https://books.google.ee/books?id=tuxQAAAAMAAJ.

[19] *CS616 – Software Engineering II / Lecture 6*. Accessed on 2022-10-25. URL: http://csis.pace.edu/~marchese/CS616/Lec6/se_l6.htm.

[20] M. Völter et al. *Model-Driven Software Development: Technology, Engineering, Management*. Wiley Software Patterns Series. Wiley, 2013. ISBN: 9781118725764. URL: https://books.google.ee/books?id=9ww%5C_D9fAKncC.

[21] Martin Fowler. *Domain-Specific Languages*. Addison-Wesley Signature Series (Fowler). Pearson Education, 2010. ISBN: 9780131392809. URL: https://books.google.ee/books?id=ri1muolw%5C_YwC.

[22] Bran Selic. “The Theory and Practice of Modeling Language Design for Model-Based Software Engineering—A Personal Perspective”. In: *Generative and Transformational Techniques in Software Engineering III: International Summer School, GTTSE 2009, Braga, Portugal, July 6-11, 2009. Revised Papers*. Ed. by João M. Fernandes et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 290–321. ISBN: 978-3-642-18023-1. DOI: 10.1007/978-3-642-18023-1_7. URL: https://doi.org/10.1007/978-3-642-18023-1_7.

[23] Object Management Group. *Unified Modeling Language*. Accessed on 2022-11-21. Dec. 2017. URL: https://www.omg.org/spec/UML/2.5.1/About-UML.

[24] Gabor Karsai et al. "Design Guidelines for Domain Specific Languages". In: (Sept. 2014).

[25] Diomidis Spinellis. "Notable design patterns for domain-specific languages". In: *Journal of Systems and Software* 56.1 (2001), pp. 91–99. ISSN: 0164-1212. DOI: https://doi.org/10.1016/S0164-1212(00)00089-3. URL: https://www.sciencedirect.com/science/article/pii/S0164121200000893.

[26] *Flex - A fast scanner generator, version 2.5*. Accessed on 2022-11-28. URL: https://ftp.gnu.org/old-gnu/Manuals/flex-2.5.4/html_mono/flex.html.

[27] Rational Software Corporation. *Rational Unified Process: Artifact: Data Model*. Accessed on 2023-05-02. URL: https://sceweb.uhcl.edu/helm/RationalUnifiedProcess/process/artifact/ar_datmd.htm.

[28] Rational Software Corporation. *Rational Unified Process: Activity: Database Design*. Accessed on 2023-05-02. URL: https://sceweb.uhcl.edu/helm/RationalUnifiedProcess/process/activity/ac_dbdes.htm.

[29] Microsoft. *Programming in AL*. Accessed on 2023-04-25. Sept. 2022. URL: https://learn.microsoft.com/en-us/dynamics365/business-central/dev-itpro/developer/devenv-programming-in-al.

[30] Microsoft. *X++ language reference*. Accessed on 2023-04-25. Aug. 2022. URL: https://learn.microsoft.com/en-us/dynamics365/fin-ops-core/dev-itpro/dev-ref/xpp-language-reference.

[31] Eiffel Language. *Design by Contract and Assertions*. Accessed on 2023-05-02. URL: https://www.eiffel.org/doc/solutions/Design_by_Contract_and_Assertions.

[32] Yilong Yang et al. "RM2PT: A Tool for Automated Prototype Generation from Requirements Model". In: *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 2019, pp. 59–62. DOI: 10.1109/ICSE-Companion.2019.00038.

[33] Object Management Group. *Object Constraint Language*. Accessed on 2023-05-08. URL: http://www.omg.org/spec/OCL/.

[34] erwin. *Stored Procedures*. Accessed on 2022-10-25. URL: https://bookshelf.erwin.com/bookshelf/public_html/2020R2/Content/User%20Guides/erwin%20Help/Stored_Procedures.html.

[35] Oracle. *Using Quick SQL*. Accessed on 2022-10-25. URL: https://docs.oracle.com/database/apex-18.1/AEUTL/using-quick-SQL.htm.

[36] Lucas Jellema. *RuleGen – the next generation framework for Data Logic (aka Data oriented business rules)?!* Accessed on 2023-05-03. URL: https://technology.amis.nl/amis/rulegen-the-next-generation-framework-for-data-logic-aka-data-oriented-business-rules/.

[37] Morten Braten. *quick-plsql*. Accessed on 2022-10-25. Sept. 2020. URL: https://github.com/mortenbra/quick-plsql.

[38] Steve Wellens. *Automatically Generate Stored Procedures with Visual Studio*. Accessed on 2022-10-25. Dec. 2011. URL: https://weblogs.asp.net/stevewellens/automatically-generate-stored-procedures-with-visual-studio.

[39] Mladen Prajdić. *CRUD stored procedures generation*. Accessed on 2022-10-25. URL: https://ssmstoolspack.com/Features?f=12.

[40] fasterzip. *Generador stored funciones postgresql*. Accessed on 2022-10-25. URL: https://sourceforge.net/p/genfuncpostgres/wiki/Home/.

[41] L. Rönnbäck et al. *Anchor modeling — Agile information modeling in evolving data environments*. Special issue on 28th International Conference on Conceptual Modeling (ER 2009). 2010. DOI: https://doi.org/10.1016/j.datak.2010.10.002. URL: https://www.sciencedirect.com/science/article/pii/S0169023X10001229.

[42] J. Martin. *Managing the Data-base Environment*. (A James Martin book). Prentice-Hall, 1983. ISBN: 9780135505823. URL: https://books.google.ee/books?id=nMgmAAAAMAAJ.

[43] *Bison 3.8.1*. Accessed on 2022-11-28. URL: https://www.gnu.org/software/bison/manual/bison.html.

[44] Terence Parr. *The Definitive ANTLR 4 Reference*. Oreilly and Associate Series. Pragmatic Bookshelf, 2012. ISBN: 9781934356999. URL: https://books.google.ee/books?id=SBXuLwEACAAJ.

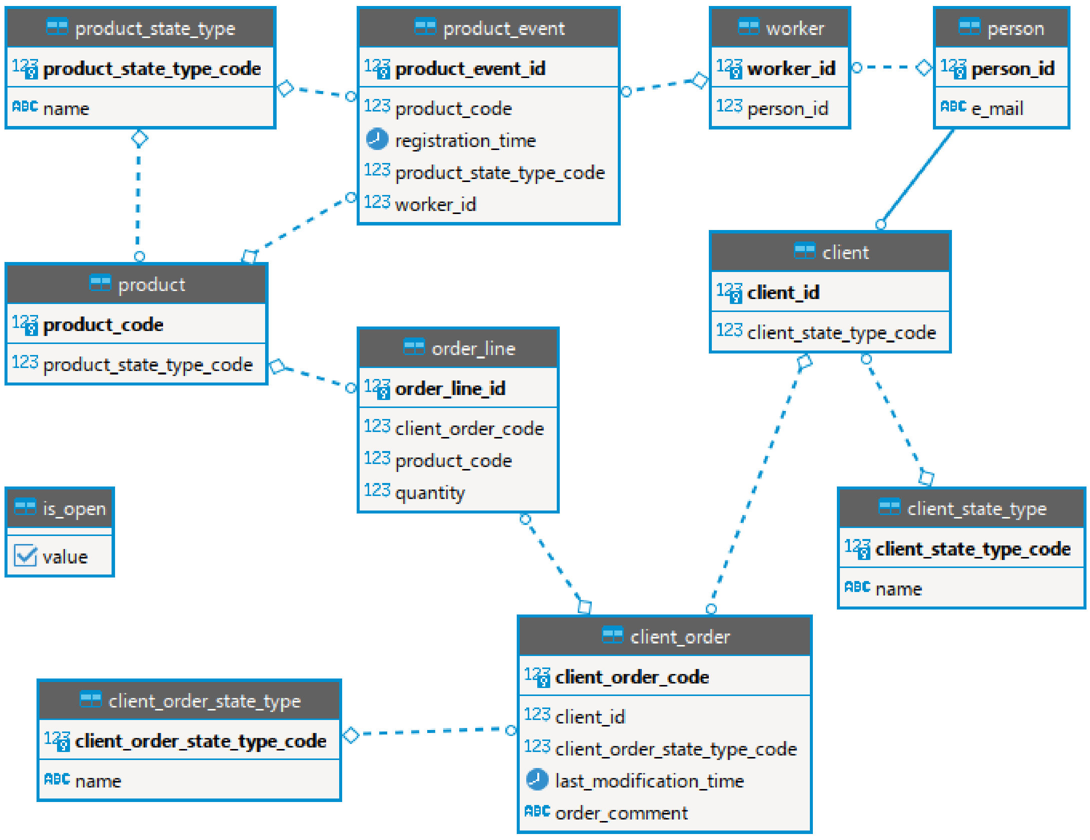
[45] Free Software Foundation. *What is Free Software?* Accessed on 2023-04-17. URL: https://www.fsf.org/about/what-is-free-software.

[46] Laurenz Albe. *Better SQL functions in PostgreSQL v14*. Accessed on 2023-05-03. URL: https://www.cybertec-postgresql.com/en/better-sql-functions-in-postgresql-v14/.

Appendix 1 – Non-Exclusive License for Reproduction and Publication of a Graduation Thesis[1]

[1]The non-exclusive licence is not valid during the validity of access restriction indicated in the student's application for restriction on access to the graduation thesis that has been signed by the school's dean, except in case of the university's right to reproduce the thesis for preservation purposes only. If a graduation thesis is based on the joint creative activity of two or more persons and the co-author(s) has/have not granted, by the set deadline, the student defending his/her graduation thesis consent to reproduce and publish the graduation thesis in compliance with clauses 1.1 and 1.2 of the non-exclusive licence, the non-exclusive license shall not be valid for the period.

Appendix 2 - Supervisor's challenge - schema



Appendix 3 - Supervisor’s challenge - OP1 contract

```
operation "Add an order line"
{
    p_order_code;
    p_product_code;
    p_quantity;
}
preconditions
{
    // Client order was in the state "Not submitted"
    exists client_order_state_type
        not_submitted_client_order_state
        (name = ’Not submitted’);
    exists client_order
        client_order(client_order_code = p_order_code);
    connection between client_order
        and not_submitted_client_order_state;

    // Client of the order was in the state "Active"
    exists client_state_type
        active_client_state(name = ’Active’);
    connection between active_client_state
        and client_order;

    // Product was in the state "Active"
    exists product_state_type
        active_product_state_type (name = ’Active’);
    exists product product
        (product_code = p_product_code);
    connection between product
        and active_product_state_type;
}
postconditions
{
    // New order line ol was created
    inserted into order_line ol
    {
        link with client_order;
        link with product;
        quantity = p_quantity;
    };
    // Client_order.last_modification_time
    // :=current date + time
    updated client_order
    {
        last_modification_time = current_timestamp;
    };
}
```

Appendix 4 - Supervisor’s challenge - OP1 generated code

```
CREATE OR REPLACE FUNCTION add_an_order_line
(
    p_order_code INTEGER,
    p_product_code INTEGER,
    p_quantity INTEGER
)
RETURNS INTEGER
LANGUAGE SQL SECURITY DEFINER
SET SEARCH_PATH TO ’public’, ’pg_temp’
BEGIN ATOMIC
    INSERT INTO
        order_line AS ol
    (
        quantity,
        client_order_code,
        product_code
    )
    SELECT
        p_quantity,
        client_order.client_order_code,
        product.product_code
    FROM
        client_order_state_type
            AS not_submitted_client_order_state,
        client_order AS client_order,
        client_state_type AS active_client_state,
        client AS client,
        product_state_type AS active_product_state_type,
        product AS product
    WHERE
        (
            not_submitted_client_order_state.name
                = ’Not submitted’
            AND client_order.client_order_code
            = p_order_code
        AND client_order.client_order_state_type_code
            = not_submitted_client_order_state
            .client_order_state_type_code
        AND active_client_state.name = 'Active'
        AND client.client_state_type_code
            = active_client_state
            .client_state_type_code
        AND client.client_id = client_order.client_id
        AND active_product_state_type.name = 'Active'
        AND product.product_code = p_product_code
        AND product.product_state_type_code
            = active_product_state_type
            .product_state_type_code
    )
;
UPDATE
    client_order AS client_order
SET
    last_modification_time = current_timestamp
FROM
    order_line AS ol,
    client_order_state_type
        AS not_submitted_client_order_state,
    client_state_type AS active_client_state,
    client AS client,
    product_state_type AS active_product_state_type,
    product AS product
WHERE
    (
        client_order.client_order_code
            = ol.client_order_code
        AND ol.product_code = product.product_code
        AND ol.quantity = p_quantity
        AND not_submitted_client_order_state.name
            = 'Not submitted'
        AND client_order.client_order_code
            = p_order_code
        AND client_order.client_order_state_type_code
```

```
                = not_submitted_client_order_state
                .client_order_state_type_code
            AND active_client_state.name = 'Active'
            AND client.client_state_type_code
                = active_client_state.client_state_type_code
            AND client.client_id = client_order.client_id
            AND active_product_state_type.name = 'Active'
            AND product.product_code = p_product_code
            AND product.product_state_type_code
                = active_product_state_type
                .product_state_type_code
        )
    RETURNING
        client_order.client_order_code
    ;
END;
```

Appendix 5 - Supervisor’s challenge - OP2 contract

```
operation "Delete an order line"
{
    p_order_line_id;
}
preconditions
{
    // Client order was in the state "Not submitted"
    exists client_order_state_type
        not_submitted_order_state
        (name = ’Not submitted’);
    exists client_order client_order;
    connection between client_order
        and not_submitted_order_state;

    // Client of the order was in the state "Active"
    exists client_state_type active_client_state
        (name = ’Active’);
    exists client client;
    connection between client and active_client_state;
    connection between client and client_order;

    exists order_line order_line
        (order_line_id = p_order_line_id);
    connection between order_line and client_order;
}
postconditions
{
    // Client_order.last_modification_time
    // :=current date + time
    updated client_order
    {
        last_modification_time = current_timestamp;
    };
    // The order line was deleted
    deleted order_line;
}
```

Appendix 6 - Supervisor’s challenge - OP2 generated code

```
CREATE OR REPLACE FUNCTION delete_an_order_line
(
    p_order_line_id INTEGER
)
RETURNS INTEGER
LANGUAGE SQL SECURITY DEFINER
SET SEARCH_PATH TO 'public', 'pg_temp'
BEGIN ATOMIC
    UPDATE
        client_order AS client_order
    SET
        last_modification_time = current_timestamp
    FROM
        client_order_state_type
            AS not_submitted_order_state,
        client_state_type AS active_client_state,
        client AS client
    WHERE
        (
            not_submitted_order_state.name
                = 'Not submitted'
            AND client_order.client_order_state_type_code
                = not_submitted_order_state
                .client_order_state_type_code
            AND active_client_state.name = 'Active'
            AND client.client_state_type_code
                = active_client_state
                .client_state_type_code
            AND client.client_id = client_order.client_id
        )
    RETURNING
        client_order.client_order_code
    ;
    DELETE FROM
```

```
        order_line AS order_line
    USING
        client_order_state_type
            AS not_submitted_order_state,
        client_order AS client_order,
        client_state_type AS active_client_state,
        client AS client
    WHERE
        (
            not_submitted_order_state.name
                = 'Not submitted'
            AND client_order
                .client_order_state_type_code
                = not_submitted_order_state
                .client_order_state_type_code
            AND active_client_state.name = 'Active'
            AND client.client_state_type_code
                = active_client_state
                .client_state_type_code
            AND client.client_id = client_order.client_id
            AND order_line.order_line_id = p_order_line_id
            AND client_order.client_order_code
                = order_line.client_order_code
        )
    ;
END;
```

Appendix 7 - Supervisor’s challenge - OP3 contract

```
operation "Reject a client order"
{
    p_order_code;
    p_comment;
}
preconditions
{
    // Client order was in the state "Submitted"
    exists client_order_state_type
        submitted_client_order_state
        (name = ’Submitted’);
    exists client_order client_order
        (client_order_code = p_order_code);
    connection between client_order
        and submitted_client_order_state;

    exists client_order_state_type
        rejected_client_order_state
        (name = ’Rejected’);
}
postconditions
{
    // Client order was in the state "Rejected"
    // Client_order.last_modification_time
    //     :=current date + time
    // Client_order.order_comment
    //     := Client_order.order_comment || p_comment
    //     /* using this expression directly
    //     in PostgreSQL means that if the comment
    //     does not exist, then new comment is not
    //     added because NULL || ’tekst’=> NULL*/
    updated client_order
    {
        link with rejected_client_order_state;
```

```
            last_modification_time = current_timestamp;
            order_comment = order_comment || p_comment;
        };
    }
```

Appendix 8 - Supervisor’s challenge - OP3 generated code

```
CREATE OR REPLACE FUNCTION reject_a_client_order
(
p_order_code INTEGER,
p_comment VARCHAR
)
RETURNS INTEGER
LANGUAGE SQL SECURITY DEFINER
SET SEARCH_PATH TO 'public', 'pg_temp'
BEGIN ATOMIC
UPDATE
client_order AS client_order
SET
last_modification_time
            = current_timestamp,
order_comment
            = (client_order.order_comment
                ||p_comment),
client_order_state_type_code
            = rejected_client_order_state
            .client_order_state_type_code
FROM
client_order_state_type
            AS submitted_client_order_state,
client_order_state_type
            AS rejected_client_order_state
WHERE
(
submitted_client_order_state.name
                = 'Submitted'
AND client_order.client_order_code
                = p_order_code
AND client_order
                .client_order_state_type_code
                = submitted_client_order_state
                .client_order_state_type_code
AND rejected_client_order_state.name
                = 'Rejected'
)
RETURNING
client_order.client_order_code
;
END;
```

Appendix 9 - Supervisor’s challenge - OP4 contract

```
operation "Drop a product from sale"
{
    p_product_code;
    p_e_mail_of_worker;
}
preconditions
{
    // Product was in the state "Active"
    exists product_state_type active_product_state
        (name = ’Active’);
    exists product p (product_code = p_product_code);
    connection between active_product_state and p;

    // Worker with p_e_mail_of_worker existed
    exists person person (e_mail = p_e_mail_of_worker);
    exists worker worker;
    connection between person and worker;

    exists product_state_type dropped_state
        (name = ’Dropped’);
    exists client_order_state_type cos
        (name not in (’Archived’, ’Not submitted’));
    exists client_order_state_type
        in_review_client_order_state
        (name = ’In review’);
    exists order_line ol;
    exists client_order co;
    connection between cos and co;
    connection between co and ol;
    connection between ol and p;
}
postconditions
{
    // All the client orders that were not in the state
    // "Archived" and "Not submitted" but have the product
    // in question were in the state "In review".
    // In case of all these client orders:
    // * Client_order.last_modification_time
    //      = current timestamp
    // * Client_order.order_comment
    //      := Client_order.order_comment
    //      || ' a product was dropped'
    updated co
    {
        link with in_review_client_order_state;
        last_modification_time = current_timestamp;
        order_comment = order_comment
            || ' a product was dropped';
    };
    // Product was in the state "Dropped"
    updated p
    {
        link with dropped_state;
    };
    // New product event was created
    // (associated with the product,
    //    new state, and given worker)
    inserted into product_event ev
    {
        registration_time = current_timestamp;
        link with p;
        link with dropped_state;
        link with worker;
    };
}
```

Appendix 10 - Supervisor’s challenge - OP4 generated code

```
CREATE OR REPLACE FUNCTION drop_a_product_from_sale
(
    p_product_code INTEGER,
    p_e_mail_of_worker VARCHAR
)
RETURNS INTEGER
LANGUAGE SQL SECURITY DEFINER
SET SEARCH_PATH TO ’public’, ’pg_temp’
BEGIN ATOMIC
    UPDATE
        client_order AS co
    SET
        last_modification_time = current_timestamp,
        order_comment =
            (co.order_comment||’ a product was dropped’),
        client_order_state_type_code
            = in_review_client_order_state
            .client_order_state_type_code
    FROM
        product_state_type AS active_product_state,
        product AS p,
        person AS person,
        product_state_type AS dropped_state,
        client_order_state_type AS cos,
        client_order_state_type
            AS in_review_client_order_state,
        order_line AS ol,
        worker AS worker
    WHERE
        (
            active_product_state.name = ’Active’
            AND p.product_code = p_product_code
            AND p.product_state_type_code
                = active_product_state
                .product_state_type_code
            AND person.e_mail = p_e_mail_of_worker
            AND person.person_id = worker.person_id
            AND dropped_state.name = 'Dropped'
            AND cos.name not in ('Archived', 'Submitted')
            AND in_review_client_order_state.name
                = 'In review'
            AND co.client_order_state_type_code
                = cos
                .client_order_state_type_code
            AND co.client_order_code
                = ol.client_order_code
            AND ol.product_code = p.product_code
    )
;
UPDATE
    product AS p
SET
    product_state_type_code
        = dropped_state.product_state_type_code
FROM
    client_order AS co,
    product_state_type AS active_product_state,
    person AS person,
    product_state_type AS dropped_state,
    client_order_state_type AS cos,
    client_order_state_type
        AS in_review_client_order_state,
    order_line AS ol,
    worker AS worker
WHERE
    (
        co.last_modification_time
            = current_timestamp
        AND co.order_comment
            = (co.order_comment
                ||' a product was dropped')
        AND co.client_order_state_type_code
            = in_review_client_order_state
```

```
                .client_order_state_type_code
            AND active_product_state.name = 'Active'
            AND p.product_code = p_product_code
            AND p.product_state_type_code
                = active_product_state
                .product_state_type_code
            AND person.e_mail = p_e_mail_of_worker
            AND person.person_id = worker.person_id
            AND dropped_state.name = 'Dropped'
            AND cos.name not in
                ('Archived', 'Not submitted')
            AND in_review_client_order_state.name
                = 'In review'
            AND co.client_order_state_type_code
                = cos.client_order_state_type_code
            AND co.client_order_code = ol.client_order_code
            AND ol.product_code = p.product_code
    )
;
INSERT INTO
    product_event AS ev
(
    registration_time,
    product_code,
    product_state_type_code,
    worker_id
)
SELECT
    current_timestamp,
    p.product_code,
    dropped_state.product_state_type_code,
    worker.worker_id
FROM
    client_order AS co,
    product AS p,
    product_state_type AS active_product_state,
    person AS person,
    product_state_type AS dropped_state,
    client_order_state_type AS cos,
        client_order_state_type
            AS in_review_client_order_state,
        order_line AS ol,
        worker AS worker
    WHERE
        (
            co.last_modification_time = current_timestamp
            AND co.order_comment =
                (co.order_comment
                ||' a product was dropped')
            AND p.product_state_type_code
                = dropped_state.product_state_type_code
            AND co.client_order_state_type_code
                = in_review_client_order_state
                .client_order_state_type_code
            AND active_product_state.name = 'Active'
            AND p.product_code = p_product_code
            AND p.product_state_type_code
                = active_product_state
                .product_state_type_code
            AND person.e_mail = p_e_mail_of_worker
            AND person.person_id = worker.person_id
            AND dropped_state.name = 'Dropped'
            AND cos.name not in ('Archived', 'Submitted')
            AND in_review_client_order_state.name
                = 'In review'
            AND co.client_order_state_type_code
                = cos.client_order_state_type_code
            AND co.client_order_code
                = ol.client_order_code
            AND ol.product_code = p.product_code
        )
    RETURNING
        ev.product_event_id
    ;
END;
```

Appendix 11 - Supervisor’s challenge - DSL Context

```
table person
{
    person_id INTEGER;
    e_mail VARCHAR;
};
identifier for person is person_id;

table worker
{
    worker_id INTEGER;
    person_id INTEGER;
};
connection between worker and person
{
    person_id = person_id;
};
identifier for worker is worker_id;

table client_state_type
{
    client_state_type_code SMALLINT;
    name VARCHAR;
};
identifier for client_state_type
    is client_state_type_code;

table client
{
    client_id INTEGER;
    client_state_type_code SMALLINT;
};
identifier for client is client_id;
connection between client and person
{
    client_id = person_id;
};
connection between client and client_state_type
{
    client_state_type_code = client_state_type_code;
};

table client_order_state_type
{
    client_order_state_type_code SMALLINT;
    name VARCHAR;
};
identifier for client_order_state_type
    is client_order_state_type_code;

table client_order
{
    client_order_code INTEGER;
    client_id INTEGER;
    client_order_state_type_code SMALLINT;
    last_modification_time TIMESTAMP;
    order_comment VARCHAR;
};
identifier for client_order
    is client_order_code;
connection between client_order
    and client_order_state_type
{
    client_order_state_type_code
        = client_order_state_type_code;
};
connection between client_order and client
{
    client_id = client_id;
};

table product_state_type
{
    product_state_type_code SMALLINT;
    name VARCHAR;
};
identifier for product_state_type
    is product_state_type_code;

table product
{
    product_code INTEGER;
    product_state_type_code SMALLINT;
    available_quantity INTEGER;
    price INTEGER;
};
identifier for product is product_code;
connection between product and product_state_type
{
    product_state_type_code = product_state_type_code;
};

table order_line
{
    order_line_id INTEGER;
    client_order_code INTEGER;
    product_code INTEGER;
    quantity INTEGER;
};
identifier for order_line is order_line_id;
connection between order_line and client_order
{
    client_order_code = client_order_code;
};
connection between order_line and product
{
    product_code = product_code;
};

table product_event
{
    product_event_id INTEGER;
    product_code INTEGER;
    registration_time TIMESTAMP;
    product_state_type_code TYPE;
    worker_id INTEGER;
};
identifier for product_event is product_event_id;
connection between product_event and product
{
    product_code = product_code;
};
connection between product_event and product_state_type
{
    product_state_type_code = product_state_type_code;
};
connection between product_event and worker
{
    worker_id = worker_id;
};
```

Appendix 12 - Supervisor’s challenge - OP5 contract

```
operation "Delete product events"
{
}
preconditions
{
    // The system must be opened
    // (i.e., is_open has value TRUE)
    exists is_open io(value = TRUE);
    exists product_event ev;
}
postconditions
{
    // All product events have been deleted.
    deleted ev;
};
```

Appendix 13 - Supervisor’s challenge - OP5 generated code

```
CREATE OR REPLACE FUNCTION
    delete_product_events
(
)
RETURNS INTEGER
LANGUAGE SQL SECURITY DEFINER
SET SEARCH_PATH TO ’public’, ’pg_temp’
BEGIN ATOMIC
    DELETE FROM
        product_event AS ev
    USING
        is_open AS io
    WHERE
        io.value = true
    RETURNING
        ev.product_event_id
    ;
END;
```