Universität Passau
Fakultät für Mathematik und Informatik
Lehrstuhl für Informations Management

Implementation of MiniCon
A Scalable Algorithm for Answering Queries Using Views
Documentation

Kevin Irmscher
currently at:
University of British Columbia
Department of Computer Science

Supervisor:
Rachel Pottinger
University of British Columbia
Department of Computer Science

Contents

1 Introduction

1.1 Outline

The objective of this project was to accomplish an implementation of the MiniCon algorithm. The work is based on the paper of Pottinger and Halevy in 2001 [1]. MiniCon is an algorithm that uses materialized views to rewrite queries for the purpose of query optimization.

The general idea of answering queries using views, also known as query reformulation, is described in section 1.3. Section 1.2 gives information about the usage of the program and how to perform automated testing. Section 1.4 explains the functioning of the MiniCon algorithm in detail. The overall design of the system, including major data structures and the interaction of the different components, is addressed in section 2. A detailed description of the algorithm's implementation is given in section 3. Section 4 contains the testing of crucial parts of the algorithm by means of test cases and their results. Finally, section 5 describes extensions of the algorithm and gives suggestions about future work that can possibly be done.

1.2 Program Usage

This project is entirely implemented in Java using J2SE 5.0.

The project is comprised of six Java packages: *antlr*, *converter*, *datalog*, *minicon*, *parser*, *testing*. Java package *minicon* contains the classes that implement the algorithm. Package *datalog* includes classes responsible for representing Datalog queries. Package *parser* and *antlr* provide the classes used to parse the input. JUnit [5] testcases are contained in package *testing*. Package *converter* contains classes to represent SQL queries and converting them into a Datalog representation.

Compile program

- tested with Java compiler **1.5.0_02**
- compiling: call **javac minicon\MiniCon.java** in the base directory

Starting the program

In the base directory call: **java minicon.MiniCon**

Program options

- no option : Datalog command line input
- ***-v*** - verbose mode : intermediate results are printed out
- ***-f FILENAME.XML ID*** - Datalog file input mode : takes xml file (with Datalog queries) as input
  (see Appendix B for XML schema file)
- ***-sql*** - SQL command line input
- ***-sql -f FILENAME.XML ID*** - SQL file input: takes xml file (with SQL queries) as input
  (see Appendix C for XML schema file)

If no option is provided, the program will prompt a command line which accepts Datalog queries as input. The user will be asked to enter the query and confirm it by pressing "Enter". If the query cannot be parsed, an error message will appear and the user will have to enter the query again. Then one or more views have to be entered. Reading input will be terminated when an empty line is followed by enter. The parser will return an error if the the input is syntactically wrong, but the semantics will not be checked.

If option *-sql* is provided as start parameter, the program assumes SQL input. First, the user will be asked to enter a schema definition for the relations contained in the SQL statements. It has the following format:

relationName(variable1, variable2, variable3,...)

Then similar to Datalog input, the user has to enter an SQL query and one or more SQL view statements with the following format:

select *alias.variable, alias2.variable2, variable3.attribute3,...*
from *relation alias, relation2 alias2, relation3 alias3,...*
where *aliasN.variableN = constant, aliasN.variableN = aliasM.variableM,...*

The *where clause* is optional. More information about SQL query representation is given in section 5.2.

If SQL input is used, the program will convert it to an equivalent DatalogQuery and run MiniCon. After finishing the algorithm, the user will be asked whether SQL statements for the resulting rewritings are to be printed out.

Use testcase file

In the base directory call: **java minicon.MiniCon -f testcases.xml *ID*** (use option *-sql* for SQL input)

This will take the provided xml file using the test case with number *ID*. Testing is explained in section 4.

Perform JUnit tests

JUnit [5] is a framework which allows automatic testing of Java projects. The package *testing* contains test cases to check crucial methods of the MiniCon algorithm. In case the Java code is modified, the correctness of the algorithm's implementation can be easily verified by running the test suite. Java class *MiniConTests* calls all test cases included in the package.

- *Install JUnit:* (see *www. junit. org* for further deatils)
  - download and install program file from JUnit website
  - add junit.jar **and** MiniCon base directory to CLASSPATH
- *Run test suite:*
  - change to MiniCon base directory
  - batch TestRunner: **java junit.textui.TestRunner testing.MiniConTests**
  - Swing based TestRunner: **java junit.swingui.TestRunner testing.MiniConTests**

All tests schould pass OK when running the test suite.

1.3 Query Reformulation

In large database systems it is vital to have efficient methods for answering queries. One approach for query optimization is the use of materialized views. In many cases using cached data is much faster than accessing the actual database relation directly. This method is used especially in data integration systems [2]. In such systems a mediated schema characterizes a set of heterogeneous databases. Views are mapped to these different data sources over the mediated schema. The schema itself does not contain any data. That is why queries that are posed by the user on this schema, have to be translated on to sources based on the views. Query reformulation is the process of translating a user query into a query over the data sources. Because of a possibly very large number of data sources it is important to process queries efficiently.

The MiniCon algorithm addresses the problem of *query reformulation*. In other words, given a query and a set of views, the algorithm will compute *rewritings* of the query using views.

The following notations for query and rewriting are used throughout this documentation:

Definition 1 *: (conjunctive) query*

- *a query that contains only selection, projection and join*
- *it is a term of the form* $q(x_1, ..., x_i) : -e_1(X_1), ..., e_n(X_n), x_k < x_l, x_{k+1} < x_{l+1}, ...$
- $q(x_1, ..., x_i)$ *is called the* ***head*** *of the query,* $x_1, ..., x_i$ *are* ***head variables***
- $e_1(X_1), ..., e_n(X_n), x_k < x_l, x_{k+1} < x_{l+1}, ...$ *is the* ***body****,* $e_1(X_1)$ *is a* ***subgoal*** *or* ***predicate***
- X_1 *is a set of variables and/or constants*
- $x_k < x_l$ *is an* ***interpreted predicate*** *(also called built-in predicate) where either* x_k *is a* ***variable*** *and* x_l *is a* ***constant*** *or* x_k *is a* ***constant*** *and* x_l *is a* ***variable***
- *if* ***x*** *appears in an interpreted predicate that it must also appear in an ordinary predicate*
- *head variables are also called* ***distinguished variables*** *and variables that only appear in the body are called* ***existential variables***
- *only safe queries are considered, i.e. every variable that appears in the head must also appear in the body*
- ***views*** *are also defined by conjunctive queries*

Definition 2 *: maximally contained rewriting*
Given a query Q and a set of views $V_1, ..., V_n$*, then* ***rewriting Q'***

- *contains a subset of the set of views and interpreted predicates in its body*
- *is a reformulation of the query and provides the maximal number of answers possible given* $V_1, ..., V_n$

Example 1
- *Query: Q(NAME,YEAR) :- student(NAME,ID), level(ID,YEAR)*
- *View 1: studentInfo(NAME,ID) :- student(NAME,ID)*
- *View 2: levelInfo(ID,YEAR) :- level(ID,YEAR)*

Rewriting: *Q'(NAME,YEAR) :- studentInfo(NAME,ID),levelInfo(ID,YEAR)*

Example 1 shows the creation of a rewriting given a query and two views. The relation *student* means that student with name *NAME* has student number *ID*. The relation *level* means that student with student number *ID* is in year *YEAR*. In case of a data integration systems these relations represent a mediated schema over data sources. The views *studentInfo* and *levelInfo* are mappings to the data sources over the mediated schema. It is not possible to access the mediated schema relations directly because they do not store any data. Instead, the query has to be rewritten so it just uses materialized views. The Datalog query *Q'* represents a rewriting of query *Q* which exclusively contains the views *studentInfo* and *levelInfo*. In this case the resulting rewriting is equivalent to the query.

However, especially in the field of data integration it is not always possible to obtain an equivalent rewriting. MiniCon instead computes a *maximally-contained rewriting*. That means, it is possible that not all answers that can be generated by the query can also be generated by the rewriting. However, a maximally-contained rewriting contains the maximal number of possible answers with respect to a particular query language.

Example 2

- *Query: Q(NAME,YEAR) :- student(NAME,ID), level(ID,YEAR)*
- *Source: S(NAME,YEAR) :- student(NAME,ID), level(ID,YEAR), inCompSci(ID)*

Rewriting: Q'(NAME,YEAR) :- S(NAME,YEAR)

Example 2 depicts the case when the result is not equivalent to the query. Source *S* is used to form the rewriting. The relation *inCompSci* means that student with studtent number *ID* is in computer science. Hence, rewriting *Q'* only includes computer science students. However, it is not possible to create a rewriting which results in more answers. Thus, it is a maximally-contained rewriting.

The MiniCon algorithm is guaranteed to be sound and complete [1] considering conjunctive queries as defined in Definition 1. Generally, data integration systems have to maintain a large number of data sources. Answering a query using a set of views possibly leads to a search through an exponential number of rewritings. Thus, the complexity of finding rewritings is NP-complete [3]. MiniCon is a scalable algorithm and performs well even in the presence of a large number of views.

1.4 Algorithm Description

The algorithm is basically divided in two major steps. The first one considers every single view and attempts to create one or more partial mappings from subgoals in the query to relevant subgoals in the view. The mappings are partial because it is not necessary to map every subgoal in the query to its corresponding view subgoal. Suppose query subgoal s_Q can be mapped to a view subgoal s_V, then in some cases the algorithm will also extend the mapping with further query subgoals in order to cover join predicates. However, the mapping will always contain only a minimal set of query subgoals. This set of subgoals is called a *MiniCon Description (MCD)*. MCDs are subject to certain constraints in order to be valid, named *MiniCon Property*, which is defined in Definition 4.

Definition 3 *: MiniCon Description (MCD)*

A MCD for a query ***Q*** *over a view* ***V*** *exhibits the following properties:*

- *a list of* ***covered subgoals****, i.e. subgoals in* ***Q*** *that take part in the mapping*
- *mapping from variables or constants of query subgoals to* ***variables*** *of view subgoals*
  → ***variable mapping***

- *mapping from variables or constants of query subgoals to* ***constants*** *of view subgoals* →***constant mapping***
- *fulfills the* ***MiniCon Property***

The second step creates rewritings of the query by combining MCDs obtained in the first step. As a possible third step redundancies in the rewritings are removed.

Example 3
- *Query: Q(NAME) :- student(NAME,ID), level(ID, 4), takes(ID,'databases')*
- *View: V(VNAME,VYEAR) :- student(VNAME,VID), level(VID,VYEAR), takes(VID,'databases')*

Resulting MCD for View 1:
variable mapping *(mapping from variable or constant to variable):*
- *NAME* → *VNAME*
- *ID* → *VID*
- *4* → *VYEAR*

constant mapping *(mapping from variable or constant to constant):*
- *'databases'*→*'databases'*

covered subgoals
- *student(NAME,ID)*
- *level(ID,4)*
- *takes(ID,'databases')*

Example 3 shows the resulting MCD for the given query and view. The answers of the query are students who are both in the 4^{th} year (subgoal *level)* and who take the course 'databases' (subgoal *takes*). Answers of the view are 'student name, year'-tuples of all students who are taking the course 'databases'. Exactly one MCD is created for the view which contains a mapping to variables as well as a mapping to a constant. The set of covered subgoals includes all query subgoals.

An MCD created by the algorithm has a minimal set of query subgoals in order to fulfill the *MiniCon Property* (Definition 4). This set of constraints assures the creation of MCDs that can be combined to form rewritings. Views that cannot be used for a rewriting will be eliminated in the first place which reduces the complexity of the combining step.

Definition 4 *: MiniCon Property*

1. ***Variable mapping:***
   given a mapping from a variable or constant $\boldsymbol{x}$ *in the query to a* ***variable*** $\boldsymbol{y}$ *in the view then following conditions must hold:*
   (a) *if* $\boldsymbol{x}$ *is a head variable in the query then* $\boldsymbol{y}$ *must be a head variable (distinguished variable) in the view*
   (b) *if* $\boldsymbol{y}$ *is existential then all query subgoals that contain* $\boldsymbol{x}$ *must be covered by the MCD*
   (c) *if* $\boldsymbol{x}$ *is a constant in the query then* $\boldsymbol{y}$ *must be a head variable in the view*
2. ***Constant mapping:***
   extension of property 1a:

(a) *if* $\boldsymbol{x}$ *is a head variable in the query and* $\boldsymbol{x}$ *is mapped to* $\boldsymbol{y}$*, then* $\boldsymbol{y}$ *must either be a head variable or a constant in the view*

3. ***Interpreted predicates:***

(a) *interpreted predicates have the form* $x < y$, $x \leq y$, $x > y$ *or* $x \geq y$

(b) *either* $\boldsymbol{x}$ *is a variable and* $\boldsymbol{y}$ *a constant or* $\boldsymbol{x}$ *is a constant and* $\boldsymbol{y}$ *is a variable*

(c) *property 1b also applies to interpreted predicates*

(d) *if an interpreted predicate* p_Q *in the query is mapped to an interpreted predicate* p_V *in the view then* p_V *must logically entail* p_Q*, i.e.* $p_V \models p_Q$

The behavior of the algorithm enforcing the MiniCon Property is shown in the subsequent examples.

Example 4 **(MiniCon Property 1)**

(a)

- *Query: Q(NAME,YEAR) :- student(NAME,ID), level(ID,YEAR)*
- *View: V(VNAME) :- student(VNAME,VID), level(VID,VYEAR)*

No MCD will be created because of the mapping ID →VID *and* ID *is a head variable but* VID *is not.*

(b)

- *Query: Q(NAME,YEAR) :- student(NAME,ID), level(ID,YEAR)*
- *View: V(VNAME,VYEAR) :- student(VNAME,VID), level(VID,VYEAR), takes(VID,'databases')*

A valid MCD will be created which covers subgoals student *and* level. *The algorithm will consider the subgoal* student *with the mappings* NAME→VNAME *and* ID→VID. *Because* VID *is existential it will also map the other subgoals in query that contains* ID, *in this case the subgoal* level.

(c)

- *Query: Q(NAME) :- student(NAME,ID), takes(ID,'databases')*
- *View: V(VNAME,VYEAR) :- student(VNAME,VID), takes(VID,VCOURSE)*

No MCD will be created because by property 1b the subgoal takes *has to be covered. This will result in the mapping from constant 'databases' → to existential variable VCOURSE which in turn violates property 1c.*

Example 5 **(MiniCon Property 2)**

- *Query: Q(NAME,YEAR) :- student(NAME,ID), level(ID,YEAR)*
- *View: V(VNAME) :- student(VNAME,VID), level(VID,3)*

Because of property 1b, both subgoals of the query have to be covered. Property 2a allows the creation of the MCD although head variable YEAR is mapped to constant 3 that does not appear in the head.

Example 6 **(MiniCon Property 3)**

- *Query: Q(NAME) :- student(NAME,ID), level(ID,YEAR), YEAR > 2*
- *View: V(VNAME) :- student(VNAME,VID), level(VID,VYEAR), VYEAR > 3*

Property 3d is fulfilled because VYEAR>3 ⇒ YEAR>2.

- *Query: Q(NAME) :- student(NAME,ID), level(ID,YEAR), YEAR ≥ 1, YEAR<4*
- *View: V(VNAME) :- student(VNAME,VID), level(VID,VYEAR), VYEAR > 2, VYEAR ≤4*

No MCD will be created because property 3d is violated. VYEAR>1 ⇒ YEAR ≥ 1 but VYEAR≤4 *⇏* YEAR < 4

After finishing the first step the algorithm eliminates redundant MCDs. Redundant MCDs are those which consider the same view and cover the same subgoals with equal mappings.

The second step is concerned with the combination of MCDs to form rewritings of the query. In order to combine a set of MCDs, the following conditions have to be fulfilled:

1. only combine MCDs which are mutually disjoint subsets of subgoals of the query
2. the set of covered subgoals resulting from the combination of the MCDs yields exactly the query subgoals

Given a set of MCDs for which condition 1 and 2 above hold, then the combination will result in a valid rewriting unless one of the two following cases apply. If one of the following cases applies, some additional work has to be done to assure correctness:

1. **The query contains interpreted predicates**
   If the interpreted predicate contains a variable that is mapped to an existential variable in the view then MiniCon property 3c applies. That means, if a query variable x is mapped to an existential variable in the view, every query predicate that contains x must be covered by the MCD including interpreted predicates. Hence, while creating the MCD and checking the MiniCon property the algorithm will make sure that it covers the relevant interpreted predicate.
   Otherwise, if the variable is mapped to a distinguished variable in the view, simply add the interpreted predicate to the rewriting. Both cases are shown in example 7.
2. **A variable or a constant in the query is mapped to a constant in the view**
   Suppose x is a variable or a constant in the query and there are two MCDs whose mappings contain x. Theses MCDs can only be combined if either they map x to the same constant or one maps x to a constant and the other one maps x to a distinguished variable. Example 8 shows the cases when a variable maps to the same and to a different constant.

Example 7 (query contains interpreted predicate)

- *Query: Q(NAME) :- student(NAME,ID), ID > 5000*
- *View: V(VNAME) :- student(VNAME,VID), VID > 6000*

Variable ID is mapped to existential variable VID. While creating the MCD the MiniCon property will be checked and this will enforce to cover 'ID > 5000' because ID is part of this interpreted predicate.

- *Query: Q(NAME) :- student(NAME,ID), ID > 5000*
- *View: V(VNAME,VID) :- student(VNAME,VID), VID > 6000*

In this example variable ID is mapped to the distinguished variable VID. Thus, interpredicate 'ID > 5000' will not be covered when the MCD is created. That is why it will be added when creating the rewriting.

Example 8 **(a variable or a constant in the query is mapped to a constant in the view)**

- *Query: Q(X) :- e1(X,Y), e2(X,Y)*
- *View 1: V1(A) :- e1(A,23)*
- *View 2: V2(A) :- e2(A,23)*

Rewriting: Q'(x) :- V1(x),V2(x)

For each view an MCD will be created. In both MCDs the variable Y maps to the numerical constant 23. So it is possible to combine both MCDs in order to form the rewriting.

- *Query: Q(X) :- e1(X,Y), e2(X,Y)*
- *View 1: V1(A) :- e1(A,23)*
- *View 2: V2(A) :- e2(A,111)*

This example is similiar to the one above. Again, for each view an MCD is created. This time variable Y is mapped to constant 23 in the MCD of view 1 but to constant 111 in the MCD of view 2. Thus, it is not possible to combine these MCDs to form a rewriting because variable Y maps to different constants.

A possible final step is the elimination of redundant subgoals from a rewriting. Though, this does not affect the number of answers that can be created by the rewriting. The problem of optimizing the rewriting by removing redundant views is addressed in section 5.

2 Design Overview

The project is divided into six Java packages:

- *minicon*
  implementation the MiniCon algorithm
- *datalog*
  representing a Datalog query
- *parser*
  parsing of Datalog and SQL input
- *antlr*
  used by ANTL parser generator
- *converter*
  representing SQL statements and converting SQL to Datalog (respectively, Datalog to SQL)
- *testing*
  JUnit test classes and test cases in form of XML-files (see section 4)

The MiniCon algorithm can be subdivided into the following parts:

1. Reading user input and parsing datalog queries (section 2.1)
2. Representation of parsed input as *DatalogQuery* objects (section 2.2)
3. Representing mappings to variables and constants (section 2.3)
4. Design of classes that implement the actual algorithm (section 2.4)

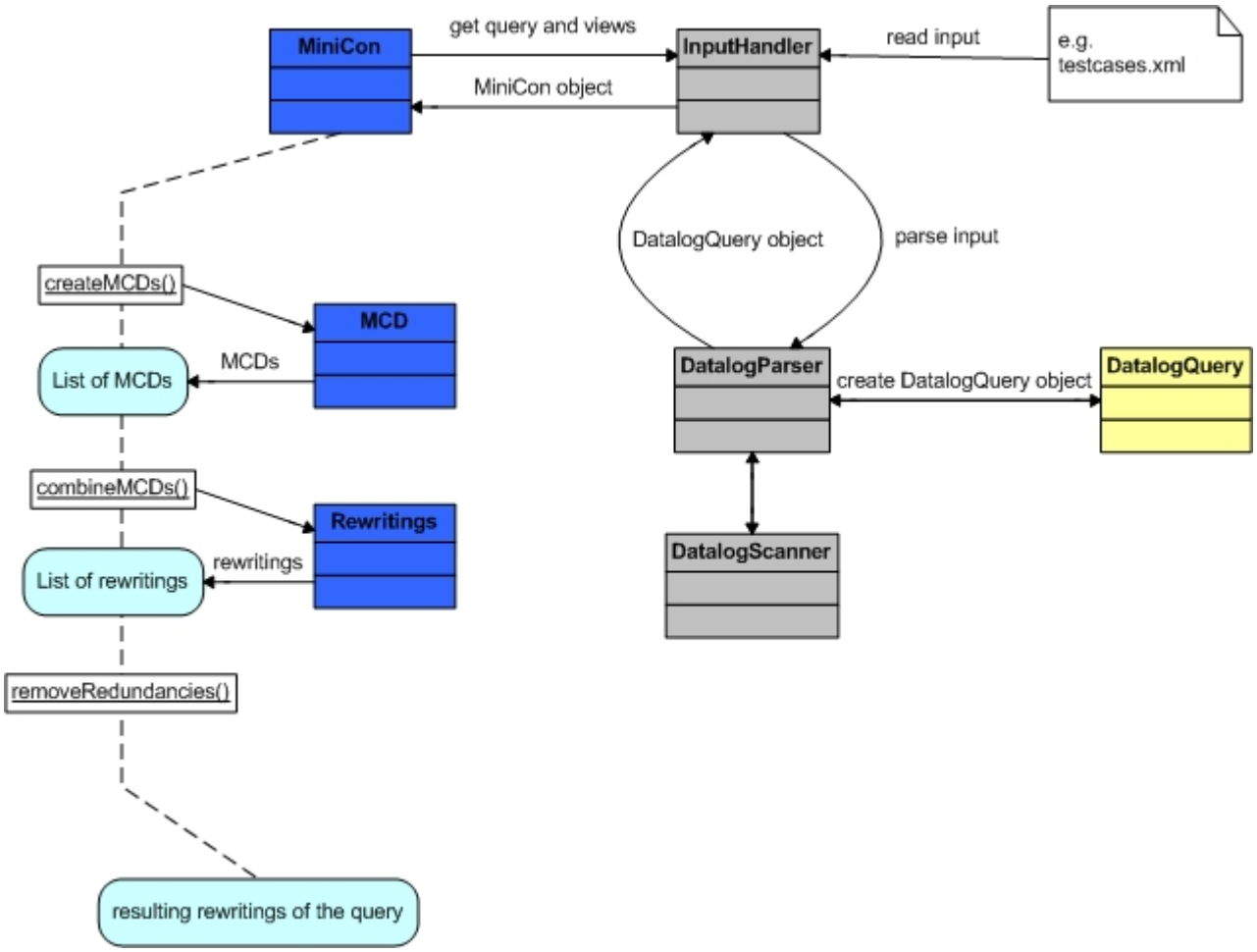


Figure 1: General functioning

Figure 1 depicts the general functioning of the program. The control flow is directed by the principle class *MiniCon*. Class *InputHandler* is used to retrieve a MiniCon object which contains the query and views in form of objects of class *DatalogQuery*. Then the algorithm processes three major steps. First, it creates *Minicon Descriptions (MCDs)* using class *MCD*. The second step involves the combination of MCDs resulting in rewritings of the query represented by objects of class *Rewriting*. The final step removes redundant elements from rewritings.

An UML diagram with classes involved in performing MiniCon can be found in Appendix A. This diagram does not contain classes used by the Datalog-to-SQL converter. This is described in section 5.2.

2.1 User Input Handling and Datalog Query Parser

Any kind of input that is provided by the user is processed by class *InputHandler*. Two different ways of reading the input are possible: 1. Using the command line after starting the program. 2. Using an XML file. The format of the XML input is determined by the XML Schema file which can be found in Appendix B (Datalog queries). The XML file with Datalog test cases is structured in the following way:

- element *testcase* - introduces a new test case
- element *id* - enables to access a test case according to its id
- element *query* - Datalog query

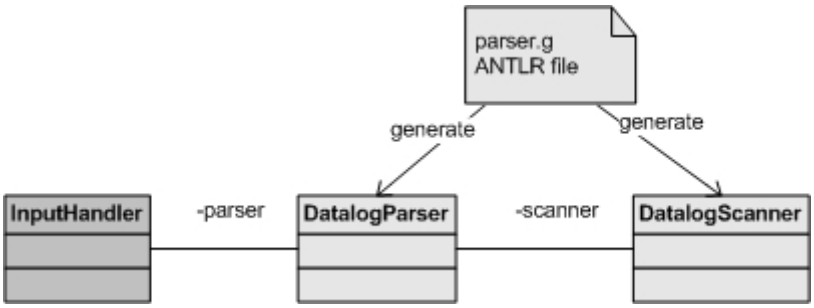


Figure 2: UML Diagram for InputHandler and Parser

- one or more elements *view* - Datalog queries representing the view(s)

Class *InputHandler* uses classes *DatalogParser* to parse the input. First, class *DatalogScanner* turns the input stream into tokens. The parser constructs an object of type *DatalogQuery* representing the query and an object of the same type for each view. Both scanner and parser were created by using the parser generator ANTLR [4]. File *parser.g* contains the syntax definitions used by the scanner and grammar rules used by the parser. Figure 2 depicts the relationship between these three classes and the parser generator file. The complete file is given in Appendix D.

Changing the grammar file does not influence the behavior of the algorithm. The actual program is independent from the grammar rules as a long as the representation of a Datalog query will not be changed.

2.2 Datalog Query Representation

Query and views are represented by objects of class *DatalogQuery* and have the following format:

$$name(headVariables) :- predicate_1, predicate_2, ..., interpretedPredicate_1, interpretedPredicate_2, ...$$

Member variable *name* represents the name of the query head which also contains a list of variables that are stored in *headVariables*. The body contains predicates and interpreted predicates. The latter is represented by class *InterpretedPredicate*. As defined in Definition 4 it consists of a variable, a constant and a comparison operator. Other possible elements of the query body are objects of class *Predicate,* whose design involves further classes. The basic elements of class *Predicate* are *PredicateElement* objects that can either be of type *Variable* or of type *Constant.* Both classes extend class *PredicateElement.* Furthermore, class *Constant* has two subclasses, which are *StringConstant* and *NumericalConstant.* Elements of the query head are of type *Variable.*

Figure 3 shows the structure described above. Class *DatalogQuery* is comprised of *Variable, Predicate* and *InterpretedPredicate*. The figure also points out the inheritance relationships between *PredicateElement, Variable* and *Constant* as well as those between *Constant, StringConstant* and *NumericalConstant.*

2.3 Mapping Representation

The MiniCon Algorithm uses three different kinds of mappings:

1. Mapping from a variable or a constant of the query to a *variable* of the view = *variable mapping*
2. Mapping from a variable or a constant of the query to a *constant* of the view = *constant mapping*

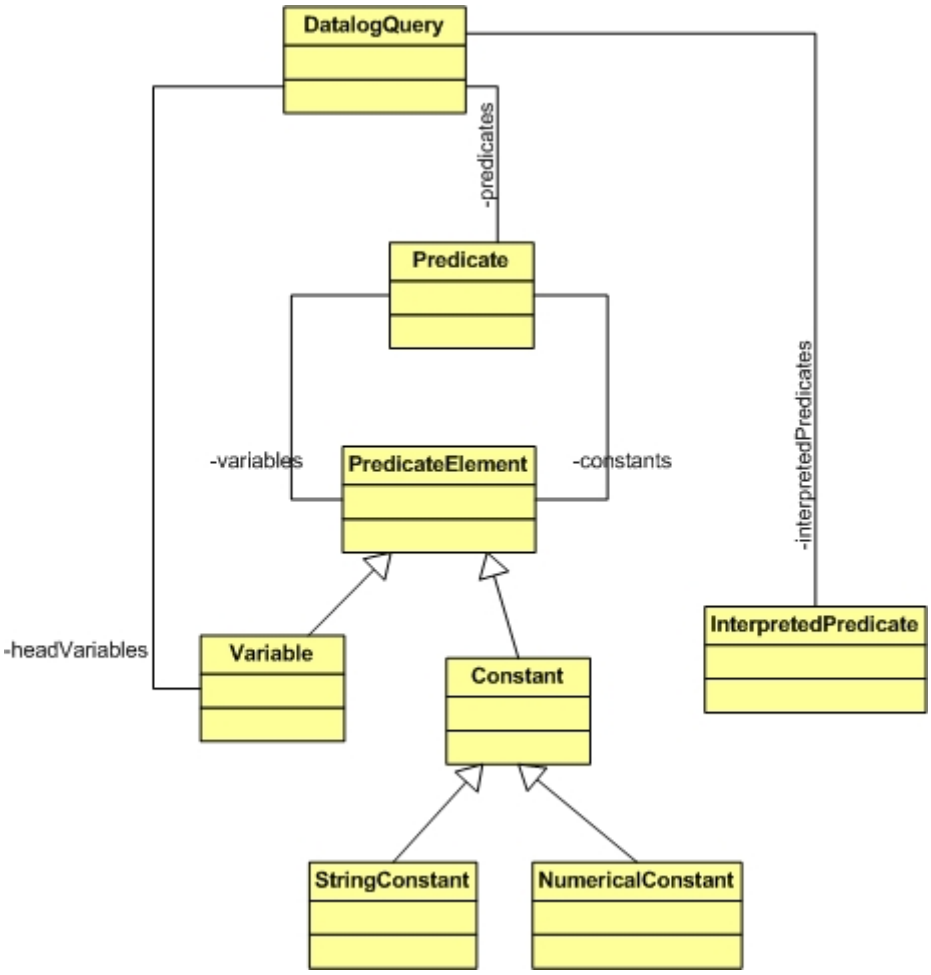


Figure 3: Representation of Datalog Query

3. Mapping from a head variable of the view to a variable or a constant of the query = *rewriting mapping*

The three kinds of mappings stated above are represented in the program as objects of class *Mapping.* This class exhibits the functionality to map elements of type *PredicateElement* to elements of the same type. Class *PredicateElement* is the super class of class *Variable* and class *Constant.* This type hierarchy makes it possible that both variables and constant can part of a mapping. Every mapping has the format *argument* $\rightarrow$ *value*, meaning that the object 'argument' is mapped to the object 'value'. Therefore, the mapping class maintains two generic lists, one for arguments (*argument list*) and the other one for values (*value list*). If there is a mapping from the variable x to the variable y then the argument list will contain the element x and the value list will contain the element y. The index position of the elements will be the same in both lists.

Figure 5 depicts an example that involves two relations *e1* and *e2.* The variable x maps to the variables a and c. The variable y maps to the variables b and d. Because they are mapped to different variables they are contained twice in the argument list.

Class *MCDMappings* consits of the three Mapping objects that are necessary for the algorithm. Object *varMap* is used for the variable mapping, *constMap* is used for the constant mapping and object *rewritingMap* represents the rewriting mapping. The last one is instatiated and used in the last stage of the algorithm when rewritings are created. Class MCDMappings contains methods that provide for functionality that involves variable and constant mappings at the same time. For example, method

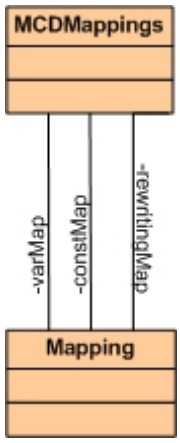


Figure 4: Mapping Representation

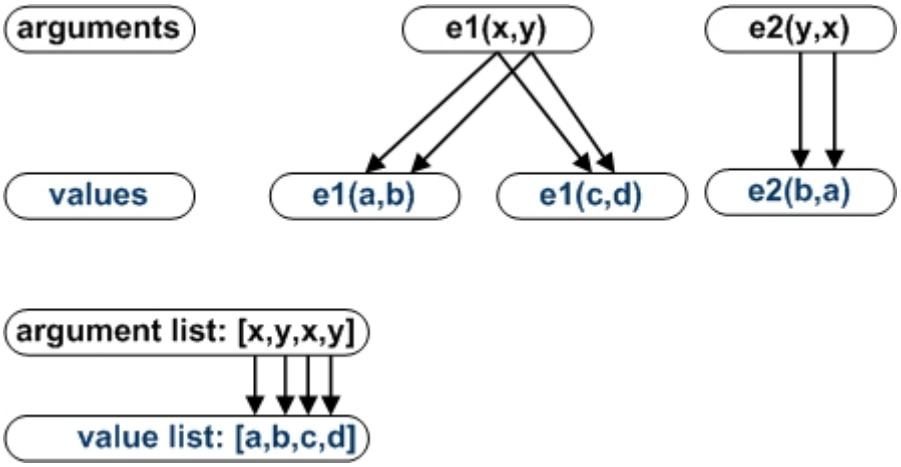


Figure 5: Mapping of variables in the relations e1 and e2 and the resulting argument and value lists

getValues(PredicateElement) returns all values of both mappings.

Figure 4 shows both classes involved in the representation of the mappings. The associations depict the member variables of class MCDMappings *varMap, constMap,* and *rewritingMap* representing the three different mapping types.

2.4 Algorithm Classes

Figure 6: Main classes executing the algorithm

The main class of the algorithm is *MiniCon*. It contains the main method to start the program. It uses class *InputHandler* to obtain parsed user input in form of a MiniCon object. Member variable *query* and a list of *views* are used by the algorithm to compute query reformulations.

The first step of the algorithm is the creation of *MCD* objects. Class *MCD* represents MiniCon descriptions used by the algorithm to obtain rewritings of the query. It contains methods to enforce the

MiniCon property (see Def. 4). Also part of the class is a DatalogQuery object *query* and a DatalogQuery object *view.* Moreover, the class maintains a list of subgoals (*coveredSubgoals*) and a list of interpreted predicates (*coveredInterpretedPredicates*) of the query that are covered by the MCD. All three kinds of mappings (variable, constant, rewriting) are encapsulated in the object *mappings* of type MCDMappings. Class *MCD* contains functionality to extend a MCD in order to make it valid, i.e. it fulfills the MiniCon Property. However, if that is not possible, the algorithm will discard the MCD so that it will not be considered in any further steps.

The second step is concerned with the combination of the MCDs obtained in the first step in order to form rewritings. An object of class *Rewriting* stores all information necessary to express a number of views as a rewriting of the query. That is a list of MCDs, the *query*, and a *list of interpreted predicates.* The union of MCDs results in the actual rewriting of the query. The class constructor sets the rewriting mapping. The member variable *interpretedPreds,* a of list interpreted predicates, contains predicates with comparisons that have not been added by the algorithm but that are necessary to complete a rewriting. These are interpreted predicates in the query that have a variable that maps to a distinguished variable in the view. The MiniCon Property does not enforce the consideration of those predicates in the first place and therefore they are stored explicitly while constructing rewritings. Example 9 shows the behavior of the algorithm in this case. Note that interpreted predicates which contain a variable that maps to an existential view variable are already included in *coveredInterpretedPredicates* contained in the MCD object. The resulting rewriting is represented by an object of class DatalogQuery.

The three main classes involved in the execution of the algorithm are depicted in figure 6. It also shows the member variables *mcds* and *rewritings* that are created during the first and the second step of the algorithm.

Example 9 (Adding interpreted predicates)
- *Query: Q(NAME) :- student(NAME,ID), level(ID,YEAR), YEAR > 2*
- *View: V(VNAME,VYEAR) :- student(VNAME,VID), level(VID,VYEAR)*

Covered subgoals: *student(NAME,ID), level(ID,YEAR)*
Rewriting: *Q(NAME) :- V(NAME,YEAR),YEAR > 2*
Although interpreted predicate YEAR > 2 *is not covered in the first place it will be added to the rewriting.*

The final stage of the algorithm is responsible for removing redundancies. While generating a rewriting the algorithm does not prune redundant subgoals in the first place. The basic implementation eliminates duplicate subgoals from the rewriting. Additional methods to tighten up rewritings are considered in section 5.

3 Implementation Details

The program starts with processing the user input which is done by class *InputHandler.* It returns the object *mc* of class MiniCon which contains the query and the views. As depicted in figure 7 class MiniCon invokes three methods on object *mc* in order to execute the algorithm:

1. *formMCDs()* (subsection 3.1)
2. *combineMCDs()* (subsection 3.2)
3. *removeRedundancies()* (subsection 3.3)

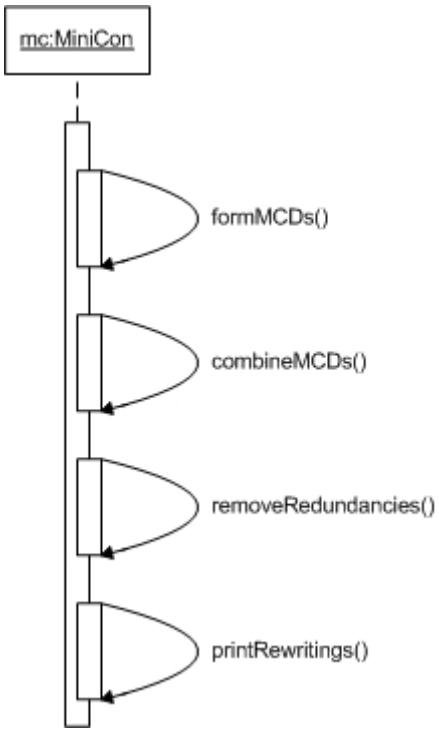


Figure 7: Three steps of MiniCon Algorithm and rewriting print-out

If the algorithm is able to form rewritings of the query using the given views, then the last method call prints out the result.

3.1 Forming MCDs

Method *formMCDs()* creates all possible MCDs for the given query and views. Every subgoal of the query is considered separately. For each subgoal, the algorithm creates all possible mappings to each view. For every valid mapping obtained, an object of class *MCD* is created. Calling method fulfillProperty() on object *mcd* checks whether the MiniCon Property is fulfilled. If necessary the MCD will be extended. If it is valid it will be added to the list of MCDs. Finally duplicate MCDs are removed from the list. Figure 8 shows the behavior of method *fromMCDs()*.

Fulfilling the MiniCon Propery

The most crucial part of the algorithm is performed by method *fulfillProperty()* of class MCD. For each mapping between a query subgoal and a view predicate, the algorithm creates an own MCD and then it invokes that method on the object in order to enforce the MiniCon Property. The following conditions have to be tested:

1. Query constants: a query constant must be mapped either to a distinguished variable or to a constant
   → method *checkQueryConstants()*
2. Valid mapping of head variables: a distinguished variable of the query must be mapped to either a distiguished variable or a constant of the view
   →method *checkHeadVariables()*

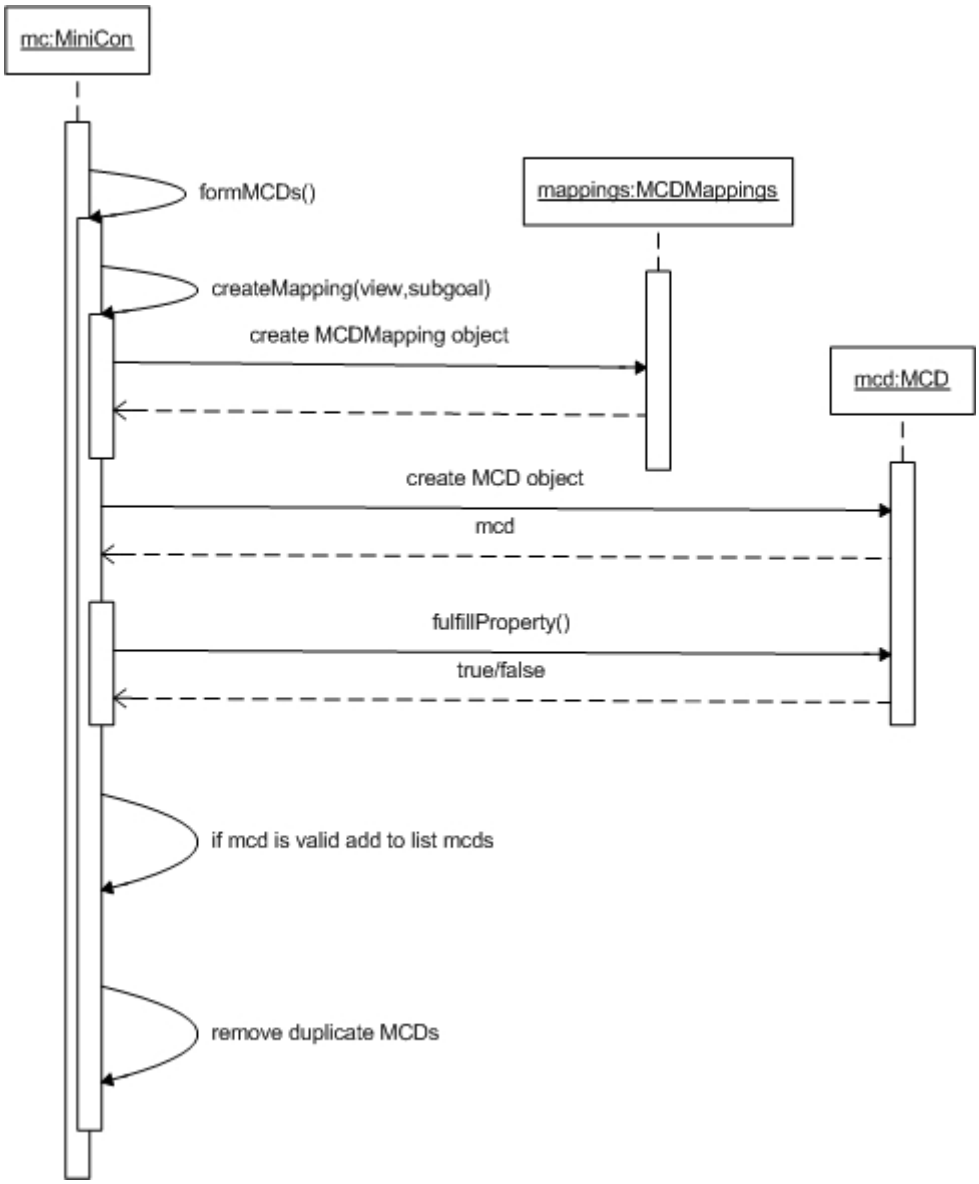


Figure 8: Method formMCDs()

3. Mappings to existential variables: if a query variable is mapped to an existential view variable, then every query subgoal that contains this variable must be covered by the MCD
   → method *coverExistentialVariables()*

4. Interpreted predicates of the query: check if interpreted predicates of the query can be satisfied by the relevant view
   → method *checkInterpretedPredicates()*

Method ***checkQueryConstants():***

This method enforces the *MiniCon property 1c* (definition 4). It checks whether a query constant contained in the mapping is mapped either to the same constant in the view or to a distinguished variable. The method only considers mappings to view variables, i.e. it ensures that constants are mapped to distinguished variables. The other case is already covered by the constant mapping, i.e. if a constant of the query is mapped to a constant of the view this mapping will be part of the constant mapping that is contained in MCDMappings. The equality of both constants has already been checked by the time they were mapped.

Method ***checkHeadVariables():***

This method checks *MiniCon Property 1a* and *2a*. Every distinguished query variable contained in the variable mapping must also be a distinguished variable in the respective view. If a distinguished variable is mapped to a constant then property 2a holds and no further checks have to be done.

Method ***coverExistentialVariables():***

This method enforces *MiniCon Property 1b*. If there is a mapping from a query variable x to an existential variable a in the view, then the MCD must also cover every subgoal that contains x. First of all, the method finds all predicates containing a variable that is mapped to variable a. Then it calls method *extendMapping(Predicate)* in order to recursively extend the mapping to fulfill the property. In more detail, this method does the following:

It tries to extend the existing mapping by adding the predicates provided as arguments. The algorithm will try to extend the existing mapping by adding in the previously mentioned predicates of the query which contain the variable a. Basically, the method extends the list of covered subgoals and then it is tested whether the MCD is still valid.

First, it searches for *mapping partners* which is done by method *findMappingPartners(Predicate)*. These are view predicates which can possibly be mapped to a query subgoal. To be able to map two predicates the following has to be fullfiled:

1. The predicate's names are equal.
2. The number of elements (variables and constants) must be the same.
3. If two constants are to be mapped, then they must be the equal.

If no mapping partner is found it will not be possible to extend the current mapping and thus the property cannot be fulfilled. Otherwise, iterate through all possible mapping partners. By cloning the current mapping, a temporary mapping *oldMap* will be created in order to be able to restore the existing mapping *varMap*. The current mapping partner *mapPartner* is added to the existing mapping. Next, *varMap* is tested for four conditions:

1. Query constants must be mapped to the same constants or to distinguished variables (MiniCon Property 1c).
2. Head variables of the query must also be head variables in the view (MiniCon Property 1a).
3. There are no variables that cannot be equated, i.e. if a query variable is mapped to an existential view variable, it cannot be mapped to any other variable of the view.
4. The current subgoal has not been covered yet, i.e. it is not in the list *coveredSubgoals*.

If one of these conditions cannot be not fulfilled the old mapping will be restored. Finally, if one or more subgoals have been added to *coveredSubgoals*, the method *findPredicates()* will be called to get possible new subgoals that have to be covered. This is necessary because after extending the list of covered subgoals, there are possibly new mappings to existential variables in the view and thus possibly new subgoals to cover.

Example 10

- *Query: Q(NAME,ADDRESS) :- student(NAME,ID), lives(ID,RESIDENCE), location(RESIDENCE,ADDRESS)*
- *View: V(VNAME,VADDR) :- student(VNAME,VID), lives(VID,VRES), location(VRES,VADDR)*

MCD V:

NAME $\rightarrow$ *VNAME*

ID $\rightarrow$ *VID*

RESIDENCE $\rightarrow$ *VRES*

ADDRESS $\rightarrow$ *VADDR*

covered subgoals: student(NAME,ID), lives(ID,RESIDENCE), location(RESIDENCE,ADDRESS)

***Rewriting**: Q(NAME,ADDRESS) :- V(NAME,ADDRESS)*

The query in example 10 gives tuples of student name and address. The view contains the same predicates as the query. Figure 9 shows the sequence diagram for example 10. It includes methods that are involved as well as their return values.

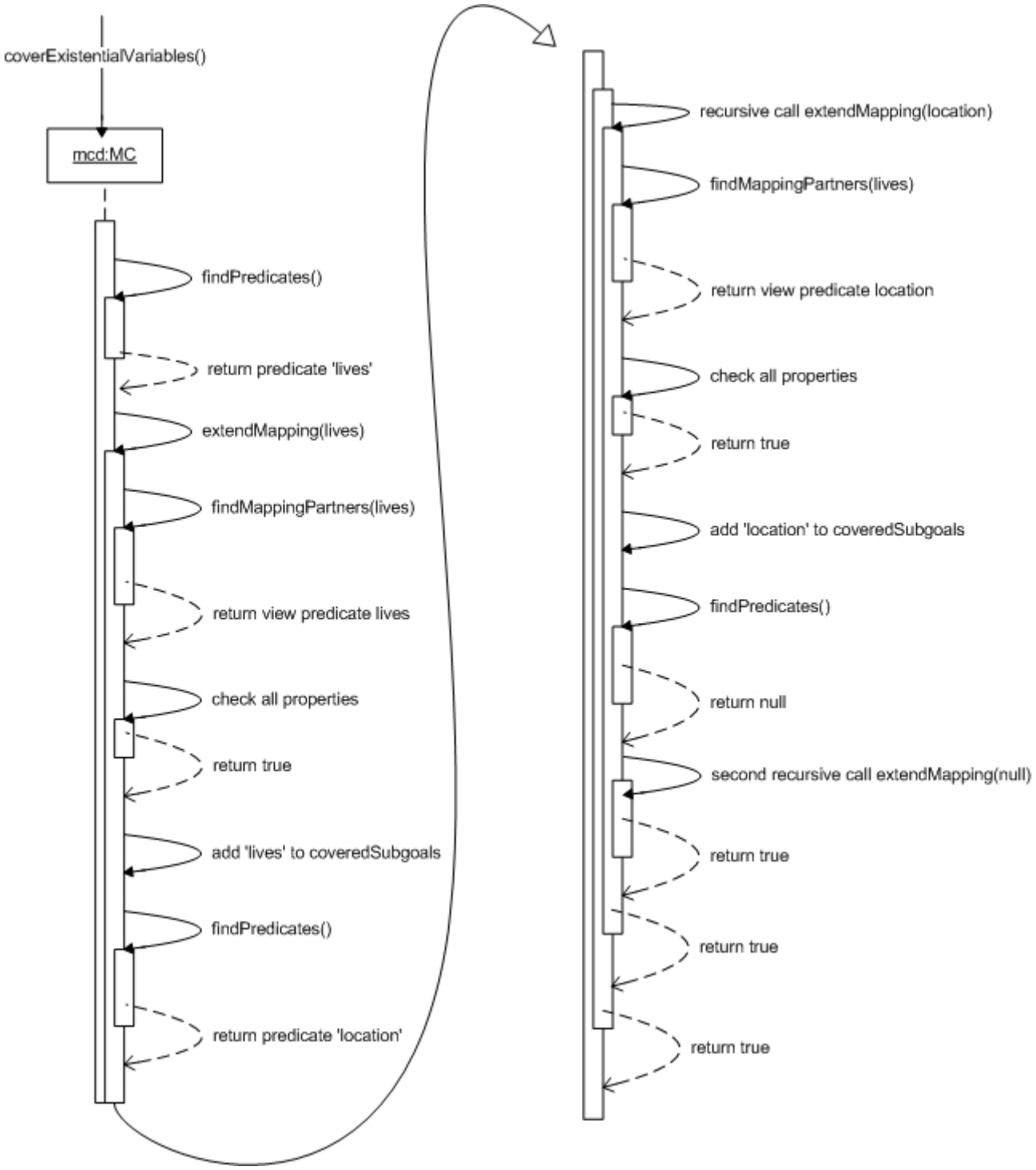


Figure 9: Sequence Diagram for cover existential variables

Covering existential variables in this example is processed as follows:

Method **coverExistentialVariables()** identifies that predicate **lives** needs to be covered (using findPredicates()) because variable **ID** is mapped to the existential variable **VID**. Then **extendMapping** will be called with predicate *lives* as the only element of the list. The method tries to find mapping partners for query predicate *lives* In this case it determines predicate *lives* in the view with variables **VID** and **VRES**. Then the mapping of the current MCD will be extended by the mappings **ID** → **VID** and **RESIDENCE** → **VRES**. It is tested whether the MiniCon Property is still fulfilled for the extended MCD. This is true and thus query predicate **lives** will be added to covered subgoals. Again, method

findPredicates() will be called because by adding new mappings to the MCD, predicate **location** has to be covered as well. Method **extendMapping(List<Predicate>)** will be called recursively with **location** as only element of the argument list. After extending the MCD no more predicates have to be covered and the method will return true.

Method ***checkInterpretedPredicates():***

This method enforces *MiniCon Property 3*. It just considers interpreted predicates in the query that contain a variable which is mapped to an existential variable in the view. If that is the case method *checkComparisionAndCover()* will be called.

The latter method invokes *findViewInterpretedPredicate()* to obtain an interpreted predicate from the view that can be mapped to the argument. If no predicate is found, it will not be possible to satisfy the query predicate. If there is such a view predicate, it must logically entail the query predicate according to *MiniConProperty 3d*. In order to check the implication four cases have to be distinguished:

x is a variable of the query which is mapped to an existential variable and N is a constant. Let a be the existential variable of the interpreted predicate in the view and M the relevant constant. The following constraints have to be satisfied in order to obtain a valid mapping of the interpreted predicates:

- 1. case: $x < N$ or $N > x$
  - if $a < M$ then $M <= N$
  - if $a <= M$ then $M < N$
  - if $M > a$ then $N >= M$
  - if $M >= a$ then $N > M$
- 2. case: $x <= N$ or $N >= x$
  - if $a < M$ then $M <= N$
  - if $a <= M$ then $M <= N$
  - if $M > a$ then $N >= M$
  - if $M >= a$ then $N >= M$
- 3. case: $x > N$ or $N < x$
  - if $a > M$ then $M >= N$
  - if $a >= M$ then $M > N$
  - if $M < a$ then $N <= M$
  - if $M <= a$ then $N < M$
- 4. case: $x >= N$ or $N <= x$
  - if $a > M$ then $M >= N$
  - if $a >= M$ then $M >= N$
  - if $M < a$ then $N <= M$
  - if $M <= a$ then $N <= M$

If the interpreted view predicate fulfills the constraints, the relevant query predicate will be added to the list of interpreted predicates which is part of an MCD object.

Example 11

- *Query: Q(NAME) :- student(NAME,ID), level(ID,YEAR), YEAR > 2*
- *View: V(VNAME) :- student(VNAME,VID), level(VID,VYEAR), VYEAR > 3*

Rewriting: *Q(NAME) :- V(NAME)*

First, method **checkInterpretedPredicate** determines that *YEAR* is part of an existential mapping and calls **checkComparisionAndCover(InterpretedPredicate)**. Interpreted predictate **VYEAR > 3** is detected to be a relevant partner by method **findViewInterpretedPredicate(InterpretedPredicate)**. The mapping between both predicates is valid because the interpreted predicate in the query follows from the corresponding one in the view, i.e. **VYEAR > 3** ⇒**YEAR > 2**.

3.2 Combining MCDs

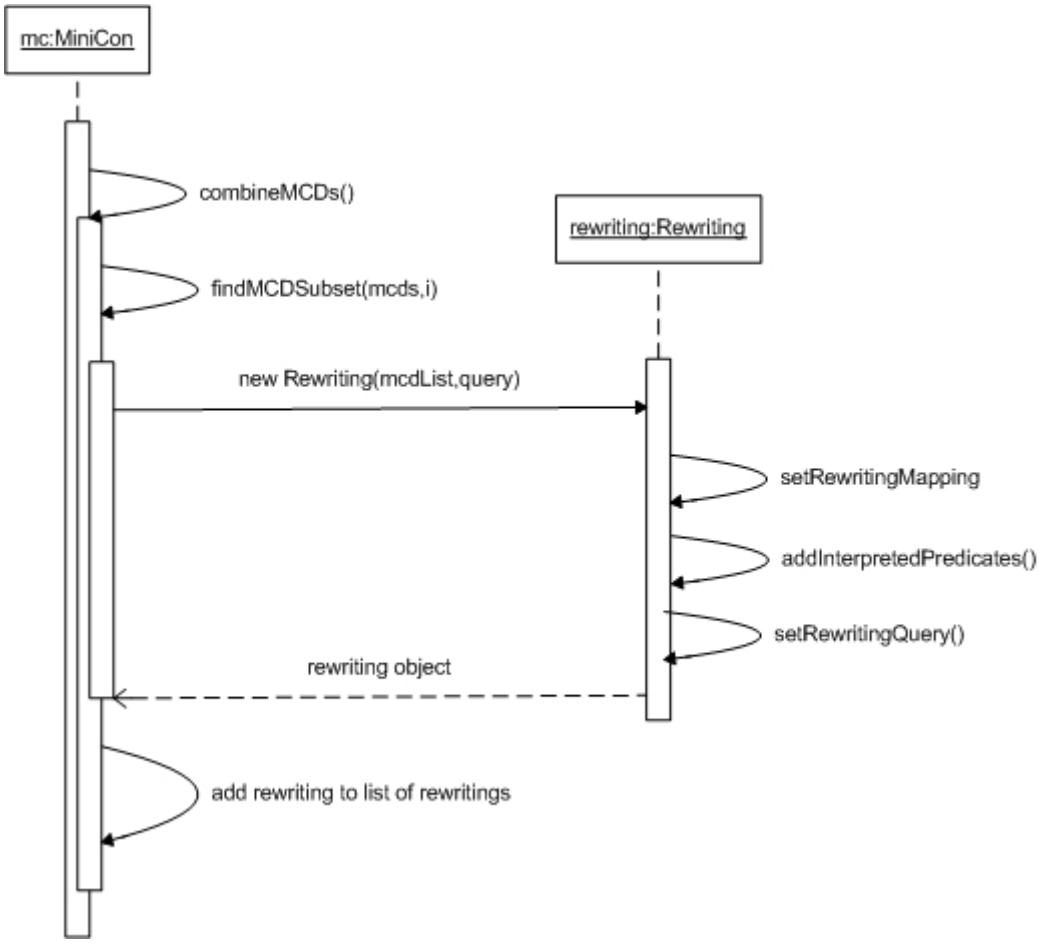


Figure 10: Sequence diagram combining MCDs

After forming MCD objects, the second part of the algorithm takes care of their combination in order to obtain rewritings of the query. Figure 10 shows the sequence diagram depicting methods and classes involved in the process of creating rewritings.

Method *combineMCDs* starts by using a basic algorithm for finding all subsets of a list of MCDs (method *findMCDSubsets*). Beginning with subsets of size one it is tested for each size whether it is possible to

combine the MCDs to a valid rewriting. A rewriting is valid if the union of the view predicates result in the set of query subgoals and if these predicates are pairwise disjoint.

The latter constraints are tested by method *isRewriting.* First the total number of predicates of the given MCDs are computed. If the number doesn't equal the number of subgoals in the query, false will be returned (interpreted predicates are not considered here). Second, every MCD is compared with every other MCD in order to test disjointness. Finally, mappings to constants will be checked for validity. If there are two or more MCDs that have a common variable and this variable is mapped to two different constants, the combination of these MCDs will not be possible.

Once a valid union of MCDs is found, a object of class *Rewriting* will be constructed. The list of MCDs as well as the query are passed as parameters. The constructor contains following methods:

1. *setRewritingMapping()*
   This method maps head variables of views contained in the rewriting to variables or constants of the query. A temporary mapping, denoted *represents*, is maintained. It is a mapping from a query variable or constant to its representative. In most cases the variable or constant is mapped to itself.
   For each MCD part of the rewriting the following will be performed:
   For every pair of *'query element, view variable*' test whether the view variable has already been mapped in the current MCD. Distinguish two cases:
   Case 1 - view variable has not been mapped: if the representative mapping does not contain the current query element add the mapping *query element* → *query element.* Also add *view variable* → *query element* to the rewriting mapping. On the other hand, if there is already a representative for the current query variable, add *view variable* → *representative* to the rewriting mapping.
   Case 2 - view variable has already been mapped: obtain the mapping value (to which the view variable has already been mapped) from the rewriting mapping. Add to the representative mapping *query elem* → *representative.* Also add *view variable* → *represent* to the rewriting mapping.

2. *addInterpretedPredicates()*
   The method adds necessary interpreted predicates to the rewriting. Every interpreted predicate of the query with a variable that is mapped to a distinguished view variable must be added to the rewriting.

3. *setRewritingQuery()*
   This method will create an object of type *DatalogQuery* that represents the actual rewriting of the query. It consists of:
   - the same head variables as the query
   - views contained in the MCDs become predicates of the body
   - rewriting mapping gives mapping from head variables of the view to variables or constants of the query; if a head variable of the view is not covered by the rewriting mapping '_' will be used instead.
   - interpreted predicates obtained by method *addInterpretedPredicates()*

Example 12 shows the creation of a rewriting that contains an interpreted predicate.

Example 12
Query: Q(NAME, YEAR) :- student(NAME,ID), level(ID,YEAR), YEAR < 3
View: V(VNAME, VYEAR) :- student(VNAME,VID), level(VID,VYEAR)
Covered subgoals: *student(NAME,ID), level(ID,YEAR)*
Rewriting: *Q(NAME,YEAR) :- V(NAME,YEAR),YEAR < 3*

Method ***combineMCDs()*** *calls* ***isRewriting()*** *for every subset of covered subgoals. The set that includes both subgoals can be combined to a rewriting. Following steps will be performed when a Rewriting object is created:*

1. ***setRewritingMapping()***
   VNAME → NAME; VYEAR → YEAR
2. ***addInterpretedPredicates()***
   add interpreted predicate YEAR < 3
3. ***setRewritingQuery()***
   head variables: NAME, YEAR; predicates: covered subgoals + interpreted predicates

3.3 Remove redundancies

MiniCon algorithm generates results which are correct rewritings of the query. However, it is often the case that a rewriting contains predicates (views) that are redundant. Removing those predicates from the rewriting does not change the semantics, i.e. the new query yields the same results as the original rewriting.

In the scope of this project one approach to remove redundancies was accomplished. More detailed information of the implementation can be found in section *Extensions of MiniCon* 5.1.

4 Testing

This section shows major test cases and the resulting behavior of the algorithm. It mostly refers to the file *testcases.xml* which can be found in the programs base directory. Each test case shows the query and one or more views. The output of the algorithm consists of the created MCDs containing variable mapping, constant mapping and covered subgoals. Resulting rewritings are printed out last. The id number in parentheses after each test case name indicates the id of the respective test case in the xml file.

The base directory also contains the file *SQLtestcases.xml* which consists test cases in SQL format.

4.1 Basic test cases

Test Case 1 (id 1)

Query: Q(x) :- e1(x,y),e2(y,x)
View: V(a) :- e1(a,b),e2(b,a)
MCD V: mapped to variables: x -> a; y -> b; mapped to constants: none; covered subgoals: e1(x,y), e2(y,x)
Rewriting(s): Q(x) :- V(x)

Query and view are identical. One-to-one mapping from query variables to view variables.

Test case 2 (id 2)

Query: Q(x,y) :- e1(x,y),e2(y,x)
View: V(a,b,d) :- e1(a,b),e2(b,c),e1(b,d)
MCD V: mapped to variables: x -> a; y -> b; mapped to constants: none; covered subgoals: e1(x,y)
MCD V: mapped to variables: x -> b; y -> d; mapped to constants: none; covered subgoals: e1(x,y)

Two MCDs for view V are formed. Both only cover subgoal e1. Subgoal e2 cannot be covered because mapping $y \rightarrow b$ violates MiniCon Property 1a. Therefore, no rewriting is created.

Test case 3 (id 3)

Query: Q1(x) :- cites(x,y),cites(y,x),sameTopic(x,y)
View: V4(a) :- cites(a,b),cites(b,a)
View: V5(c,d) :- sameTopic(c,d)
View: V6(f,h) :- cites(f,g),cites(g,h),sameTopic(f,g)
MCD V6: mapped to variables: x -> f; y -> g; x -> h; mapped to constants: none; covered subgoals: cites(x,y), cites(y,x), sameTopic(x,y)
MCD V5: mapped to variables: x -> c; y -> d; mapped to constants: none; covered subgoals: sameTopic(x,y)
Rewriting(s): Q1(x) :- V6(x,x)

This test case is identical to the one in [1] page 188.

Test case 4 (id 5)

Query: Q(x) :- e1(x,y),e2(y,x),e3(x,z)
View: V1(a,b) :- e1(a,b)
View: V2(a,b) :- e2(a,b)
View: V3(a,b) :- e3(a,b)
MCD V1: mapped to variables: x -> a; y -> b; mapped to constants: none; covered subgoals: e1(x,y)
MCD V2: mapped to variables: y -> a; x -> b; mapped to constants: none; covered subgoals: e2(y,x)
MCD V3: mapped to variables: x -> a; z -> b; mapped to constants: none; covered subgoals: e3(x,z)
Rewriting(s): Q(x) :- V1(x,y),V2(y,x),V3(x,z)

For each subgoal in the query an own MCD is created. The three resulting MCDs are combined to the rewriting.

4.2 Existential Variables

Test case 1 (id 4)

Query: Q(x) :- e1(x,y),e2(y,z),e3(z,x)
View: V(a,c) :- e1(a,b),e2(b,c),e3(b,c)
MCD V: mapped to variables: x -> a; y -> b; z -> c; mapped to constants: none; covered subgoals: e1(x,y), e2(y,z)

If you map query subgoal e1 then e2 and e3 also have to be covered because of MiniCon property 1b. However, it is not possible to map $x \rightarrow c$ because x is in the head of the query and c is existential in the view (MiniCon Property 1a).

Test case 2 (id 6)

Query: Q(x) :- e1(x,y),e2(y,x)
View: V1(a) :- e1(a,b)
View: V2(a,b) :- e1(a,b)
View: V3(b) :- e1(a,b),e2(b,a)
View: V4(a,b) :- e1(a,b),e2(b,a)
MCD V2: mapped to variables: x -> a; y -> b; mapped to constants: none; covered subgoals: e1(x,y)
MCD V4: mapped to variables: x -> a; y -> b; mapped to constants: none; covered subgoals: e1(x,y)
MCD V4: mapped to variables: y -> b; x -> a; mapped to constants: none; covered subgoals: e2(y,x)
Rewriting(s):
Q(x) :- V2(x,y),V4(x,y)
Q(x) :- V4(x,y)

One MCD for V2 and two MCDs for V4 are created. It is not possible to form a MCD for V1 because subgoal e2 cannot be covered. Neither is it possible to form an MCD for V3 because variable a in the view is existential.

4.3 Rewritings

Test case 1 (id 9)

Query: Q(x) :- e1(x,y),e2(y,z)
View: V(a,c,d) :- e1(a,b),e2(b,c),e2(b,d)
MCD V: mapped to variables: x -> a; y -> b; z -> c; mapped to constants: none; covered subgoals: e1(x,y), e2(y,z)
MCD V: mapped to variables: y -> b; z -> d; x -> a; mapped to constants: none; covered subgoals: e2(y,z), e1(x,y)
Rewriting(s):
Q(x) :- V(x,z,_)
Q(x) :- V(x,_,z)

MiniCon generates two MCDs for view V. Both cover the same subgoals but exhibit different mappings. The first one maps z to d. The resulting rewriting maps head variables a and c of the view to x and z, respectively. However, variable d is not part of the mapping and thus '_' is used to fill to empty spot in the view head. Similarly, the second MCD maps z to d. That means c is not contained in the mapping.

Test case 2 (id 10)

Query: Q(x) :- e1(x,y),e2(y,z),e3(z,w)
View: V(x,w,v) :- e1(x,y),e2(y,z),e3(z,w),e2(y,q),e3(q,v)
MCD V: mapped to variables: x -> x; y -> y; z -> z; w -> w; mapped to constants: none; covered subgoals: e1(x,y), e2(y,z), e3(z,w)
MCD V: mapped to variables: y -> y; z -> q; x -> x; w -> v; mapped to constants: none; covered subgoals: e2(y,z), e1(x,y), e3(z,w)
Rewriting(s):
Q(x) :- V(x,w,_)
Q(x) :- V(x,_,w)

This test case is similar to test case 1 (id 9). However, as the view consists of two subgoals for predicate e2 and e3, the algorithm considers more mappings in the first place. In addition to the mappings that are already part of the MCDs the following would also be possible:
x -> x, y -> y, **z -> z**, **z -> q**, w -> v and x -> x, y -> y, **z -> q**, **z -> z**, w -> w
Both contain the mappings z -> z and z -> q. Since variable z in the query is mapped to different existential variables of the view, it is not possible to equate z.

Test case 3 (id 11)

Query: Q(x) :- e1(x,y),e2(y,r)
View: V1(c,d) :- e2(c,d)
View: V2(a) :- e1(a,a)
MCD V2: mapped to variables: x -> a; y -> a; mapped to constants: none; covered subgoals: e1(x,y)
MCD V1: mapped to variables: y -> c; r -> d; mapped to constants: none; covered subgoals: e2(y,r)
Rewriting(s): Q(x) :- V2(x),V1(x,r)

The MCD of V2 and the MCD of V1 are combined to form the rewriting. The former maps variables x and y to variable a of the view. Although in MCD V1 y is also mapped to c, subgoal V1 of the rewriting must still take x as head variable in order to be valid.

Test case 4 (id 12)

Query: Q(x) :- e1(x,y),e2(y,x)
View: V4(a,b) :- e1(a,b),e2(b,a)
MCD V4: mapped to variables: x -> a; y -> b; mapped to constants: none; covered subgoals: e1(x,y)
MCD V4: mapped to variables: y -> b; x -> a; mapped to constants: none; covered subgoals: e2(y,x)
Rewriting(s): Q(x) :- V4(x,y)

This test case shows the elimination of a redundant subgoal in the rewriting. The algorithm creates an MCD for each subgoal in the query. The resulting rewriting contains view V4 twice with the same variables and mappings. Therefore it is possible to remove one subgoal without changing the expressiveness of the rewriting.

4.4 Constants

Test case 1 (id 13)

Query: Q(x) :- e1(x,23),e2(y,x)
View: V(a) :- e1(a,b),e2(b,a)
MCD V mapped to variables: y -> b; x -> a; mapped to constants: none; covered subgoals: e2(y,x)

An rewriting is not created because it is not possible to cover subgoal e1. Doing so would violate MiniCon Property 1c. That is, if a constant is mapped to a variable, then it has to be a distinguished variable.

Test case 2 (id 14)

Query: Q(x) :- e1(x,23),e2(y,x)
View: V(a) :- e1(a,23),e2(b,a)
MCD V: mapped to variables: x -> a; mapped to constants: 23 -> 23; covered subgoals: e1(x,23)

MCD V: mapped to variables: y -> b; x -> a; mapped to constants: none; covered subgoals: e2(y,x)
Rewriting(s): Q(x) :- V(x)

The first MCD includes a constant mapping and covers subgoal e1. Together with the second MCD which covers e2, the rewriting is formed.

Test case 3 (id 15)

Query: Q(x,y) :- e1(x,y),e2(y,x)
View: V(a) :- e1(a,"CONSTANT"),e2(b,a)
MCD V mapped to variables: x -> a; mapped to constants: y -> "CONSTANT"; covered subgoals: e1(x,y)

Mapping *y -> "CONSTANT"* is still valid, even though y is a distinguished variable in the view (MiniCon Property 2a). However, it is not possible to cover subgoal e2 because y is a distinguished variable in the view which would be mapped to the existential variable b.

Test case 4 (id 16)

Query: Q(x,y) :- e1(x,"CONSTANT"),e1(z,y),e2(z,x)
View: V(a,b) :- e1(a,b),e2(b,a)
MCD V: mapped to variables: x -> a; "CONSTANT" -> b; mapped to constants: none; covered subgoals: e1(x,"CONSTANT")
MCD V: mapped to variables: z -> a; y -> b; mapped to constants: none; covered subgoals: e1(z,y)
MCD V: mapped to variables: z -> b; x -> a; mapped to constants: none; covered subgoals: e2(z,x)
Rewriting(s): Q(x,y) :- V(x,"CONSTANT"),V(z,y),V(x,z)

In order to cover both e1-subgoals of the query two separate MCD are formed. One maps "CONSTANT" to the distinguished variable b and the other one maps y to the same variable.

Test case 5 (id 17)

Query: Q(x,y) :- e1(x,y),e2(y,x)
View: V1(a) :- e1(a,"CONSTANTx"),e2(b,a)
View: V2(a) :- e2("CONSTANT",a)
MCD V1: mapped to variables: x -> a; mapped to constants: y -> "CONSTANTx"; covered subgoals: e1(x,y)
MCD V2: mapped to variables: x -> a; mapped to constants: y -> "CONSTANT"; covered subgoals: e2(y,x)

In this test case it is not possible to create a rewriting although the union of both MCDs would cover the query subgoals. Variable y is mapped to different constants and therefore it cannot be equated for the rewriting.

4.5 Interpreted Predicate

Test case 1 (id 18)

Query: Q(x) :- e1(x,y),e2(y,x),x < 23
View: V1(a) :- e1(a,b),e2(b,a)
MCD V1: mapped to variables: x -> a; y -> b; mapped to constants: none; covered subgoals: e1(x,y), e2(y,x)
Rewriting(s): Q(x) :- V1(x),x < 23

Interpreted predicate 'x < 23' is not explictly covered in the MCD. However, since x is mapped to a distinguished variable the predicate can be added to the rewriting.

Test case 2 (id 19)

Query: Q(x) :- e1(x,y),e2(y,x),y < 23
View: V1(a) :- e1(a,b),e2(b,a)
No MCDs created

As opposed to test case 1 (id 18), variable y of the interpreted predicate is mapped to an existential variable in the view. Thus, the interpreted predicate must be covered by the MCD. In that case that is not possible because there is no corresponding interpreted predicate in the view.

Test case 3 (id 20)

Query: Q(x) :- e1(x,y),e2(y,x),y < 23
View: V1(a) :- e1(a,b),e2(b,a),b < 23
MCD V1: mapped to variables: x -> a; y -> b; mapped to constants: none; covered subgoals: e1(x,y), e2(y,x), b < 23
Rewriting(s): Q(x) :- V1(x)

Similary to test case 2 (id 19) variable y is mapped to the existential variable b in the view. In this case the view also contains an interpreted predicate that includes variable b. Hence, the algorithm will cover the interpreted predicate in the view in order to fulfill MiniCon property 3c.

Test case 4 (id 21)

Query: Q2(x) :- inSIGMOD(x),cites(x,y),year(x,r1),year(y,r2),r1 >= 1990,r2 <= 19 85
View: V9(a,s1) :- inSIGMOD(a),cites(a,b),year(a,s1),year(b,s2),s2 <= 1983
View: V10(a,s1) :- inSIGMOD(a),cites(a,b),year(a,s1),year(b,s2),s2 <= 1987
MCD V9: mapped to variables: x -> a; mapped to constants: none; covered subgoals: inSIGMOD(x)
MCD V10: mapped to variables: x -> a; mapped to constants: none; covered subgoals: inSIGMOD(x)
MCD V9: mapped to variables: x -> a; y -> b; r2 -> s2; mapped to constants: none; covered subgoals: cites(x,y), year(y,r2), s2 < 1983
MCD V9: mapped to variables: x -> a; r1 -> s1; mapped to constants: none; covered subgoals: year(x,r1)
MCD V10: mapped to variables: x -> a; r1 -> s1; mapped to constants: none; covered subgoals: year(x,r1)
MCD V9: mapped to variables: y -> a; r2 -> s1; mapped to constants: none; covered subgoals: year(y,r2)
MCD V10: mapped to variables: y -> a; r2 -> s1; mapped to constants: none; covered subgoals: year(y,r2)
Rewriting(s):
Q2(x) :- V9(x,_),V9(x,r1),r1 >= 1990
Q2(x) :- V9(x,_),V10(x,r1),r1 >= 1990
Q2(x) :- V10(x,_),V9(x,_),V9(x,r1),r1 >= 1990
Q2(x) :- V10(x,_),V9(x,_),V10(x,r1),r1 >= 1990

This test case is identical to the one in [1] page 192.

4.6 Other test cases

Test case 1

Query: q(x) :- e(x), x < 10
View: v(a) :- e(a), a > 20

MCD v: mapped to variables: x -> a; mapped to constants: none; covered subgoals: e(x)
Rewriting(s): q(x) :- v(x),x < 10

MiniCon creates a valid rewriting for this test case. As x is mapped to the distinguished variable a, MiniCon Property 3d does not apply. Hence, it is not checked whether the interpreted predicate in the view implies the one in the query. Although the rewriting is valid, it will no containt any answers.

5 Extentions of MiniCon

5.1 Remove Redundancies from Rewritings

The general MiniCon algorithm does not remove redundant subgoals from the rewritings from the start. However, it is possible to prune a query without affecting its semantics. The program has to be started with a special option in order to perform redundancy reduction. This is described in section 1.2.
Note, after running the reduction the query may still contain redundant subgoals. Thus, it is possible to perform further pruning steps which, however, are not addressed in this implementation.

Two subgoals *pred1* and *pred2* of a rewriting are considered to be redundant if

- they have the same predicate name and
- they have an equal number of predicate elements
- if *pred1* (respectively, *pred2*) has variable v at position i then *pred2* (respectively, *pred1*) must have the same variable or an underscore at this position. (Note, in this implementation an underscore means that the particular view variable was not part of the mapping of the MCD, i.e. it was not mapped to a query variable).

The redundancy reduction is performed in class *Rewriting* and involves following methods:

- removeRedundancies()
- insertNonRedundant(List<Predicate>)
- canSubstitutePred(Predicate, Predicate)
- findUnifier(Predicate, Predicate)

Method *removeRedundancies()*

The method iterates through the list of subgoals of the rewriting and uses method *insertNonRedundant* to add non redundant subgoals to a list. In the end the list represents a new set of subgoals of the rewriting with no redundancies. Note, further reductions may still be possible but have not been implemented.

Method *insertNonRedundant(List<Predicate>)*

The method gets a single predicate and a list of predicates as arguments. It iterates through the list and checks whether it is possible to substitute an element of the list with the given predicate. This is tested by using method *canSubstitutePred*. If that is the case, method findUnifier will be called to obtain the unifier between the new predicate and the relevant predicate in the list. In the other case, if the predicate cannot subsitute any of the list members, it will be added as a new element to the list.

Method *canSubstitutePred(Predicate, Predicate)*

This method gets two predicates as arguments which are compared with each other. First of all, if the predicates' names are not equal and the number of predicate elements is not the same, **false** will be returned immediately. Otherwise, the elements of both predicates are tested against each other. **False** will be returned if the predicate elements do not equal and the element of the second predicate is not an underscore ('_').
1. Example: *pred(x,y,z)* - *pred(x,y,_)* → x = x, y = y, z = _ → **true**
2. Example: *pred(x,_,z)* - *pred(x,_,y)* → x = x, _ = _, z != y →**false**

Method *findUnifier(Predicate, Predicate)*

Two arguments are provided for this method: *Predicate pred*, *Predicate oldPred*
Precondition: Predicate *oldPred* can be substituted with Predicate *pred*. The method finds a unifer for the arguments. A new *Predicate* object is created and elements of either *pred* or *oldPred* are added. In case the element of *oldPred* is an underscore ('_'), the relevant element of *pred* is added to the new Predicate object. Otherwise the element of *pred* is chosen to be added.
Example: *pred(x,_,z)* - *pred(x,y,_)* → *pred(x,y,z)*

5.2 SQL-Datalog converter

In order to process SQL queries by the MiniCon algorithm, they have to be converted into Datalog queries. An SQL query can be transformed to an equivalent query in Datalog notation. This can be done automatically by the *SQL-Datalog converter* which classes are located in the package *converter*. Besides the actual converter it also contains classes to represent an SQL query.

SQL query representation

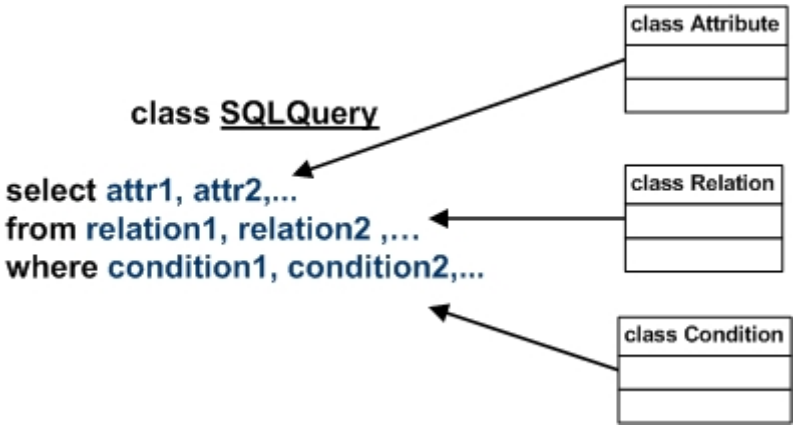


Figure 11: SQL query representation

Generally, an SQL query is given as follows:

select [list of attributes] from [list of relations] where [list of conditions]

The converter needs to be provided with SQL queries of the following format:

select *alias.variable, alias2.variable2, variable3.attribute3,...*
from *relation* ***as*** *alias, relation2* ***as*** *alias2, relation3* ***as*** *alias3,...*
where *condition1, condition2, condition3,...*

The *where clause* is optional. A condition can either involve a constant or a variable on the right side of the comparator sign:

aliasN.variableN = constant or *aliasN.variableN = aliasM.variableM*

The package converter is comprised of four classes to represent an SQL statement. These are:

- class SQLQuery
- class Attribute
- class Relation
- class Condition

Figure 11 depicts how these classes are correlated with the actual SQL query. Class *SQLQuery* exhibits three member variables (*select*, *from*, *where*) which represent the clauses of an SQL query.

Member variable *select* holds a list of Attribute objects. Class Attribute represents elements of the *select clause*. It is composed of the *alias name* of a relation and a *variable* of this relation. The string representation of an Attribute object is: *aliasName.variable*

Member variable *from* holds a list of Relation objects. Class Relation represents elements of the *from clause* of an SQL statement. Member variable *relation* holds the actual relation and member variable *alias* denotes an alias name that is used in the SQL statement. They are separated by the keyword **as** or **AS**.

Member variable *where* holds a list of Condition objects. Class Conditon represents the *where clause* of an SQL query. It is composed of three parts. The left side consists of a relation *alias* and the relevant *variable* (alias.variable). The right side also has a relation *alias* and an object of class *PredicateElement* which can be either a *variable* or a *constant*. However, a constant on the right side of a condition does not belong to a particular relation. In that case the alias part of the right side is just an empty string.

Parsing

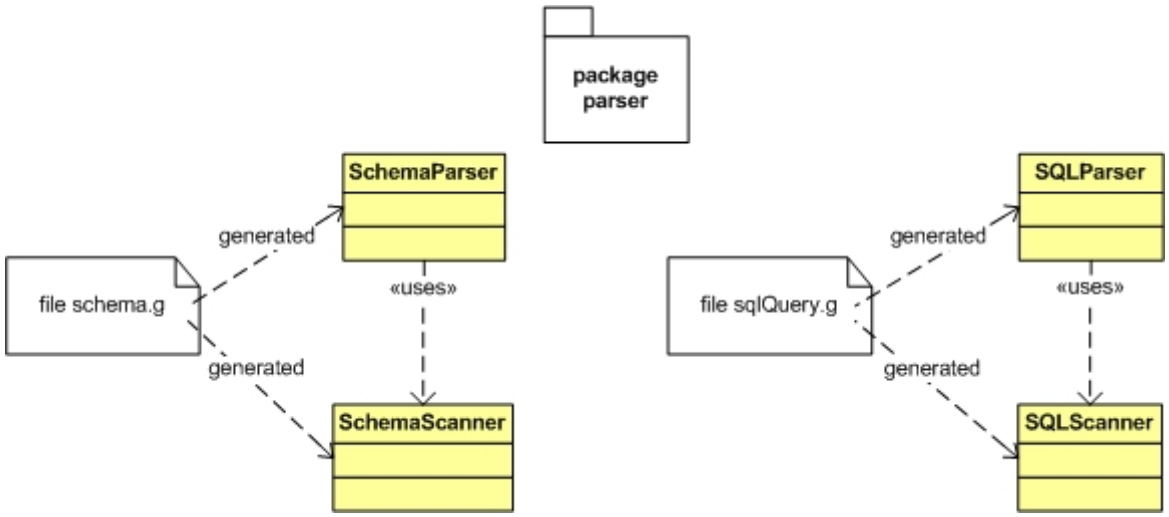


Figure 12: Parser used by class InputHandler to parse a database schema and an SQL query

Similarly to the user input in form of Datalog queries, class *InputHandler* also takes care of SQL input. SQL queries are entered either through the command line or by reading in an XML file. The XML schema definition file can be found in Appendix C.
Package *parser* contains the classes necessary to parse the database schema as well as the SQL query. Both

parsers were created with ANTLR [4] by using the files *schema.g* for the schema parser and *sqlQuery.g* for the SQL parser. Both can be found in the parser package. Figure 12 depicts the files and classes used to parse the input.

Convert SQL to Datalog

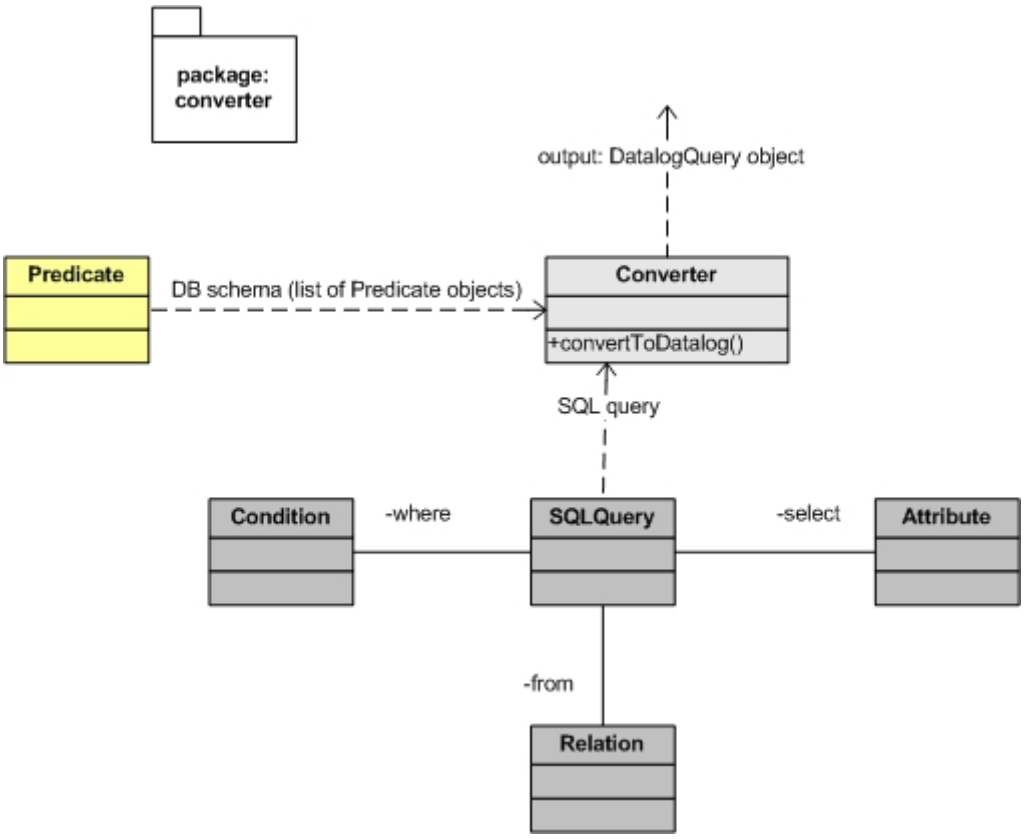


Figure 13: Converting SQL to Datalog

The process of converting an SQL query into a Datalog query is depicted in figure 13. It also shows the UML structure of the SQL query representation.

After parsing the input, class InputHandler calls the converter in order to obtain an object of class Datalog. This object will then be processed by the algorithm in the usual way. The main method converts an SQL query to its Datalog representation is and called:
convertToDatalog(String, List<Predicate>, SQLQuery)
It takes a query name as argument which will denote the head of the resulting query. Also, a database schema has to be provided which contains all relations used in the SQL query. The method instantiates a new DatalogQuery object and uses helper functions to gradually process the conversion:

1. *addHeadVariables(DatalogQuery,List<Attribute>)*
   The method adds head variables to the DatalogQuery object. It takes variables from the select clause of the SQL query which are renamed before they are added.

2. *addBodyPredicates(DatalogQuery, SQLQuery, List<Predicate>)*
   The method adds body predicates to the Datalog query. Relations of the *from clause* of the SQL query are compared with the ones contained in the schema definition. If a match is found, it will be added as Predicate object to the body of the Datalog query. The predicate added to the query

will be named with the relation alias. If no match is found, an error message will be printed out. Then, variables of the relation as defined in the schema will be added to the Datalog predicate. All variable names will have the form *relationAliasName_variableName.*

3. *addConditions(DatalogQuery, SQLQuery)*
   The method incorporates conditions of the *where clause* of the SQL query into the Datalog query. In particular, it does the following:
   • substitute predicate variables with constants: if the right side of the condition is a constant, replace variable with this constant
   • exchange variables: if the condition contains two variables replace the left variable with the right variable or its representatives (method *setRepresentatives* is used)
   • add interpreted predicates: the condition is an unequation which is added to the Datalog query as an interpreted predicate

4. *simplifyHeadVariables(DatalogQuery)*
   The method renames head variables and relevant variables in the body in order to increase legibility. The first variable is named *x1*, the second *x2*, the third *x3,* and so on. Head variables that also appear in the body get the same names.

5. *equatePredicates(DatalogQuery, SQLQuery)*
   The method equates predicate names of the Datalog query which belong to the same relation in the SQL query.
   Example: *select e1.x, e2.y from MyRelation e1, MyRelation e2*
   Resulting Datalog query before predicates have been equated: *q(x1,x2) :- e1(x1,e1_b), e2(e2_a,x2)*
   As *e1* and *e2* are alias names of the same relation (*MyRelation*) they are equated:
   q(x1,x2) :- e1(x1,e1_b), e1(e2_a,x2)

Example 13 shows the step-by-step conversion of an SQL query and an SQL view to their equivalent Datalog representations.

Example 13 *(Convert SQL to Datalog)*

Database schema:
Relation 1: rel1(a,b)
Relation 2: rel2(a,b)
SQL query:
select e1a.a, e1a.b
from rel1 *as* e1, rel1 *as* e1a, rel2 *as* e2
where e2.b = e1.a *and* e1.b = "CONSTANT" *and* e1a.a = e2.a
SQL view:
select e1.a, e1.b *from* rel1 *as* e1, rel2 *as* e2 *where* e2.b = e1.a *and* e2.a = e1.b

1. step - add head variables:
SQL query: Q(e1_a,e1a_b) :-
SQL view: V1(e1_a,e1_b) :-

2. step - add body predicates:
SQL query: Q(e1_a,e1a_b) :- e1(e1_a,e1_b),e1a(e1a_a,e1a_b),e2(e2_a,e2_b)
SQL view: V1(e1_a,e1_b) :- e1(e1_a,e1_b),e2(e2_a,e2_b)

3. step - add conditions:
SQL query: Q(e1_a,e1a_b) :- e1(e1_a,"CONSTANT"),e1a(e2_a,e1a_b),e2(e2_a,e1_a)
SQL view: V1(e1_a,e1_b) :- e1(e1_a,e1_b),e2(e1_b,e1_a)

4. step - simplify head variables:
SQL query: Q(x1,x2) :- e1(x1,"CONSTANT"),e1a(e2_a,x2),e2(e2_a,x1)
SQL view:V1(x1',x2') :- e1(x1',x2'),e2(x2',x1')

5. step - equate predicates:
SQL query: Q(x1,x2) :- e1(x1,"CONSTANT"),e1(e2_a,x2),e2(e2_a,x1)
SQL view: V1(x1,x2) :- e1(x1,x2),e2(x2,x1)

Convert Datalog to SQL

The output after running MiniCon will be an object of class *Rewriting*. The actual rewriting of the Datalog query is held in a member variable of this object and its type is *DatalogQuery*. Besides converting an SQL statement to a Datalog query, it is possible to do the conversion in the other direction which is shown in figure 14. Class InputHandler will use the converter method *convertToSQL(Rewriting)* to obtain objects of class SQLQuery, provided that after running MiniCon the user requests a printout SQL statements for the resulting rewritings. After instantiating a new SQLQuery object, the method invokes the following helper methods:

1. *addSelectClause(SQLQuery, DatalogQuery)*
   The method adds attributes to the *select clause* of the SQL query. Head variables of the Datalog query are the ones to be added to the *from clause*. It creates a new Attribute object which has to be provided with an *alias name* of the relevant relation and the *variable name*. In order to find out

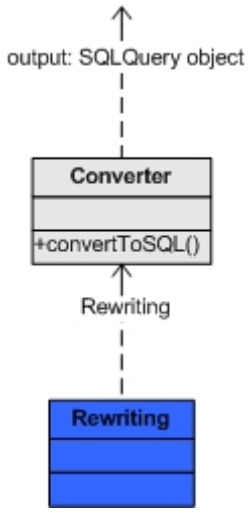


Figure 14: Converting Datalog to SQL

the alias name, it iterates through the list of Datalog predicates. The alias name is determined by the position of the predicate in the Datalog query. That means, the first predicate has alias *e1*, the predicate at the second position has alias *e2* and so on.

2. *addFromClause(SQLQuery, DatalogQuery)*
   The method adds relations to the *from clause* of the SQL query. It iterates through the list of predicates of the Datalog query. For each predicate it creates an object of type Relation. The alias name for each relation is *e1* for the first relation, *e2* for the second one and so on. This convention for alias naming agrees with the one used in method *addSelectClause*.

3. *addWhereClause(SQLQuery, DatalogQuery)*
   The method adds conditions to the *where clause* of the SQL query. In order to access the head variables of the views (i.e. subgoals in rewriting) following schema is assumed: *ViewName(x1,x2,x3,...)*. The method is split up in three parts. Both, part one and two are both performed in the first loop of the method. For each part conditions in form of objects of class *Condition* are added to the SQL query.

   1. Constants in the views are added as conditions
   Example: Rewriting: Q'(x) :- V(x,23) → SELECT e1.x from V AS e1 WHERE e1.x2 = 23

   2. Variables in the views are renamed. If variable i in the rewriting is named xi (according to the schema), nothing will be done. Otherwise a condition will be added to rename the variable in the view.
   Example:
   Q'(x1,x2) :- V1(x1,e1_b), V2(x1,x2) → SELECT e1.x1, e2.x2 FROM V1 AS e1, V2 AS e2 WHERE e1.x2 = e1_b

   3. For each interpreted predicate of the Datalog query, a condition is added to the SQL query.
   Example:
   Q'(x1) :- V1(x1,x2), V2(x1,x2), x2 > 23 → SELECT e1.x1, e2.x2 FROM V1 AS e1, V2 AS e2 WHERE e1.x2 > 23

Example 14 shows the step-by-step conversion a Rewriting to an SQL query.

Example 14 *(Convert Datalog to SQL)*

Rewriting:
Q(x1,x2) :- V1(x1,"CONSTANT"),V1(e2_a,x2),V1(x1,e2_a)

1. step - add select clause:
SELECT e1.x1, e2.x2
FROM

2. step - add from clause:
SELECT e1.x1, e2.x2
FROM V1 AS e1, V1 AS e2, V1 AS e3

3. step - add where clause:
SELECT e1.x1, e2.x2
FROM V1 AS e1, V1 AS e2, V1 AS e3
WHERE e1.x2 = "CONSTANT", e2.x1 = e2_a, e3.x2 = e2_a

5.3 Possible Future Work

The subsection gives suggestions about further work on MiniCon which is not within the scope of this project.

- *Extension of MiniCon in order to answer queries using views in the context of query optimization*
  In this context the goal is to achieve the cheapest rewriting. This rewriting may include subgoals which do not contribute to the logical correctness but optimize the query. [1] shows two approaches to extend MiniCon. First, a modification of *GMAP* uses MCDs to improve this algorithm. It reduces cost but does not necessarily find an optimal rewriting. The second approach is a heuristic algorithm. It considers views which are not needed for logical correctness but lead to an optimal query.

- *Interpreted predicates*
  This implementation can only handle interpreted (built-in) predicates with the shape:
  $x_k < x_l$ or $x_k \leq x_l$ where either x_k is a **variable** and x_l is a **constant** or x_k is a **constant** and x_l is a **variable**.
  The algorithm is guaranteed to be completeness for these cases. It has to be found out which other cases still guarantee complete and which not.

- *Multi-query execution*
  Extend MiniCon to the context of multi-query execution. That is, a number of queries are being executed simultaneously. Combine MiniCon with other techniques for multi-query optimization.

References

[1] Rachel Pottinger and Alon Y. Halevy: MiniCon: A Scalable Algorithm for Answering Queries Using Views. VLDB Journal 10(2-3) (special edition on best of VLDB 2000), 2001. pp. 182-198.

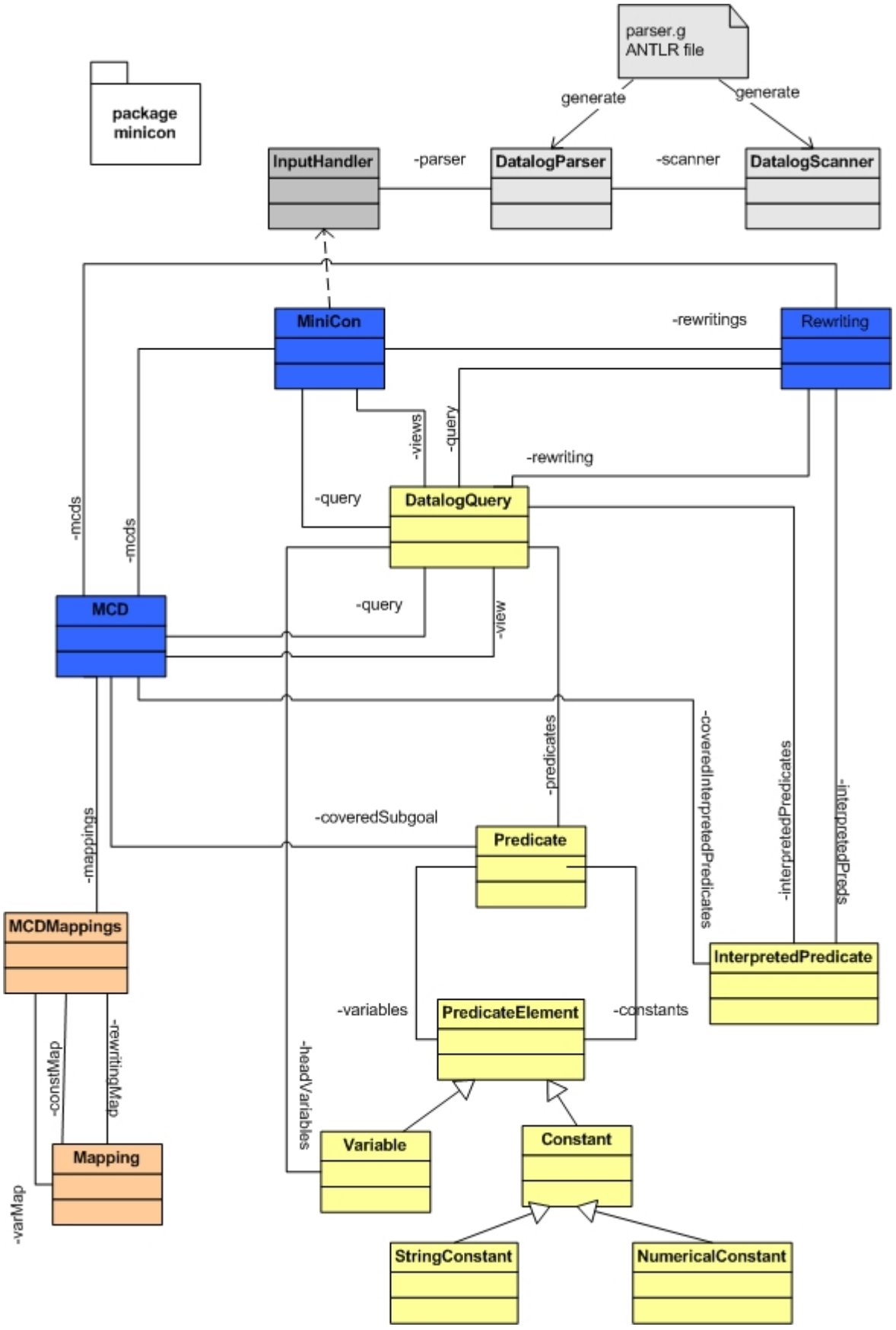
[2] http://data.cs.washington.edu/integration/minicon/

[3] Levy A.Y., Mendelzon A.O., Sagiv Y., Srivastava D. Answering queries using views. In: PODS, pp. 95104, 1995

[4] ANTLR Parser Generator - www.antlr.org

[5] Regression testing framework - www.junit.org

A Complete Class Diagram



B XML Schema for Datalog test cases

File *testsDefinition.xsd*

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
  <xs:element name="testcases">
    <xs:complexType>
      <xs:sequence maxOccurs="unbounded">
        <xs:element name="testcase" type="testcaseType"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:complexType name="testcaseType">
    <xs:sequence>
      <xs:element ref="id"/>
      <xs:element ref="query"/>
      <xs:choice minOccurs="0" maxOccurs="unbounded">
        <xs:element ref="view"/>
      </xs:choice>
      <xs:choice minOccurs="0" maxOccurs="unbounded">
        <xs:element ref="information"/>
      </xs:choice>
    </xs:sequence>
  </xs:complexType>
  <xs:element name="id" type="xs:integer"/>
  <xs:element name="query" type="xs:string"/>
  <xs:element name="view" type="xs:string"/>
  <xs:element name="information" type="xs:string"/>
</xs:schema>
```

C XML Schema for SQL test cases

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
  <xs:element name="testcases">
    <xs:complexType>
      <xs:sequence maxOccurs="unbounded">
        <xs:element name="testcase" type="testcaseType"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:complexType name="testcaseType">
    <xs:sequence>
      <xs:element ref="id"/>
      <xs:element name="DBschema" type="DBschemaType"/>
      <xs:element ref="SQLquery"/>
      <xs:choice minOccurs="0" maxOccurs="unbounded">
```

```
      <xs:element ref="SQLview"/>
    </xs:choice>
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:element ref="information"/>
    </xs:choice>
  </xs:sequence>
 </xs:complexType>
 <xs:complexType name="DBschemaType">
  <xs:sequence>
    <xs:element ref="relation" maxOccurs="unbounded"/>
  </xs:sequence>
 </xs:complexType>
 <xs:element name="id" type="xs:integer"/>
 <xs:element name="relation" type="xs:string"/>
 <xs:element name="SQLquery" type="xs:string"/>
 <xs:element name="SQLview" type="xs:string"/>
 <xs:element name="information" type="xs:string"/>
</xs:schema>
```

D ANTLR file: parser.g

```
header
{
  package minicon;
  import java.util.ArrayList;
  import java.util.List;
}

class DatalogParser extends Parser;
options {defaultErrorHandler=false;k=2;}
{
  private DatalogQuery query = new DatalogQuery();
  private List<PredicateElement> tempPredElems = new ArrayList<PredicateElement>();
}

query returns [DatalogQuery quer = null] : head COLONDASH body (PERIOD)? (NEWLINE)?
{
  quer = query;
};

head : n:VARIABLE LPAREN head_variables RPAREN
{
  query.setName(n.getText().trim());
};

head_variables : head_var (COMMA head_var)* ;

head_var : n:VARIABLE
{
  String name = n.getText().trim();
```

```
  query.addHeadVariable(new Variable(name));
};

body : predicate (COMMA predicate)*;

predicate : (regular_pred | interpreted_pred);

regular_pred : n:VARIABLE LPAREN vars_or_cons RPAREN
{
  String name = n.getText().trim();
  Predicate pred = new Predicate(name);
  pred.addAllElements(tempPredElems);
  tempPredElems.clear(); query.addPredicate(pred);
};

interpreted_pred : (const_compar_var | var_compar_const);

const_compar_var : constant c:COMPARISON variable
{
  PredicateElement left = tempPredElems.get(0);
  PredicateElement right = tempPredElems.get(1);
  String comparator = c.getText();
  InterpretedPredicate pred = new InterpretedPredicate(left,right,comparator);
  tempPredElems.clear();
  query.addInterpretedPredicate(pred);
};

var_compar_const : variable c:COMPARISON constant
{
  PredicateElement left = tempPredElems.get(0);
  PredicateElement right = tempPredElems.get(1);
   String comparator = c.getText();
   InterpretedPredicate pred = new InterpretedPredicate(left,right,comparator);
   tempPredElems.clear(); query.addInterpretedPredicate(pred);
};

vars_or_cons : (variable | constant) (COMMA (variable | constant))*;

variable : n:VARIABLE
{
  String name = n.getText().trim();
  tempPredElems.add(new Variable(name));
};

constant : (string_constant | numerical_constant);

string_constant : n:STRING_CONST
{
  String name = n.getText().trim();
  tempPredElems.add(new StringConstant(name));
};

numerical_constant : n:NUMERICAL_CONST
{
  String name = n.getText().trim();
```

```
  tempPredElems.add(new NumericalConstant(name));
};

class DatalogScanner extends Lexer;
options {defaultErrorHandler=false;k=2;}

COLONDASH: ":-";
LPAREN: '(';
RPAREN: ')';
COMMA : ',';
PERIOD: '.';

NUMERICAL_CONST : (DIGIT)+;

VARIABLE: ( LETTER | DIGIT)+ ;
LETTER: (LOWERCASE | UPPERCASE);

QUOTE : "" | """;
STRING_CONST : QUOTE VARIABLE QUOTE;

COMPARISON : ('<' | '>' | "<=" | ">=") ;

LOWERCASE : 'a'..'z';
UPPERCASE : 'A'..'Z';
DIGIT : '0'..'9';

NEWLINE :
  '\r' '\n' // DOS
  | '\n' // UNIX ;

WS : (' ' | '\t' | '\n' | '\r')
{
  _ttype = Token.SKIP;
};
```