High Level Database Interface with Application to GIS

Mads Johnsen, s991179

LYNGBY 2005
M.Sc. Thesis
NR. 22/05

IMM

Printed at IMM, DTU

Abstract

This thesis presents a high level language HLCL (High Level Constraint Language) which facilitates the formulation of constraints imposed on a relational database. The language has been designed with a syntax very similar to natural language and has been developed to be as intuitive as possible. It is based on the so-called "Peirce Product" known from algebraic logic, which gives clear and unambiguous semantics. The language is intended to help trained domain specialists, who are not necessarily logic specialists, formulate constraints correctly.

Constraints are formulated in HLCL on the basis of a conceptual model (E/R-diagram), which has a specified map into the actual database schema. The specification of the conceptual model, database schema and the mapping between the two are referred to as the database model.

The thesis also describes a compiler system, which given a database model can compile constraints formulated in HLCL to a directly executable SQL query and well-defined DATALOG$^{\neg}$ interface. A proof-of-concept compiler system is implemented in PROLOG and tested with actual constraints from the Geographic Information Domain.

Resumé

Dette speciale præsenterer et højniveau sprog HLCL (High Level Constraint Language) med henblik på at formulere constraints på en relationel database. Sproget er designet med en syntaks meget tæt på naturligt sprog og er udviklet til at være så intuitivt som muligt. Sproget er baseret på det såkaldte "Peirce Product" kendt fra algebraisk logik, som giver en klar og entydig semantik. Sproget er tænkt som en hjælp til trænede domænespecialister, der ikke nødvendigvis er specialister i logik, så constraints formuleres korrekt.

Constraints er formuleret i HLCL på basis af en konceptuel model (E/R-diagram), der har en specificeret afbildning til databasens tabeller. Specifikationen af den konceptuelle model, databasens tabeller og afbildningen mellem de to bliver samlet referet til som database modellen.

Specialet beskriver også et oversættersystem, der givet en database model kan kompilere constraints formuleret i HLCL til en direkte eksekverbar SQL forespørgsel og en veldefineret DATALOG$^{\neg}$ grænseflade. Et "proof-of-concept" kompiler system er udviklet i PROLOG og testet med faktiske constraints fra det geografiske domæne.

Preface

The enclosed report constitutes the M.Sc. Thesis by Mads Johnsen. The project was carried out at the Computer Science and Engineering division (CSE), Institute of Informatics and Mathematical Modelling (IMM) at the Technical University of Denmark (DTU). The duration of the project has been six months from the 1^{st} of October 2004 to the 31^{st} of March 2005, and corresponds to a workload equivalent of 30 ECTS points. The work has been supervised by Professor Jørgen Fischer Nilsson and Associate Professor Hans Bruun.

I would like to thank my supervisors for valuable guidance and constructive criticism during the project. I would also like to thank Jesper Vinther Christensen, Per Larsen and Thomas Bolander for valuable input and support during my work with the thesis.

Kgs. Lyngby, April 5, 2005

Mads Johnsen

Contents

1 Introduction **1**
- 1.1 Reading Guide . . . 2
- 1.2 Thesis Structure . . . 5
- 1.3 Abbreviations . . . 6

2 Background **9**
- 2.1 Database Constraints . . . 9
- 2.2 Application Domain . . . 10

3 General Idea **13**
- 3.1 The HLCL System . . . 13
- 3.2 The Conceptual Model . . . 14
- 3.3 HLCL Syntax . . . 15
  - 3.3.1 Informal Description . . . 16
  - 3.3.2 Macro Functionality . . . 25
- 3.4 Language Design Decisions . . . 26
  - 3.4.1 HLCL Expressions and Intuition . . . 28
- 3.5 Formal Description . . . 29
- 3.6 Model Theoretic Semantics . . . 31

4 Running Examples **37**
- 4.1 Conceptual Model . . . 38
- 4.2 Constraint Examples . . . 38

5 Related Work **43**
- 5.1 Colan . . . 44
- 5.2 Description Logic . . . 46
- 5.3 OCL . . . 47
- 5.4 EER . . . 50
- 5.5 Concluding Remarks . . . 50

6 From HLCL to Predicate Logic **53**
- 6.1 Macro Functionality . . . 53
- 6.2 Simple Translating Strategy . . . 53

6.3 Advanced Translating Strategy . . . 58
6.3.1 Issues . . . 58
6.3.2 Strategy . . . 59
6.4 Full Translating Strategy . . . 65

7 Intermediate Steps **69**
7.1 Interface Definitions . . . 70
7.1.1 Extended DATALOG . . . 70
7.1.2 DATALOG$^\neg$ . . . 71
7.2 From Predicate Logic to Extended DATALOG . . . 71
7.3 From Extended DATALOG to DATALOG$^\neg$ . . . 73
7.4 Attributes in DATALOG$^\neg$ . . . 76

8 Database Representation **79**
8.1 Formal Description . . . 79
8.2 Informal Description . . . 81
8.2.1 Conceptual Model . . . 81
8.2.2 Database Model . . . 83
8.2.3 Mapping . . . 84
8.3 Wellformedness . . . 87
8.4 Shortcomings of Representation . . . 87

9 From Extended DATALOG to SQL **89**
9.1 Issues . . . 89
9.1.1 Expressiveness . . . 90
9.1.2 Safety . . . 90
9.2 Translation Strategy . . . 91
9.2.1 ISA-structures . . . 99

10 Implementation **103**
10.1 PROLOG and Logic Programming . . . 103
10.2 Definite Clause Grammars . . . 104
10.3 λ-Calculus in PROLOG . . . 104
10.4 Variables . . . 105

11 Future Work **107**
11.1 Improvements of Current System . . . 107
11.1.1 Optimizations of Queries . . . 107
11.1.2 Expressiveness . . . 108
11.1.3 Optimization of Database Representation . . . 108
11.2 Other Applications of Current System . . . 108
11.2.1 As a Basis for Yet Another Interface . . . 108
11.2.2 Deducting HLCL Constraints . . . 108
11.2.3 Logical Relationships Between Constraints . . . 109

12 Conclusion **111**

Bibliography **113**

A Concepts Explained **119**
A.1 Constraints in the E/R-model . . . 119
A.2 Anaphora and "Donkey-sentences" . . . 120
A.3 SQL . . . 121
A.3.1 Spatial Data and SQL . . . 122
A.4 Problematic Representations . . . 124
A.4.1 ISA-structures . . . 124
A.4.2 Mapping Weak Entity Sets . . . 126
A.4.3 Computed Relations . . . 127
A.5 λ-Calculus . . . 128
A.6 Skolem Functions . . . 129

B Detailed Implementation **131**
B.0.1 Notation . . . 131
B.0.2 Coding Convention . . . 132
B.0.3 Interfaces . . . 133
B.1 Overview - HLCL to SQL/DATALOG$^\neg$ . . . 133
B.2 HLCL to Predicate LogicTranslation . . . 134
B.3 Wellformedness Checking . . . 137
B.4 Intermediate Steps . . . 139
B.5 Extended DATALOG to SQL . . . 140
B.6 Extended DATALOG to DATALOG$^\neg$ . . . 141
B.7 Settings . . . 143
B.8 Other Predicates . . . 144
B.8.1 I/O Predicates . . . 144
B.8.2 Test Predicates . . . 144
B.9 Auxiliary Predicates . . . 144

C Userguide and CD Contents **149**
C.1 CD Contents . . . 149
C.2 User Guide . . . 149
C.2.1 Installing the HLCL System . . . 149
C.2.2 Running the HLCL System . . . 150

D Test Cases **151**
D.1 Overview . . . 151
D.2 Actual Tests . . . 157

E Sourcecode **191**
E.1 HLCL-system 191
E.2 User Settings 231
E.3 Help Functions 236
E.4 Test Functions 239

List of Figures

1.1 Walkthrough Conceptual Model 3
1.2 Walkthrough Database Schema 4

3.1 An overview of the HLCL System 14
3.2 Two Classes 16
3.3 A simple relation 17
3.4 A path through relations 17
3.5 Compound Relations 20
3.6 An example conceptual model for illustrating use of brackets 22

4.1 The domain for the Constraint Examples 38

5.1 The domain for the OCL constraints 48

8.1 Conceptual Model Example 81
8.2 Database Tables corresponding to figure 8.1 82

A.1 E/R Diagram Constraint Examples 119
A.2 Topological Relations 123
A.3 Topological Classes 124
A.4 An example ISA-structure 125
A.5 A Conceptual Model Fragment having Weak Entity Sets . . . 126

B.1 Notation 132
B.2 HLCL to Predicate Logic Translation 135
B.3 Wellformedness Checking 137
B.4 Intermediate steps 139
B.5 Extended DATALOG to DATALOG$^\neg$ Translation 142

List of Tables

3.1 HLCL Reserved Keywords 25
3.2 Abstract Syntax Constructs for HLCL 30

A.1 The topological operators in Oracle Spatial 130

B.1 Naming Convention of Interfaces in Documentation 133
B.2 The cases in Extended DATALOG to SQL Translation 145

D.1 Test Cases for the HLCL-system 152

Chapter 1

Introduction

As computer systems are used more widely in our society, the databases supporting the systems become larger and more complex. To keep increasingly complex databases consistent and homogenous is a difficult and advanced task. To ensure consistency, database professionals define so-called integrity constraints on the objects of the database, which will reveal any data sets that are incorrect.
The constraints are usually deducted from extensive specifications written in plain natural language. The natural language specifications are easy to read and understand for non-database experts, and are relieved from any unnecessary implementation details. But due to the nature of natural language, they can be ambiguous and are, in general, not sufficiently precise.
Constraints on the other hand are typically formulated in first order predicate logic or in a database query language, and are very well-defined and unambiguous. This means that there exists a gap between the informal specification and the very formal constraints definition. Consequently, the deducted constraints are not optimal in all cases, perhaps even wrong, and this will ultimately result in loss of data quality.
In this report we present a language for formulation of constraints, which can fill the gap between specifications and the actual constraints. The language we have designed is called HLCL (High Level Constraint Language) and is targeted to be as similar to natural language as possible, but its semantic principles are built on a well known and well-defined property called "Peirce Algebras".
The language is intended for domain specialists, who have a basic background in databases. It is not intended for the absolute beginner, since it has a fixed syntax and requires some training to use.

The user will formulate a constraint in the context of a conceptual model, an Entity-Relationship Diagram (E/R Diagram), which is made available to the user. A compiler with information about the conceptual model, the

actual database schema and the mapping between the two, will compile the constraint into an executable query in the target database. The mapping between the conceptual model and database schema will be very loosely coupled, so that any changes in the low-level database schema can be accommodated by the map, resulting in no change in the conceptual model nor constraint.
There is a number of benefits from introducing the High Level Constraint Language we have designed: First of all HLCL will help the user to formulate the constraints in an easier and more natural language. A wellformedness check on the constraints will be able to catch many errors in the specifications. Second, the constraints will be clear and unambiguous, directly executable and implementation independent. Finally the actual constraints are formulated in the context of the conceptual knowledge, thereby being free of any implementation specific details, making them simpler and easier to understand.
In the present report the syntax and design choices of HLCL will be discussed, this includes a formal language specification and a semantic model. A proof-of-concept compiler compiling HLCL into DATALOG$^\neg$ and SQL will be supplied and explained.

As a case study constraints from the "The National Survey and Cadastre" (KMS) Geographical Information System (GIS) database will be used. It is important to note that the proposed constraint language is very general, and that it could be used on any system, but GIS is chosen since even quite complex constraints are fairly easy to comprehend for nonspecialists.

1.1 Reading Guide

The structure of this report corresponds to steps taken in the process of deducting a constraint from a specification to checking if it holds for a given database. In order to give an overview of the report structure and the full compiler process the following chapter will show how a simple constraint is compiled step by step by the HLCL-system.

The starting point is a constraint formulated in natural language, and a description of the target database. For a discussion of constraints in general and the application domain we have chosen the reader is referred to chapter 2 on page 9.

Suppose the target database has a conceptual model as in figure 1.1 on the next page. In our conceptual model we only operate with entity classes[1] and relations. For a full discussion of the requirements for the conceptual model, the reader is referred to chapter 3.2 on page 14.

[1] In this report we will use the shorter term "classes" for "entity classes"

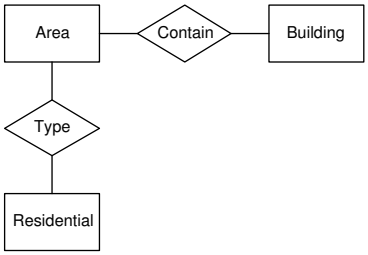

Figure 1.1: Walkthrough Conceptual Model

Let us imagine we want to express the constraint that: *"all areas must contain at least one building"*. This report defines a language, HLCL for describing such a constraint. An introduction to the constructed language HLCL can be found in chapter 3 on page 13. The constraint would be formulated in HLCL as:

```
all area type residential must contain building
```

HLCL expressions cannot be checked directly in the database. In order to check our constraint, we need to translate HLCL into an executable query language. To support this process an HLCL-compiler has been developed which takes HLCL as input and transforms it to SQL and DATALOG$^{\neg}$. The source code for this system can be found in Appendix E on page 191 and implementation details can be found in chapter 10 on page 103 and appendix B on page 131. The installation guide for the system can be found in appendix C on page 149. The rest of this chapter describes the steps the HLCL compiler takes:

The first step is to translate the HLCL expression into an intermediate Predicate Logic form - this is shown in chapter 6 on page 53. The above sentence has the corresponding Predicate Logic expression:

$$\forall X area(X) \wedge type(X,' residential') \rightarrow \exists Y(contain(X,Y) \wedge building(Y))$$

Then follows a series of logical rewritings, in which the Predicate Logic expression is transformed to a SQL-prepared form, called Extended DATALOG. The rewritings are shown in chapter 7 on page 69. This leads to the expression:

$$area(X) \wedge type(X,' residential') \wedge \neg\exists Y(building(Y) \wedge contain(X,Y))$$

In order to translate this into SQL, the compiler uses the database schema

Area

AreaID	Type	Size
1055	Residential	10
1056	Commercial	135
...	...	...

Contain

AreaID	BuildingID	Zipcode
1055	2055	2200
1056	2056	2100
...	...	...

Building

BuildingID	Zipcode	Size
2055	2200	5
2056	2100	30
...	...	...

Figure 1.2: Walkthrough Database Schema

specification and mapping. The specification of the description is given in chapter 8 on page 79. The SQL translation strategy is shown in chapter 9 on page 89. Say the database actually consists of three tables as shown in figure 1.2, then the corresponding SQL would look like the following:

```
SELECT *
FROM area a
WHERE
   a.type = 'residential'
AND NOT EXISTS(
   SELECT *
   FROM contain b
   WHERE b.areaID = a.areaID
   AND WHERE EXISTS(
      SELECT *
      FROM Building c
      WHERE c.BuildingID = b.BuildingID))
```

This SQL query would be the output of the compiler system, and could be used directly to query the database. The result of the query would be any data sets which do not conform to the constraint, hence if all the data conforms to the constraint then the database returns an empty: "0 rows returned" result.

Besides being translated into SQL, the HLCL constraint is also translated

into DATALOG$^\neg$ from the Extended DATALOG. The steps for translating Extended DATALOG to DATALOG$^\neg$ are described in chapter 7.3 on page 73. The corresponding DATALOG$^\neg$ can be seen below:

$$\textit{error} \leftarrow \texttt{Area(X)} \wedge \texttt{type(X,'residential')} \wedge \neg \texttt{f}_1\texttt{(X)}$$
$$\texttt{f}_1\texttt{(X)} \leftarrow \texttt{Building(Y)} \wedge \texttt{Contain(X,Y)}$$

This concludes the walkthrough of the process and report.

1.2 Thesis Structure

The rest of this thesis is structured as follows:

Chapter 2 - Background
Contains an introduction to Integrity Constraints for databases and the application domain.

Chapter 3 - General Idea
A new constraint specification language HLCL is introduced. The chapter contains both an informal and a formal definition of the syntax of HLCL.

Chapter 4 - Running Examples
A number of integrity constraint examples from the GIS domain are introduced, which will show various aspects of the language, and will be used as a basis for explaining transformations done in chapter 6 and 9.

Chapter 5 - Related Work
Current research in integrity constraints and approaches to help users formulate them is summarized. Particular attention is given to the four most widely used constraint languages: Description Logic, COLAN, OCL and EER.

Chapter 6 - From HLCL to Predicate Logic
The issues concerned with translating HLCL to Predicate Logic are discussed, and a formal translating strategy is given.

Chapter 7 - Intermediate Steps
The issues concerned with translating Predicate Logic to Extended DATALOG and DATALOG$^\neg$ are discussed, and the logical rewriting steps are given.

Chapter 8 - Database Representation
The details of the database representation and mapping between the conceptual model and to the actual database schema are discussed.

Chapter 9 - From Extended DATALOG to SQL
The issues concerned with translating Extended DATALOG to SQL are discussed, and the translating strategy is explained through examples.

Chapter 10 - Implementation
Overall issues and strategies concerned with the actual implementation of the HLCL system are discussed, such as choice of programming language, compiler strategy, etc.

Chapter 11 - Future Work
Suggestions for future research are given.

Chapter 12 - Conclusion
This chaper concludes the thesis and the contributions of the thesis are reviewed.

Appendix A - Concepts Explained
Concepts and notations used in the report are explained, such as "donkey sentences", "anaphora" and λ-calculus.

Appendix B - Detailed Implementation
Detailed implementation issues such as algorithms and choice of data-structures are explained.

Appendix C - User Guide and CD Contents
Contains a user Guide for installing and running the HLCL system

Appendix D - Test Cases
The test environment and the test results of the HLCL system are described.

Appendix E - Source Code
Contains the actual source code of the HLCL-system.

1.3 Abbreviations

A list of the most commonly used abbreviations in this report are explained below:

DL	Description Logic, see chapter 5.2 on page 46
DATALOG$^{\neg}$	DATALOG with negation, see chapter 7.1.2 on page 71
Extended DATALOG	The SQL prepared DATALOG, see chapter 7.1.2 on page 71
FOL	First Order Logic, Predicate Logic.
GIS	Geographic Information System, see chapter 2.2 on page 10
HLCL	High Level Constraint Language, see chapter 3 on page 13
ISA-structures	A relationship in the E/R-model denoting inheritance.
KMS	"Kort og Matrikelstyrelsen", National Survey and Cadastre. See www.kms.dk
SQL	Structured Query Language, see appendix A.3 on page 121
TOP10DK	"TOPografisk kortdabase 1:10.000 Danmark". A specification of the map of Denmark, see chapter 2.2 on page 10
XML	eXtensible Markup Language

Chapter 2

Background

In this chapter a brief background description will be given, explaining both integrity constraints in general and the application domain.

2.1 Database Constraints

Traditionally database integrity constraints have been divided into four different kinds. This view is presented in [GMUW02] and [AHV95], where constraints are defined in the following categories:

Single-value constraints This is the most basic constraint type. Single-value constraints requires that a certain value should be unique within a certain context. The most common single-valued constraint type is the key-constraints. A key-constraint specifies that an attribute, or a set of attributes, constitutes a unique key, so that two entries cannot have the same value in their key-attribute(s). Other single-value constraints are "many-one" relationships, which constrain a relation to relate each entity to at most one other entity.

Domain constraints These constraints can be seen as "value-limiting" on the attribute values. The constraints specify that an attribute should be within a certain value interval.

Referential constraints Referential constraints can constrain multiple tables, such that an attribute value in one table should also be a value in another table. A subclass of these constraints are the "foreign key constraints" which can impose that an attribute in one table should also be an attribute key in another table.

Schema-level constraints also called "General Constraints". These constraints define the interactions between the entities in the conceptual model, i.e. that one class must be related to another class in a specific way.

The first two of the above constraint types are relatively simple: They only refer to one or two tables in the actual database, and therefore do not require a lot of computational effort to enforce. More importantly, due to their simplicity they are easier for a database designer to realize and formulate correctly. These constraints are typically built-in in the database implementation, such that the database does not accept new entries which violate the constraints. Therefore checking these constraints at a later stage is pointless.
The third and fourth type: Referential and Schema-level constraints can be nontrivial: They can potentially extend throughout the conceptual model, and can consist of numerous intertwined relations between various classes, and although they can also be built into the database, it is usually not the case. These are the type of constraints we will examine and define a language for in the present report.

[GMUW02] views constraints as an "active" element, where a constraint is a query which is written once and then stored in the database. When new data are entered, the database checks if the data conforms to the stored constraints: The constraints are used as so-called "triggers". To build the constraints into the database implementation raises a number of issues: How to store the constraints? Which constraints should be checked, how should they be checked? How many and in what order should the constraints be tested towards the data, etc. This is an ongoing research-topic, and we will not look at these issues in the present report.
Instead we will adopt the view that an integrity constraint is simply a query which is not necessarily built into the database. The user can execute the query on the database at any time, and the result will contain any data-sets breaking the constraint. Hence if the constraint is fulfilled, the query should return no rows.

2.2 Application Domain

The proposed constraint language HLCL and corresponding system rely on general database principles which could be used in any domain. In this report we have chosen to look more closely at the geographic data domain, and the example material is based on typical constraints one would meet in the Geographic Information Systems (GIS). This chapter will contain a brief discussion of the properties and special considerations needed when handling geographic data.

Geographic Information Systems (GIS) is a broad term covering computer systems specialized in managing geographic data. This includes a variety of functions; such as storing the data, making various textual and graphical

extracts, having interfaces for users to update information, etc. Common for most of these functions are that they are built on top of an underlying database which stores the actual data.
Geographic databases are usually regular databases extended with datatypes supporting spatial properties. The spatial properties can store information about objects with respect to their location, or "maps" of these objects and their properties. Typically, a geographic database represents a model or a map of a landscape, containing information about both natural terrain such as forests and lakes, and of man-made entities such as buildings and roads.
The spatial properties are typically modelled within the "topological database model". In the topological database model one operates with objects such as points, lines and polygons. These geometric shapes all fulfill the topological property, which means that they are invariant in respect to scaling, rotation and affine transformations[1]. The objects have mutual topological relations such as "overlap", "contain", etc.[2]. These shapes and their properties are used as the basis of geographic objects, i.e. a road can be represented as a line. For a more comprehensive description of topological properties, the reader is referred to appendix A.3.1 on page 122.

The geographic database which KMS use consists of around ten million objects, and follows the "Danish TOP10DK specification" [SC01], which define around 60 object categories. Besides the object definitions, the TOP10DK specification also contains some constraints on the objects formulated in natural language.
The data in the topological database typically originates from aerial photos. The photos are analyzed, classified and manually entered in the database by specialized personnel. This procedure will always give rise to a number of mistakes, which need to be found at a later stage. At the same time, geographic data is used more widely than ever. With the introduction of computers and databases, GIS has grown and now spans over a much larger domain. GIS has grown from being used to create a printed visual map used by humans, to supply data to numerous of other applications, such as navigation systems in cars, traffic planning portals on the internet, etc. When the data is being used by other applications, it is more crucial than ever that the data is absolutely correct, and since the data collecting method is error-prone, integrity constraints are needed to ensure that the data is well-formed.
Research in GIS related problems is a research field of its own, and as a result the vendors making GIS systems have produced their own standards. Topo-

[1] Affine transformations are transformations that preserves lines and parallelism, e.g. parallel lines are also parallel after the transformation

[2] A definition of the full topological set of relations can be found in appendix A.3.1 on page 122

logical Databases are typically implemented in proprietary non-standard databases, such as "MapInfo" or "ArcGIS", and have their own query and constraint language, which can only be used within that implementation. Consequently it is crucial that the HLCL-system, besides the standard SQL interface, also has a another well-defined output interface. Therefore the HLCL-system also translates HLCL to a well-defined DATALOG$^\neg$ interface which can then be translated into any specific query languages.

To read more about GIS in general the reader is referred to [BS94]. Furthermore a more detailed explanation of the problems in modelling data in GIS can be found in [Chr05].

Chapter 3

General Idea

The overall task is to design a system where non-database experts can formulate integrity constraints easily. The system uses a High-Level Constraint Language, referred to as HLCL in this report. The system is designed such that it lets the user formulate a constraint in HLCL, which then is translated to form a directly usable integrity constraint query in SQL. In order to do so, the system also needs access to a database model. The overall idea of the system is sketched in figure 3.1 on the next page.

3.1 The HLCL System

The main task of the HLCL system is to translate HLCL into usable database queries, which in our case are queries in SQL. The translation procedure is broken up in two steps, where an intermediate language called Extended DATALOG is used. HLCL is first translated into Extended DATALOG then from Extended DATALOG to either SQL or DATALOG$^\neg$. This approach has been chosen since there are some common steps in the translation procedure to both DATALOG$^\neg$ and SQL and a translation of HLCL directly to SQL would be more difficult to understand.

$$\text{HLCL} \rightarrow \text{Predicate Logic} \rightarrow \text{Extended DATALOG} \quad \begin{matrix} \rightarrow \text{SQL} \\ \searrow \text{DATALOG}^\neg \end{matrix}$$

By introducing this intermediate step, the process of translating from HLCL to SQL should be more clear and understandable. SQL is the de facto standard query language for commercial relational DBMS systems. Allthough SQL is not the lowest level of abstraction for database queries, it has still been chosen as the target query language, since optimizing SQL for databases has been an on-going research topic for a long time. Therefore letting the DBMS do the actual querying by SQL would give the best performance. In general, optimization of the generated queries has not been researched

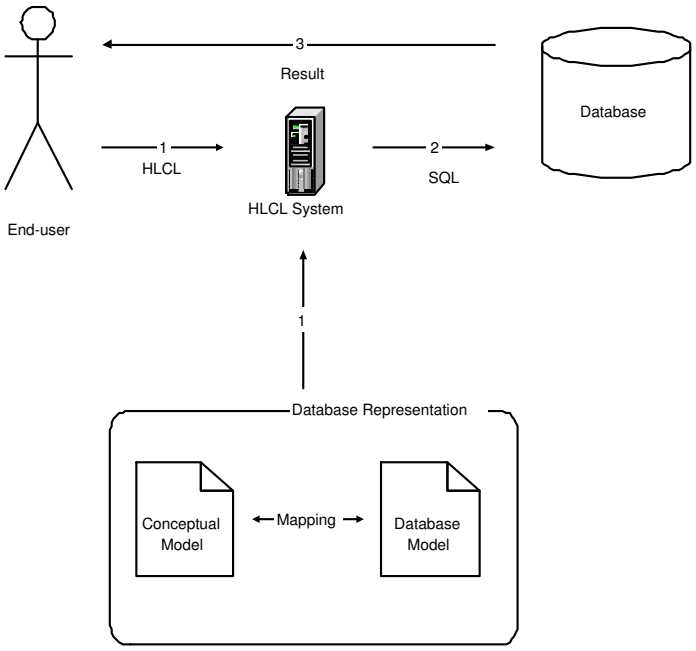

Figure 3.1: An overview of the HLCL System

extensively in this thesis.
Furthermore a subset of the Extended DATALOG is also translated into DATALOG¬. The DATALOG¬ interface provides an clear and easy-to-read alternative to the SQL query, which can become quite comprehensive even for small constraints. Translating numerical quantifications into DATALOG¬ would result in a lot of extra predicates in the DATALOG¬, which would make the DATALOG¬ expression less readable. We have chosen not to translate Extended DATALOG expressions with numerical quantifications into DATALOG¬. The DATALOG¬ interface also makes the HLCL system easier to expand, since the DATALOG¬ is a good starting point from which the constraint could be translated into other query languages, depending on the actual database implementation. HLCL is designed so that it can be used as a constraint language not only for traditional relational databases but also for other kinds of databases, i.e. Object Oriented Databases.

3.2 The Conceptual Model

It is important to realize that HLCL expressions are formulated in the context of a certain conceptual model. The conceptual model decides much of the syntax available, since the majority of HLCL expressions consist of names of classes and relations. Conceptual models for databases are usually repre-

sented in Entity-Relationship models (E/R-diagrams), which is a graphical notation for representing database structure and concepts. An introduction to E/R-diagramming rules and concepts can be found in [GMUW02]. The conceptual model itself already defines a number of constraints: Which classes could possibly be related, which classes are different, etc. Actually a conceptual model can be seen as just a collection of constraints. In this report we think of the constraints the conceptual model defined as very basic, they are all definitions of the classes involved, a kind of "type system", and HLCL is not designed to formulate these constraints. An exception is constraints which defines multiplicity of relations and some kinds of referential constraints, which can and should be formulated in HLCL, see appendix A.1 on page 119 for further details. HLCL is not made for defining simple constraints, instead HLCL defines schema-level constraints, and relies on the "type system" already defined in the conceptual model. Therefore the most basic HLCL constraint contains at least two classes, the "types" area and building. There are a number of requirements to the conceptual model which are listed below:

- The E/R diagram can consist of any number of entities and binary relations.
- In order to streamline the model, attribute values possessed by entities, should be modelled as relations to entity classes. The name of the relation should be the name of the attribute, and the connected class should be the value domain for that attribute.
- Names of relations do not need to be distinct, but different relations connecting the same entities should have unique names.
- HLCL does not operate with inverse relations, therefore both a relation and its inverse relation should be explicitly defined.
- ISA-structures [1] are supported, but only as hierarchies. This means that multiple inheritance is not supported in the current HLCL-system.
- Relations which are shared between children in an ISA-structure should always be connected from the parent in the ISA-structure. E.g. common relations should always be put at the top-most level in the structure.

3.3 HLCL Syntax

The general HLCL syntax is very simple; in fact, HLCL makes use of less than 10 keywords and structures. In the following chapter we describe how

[1] Class inclusion, see [GMUW02]

constraints are formulated in HLCL within a given corresponding conceptual model. The description is divided into two chapters: This chapter will introduce HLCL informally and to non-technical users, whereas chapter 3.5 on page 29 will give the formal specification of the language and is targeted at readers with a background in logic and semantic models.
The HLCL will be explained by using a number of examples each labelled with a **C** and a number. The numbers are not consecutive in this chapter. These examples will be used throughout the report as "running examples" to explain how HLCL is converted into DATALOG$^\neg$ and SQL, and can all be found in chapter 4 on page 37.

3.3.1 Informal Description

Minimal Sentence

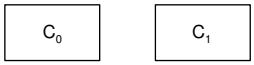

Figure 3.2: Two Classes

HLCL sentences are made up by two almost identical parts. The first part selects which class the constraint should apply to, and the second part specifies the actual constraint. The two parts are divided with a ‘‘`must`’’ keyword, so that the overall sentence structure looks like:

`all` ⟨first part⟩ `must` ⟨second part⟩.

This report will refer to the *"left-hand side"* of the expression when referring to the first part, and similarly the *"right-hand side"* when referring to the second part of the expression. In the simplest case each part are simply a class as seen below:

`All` c_0 `must` c_1.

A simple sentence from the GIS domain could be:

```
All house must building
```

The above constraint states that *"all houses must be buildings"*. The corresponding conceptual model should look as figure 3.2 with c_0 as "house" and c_1 as "building". The first ‘‘`all`’’ quantifier can also be replaced by a ‘‘`no`’’ quantifier, which gives the directly opposite constraint. The ‘‘`must`’’ keyword is correspondingly replaced by a ‘‘`may`’’, in order to

reflect correct English:

```
No area may house
```

The above constraint states that *"no areas may be houses"*. These two sentence types constitute the most basic sentence one can formulate in HLCL, so the minimal HLCL constraint involves at least two classes.

Paths

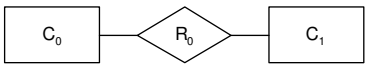

Figure 3.3: A simple relation

The constraints consisting of just two classes are usually not interesting, and can only form trivial constraints. The constraints can get more advanced when we look at related classes. In the context of the conceptual model in figure 3.3, an HLCL expression could be:

`All` c_0 `must` r_0 c_1.

A simple sentence from the GIS domain could be:

C2 `all area must contain building`

The above constraint formulates that every "area" must have a relation "contain" to a building (*"every area must contain a building"*). The above constraint actually implies an existential quantifier, so that we actually state: "an area must contain at least one building". This existential quantifier is implicit in the HLCL syntax, and one needs not explicitly to put it there. In order to select more specific class instances, HLCL uses a structure which

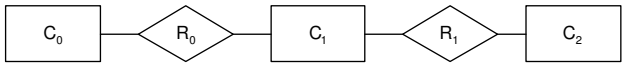

Figure 3.4: A path through relations

can be seen as a path through relations and classes. In a path we navigate from class to class through their relations. The syntax is simple, the path is simply written by the class followed by the relation followed by the next class, etc. A path-expression corresponding to the path seen in figure 3.4 can be seen below:

c_0 r_0 c_1 r_1 c_2

It is important to realize that one can only make path expressions that have a corresponding path in the conceptual model. A full HLCL-expression using the path in figure 3.4 on the preceding page could for instance be:

`no` c_0 `may` r_0 c_1 r_1 c_2

If nothing is specified between the classes and relations, the HLCL-system interprets it as existentially quantified classes, so that the above expression would be understood as: *"no c_0 should be related by r_0 to at least one c_1 which is related by r_1 to at least one c_2"* An example from the GIS domain could be:

C5 `no building may containedin area contain lake`

The above constraint expresses that: *"no buildings may be contained in an area which contain a lake"*. This makes the user able to select precisely which class the constraints should function on. But more importantly, it makes it possible for the user to access properties which are connected to the class through related classes. The paths can be used on both the left-hand and right-hand side of the expression.

Quantifiers

As previously mentioned, the existential quantifier is implicit in the HLCL-language, but if one wants to express *"an area must contain all buildings"*, then a universal quantifier must explicitly be formulated in the HLCL-expression. This is done with the `''all''` keyword, which is put between the relation and the class as seen below.

r_0 `all` c_1

This construct can be put instead of a "r_0 c_1" construct at any depth in a path. An example from the GIS domain could be:

C10 `no area may contain all building`

The above constraint expresses that: *"no areas are allowed to contain every building there exists in the database"*. If we continue the path expression after the class containing the `''all''`-keyword, we can further select the class.

```
No area must contain all building type residential
```

This constraint expresses that: *"no areas should contain all buildings which are of type residential"*. Notice how the extra "type residential" limits the

set of buildings which should not be contained in the areas.

Besides having the possibility to express that a class should be related to at least one, or all entities of another class, HLCL also allows numerical quantifiers, namely expressing exactly how many classes should be related. This is done by using one of the three numerical quantifiers placed in between the relation and the class as seen below:

r_0 `exactly` ⟨integer⟩ c_1.
r_0 `at least` ⟨integer⟩ c_1.
r_0 `at most` ⟨integer⟩ c_1.

The keywords should be self explanatory, but let us look at a couple of examples from the GIS domain:

```
all area must contain exactly 4 building
all area must contain at least 4 building
all area must contain at most 4 building
```

The top one of the three constraints, states that: *"all areas must contain exactly 4 buildings"*, the middle one states: *"all areas must contain at least 4 or more buildings"*, and the final constraint expresses: *"all areas must contain at most 4 or less buildings"*. All of these quantifications, can be set at any level in the path.

The "Solely" keyword

HLCL uses the keyword ``` ``solely'' ``` for limiting relations to a single class. The ``` ``solely'' ``` keyword is put at the same place in the HLCL expression as the other quantifiers, as seen below:

r_0 `solely` c_1

An example from the GIS domain could be:

```
all area must contain solely building
```

The above constraint states that *"areas are not allowed to contain anything else than buildings"*, basically that the contain relation in the conceptual model is only allowed to relate to buildings, and no other classes. It is important to notice that this keyword is only relevant because we allow multiple relations to have the same name. If we had stricter requirements to the conceptual model, such as all relations should have unique names, then the ``` ``solely'' ``` keyword would not be necessary.

The ‘‘`solely`’’ keyword is not simple to use, in order to illustrate this let us look at a more complicated example:

```
all area type residential must
contain solely building type residential
```

This HLCL-expression above means that *“all residential areas must only contain residential buildings, and nothing else”*. But what if one wanted to express that a residential area could contain lots of things, but all the buildings that it contained should be residential. One might be tempted to formulate a constraint as the one seen below:

```
all area type residential must
contain building type solely residential
```

But this constraint does not express the above constraint, instead it expresses: *“all residential areas must contain at least one building which is of solely type residential”*. The trick in formulating this constraint is to realize that the class we want to put a constraint on is not areas but in fact buildings. The correct constraint is shown below:

```
all building containedin area type residential must
type residential
```

The above constraint formulates: *“all buildings which are contained in a residential area must be residential”*. Here ‘‘`containedin`’’ is the inverse relation of ‘‘`contain`’’.

Compound statements

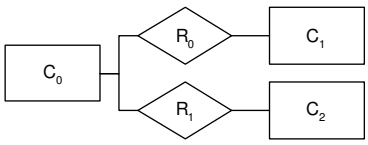

Figure 3.5: Compound Relations

Furthermore one can also make compound statements in HLCL, using one of 4 boolean compound operators: ‘‘`and`’’,‘‘`or`’’,‘‘`andnot`’’ and ‘‘`ornot`’’. These can split sections of paths expressions apart. Suppose a conceptual model like figure 3.5 is given, then HLCL allows us to formulate the following:

`all` c_0 `must` r_0 c_1 ⟨boolean operator⟩ r_1 c_2

An actual constraint from the GIS domain could be:

C6 `all area must contain building or contain lake`

The above constraint expresses that: *"all areas must either contain a building or contain a lake"*. The compound statements can be used on both the left-hand side and the right-hand side. But operators are only allowed in between relational paths. It is not allowed to use operators between classes, i.e. even if both relations were called r_0 in figure 3.5 on the facing page, we would not be allowed to write:

WRONG: `all` c_0 `must` r_0 c_1 ⟨boolean operator⟩ c_2

So if we look at the previous example from the GIS domain, we would not be allowed to write:

WRONG: `all area must contain house or lake`

In HLCL it is required that you "spell out" the whole expression as done in constraint example **C6**.

HLCL does not have an "exclusive or" operator explicitly defined. Instead it can be made with `''or''` and `''solely''` keywords. Take a look at the two expressions below:

```
all area must contain house exclusive-or contain lake
all area must contain solely house or contain solely lake
```

These two expressions are logically identical, and therefore the user should use the `solely` keyword, instead of a "exclusive or" operator.

Grouping Expressions with Brackets

Introducing compound operators also introduces a potential problem with scoping. Check out the HLCL expression below in the context of figure 3.6 on the next page:

`all` c_0 `must` r_0 c_1 r_1 c_2 `or` r_2 c_3

The above expression could be misunderstood, since there are two possibilities for the class "c_3", labelled A and B in figure 3.6 on the following page. The HLCL-system is right-associative, meaning that the expression will be read from right to left, so that "c_3" in the above HLCL expression

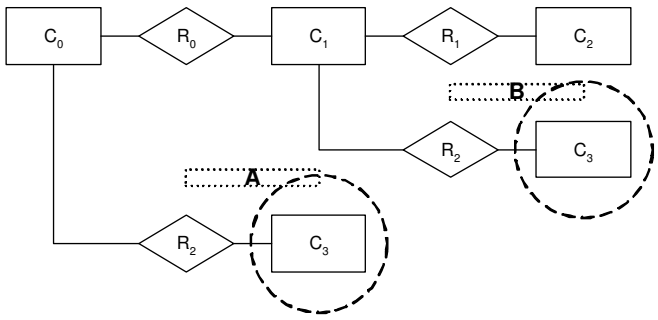

Figure 3.6: An example conceptual model for illustrating use of brackets

is automatically understood as the class labelled B. If we want "c_3" to be understood as the class labelled A, we need to use brackets, as seen below:

`all` c_0 `must (` r_0 c_1 r_1 c_2 `) or` r_2 c_3

In the above expression the class c_3 refers to the class labelled A in figure 3.6. HLCL needs brackets when we want the expression to be read from left to right, but it is also possible to use brackets even when they are not needed, i.e. take a look at the expression below

`all` c_0 `must` r_0 c_1 `(` r_1 c_2 `or` r_2 c_3`)`

The above expression is a valid HLCL expression. The brackets optional so that the user can put them there anyway to improve readability.

User-defined Variables

The sentence structures which have been introduced so far can only express existence of relations and classes. It is not possible to express equality of any of the involved classes. This is solved in HLCL by allowing optional user-defined variables. Introducing these gives rise to a number of questions and problems, which will be handled separately in this section. Let us look at an sentence example from the GIS domain:

```
all area intersectedby road must
contain building intersectedby road
```

The sentence states: *"all areas intersected by a road must also contain a building which is intersected by a road"*. The sentence only imposes that the roads should exist. But what if we also want to make the constraint express the fact that the two roads involved should be the same? One ap-

proach is to add something extra at the end of the sentence, i.e.: ``...and the earlier mentioned roads should be equal'', but anaphoric constructs[2] like these are not very easy to formulate, nor easy to handle in the HLCL system.
The solution is to allow the user to define their own variables in the expression. Variables are capitalized letters which can be put right after a class. A constraint expressing that the roads should be equal would look like:

C11
```
all area intersectedby road R must
contain building intersectedby road R
```

The user-defined variable "R" clarifies that the selected roads should be the same. The user-defined variables make HLCL expressions succinct and precise.
User-defined variables can only be used in existentially quantified classes, i.e. it is not possible to use the ``all/solely/numerical'' keyword and a variable for the same class. The argument can be seen below. As an example take a look at the HLCL fragment below:

```
...area contain building A...
```

The above HLCL fragment makes sense, it states that all areas which contain a building, which we call A for later reference. Here the building A is existentially quantified[3]. Let us look at the same fragment when the class is universally quantified:

```
...  area contain all building A...
```

This fragment is confusing. We want to impose a constraint on areas which contain all buildings, and all of these buildings are labelled A for later reference. But if we really need to refer to all of these buildings at a later stage in the constraint, it would have nothing to do with the original areas we are imposing the constraint on. Then we are actually looking at two constraints disguised as one, and this should be split up in two: A constraint dealing with the areas, and a constraint dealing with all those buildings, which the latter constraint would start with in our case:

```
All building A containedin area ...
```

This leads to an exception to the rule: That the only class which can have a universally quantified variable is the class we want to put the constraint on.

[2] An explanation of anaphora can be found in appendix A.2 on page 120
[3] Remember that existential quantification is implicit in the HLCL syntax

An HLCL expression can use an arbitrary number of user-defined variables, to relate as many classes as needed. When a user introduces more than one variable, the system will recognize different variable names as different instances. For instance in the expression:

```
all building A neighbour building B must
ZDifferenceLargerThan5(A,B)
```

The constraint[4] expresses the fact that that two different buildings must have a zdifference larger than 5. In this expression one does not need to explicitly add an extra condition saying A and B should be different. The HLCL-system will interpret different variable names as different entities.

Notice that usage of variables is optional; it is not required for simple queries to use variables, but as one makes larger queries in HLCL it becomes necessary to use variables.

User-defined Predicates

Finally we introduce the possibility to use user-defined predicates. These typically formulate topological and arithmetic properties, and should be defined functions available in the target database. In the current HLCL implementation only unary and binary predicates are allowed, but this could easily be extended to n-ary predicates. User-defined predicates can be put instead of class expression, although not instead of the very first class. They are defined by a name, which is followed by an argument containing previously declared variables:

```
somepredicate(firstvariable,secondvariable,....,lastvariable)
```

Let us take a look at the following example:

```
all building A neighbour building B must
ZDifferenceLargerThan5(A,B)
```

The built-in predicates are always used together with variables, which indicates which classes the function should be applied to.

HLCL Keywords

A summary of all reserved keywords in HLCL can be seen in table 3.1 on the facing page.

[4]This constraint also uses user-defined predicates, which are explained in chapter 3.3.1

all	no	must
may	solely	at
least	most	exactly

Table 3.1: HLCL Reserved Keywords

3.3.2 Macro Functionality

Definitions

In order to keep HLCL expressions short and compact, and to make it simpler to formulate constraints, the HLCL-system allows the user to make class definitions. Class definitions are simply a shorthand expressions for complicated class specializations. The general structure can be seen below:

```
Classdef = Class Expression
```

The left-hand side of the above expression should be a single-word unique label, and the right-hand side should be an HLCL class expression, which should abide the same well-formedness rules as general HLCL. All of the above HLCL constructs except variables and user-defined predicates are allowed. An example of a class definition can be seen below:

```
ResidentialBuildings = building type residential
```

The definitions can be arbitrarily complex, a more advanced example can be seen below:

```
SpecialBuildings = building type residential or
  touch building size at least 100
```

The class definition label can then be used in HLCL expression where classes normally would be placed, i.e. the definition of ‘‘ResidentialBuildings’’ could be used as seen below:

```
all area must contain SpecialBuildings
```

The above would be the same as writing:

```
all area must contain type residential or
  touch building size at least 100
```

‘‘SpecialBuildings’’ is a specilization of building entities, and can there-

fore only be put where the class ‘‘building’’ normally could be put. Class definitions can be recursively defined, e.g. class definitions can use other class definitions, such that definitions can “build on top of each other”. One should be careful not to build recursive defintions - definitions that indirectly define them self.

3.4 Language Design Decisions

In the following subchapters design discussions which we have encountered in finding the most optimal language will be discussed.

Variable Specification in User-Defined Predicates

The user-defined predicates style is somewhat closer to programming style, than to natural language. In the design phase it was considered if it would be better simply to write the name of the predicates without variables. I.e. instead of writing

```
all building A neighbour building B must
ZDifferenceLargerThan5(A,B)
```

One would simply write:

```
all building A neighbour building B must
ZDifferenceLargerThan5
```

The HLCL system could enforce a rule that specified that whatever variables set on the left-hand side, would automatically be used to the user-defined predicate on the right-hand side. This would actually make user-defined predicate constructs without variables possible.
This construction was discarded, because even though this would be closer to natural language, it would limit the HLCL expressiveness, i.e. one could imagine an HLCL expression where not all user-defined variables would be used in the predicate (As seen in constraint example **C12**), and the language would not be very clear. Therefore the design decision was to deviate from the natural language feel, and require that users declare the variables in the user-defined predicates in a programming language manner.

Negations

HLCL does not have a separate negation operator, instead negation is always part of another construct, either as the top construct ‘‘no-may’’, or in conjunction with one of the operators as ‘‘andnot’’ or ‘‘ornot’’. A

separate negation operator is avoided because floating negations could potentially make the expression counter-intuitive and unsafe. Leaving out the negation operator does not limit the expressiveness of HLCL as shown in the following.
Let us imagine that we had a negation operator: ‘‘no’’, which could be placed between the class and the relation (same place as the ‘‘all/solely’’ operator). Take a look at the expression below:

`all` c_0 `must` r_0 `no` c_1

The above expression is equivalent to one with a ‘‘no-may’’ construct as seen below:

`no` c_0 `may` r_0 c_1

Similarly if the ‘‘no’’ construct was part of a conjugated/disjunctive relational path, the ‘‘andnot’’/‘‘ornot’’ constructs could be used instead.

Path Expression

In HLCL we have chosen to use a very simple notation for paths. This is chosen because its resemblance to natural language, and because it is easier to type for the user. In the design phase several other ways of representing paths were considered, at first brackets were used, so that a path would be formulated:

c_0`(`r_0 c_1 `(`$r_1 c_2$`))`

The above notation is actually still allowed in HLCL as explained in 3.3.1 on page 21, but it is not mandatory. Other possible notations could have been paths as seen in XPath[5]:

$c_0/r_0/c_1/r_1/c_2$

Or it could be completely different notations, i.e. using arrows or other special characters:

$c_0 \mapsto r_0 \mapsto c_1 \mapsto r_1 \mapsto c_2$

But both of these constructs would break the “natural language feel” that HLCL has, and furthermore make it harder to type into the system, therefore the design decision is to stick with nothing else than spaces in between the

[5]XPath is a query language for XML - further information can be found online at: http://www.w3.org/TR/xpath

classes and relations in paths.

3.4.1 HLCL Expressions and Intuition

The goal of using HLCL language for specification is to make it easier for the user to formulate these constraints, therefore it is a key aspect that the language expresses constraints in an intuitive manner. But some of the constructs in HLCL are counter-intuitive, especially combinations of ``no-may'' and ``or''. Say the user wants to express the constraint that *"No buildings or houses may overlap an area"*, this would intuitively be defined as:

WRONG: `No building or house may overlap area`

But this is wrong, the above constraint is fulfilled even if a building overlaps an area, as long as no house overlaps an area, and vice-versa. The user actually meant logical "and" instead of logical "or", yet in natural language an "or" is used. The correct sentence can be seen below:

```
No building and house may overlap area
```

To solve this problem, HLCL does not allow operators between classes on the left-hand side in the constraints and this forces the user to split a constraint up in two, thereby getting:

```
No building may overlap area
No house may overlap area
```

Which is the correct interpretation of the constraint. The problem persists throughout though, and the user still needs to be aware of constructs where operators are used at a deeper path level, as seen in the expression below:

```
No building overlap area or contain area may ...
```

The expression above is allowed in HLCL, it is only operators at the outermost level which are not alllowed.
This concludes the language design decisions and thereby the informal description of the constructed HLCL language. In the following chapters a more formal definition of HLCL will be given.

3.5 Formal Description

In this chapter a formal definition of HLCL in both a concrete and an abstract syntax is given. The concrete syntax is a good reference for realizing which HLCL expressions are allowed, and how they can be built up. The abstract syntax is useful in the translation process of HLCL, because it ignores constructs which are not directly used in the translation. The constructs used in the abstract syntax will be used in the semantics and the translation strategy of HLCL. An introduction to abstract syntax can be found in: [SK]

Concrete Syntax

⟨expression⟩::=**"all"** ⟨class exp⟩ **"must"** ⟨class exp⟩ |
"no" ⟨class exp⟩ **"may"** ⟨class exp⟩

⟨class exp⟩::=⟨class⟩ [⟨rel class⟩] |⟨function⟩ |⟨varclass exp⟩

⟨varclass exp⟩::=⟨class⟩ ⟨variable⟩ [⟨rel class⟩]

⟨rel class⟩::=**"("**⟨rel class⟩**")"** |
⟨relation⟩ [⟨int quant⟩] ⟨class exp⟩ [⟨operator⟩ ⟨rel class⟩] |
⟨relation⟩ ⟨varclass exp⟩ [⟨operator⟩ ⟨rel class⟩] |
⟨attribute⟩ ⟨value⟩ |
⟨attribute⟩ ⟨numerical relation⟩ ⟨integer⟩ ⟨value⟩

⟨int quant⟩::=**"all"** |**"solely"** | ⟨numerical relation⟩ ⟨integer⟩

⟨operator⟩::= **"and"** | **"or"** | **"andnot"** | **"ornot"**

⟨numerical relation⟩::=**"exactly"** |**"at least"** |**"at most"**

⟨variable⟩::= **"A"** | ... |**"Z"**

⟨integer⟩::= **"1"** |**"2"** | ...

⟨class⟩::= **"building"** | **"area"** | ...

⟨attribute⟩::= **"type"** | **"size"** | ...

⟨relation⟩::= **"contain"** | ...

⟨function⟩::= **"zdifflargerthan"** | ...

⟨value⟩::= ...

The values in ⟨class⟩, ⟨relation⟩, ⟨attribute⟩ and ⟨value⟩ depend on the corresponding conceptual model. The values in ⟨function⟩ depend on the database model.
Please see 3.5 on page 31 in order to see well-formed limitations on the grammars.

Abstract Syntax

HLCL	**Abstract Syntax**
all P must Q	**allmust**(P,Q)
no P may Q	**nomay**(P,Q)
Relation ClassExp	**exrelclass**(Relation,ClassExp)
Relation all ClassExp	**allrelclass**(Relation,ClassExp)
Relation solely ClassExp	**solelyrelclass**(Relation,ClassExp)
Relation exactly N ClassExp	**numrelclass**(eq,N,Relation,ClassExp)
Relation at least N ClassExp	**numrelclass**(ge,N,Relation,ClassExp)
Relation at most N ClassExp	**numrelclass**(le,N,Relation,ClassExp)
Attribute Value	**attribute**(Attribute,eq,Value)
Attribute exactly N	**attribute**(Attribute,eq,Value)
Attribute at least N	**attribute**(Attribute,ge,Value)
Attribute at most N	**attribute**(Attribute,le,Value)
Class RC	**class**(Class,RC)
Class Var RC	**varclass**(Class,Var,RC)
Predicate(Var)	**unarypredicate**(Predicate,Var)
Predicate(Var1,Var2)	**binarypredicate**(Predicate,Var1,Var2)
P and Q	**and**(P,Q)
P or Q	**or**(P,Q)
P andnot Q	**andnot**(P,Q)
P ornot Q	**ornot**(P,Q)

Table 3.2: Abstract Syntax Constructs for HLCL

⟨E⟩::= **allmust(**⟨CE⟩**,**⟨CE⟩**)** | **nomay(**⟨CE⟩**,**⟨CE⟩**)**

⟨CE⟩::= **class(**C_{id}**,**⟨RC⟩**)** | **varclass(**C_{id}**,**V_{id}**,**⟨RC⟩**)** | ⟨F⟩ | ⟨RC⟩

⟨RC⟩::= **exrelclass(**R_{id}**,**⟨CE⟩**)** | **allrelclass(**R_{id}**,**⟨CE⟩**)** | **solelyrelclass(**R_{id}**,**⟨CE⟩**)**
| **numrelclass(**Int**,**⟨Comp⟩**,**R_{id}**,**⟨CE⟩**)** | **attribute(**A_{id}**,**⟨Comp⟩**,**Val**)**
| **or(**⟨RC⟩**,**⟨RC⟩**)** | **and(**⟨RC⟩**,**⟨RC⟩**)** | **ornot(**⟨RC⟩**,**⟨RC⟩**)** | **andnot(**⟨RC⟩**,**⟨RC⟩**)**
| []

⟨F⟩::= **unarypredicate(**P_{id}**,**V_{id}**)** | **binarypredicate(**P_{id}**,**V_{id1}**,**V_{id2}**)**

⟨Comp⟩::= **eq** | **ge** | **le**

This abstract grammar introduces a number of predicates used for the basic constructions in HLCL. Most of the names are self explanatory and straightforward, and the explanation of each construct can be seen in table 3.2.

Language Wellformedness

Besides fulfilling the grammars above, HLCL sentences should also fulfill a number of wellformedness requirements, which cannot be expressed by the grammar. These can be seen below:

WFF1 Queries using paths should only be able to "walk" in existing paths, meaning that "c_0 r_0 c_1" structures are only allowed iff there exists a relation r_0 between c_0 and c_1 in the conceptual model.

WFF2 Variables should be declared at least two times in the HLCL expression. Using a variable just one place in the expression, makes no sense since we need at least two in order to make a comparison. Usage of the same variable more than two times is allowed.

WFF3 The variables should be of the same type, meaning that multiple occurrences of the same variable should refer to the same type of class/attribute.

WFF4 HLCL expressions are only allowed to start with a class expression, i.e they are not allowed to start on a user-defined predicate or a relational path.

WFF5 HLCL expressions are allowed to start on a relational path on the right-hand side of the expression. This will be understood as a relational path continuing from first class expression given on the left-hand side.

WFF6 HLCL expressions are not allowed to have compound operators on the left-hand side of the expression. For a further explanation the reader is referred to chapter 3.4.1 on page 28

WFF7 User-defined variables are not allowed in operands of the `''all''`, `''solely''` and the nummerical quantifications operators.

3.6 Model Theoretic Semantics

The following chapter will contain a formal specification of the model theoretic semantics for HLCL. The model will use the constructs introduced in the abstract syntax in chapter 3.5 as basis for the set theoretic model. The constructs for user-defined variables and user-defined predicates will not be modelled in the following, since it increases the complexity of the model significantly. The modelling of the user-defined predicates has not pursued further in this thesis.

Basic Form

The two different sentence forms in HLCL expressions are represented by the two top-predicates *"allmust"* and *"nomay"*, which are defined below:

$$[\![allmust(P,Q)]\!] = [\![P]\!] \subseteq [\![Q]\!]$$
$$[\![nomay(P,Q)]\!] = [\![P]\!] \cap [\![Q]\!] = \emptyset$$

The first construct: "*allmust*" specifies class inclusion: That the set the left-hand side captures must be a subset of the set the right-hand side captures. The second construct: "*nomay*" expresses the opposite, namely that there should be no overlap between the sets, or more formally: The intersection between the set on the left-hand side and the set on right-hand side should be empty. The simplest HLCL expressions has only a class as P and Q, which is specified as follows:

$$[\![class(C_{id}, [])]\!] = C_{id}$$

Combining the simplest class expression with the two top-predicates yields the two basic sentence types which can be seen below:

$$[\![allmust(class(C_P, []), class(C_Q, []))]\!] \rightarrow C_P \subseteq C_Q$$
$$[\![nomay(class(C_P, []), class(C_Q, []))]\!] \rightarrow C_P \cap C_Q = \emptyset$$

Peirce Product

The basic form seen above can be extended with relational paths. A class containing a relational path has a "RC" component in the second argument as seen below:

$$[\![class(C_{id}, RC)]\!] = \{x | x \in C_{id} \wedge x \in [\![RC]\!]\}$$

In the simplest case the "RC" component is an existentially quantified relational path, a "*exrelclass*" construct in the abstract syntax. The semantic interpretation of the "*exrelclass*" is defined as:

$$[\![exrelclass(R_{id}, CE)]\!] = \{x | \exists y, y \in [\![CE]\!] \wedge (x,y) \in R_{id}\}$$

The above set expression can be read as: *"The entities where there exists a relation R_{id} to an entity of class CE"*. Where "CE" can be a simple class identifier or yet another relational path. This property is also known as the "Peirce product", which in [BBS94] would be expressed as "(R_{id} : CE)". The Peirce Product is a dyadic operator taking a binary relation and a set as arguments, resulting in a set. For a more detailed introduction the

reader is referred to [BBS94].
This means that a relational full path, i.e. "$c_0\ r_0\ c_1$", with the following abstract syntax:

$$[\![class(C_0, exrelclass(R_0, class(C_1, [])))]\!]$$

Would have the following semantic meaning

$$\{x|x \in C_0 \wedge x \in \{x|\exists y, y \in C_1 \wedge (x,y) \in R_0\}\}$$

Which reads: *"The set of entities of class C_0 which relate to at least one entity of class C_1 through the relation R_0".*

Peirce Product Variations

Besides existentially quantified relational paths, HLCL also uses paths which are similar to the Peirce Product, but differently quantified. These will be explained in this section. First we look at the dual operator to the Peirce product, namely the *"solelyrelclass"*, which has the following meaning:

$$[\![solelyrelclass(R_{id}, CE)]\!] = \{x|\forall y, y \in (x,y) \in R_{id} \rightarrow y \in [\![CE]\!]\}$$

As it can be seen the "*solelyrelclass*" looks similar to the *"exrelclass"* except that it uses a universal quantifier instead. The set expression can be read as: *"all entities which are related through the relation R_{id} to entities of solely class CE".*
Furthermore HLCL allows another universal quantified path, namely "*allrelclass*", where implication is reversed as seen below:

$$[\![allrelclass(R_{id}, CE)]\!] = \{x|\forall y, y \in [\![CE]\!] \rightarrow (x,y) \in R_{id}\}$$

This can be read as *"all the entities related to all entities of class CE through R_{id}".* Finally HLCL allows relational paths which use numerical quantifications. The meanings of these can be seen below:

$$[\![numrelclass(eq, N, R_{id}, CE)]\!] = \{x|\exists y, card(\{y|R_{id}(x,y) \wedge y \in [\![CE]\!]\}) = N)\}$$

$$[\![numrelclass(le, N, R_{id}, CE)]\!] = \{x|\exists y, card(\{y|R_{id}(x,y) \wedge y \in [\![CE]\!]\}) < N)\}$$

$$[\![numrelclass(ge, N, R_{id}, CE)]\!] = \{x|\exists y, card(\{y|R_{id}(x,y) \wedge y \in [\![CE]\!]\}) > N)\}$$

The three numerical quantifications are very similar, except there is a difference in the comparison to the given integer 'N'. The top numerical quantified set expression can be read as: *"the set of entities which exactly N entities of class CE is related to through R_{id}"*.

Operators

HLCL allows operators between the relational paths, in order to make compound expression. The compound operators has a straightforward semantic meaning and can be expressed directly in set theory:

$$[\![and(P,Q)]\!] = [\![P]\!] \cap [\![Q]\!]$$
$$[\![or(P,Q)]\!] = [\![P]\!] \cup [\![Q]\!]$$
$$[\![andnot(P,Q)]\!] = [\![P]\!] \cap [\![Q]\!]^C$$
$$[\![ornot(P,Q)]\!] = [\![P]\!] \cup [\![Q]\!]^C$$

Attributes

Finally there is the option that a relational path could in fact be an attribute, which would have the semantic meaning seen below:

$$[\![attribute(A_{id}, eq, value)]\!] = \{x | A_{id}(x, value)\}$$
$$[\![attribute(A_{id}, le, value)]\!] = \{x | \exists y, A_{id}(x,y) \wedge y < value)\}$$
$$[\![attribute(A_{id}, ge, value)]\!] = \{x | \exists y, A_{id}(x,y) \wedge y > value)\}$$

Let us look at a couple of examples illustrating the above points.

Example 3.6.1
Let us try to look at the semantic meaning for one of the simplest HLCL expression as seen below:

C2 `all area must contain building`

The semantic meaning can be reduced as seen below:

$$[\![allmust(class(area, []), exrelclass(contain, class(building, [])))]\!]$$
$$[\![class(area, [])]\!] \subseteq [\![exrelclass(contain, class(building, []))]\!]$$
$$area \subseteq \{x | \exists y, y \in [\![class(building, [])]\!] \wedge (x,y) \in contain\}$$
$$area \subseteq \{x | \exists y, y \in building \wedge (x,y) \in contain\}$$

The final set expression can be read as: *"the entities of class area should be a subset of the entities which relate to at least one building through contain"*. This is clearly the correct interpretation.

Example 3.6.2
Let us look at a more complex HLCL expression:

C4 `all building type industrial must beusedby company`

The semantic meaning could be reduced as seen below

$$\begin{aligned}&[\![allmust(class(building, attribute(type, eq, industrial)),\\ &\quad exrelclass(beusedby, class(company, [])))]\!]\\ &[\![class(building, attribute(type, eq, industrial))]\!] \subseteq\\ &\quad [\![exrelclass(beusedby, class(company, []))]\!]\end{aligned}$$

The left-hand side of the $\subseteq$ is reduced into:

$$\begin{aligned}&\{x|x \in building \wedge x \in [\![attribute(type, eq, industrial)]\!]\}\\ &\{x|x \in building \wedge x \in \{x|type(x, industrial)\}\end{aligned}$$

The right-hand side of the $\subseteq$ is treated similar to the previous example:

$$\begin{aligned}&[\![exrelclass(beusedby, class(company, []))]\!]\\ &\{x|\exists y, y \in company \wedge (x, y) \in beusedby\}\end{aligned}$$

This gives the full expression as seen below

$$\begin{aligned}&\{x|x \in building \rightarrow x \in \{x|type(x, industrial)\}\\ &\quad \subseteq \{x|\exists y, y \in company \wedge (x, y) \in beusedby\}\end{aligned}$$

This is also clearly the correct interpretation.

Chapter 4

Running Examples

To help show the usage of HLCL, a number of constraint examples from the GIS domain will be defined in the following section. These constraints will first be defined in natural language and then their HLCL counterpart will be explained. Some of the constraints are from [Chr04], others are extracted from the "KMS TOP10DK Specification" [SC01].

4.1 Conceptual Model

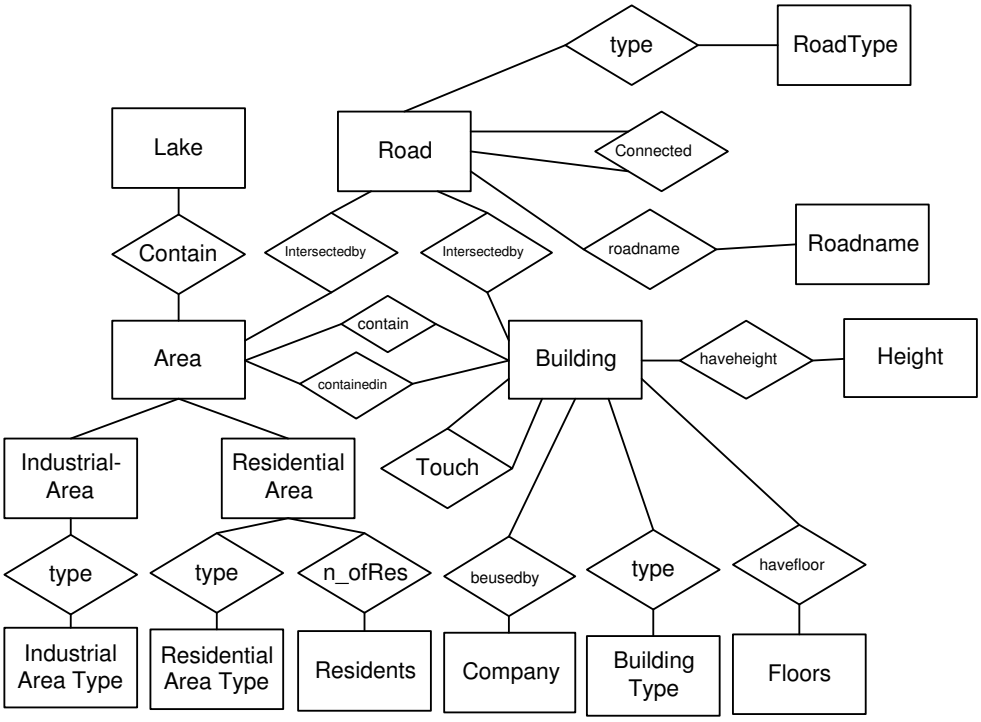

Figure 4.1: The domain for the Constraint Examples

A small, but representative part of the GIS domain has been modelled in the E/R-diagram seen in figure 4.1. All classes and relations referred to in the constraint examples, are shown in this diagram. The constraints evolve around the three main classes "Area", "Building" and "Road". Furthermore the "Area" class has two specializations: "IndustrialArea" and "ResidentialArea". The rest of the classes and relations should have names which are self explanatory.

4.2 Constraint Examples

In this section a number of constraint examples will be defined. Constraints are identified by **C** followed by a number for further reference, this identification will be used throughout the report. The constraints have been chosen as a representative set for explaining the HLCL concepts in the report, but should not be viewed as a complete set for testing the actual system. The constraints used for testing the system can be seen in appendix D on page 151.

Constraint Example C1 *"No industrial areas may be residential areas at the same time"*. In HLCL this would be formulated as:

C1 `no industrialArea may residentialArea`

The translation into HLCL is straightforward, a `‘‘no-may’’` construct is used, and the classes are directly available in the conceptual model.

Constraint Example C2 *"all area must contain at least one building"*. In HLCL this would be formulated as:

C2 `all area must contain building`

The translation uses the `‘‘all-must’’` construct, and uses the relation "contain" to relate classes "area" and "building".

Constraint Example C3 *"no areas are allowed to contain lakes"*. In HLCL this would be formulated as:

C3 `no area may contain lake`

The translation is similar to **C2**, except the inverse `‘‘no-may’’` construct is used

Constraint Example C4 *"all industrial buildings should be used by a company"*. In HLCL this would be formulated as:

C4 `all building type industrial must beusedby company`

The translation makes use of the type attribute for building, thereby restricting the constraint to only concern buildings which have the type "industrial".

Constraint Example C5 *"no buildings must be contained in areas which also contains lakes"*. In HLCL this would be formulated as:

C5 `no building may containedin area contain lake`

The translation makes use of a longer path, first through the "containedin" relation, then through the "contain" relation. Such that the above constraint can be read as no building may be contained in at least one area with at least one lake in it.

Constraint Example C6 *"areas has to either contain a building or a lake"*. In HLCL this would be formulated as:

C6 `all area must contain building or contain lake`

The translation uses composite operators, notice that both "contain" relations relate to the "area" class.

Constraint Example C7 *"residential areas must contain maximum one commercial building"*. In HLCL this would be formulated as:

```
C7 all residentialArea must contain at most 1 building
     type industrial
```

The translation makes use of the numerical quantifications, so that every residentialArea must only contain at least one entity type commercial. In this expression it is important to realize, that it is perfectly all right that the residentialArea contains more of other entities, we only limit commercial buildings specifically

Constraint Example C8 *"industrial areas may only contain industrial buildings"*. In HLCL this would be formulated as:

```
C8 all industrialArea must contain solely building
     type industrial
```

The translation makes use of the "solely" keyword for limiting the industrialArea's contain relation. Notice the difference between this constraint and **C9**: This constraint expresses the fact that industrial areas can only contain buildings type industrial and nothing else, whereas **C9** expresses that all the buildings contained in an industrial area should be of the type industrial.

Constraint Example C9 *"industrial areas may only contain buildings that are industrial, but are allowed to contain other things as well"*. In HLCL this would be formulated as:

C9 `all building containedin industrialArea must type industrial`

See **C8** for an explanation.

Constraint Example C10 *"no areas are allowed to contain every single building in the database"*. In HLCL this would be formulated as:

C10 `no area may contain all building`

The translation uses the ``‘‘all’’`` keyword to quantify all objects in the database of a certain kind.

Constraint Example C11 *"No road must intersect an area unless it intersects a building also within that area."*. In HLCL this would be formulated as:

C11
```
all area intersectedby road R must
    contain building intersectedby road R
```

The translation uses the user-defined variable "R" to keep track of the road. Notice how the variable allows us to make a reference to the same road multiple places in the expression. For more information about variables please refer to section 3.3.1 on page 22.

Constraint Example C12 *"If two buildings of the same type are neighbors then the z-difference between the two buildings must be larger than 5 meters"*. In HLCL this would be formulated as:

C12
```
all building A type T touch building B type T
    must havezdifferencebiggerthan5(A,B)
```

The translation uses the variable T to ensure that the buildings have the same type, and variables A and B in a special user-defined predicate havezdifferencebiggerthan5(A,B) which makes the arithmetic comparison.

Constraint Example C13 *"All Residential Buildings within Low Residential Areas must have at most 3 floors or be lower than 12 meters."*. In HLCL this would be formulated as:

C13
```
all building type residential and containedin area
    type lowresidential must havefloor at most 3 or haveheight
    at most 12
```

This is a more advanced constraint illustrating several of the above principles at the same time.

Constraint Example C14 *"all buildings must be blockbuildings and are only allowed to touch other buildings"*. In HLCL this would be formulated as:

C14
```
all building must type blockbuildings and
    touch solely building
```

The translation is straightforward according to the above principles

Chapter 5

Related Work

This chapter contains a brief discussion of the efforts that have been made in the database integrity constraint formulation field over the last twenty years, and particular attention will be given to the four newest and most popular approaches, namely: Description Logic, COLAN, OCL and EER.

There is a general consensus that formulating queries in low-level database languages is too hard for end-users. Typically the users specifying the queries are domain experts, but not necessarily database/computer experts. Difficulty in formulating queries directly in a database implementation has given rise to languages such as SQL, which is an abstraction over the algorithmic details of the database query[1]. But complicated queries are hard to formulate even in SQL, where they quickly become very big and confusing.
In general there are a lot of approaches as to make query formulation easier, both by the means of the query language used, and in the process used to formulate the constraint.
One approach is to change the process in which the user formulates a query: Instead of having the user to formulate a complete query in one step, some try to make an interactive system, where the user composes a query step by step with guidance from the system.
This approach is used in a system called "Kaleidoscope" [Cha90]. In "Kaleidoscope" the user specifies the query via a menu-driven interface. The user starts off by selecting a small fragment of the query, the system then calculates the next possible fragments, and displays these to the user. From this list the user selects yet another fragment. By this process the user composes a query step by step, and the system ensures that the query is valid by only allowing the user to choose valid query fragments.
Another similar approach is using "SQL FORMS" [AG97], which consists of page with a number of fields, similar to a form on the internet. The user navigates through "blocks" of these forms, also generating a query. The

[1] See appendix A.3 on page 121 for more details on SQL

advantages are the same as with the "Kaleidoscope" approach.

Another approach is to use a graphical interface, instead of a textual one, such as an E/R-diagram, where the user can compose a query by clicking or "drag-n-drop'ing" a query. For a good survey of graphical systems the reader is referred to [CERE90] which has a long list of references to graphical systems.
A lot of research has also been devoted to making "real" natural language interfaces to databases. The idea is that a user could just use his everyday language to access the database and there is no fixed syntax for the query. The interfaces typically recognizes keywords and tries to guess what the user wants. But without a fixed syntax there will always be some uncertainty in the query. For a user who needs to extract information from a database, this might be acceptable, since the query can be rephrased after the query results are shown. But this is not acceptable for a constraint specification system, since the user might not be able to realize from the result if the constraint is correctly formulated or not. Therefore we need a fixed syntax which is not as free as natural language, but which is still easy to learn and understand due to its resemblance to natural language.
There is also a number of approaches which try to make high-level interfaces closer to the one we suggest, but where it does not function on the relational data model. One of these is COLAN which is handled in the next section. ALICE [Urb89] is yet another approach with a language similar to HLCL, but ALICE maps into a database model called CORAL, which is a deductive database. [Red93] suggests a language PFL, which maps into a functional database called PFL. Among the other systems, [GEHK99] refers to CIAO++ [JQ92] which maps into a C++-inspired database-model and PRISM [SK84].

There are in fact a number of research articles which propose high-level query languages for the E/R-model, [GH91] cites languages such as ERROL [MR83], Despath [Roe85] and GORDAS [EW81], but these were invented in the early 1980's and have not been cited much since in the litterature. Lately, most of the litterature on integrity constraints has been on OCL, and the OCL litterature does not contrast OCL to these languages. Therefore we have chosen not to pursue the ideas in these languages, since they are most likely contained in the four languages shown below.

5.1 Colan

One of the attempts similar to the one we propose is the language COLAN [BG95, GEHK99]. COLAN uses a language similar to HLCL, and works on a database model called "P/FDM" (Prolog/Functional Data Model) data

model, where the query language is called DAPLEX[2]. The constraint language COLAN can be seen as an extension of DAPLEX, and is sometimes also referred to as Extended-DAPLEX [EG96].
The P/FDM model is a functional database implemented in PROLOG, but the articles state that it is similar to the relational data model. In an rather new article, [GHP01] which is concerned with semi-structured XML data, it is also stated that the P/FDM is similar to a semi-structured database. Both similarities are somewhat unclear, since they are not described well in the articles, and furthermore it is not well-defined how COLAN should be translated into a query language such as SQL.
Let us look at an example to get a better understanding of COLAN. In [GEHK99] an example is given from the university domain, where a constraint expresses that *"every member of the staff with a certain status, should only advice honours students"*. This is in COLAN expressed as:

```
constrain each s in staff such that astatus(s)=‘‘honour’’
   so that each s1 in studadvised(s)
has status(s1)=‘‘honour’’;
```

The corresponding Predicate Logic expression looks like the following:

$$\forall S\,staff(S) \wedge astatus(S, honour) \rightarrow \forall S1(studadvised(S1, S) \\ \rightarrow (student(S1) \wedge status(S1, honour)))$$

It can be seen that COLAN is very close to Predicate Logic, there is almost a linear correspondence: Variables are introduced and quantified in the same order as they would be in Predicate Logic: COLAN is basically "sugared logic". One striking difference between COLAN and HLCL is that COLAN requires that you explicitly define a variable for each class which is used: There is no option of leaving out variables in simple expressions as you can in HLCL. The above example illustrates the point well. When formulating the example in HLCL it becomes a much simpler expression as seen below:

```
All staff astatus honour must advise
solely student status honour
```

In the above expression "`advise`" is the inverse relation of "`studadvised`". It can be seen that our corresponding HLCL-expression does not need any user-defined variables, and thereby creates a more simple and readable constraint.

[2]FDM is an ongoing research database framework, further information can be found online: http://www.csd.abdn.ac.uk/∼pfdm/

5.2 Description Logic

Description logic (DL) is a logic-based knowledge representation, which is a descendant of the KL-ONE system. There exists a number of different description logics, i.e. $\mathcal{FL}$, $\mathcal{ALC}$, which all vary in decidability, computational complexity and expressiveness. In general there is a trade-off between these two properties: The more expressive the language, the more computationally complex a language is.
Description logics work on semantic networks, which are similar to E/R-models, and can therefore easily be used to model database structures, and can even be used as a query language. Description logics operates with concepts, which are unary predicates which can be viewed as classes, and roles which are binary predicates which can be viewed as relations between classes. Furthermore there are a number of operators, which differ depending on how expressive the description logic is. A good introduction to Description Logics can be found in [BMNPS02].
Description Logic can also be used to model constraints, for instance the simple HLCL expression:

C2 `all area must contain building`

The expression above would in DL look like:

$$area \sqsubseteq \exists contain.building$$

Which would be read as: *"The set of all areas is a subset of the set of all areas that contain a building"*. The opposite constraint seen below:

C3 `no area may contain lake`

Would in DL be formulated as:

$$\bot \doteq area \sqcap \exists contain.building$$

Which reads: *"The intersection of the set of all areas and the set of areas which contains a building is empty"*.
Modelling more complex constraints with Description Logic is problematic. Description Logic basically only consists of unary predicates. There are binary predicates, but these are the roles, which can only be used in conjunction with existential and universal quantifications or perhaps some of the built-in operators. It is not possible to use the relations on their own, and therefore it is only possible to get certain structures in the corresponding First Order Logic or ultimately SQL query. For example let us look at the constraint: *"An area is not allowed to contain two buildings that overlap"*. In HLCL we would formulate this like:

```
no building containedin area A may
   overlap building containedin area A
```

The similarity in structure is obvious. This constraint cannot possibly be formulated in Description Logic. One can extend Description Logic with variables, so that the above expression could be formulated as:

$$\bot \doteq \exists containedin.(V) \sqcap \exists overlap.(\exists containedin.(V))$$

This expression reads: *"The intersection of the set of buildings contained in area V and the set buildings which overlap the buildings contained in area V is empty"*. But the above sentence is not Description Logic, due to the usage of variables. A more formal definition of the problem of expressiveness can be seen in [Bor96], which points out a number of FOL expressions that cannot be expressed in Description Logic. Recently a Description Logic called $\mathcal{DLR}$ has been proposed, which allows n-ary relations. This means that it should be able to formulate the constraints seen above, but unfortuantly it is only tractable in ExpTime, and it has therefore not been further examined. Due to the limitations in expressiveness we have chosen not to use Description Logic.

It should be noted that Description Logics were made with another intention in mind, and Description Logics have some characteristics which HLCL does not have: Description Logic has a resolution engine and can therefore reason about its knowledge. HLCL functions on the other hand as a language, which is compiled into SQL without any reasoning or optimizations. There is no framework from extracting extra knowledge out of our HLCL statements. However we do not need this in our project, where HLCL has been used more as a basis for parsing.

5.3 OCL

One of the more popular approaches to specifying integrity constraints is by using a language called "Object Constraint Language" (OCL). OCL is part of the UML Specification, which is maintained by the Object Management Group[3]. The UML standard consists of a number of different diagram notations, mainly used in Software Development, but the concepts can also be used in database design.
UML and OCL are evolving standards, UML is currently in version 1.5, but a version 2.0 is on the drawing board. The OCL part of UML 2.0 can already be found in [Gro].

[3] www.omg.org

Example sentences in OCL

In OCL it is possible to specify 4 different kinds of constrains: Invariants, Guards, Pre- and Postconditions. Invariants are the constraints that should be fullfilled at all times for the database, and therefore we will only focus on this type of constraint in OCL. Guards, Pre- and Postconditions are not relevant, since they are used in database update queries.
To explain the syntax of OCL two constraint examples in OCL will be briefly

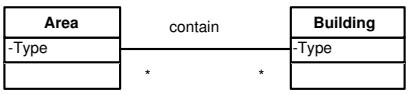

Figure 5.1: The domain for the OCL constraints

sketched, the used domain for the constraints can be seen in figure 5.1. A simple sentence in HLCL is:

C2 `all area must contain building`

This would in OCL look like:

```
context area inv
self.contain->size() >= 1
```

In OCL expressions one starts by defining a context, which is the base object/class that is affected by the constraint. In our case this is "area". Then follows a path expression: "self.building", meaning we navigate from "self" (a reference to context, area) to building. "size()" is a built-in predicate, which returns the number of sets which are in this path. In our case the number of buildings contained by every area, should be at least one.
Let us look at a more complicated example:

```
all area type residential must contain
  at least 5 building type residential
```

Which in OCL would turn out to:

```
context area inv
  self.type = residential implies self.contain ->
        select(type = residential)->size() >= 5
```

In this query the context is the same. The OCL expression makes use of the "implies" construct, which is available together with existential and universal quantification in OCL. In the above case the universal quantification is implicit because we are quantifying the context class. It can be seen how the handling of sets is difficult in OCL, we need to add selection criterions

in the middle of a path, and again use the built-in size() function. In general OCL relies heavily on these built-in functions. With these constructs every sentence we can formulate in HLCL can be formulated in OCL as well, although it seems more difficult to do so.

Problems with OCL

In this project we have chosen not to let the users formulate constraints in OCL, nor using OCL as an intermediate language between HLCL interface and SQL. There is a number of reasons for this:
The main reason is that the semantic model of OCL is not very well-defined. A number of problems with the semantics of OCL have been pointed out in [GR98]. Some of these problems have been fixed in OCL 2.0, but it is difficult to conclude that OCL 2.0 has a completely well defined semantic model.
OCL is still a very "young" language, and it is likely that further problems with its semantic model will arise during this evolving phase. Due to the uncertain semantic model, an actual approach for converting OCL constraints into SQL is also somewhat unclear. Attempts to convert OCL to SQL have been made in [DHL01], but the algorithm has a somewhat "dirty" feel. It is still in a prototype version, it can only convert a subset of OCL and the article does not mention any formal strategy for the translation.
In [BKS02] the authors sketch an approach for converting OCL into First Order Logic, and they present an OCL to First Order Logic algorithm. Whether this subset of FOL in affect can be translated further on into SQL is unclear[4].
Attempts to formulate GIS constraints in OCL has been made in [CWD00], but the authors realize that OCL do not fit their GIS needs, and modifies the OCL syntax accordingly, again leaving the semantics unclear.
Finally, from the examples mentioned above it is clear that OCL syntax is closer to Programming Languages syntax such as Java and C#, than it is to natural language. OCL is part of a larger system, since OCL is a subset of UML - which from our database constraint point of view has a lot of unnecessary functionality. This results in a comprehensive syntax, with a lot of types, built-in functions and structures that the end-user needs to learn, in order to formulate correct constraints.
HLCL should be closer to natural language, hence easier to use right away. Furthermore it should only consist of the most necessary syntax structures, which should be as simple as possible.

[4]See chapter 9.1 on page 89 for a discussion of the issues concerned with FOL to SQL translation

5.4 EER

To use the Extended Entity Relationship (EER) model is yet another approach for formulating constraints, which is proposed in [GH91]. EER is both a diagram notation, and a very well-founded semantic calculus for queries. The basis for EER is the E/R-diagram, but it has been extended so that it can model more complex relations and express most constraints. In terms of semantic wellfoundnedness, EER is the direct opposite of OCL, a comparison between OCL and EER can be found in [GR98]. But in terms of ease and readability of constraints the syntax is not very simple, to illustrate the point, look at the constraint below.

C2 `all area must contain building`

In order to formulate this in EER, the relation "contain" should be defined as an attribute, which will then be used in the real constraint:

buildings: AREA → set(BUILDING)
buildings(a:AREA) := BTS -[b ‖ (b:BUILDING) ∧ contain(a,b)]-

This would be used in the actual constraint:

∀ (a:AREA) CNT -[b ‖ (b:buildings(a)]- ≥ 1

From the example it can be seen that it is even more difficult to formulate queries in EER than in Predicate Logic or First Order Logic! The end user needs to formulate intermediate functions representing relations, and be comfortable with usage of operators such as CNT (Count), BTS (Bag To Set), and a number of other built-in functions in EER. Therefore EER does not seem as a good choice for making constraint formulation easier.

5.5 Concluding Remarks

We have researched through articles from the last 20 years, and yet it seems there is not a single integrity constraint language which suits all our needs at the same time. COLAN has a syntax close to HLCL, but it is not as simple and close to natural language as HLCL. Description Logic is well-defined and can even reason with the defined constraints, but is not expressive enough for our needs. EER has a well-defined semantic model, but is too complicated to use to fill the gap between natural language and Predicate Logic. OCL is probably the most widely used constraint language currently, but it seems to have been developed as a constraint *specification* language, not an actual constraint language, and its underlying semantic model is ambiguous and unclear. We can therefore conclude that it seems there is a need for a new

constraint language like HLCL which is both easy to use and has a formal semantic model.

Chapter 6

From HLCL to Predicate Logic

The following sections will contain a formal strategy for translating HLCL expressions into Predicate Logic. The translation strategy is split up in two parts: First a "simple translation strategy" is presented which deals with HLCL sentences without user-defined variables and predicates. Second an "advanced translation strategy" is presented, which solves the problems that user-defined variables raise.
We have chosen this approach since it is easier to understand the general ideas behind the translation when looking at the simple strategy, and then be gently introduced to more "cluttered" advanced strategy.
Finally the two strategies will be merged in the "Full Translation Strategy" chapter which summarizes the full strategy. When formalizing the strategies λ-notation will be used, which is explained in chapter A.5 on page 128.

6.1 Macro Functionality

Before any of the translation strategies are applied to an HLCL-expression, the class definitions are subsidized with their full HLCL-definitions. The functionality behind class definitions is very simple: The HLCL-system searches the HLCL-expression for any definitions. For each definition found, it substitutes the definition name with the full HLCL-definition. The substitutions are applied recursively, such that before a substitution is made, it is checked if the substituted definition it self has any definitions, which are then substituted etc. If the HLCL-expression contains no definitions, the expression is left untouched.

6.2 Simple Translating Strategy

The top sentence constructs in our abstract syntax are the *"allmust"* and *"nomay"* constructs. The strategy for translating these two constructs is close to our semantic model: *"allmust"* is being translated as implication,

and *"nomay"* is being translated as an conjunction. This can be seen below:

$$\mathcal{T}[allmust(P,Q)] = \forall X \mathcal{T}[P](X) \rightarrow \mathcal{T}[Q](X)$$
$$\mathcal{T}[nomay(P,Q)] = \neg\exists X \mathcal{T}[P](X) \wedge \mathcal{T}[Q](X)$$

In the simplest case, the P and Q from above is *"class"* constructs. These are translated into a λ-expression as seen below.

$$\mathcal{T}[class(C_{id}, [])] = (\lambda x.C_{id}(x))$$

If the class expression contains a relational path, the class expression is translated as above, and conjugated with the translated relational path as seen below:

$$\mathcal{T}[class(C_{id}, RC)] = (\lambda x.C_{id}(x) \wedge \mathcal{T}[RC](x))$$

An example is shown below to illustrate the concepts, before we move on to the more advanced constructs

Example 6.2.1
The simplest HLCL expression can be seen below

`all` c_0 `must` c_1

This expression would in the abstract syntax look as seen below

allmust(class(c_0*,[]),class(*c_1*,[])*

Applying $\mathcal{T}$ to the *"allmust"* construct results in:

$$\forall X \mathcal{T}[class(c_0, [])](X) \rightarrow \mathcal{T}[class(c_0, [])](X)$$

Which would be further translated into

$$\forall X(\lambda x.C_0(x))(X) \rightarrow (\lambda x.C_1(x))(X)$$

Now the HLCL-expression is translated, but in order to get the actual Predicate Logic-expression we need to recursively apply λ-reduction. In this example we only need one λ-reduction which gives us the final correct result:

$$\forall X C_0(X) \rightarrow C_1(X)$$

Looking at the example above, one should notice how the the variable X is quantified and passed along to the next iteration of $\mathcal{T}$ in the top constructs.

Furthermore the $class(C_{id}, [])$ structure will always be the innermost construct in an HLCL sentence; notice it translates into a λ-function without argument. This is because $\mathcal{T}$ eventually will evaluate to a λ-expression, to which the X will be used as an argument, which will ensure that the subexpression uses the correct variable names.

All the *"relclass"* constructs introduce a new variable Y which they quantify and pass on the following class expression. The variable Y should always be the next available variable. The *"exrelclass"* has a translation similar to the semantic model, which can be seen below

$$\mathcal{T}[exrelclass(R_{id}, CE)] = (\lambda x.\exists Y(R_{id}(x, Y) \wedge \mathcal{T}[CE](Y)))$$

The *"allrelclass"* is translated similar to *"exrelclass"* but with a different quantifier as seen below.

$$\mathcal{T}[allrelclass(R_{id}, CE)] = (\lambda x.\forall Y(R_{id}(x, Y) \rightarrow \mathcal{T}[CE](Y)))$$

The *"solelyrelclass"* quantifies the expression similar to the "all" keyword, but has a reversed implication.

$$\mathcal{T}[solelyrelclass(R_{id}, CE)] = (\lambda x.\forall Y(\mathcal{T}[CE](Y) \rightarrow R_{id}(x, Y)))$$

Finally there is the numerical quantifications, where we use a special numerical quantifier ($\exists_{CompID,Integer}$) to illustrate the usage. This numerical quantifier is not part of Predicate Logic nor DATALOG$^{\neg}$ and will therefore only be used for SQL translation purposes. The translation can be seen below

$$\mathcal{T}[numrelclass(eq, N, R_{id}, CE)] = (\lambda x.\exists Y_{=N}(R_{id}(x, Y) \wedge \mathcal{T}[CE](Y)))$$
$$\mathcal{T}[numrelclass(ge, N, R_{id}, CE)] = (\lambda x.\exists Y_{\geq N}(R_{id}(x, Y) \wedge \mathcal{T}[CE](Y)))$$
$$\mathcal{T}[numrelclass(le, N, R_{id}, CE)] = (\lambda x.\exists Y_{\leq N}(R_{id}(x, Y) \wedge \mathcal{T}[CE](Y)))$$

Attributes are handled similar to their semantic model. "Regular Attributes", i.e. attribute paths which states equality can be expressed directly in Predicate Logic as seen below:

$$\mathcal{T}[attribute(A_{id}, eq, val)] = (\lambda x.A_{id}(x, val))$$

Numerically quantified attribute paths has an extra argument stating the comparison method. These can, as in the case of numerical quantifiers, not be used in DATALOG$^{\neg}$, but are also used for SQL purposes. The translation strategy can be seen below:

$$\mathcal{T}[attribute(A_{id}, Comp_{id}, val)] = (\lambda x.A_{id}(x, Comp_{id}, val))$$

The logical operators can be translated directly into Predicate Logic:

$$\mathcal{T}[or(P,Q)] = (\lambda x.\mathcal{T}[P](x) \vee \mathcal{T}[Q](x))$$
$$\mathcal{T}[and(P,Q)] = (\lambda x.\mathcal{T}[P](x) \wedge \mathcal{T}[Q](x))$$
$$\mathcal{T}[ornot(P,Q)] = (\lambda x.\mathcal{T}[P](x) \vee \neg\mathcal{T}[Q](x))$$
$$\mathcal{T}[andnot(P,Q)] = (\lambda x.\mathcal{T}[P](x) \wedge \neg\mathcal{T}[Q](x))$$

In order to get a better feel for the translation strategy a couple of examples will be given in the following sub chapters.

Example 6.2.2
Let us take a look at one of the simplest running examples:

C2 `all area must contain building`

In our abstract syntax the sentence above would look like this:

$$allmust(class(area, []), exrelclass(contain, class(building, [])))$$

By using the simple strategy presented above we get the following:

$$\forall X\mathcal{T}[class(area, [])](X) \rightarrow \mathcal{T}[exrelclass(contain, class(building, []))](X)$$
$$\forall X(\lambda x.area(x))(X) \rightarrow (\lambda x.\exists Y(contain(x,Y) \wedge \mathcal{T}[class(building, [])](Y))(X)$$
$$\forall X(\lambda x.area(x))(X) \rightarrow (\lambda x.\exists Y(contain(x,Y) \wedge (\lambda x.building(x))(Y))(X)$$

Then we can apply λ-reduction

$$\forall X(\lambda x.area(x))(X) \rightarrow (\lambda x.\exists Y(contain(x,Y) \wedge building(Y)))(X)$$

$$\forall X area(X) \rightarrow \exists Y(contain(X,Y) \wedge building(Y))$$

The above Predicate Logic statement can be read as: *"for all areas there exists a building which is contained in the area"*, which is obviously the correct interpretation of the above HLCL expression.

Example 6.2.3
Then we can move on and check a more complicated example, namely one using relational paths:

C4 `all building type industrial must beusedby company`

In our abstract syntax this sentence would look like:

$$allmust(class(building, attribute(type, eq, building),\\ exrelclass(beusedby, class(company, []))))$$

By using the simple strategy presented above we get the following:

$$\forall X \mathcal{T}[class(building, attribute(type, eq, industrial)](X) \rightarrow\\ \mathcal{T}[exrelclass(beusedby, class(company, []))](X)$$

The right-hand side can be treated similar to example 6.2.2 on the preceding page, so in the following we will only look at the left-hand side:

$$\forall X \mathcal{T}[class(building, attribute(type, eq, building)](X)\\ \forall X(\lambda x.building(x) \wedge \mathcal{T}[attribute(type, eq, industrial)])(x)(X)\\ \forall X(\lambda x.building(x) \wedge (\lambda x.type(x, industrial))(x)(X)$$

Then we can apply λ-reduction:

$$\forall X(\lambda x.building(x) \wedge type(x, industrial))(X)\\ \forall X building(X) \wedge type(X, industrial)$$

The full expression would then look like (translating the right-hand side as example 6.2.2 on the facing page)

$$\forall X building(X) \wedge type(x, industrial)\\ \rightarrow \exists Y(beusedby(X, Y) \wedge company(Y)$$

This reads as *"all buildings which are of type industrial must be used by a company"*. This is the correct interpretation.

Example 6.2.4
Let us look at an example with composite structures. Take a look at the HLCL expression below:

C6 `all area must contain building or contain lake`

In our abstract syntax this would be represented as:

$$allmust(class(area, []), or(exrelclass(contain, class(building, [])),\\ exrelclass(contain, class(lake, []))))$$

Using the translation strategy results in:

$$\forall X \mathcal{T}[class(area, [])](X) \rightarrow\\ \mathcal{T}[or(exrelclass(contain, class(building, [])),$$

$$exrelclass(contain, class(lake, []))](X)$$

The left-hand side can be translated as in example 6.2.2 on page 56, in the following we will only look at the right-hand side:

$$\mathcal{T}[or(exrelclass(contain, class(building, [])),\\ exrelclass(contain, class(lake, [])))](X)$$

$$\lambda x.(\mathcal{T}[exrelclass(contain, class(building, []))](x)\\ \vee \mathcal{T}[exrelclass(contain, class(lake, []))](x))(X)$$

$$\lambda x.(\lambda x.(\exists Y(contain(x, Y) \wedge (\lambda x.building(x))(x)))(x)\\ \vee(\lambda x.(\exists Y(contain(x, Y) \wedge (\lambda x.lake(x))(x)))(x))(X)$$

Then we can apply λ-reduction:

$$\lambda x.(\lambda x.(\exists Y(contain(x, Y) \wedge building(x)))(x)\\ \vee(\lambda x.(\exists Y(contain(x, Y) \wedge lake(x)))(x))(X)\\ \lambda x.(\exists Y(contain(x, Y) \wedge building(x)) \vee\\ (\exists Y(contain(x, Y) \wedge lake(x)))(X)\\ \exists Y(contain(X, Y) \wedge building(X)) \vee (\exists Y(contain(X, Y) \wedge lake(X))$$

The total expression is then

$$\forall X area(X) \rightarrow\\ \exists Y(contain(X, Y) \wedge building(X)) \vee (\exists Y(contain(X, Y) \wedge lake(X))$$

This reads: *"all areas must contain either a building or contain a lake"*, which is the correct result in Predicate Logic.

6.3 Advanced Translating Strategy

In this chapter we will deal with the problems that arise when introducing user-defined variables.

6.3.1 Issues

Introducing user-defined variables is not just a matter of changing the outer variable in our original translation strategy with the user-defined variable. In order to illustrate the problem, let us look at an *"exrelclass"* which relates to a user-defined variable. From our simple strategy we have a rule which looks like:

$$\mathcal{T}[exrelclass(R_{id}, CE)] = (\lambda x.\exists Y(R_{id}(x, Y) \wedge \mathcal{T}[CE](Y)))$$

In order to adapt this to user-defined variables, the first step is to replace the existentially quantified Y to the correct name of the user-defined variable, which would result in the rule below:

$$\mathcal{T}[exrelclass(R_{id}, varclass(C_{id}, V_{id}, RC)] = \\ (\lambda x.\exists V_{id}(R_{id}(x, V_{id}) \wedge \mathcal{T}[varclass(C_{id}, V_{id}, RC)](V_{id})))$$

If we used the above rule alone we would end up with an expression of the form:

$$...\exists V_{id}(class(V_{id})...V_{id}...)... \rightarrow ...\exists V_{id}(class(V_{id})...V_{id}...)...$$

This looks correct, but reveals a new problem - namely quantifier scoping of the existentially quantified V_{id}. This problem relates to the so-called: "donkey sentences" which are explained in appendix A.2 on page 120. This expression has two existentially quantified variables V_{id}, but the expression fails to express the fact that that the variables V_{id} should be equal. This is solved by using a single global universal quantifier instead of the two local existential quantifiers. This will express that the variables should be equal. Furthermore we should also get rid of the double class expression: "$class(V_{id})$". It should only be initiated on the left-hand side, such that we get an expression, similar in structure to the one seen below:

$$\forall V_{id}...class(V_{id})(...V_{id}...) \rightarrow (...V_{id}...)$$

In the following section we will extend the simple translation strategy with the modifications mentioned above. The result is a somewhat more "cluttered" translation strategy which can be harder to understand. The key is to remember that the only difference is, that the advanced strategy quantifies all user-defined variables globally, and moves the class predicates to the beginning of the Predicate Logic expression.

6.3.2 Strategy

Sub-strategy

In order to make the above mentioned modifications, an extra sub strategy $\mathcal{S}$ is introduced. $\mathcal{S}$ takes two arguments: The first argument is an HLCL expression, and the second argument is a variable list. $\mathcal{S}$ searches through the given HLCL expression and collects all user-defined variables which are not in the given list. The result is a Predicate Logic expression which consists of all the user-defined variables universally quantified together with class expression. An example of how $\mathcal{S}$ works can be seen below:

$$\mathcal{S}[varclass(Building, A, [])][\,] = \forall A Building(A)$$

$$\mathcal{S}[varclass(Building, A, [])][A] = [\,]$$

$$\mathcal{S}[or(varclass(Building, A, []), varclass(Area, B, []))] = \forall A Building(A) \wedge \forall B Area(B)$$

$\mathcal{S}$ only returns one class and quantification for each user-defined variable, so even though it encounters the same user-defined variable twice, only a single class expression is returned for each user-defined variable. The formal specification of $\mathcal{S}$ can be found in chapter 6.4 on page 65.

Actual Strategy

In general the advanced rules extend the rules from our simple strategy. Instead of operating with Ps and Qs as we did in the simple strategy, we use specific construct-matching, such as *"allmust(varclass(_,_,_))"* for matching an *"allmust"* construct with a *"varclass"* construct. The topconstruct *"allmust"* can be defined as:

$$\mathcal{T}[allmust(class(C_{id}, RC), Q)] = \forall X(\lambda x.C_{id}(x) \wedge \mathcal{S}[and(RC, Q)][] \wedge \mathcal{T}[RC](x))(X) \rightarrow \mathcal{T}[Q](X)$$

The *"allmust"* construct uses the $\mathcal{S}$ to universally quantify variables and pull out "class expressions" as explained above. Otherwise the expression is very similar to the one in the simple strategy. If the expression starts with a classexpression which uses a user-defined variable, the strategy looks as the following:

$$\mathcal{T}[allmust(varclass(C_{id}, V_{id}, RC), Q)] = \forall V_{id}(\lambda x.C_{id}(x) \wedge \mathcal{S}[and(RC, Q)][V_{id}] \wedge \mathcal{T}[RC](x))(V_{id}) \rightarrow \mathcal{T}[Q](V_{id})$$

$$\mathcal{T}[allmust(varclass(C_{id}, V_{id}, []), Q)] = \forall V_{id} C_{id}(V_{id}) \wedge \mathcal{S}[and(RC, Q)][V_{id}] \rightarrow \mathcal{T}[Q](V_{id})$$

The above strategy quantifies the first user-defined variable V_{id}, and uses it as the second argument in $\mathcal{S}$ since the it is already correctly scoped. Furthermore V_{id} is used as argument in the λ-functions, such that the classes uses the user-defined variable. The second case is simply a specialization of the first one, namely that if the left-hand side only contains a *"varclass* component, then the overall strategy is a bit simpler.
The *"nomay"* constructs have similar rules seen below:

$$\mathcal{T}[nomay(class(C_{id}, RC), Q)] = \\ \neg\exists X(\lambda x.C_{id}(x) \wedge \mathcal{S}[and(RC, Q)][] \wedge \mathcal{T}[RC](x))(X) \wedge \mathcal{T}[Q](X)$$

$$\mathcal{T}[nomay(varclass(C_{id}, V_{id}, RC), Q)] = \\ \neg\exists V_{id}(\lambda x.C_{id}(x) \wedge \mathcal{S}[and(RC, Q)][V_{id}] \wedge \mathcal{T}[RC](x))(V_{id}) \wedge \mathcal{T}[Q](V_{id})$$

$$\mathcal{T}[nomay(varclass(C_{id}, V_{id}, []), Q)] = \\ \neg\exists V_{id} C_{id}(V_{id}) \wedge \mathcal{S}[and(RC, Q)][V_{id}] \wedge \mathcal{T}[Q](V_{id})$$

The *"varclass"* constructs are treated just like *"class"* constructs, except the variable inside the class which is already given.

$$\mathcal{T}[varclass(C_{id}, V_{id}, [])] = (\lambda x.C_{id}(V_{id}))$$
$$\mathcal{T}[varclass(C_{id}, V_{id}, RC)] = (\lambda x.C_{id}(V_{id}) \wedge \mathcal{T}[RC](V_{id}))$$

Notice how the first *"varclass"* strategy is a constant λ-expression. Nevertheless we still need to have the full λ-expression, so number of arguments and λ-expressions add up in the translation strategy. The *"exrelclass"* is extended as explained the issues section.

$$\mathcal{T}[exrelclass(R_{id}, varclass(C_{id}, V_{id}, []))] = (\lambda x.R_{id}(x, V_{id}))$$
$$\mathcal{T}[exrelclass(R_{id}, varclass(C_{id}, V_{id}, RC)] = (\lambda x.R_{id}(x, V_{id}) \wedge \mathcal{T}[RC](V_{id}))$$

These rules do not quantify the expression with the user-defined variable, nor add a classs expression corresponding to $C_{id}(V_{id})$, this is done by the sub-strategy $\mathcal{S}$ which was applied in the translation of the topconstructs. Variables can only be used with existentially quantified classes, so the rules concerning the other relational paths stays the same. Finally there is the predicates which are translated as seen below:

$$\mathcal{T}[unarypredicate(P_{id}, V_{id})] = (\lambda x.P_{id}(V_{id}))$$
$$\mathcal{T}[binarypredicate(P_{id}, V_{id1}, V_{id2})] = (\lambda x.P_{id}(V_{id1}, V_{id2}))$$

In order to understand the presented strategies better, let us look at a few example translations seen below.

Example 6.3.1
Let us look at an example translation of an HLCL expression using variables:

```
all building B must touch building B
```

In abstract syntax the above expression would look like:

allmust(varclass(building,B,[]),exrelclass(touch,varclass(building,B,[])))

Applying the translation strategy results in

$$\forall B\, building(B) \wedge \mathcal{S}[and(varclass(building, B, []),\\ exrelclass(touch, varclass(building, B, [])))][B] \rightarrow\\ \mathcal{T}[exrelclass(touch, varclass(building, B, []))](B)$$

The left-hand side of the implication in the expression uses the strategy $\mathcal{S}$, which evaluates into nothing, since there are no further user-defined variables in the expression. The result is that the full left-hand side of the expression evaluates into the Predicate Logic expression seen below:

$$\forall\, BbuildingB$$

The right-hand side of the expression can seen below

$$\mathcal{T}[exrelclass(touch, varclass(building, B, []))](B)$$

Using the advanced strategy $\mathcal{T}$ is translated into the λ-expression seen below:

$$(\lambda x.touch(x, B))(B)$$

Using λ-reduction we reach:

$$touch(B, B)$$

Putting the left-hand and the right-hand side together we get the final Predicate Logic expression:

$$\forall B\, buildingB \rightarrow touch(B, B)$$

Which is clearly correct.

Example 6.3.2
Let us look at an example translation of an HLCL expression using user-defined predicates:

```
no area A must someareafunction(A)
```

In abstract syntax the above expression would look like:

nomay(varclass(area,A,[]),unarypredicate(someareafunction,A))

Applying the translation strategy results in

$$\neg\exists A\, area(A) \wedge \mathcal{S}[and(varclass(area, A, []),\\ unarypredicate(someareafunction, A))][A] \rightarrow\\ \mathcal{T}[unarypredicate(someareafunction, A)](A)$$

The left-hand side of the implication in the expression uses the strategy $\mathcal{S}$, which evaluates into nothing, since there are no further user-defined variables in the expression. The full left-hand side of the expression evaluates into the Predicate Logic expression seen below:

$$\neg\exists A\, area(A)$$

The right-hand side of the expression as seen below

$$\mathcal{T}[unarypredicate(someareafunction, A)](A)$$

Using the new strategy the above is translated into the λ-expression seen below:

$$(\lambda x.someareafunction(A))(A)$$

Using λ-reduction (in where we reduce a constant function) we reach:

$$someareafunction(A)$$

Putting the left-hand and the right-hand side together we get the final Predicate Logic expression:

$$\neg\exists A\, area(A) \rightarrow someareafunction(A)$$

Which is clearly correct.

Example 6.3.3
Then let us look at an example, which uses both simple and advanced rules. Take a look at the HLCL expression seen below:

C11
```
all area intersectedby road R must
contain building intersectedby road R
```

In abstract syntax this would look like

$$allmust(class(area, exrelclass(intersectedby, varclass(road, R, []))),\\ exrelclass(contain, class(building,\\ exrelclass(intersectedby, varclass(road, R, [])))))$$

Applying the strategy $\mathcal{T}$ results in the following, quite confusing, fragment seen below:

$$\forall X(\lambda x.area(x) \wedge \\ \mathcal{S}[and(exrelclass(intersectedby, varclass(road, R, [])), \\ exrelclass(contain, class(building, exrelclass(intersectedby, \\ varclass(road, R, [])))))][] \wedge \mathcal{T}[exrelclass(contain, class(building, \\ exrelclass(intersectedby, varclass(road, R, [])))))](x))(X) \\ \rightarrow \mathcal{T}[exrelclass(contain, \\ class(building, exrelclass(intersectedby, varclass(road, R, [])))))](X)$$

In order to get an overview, the expression above is split it up in the left- and right-hand side of the implication. If we look at the left-hand side, then the substrategy $\mathcal{S}$ will simply return

$$\forall Rroad(R)$$

So the entire left-hand side becomes:

$$\forall X(\lambda x.area(x) \wedge \forall Rroad(R) \wedge \mathcal{T}[exrelclass(contain, \\ class(building, exrelclass(intersectedby, varclass(road, R, [])))))](x))(X)$$

Applying $\mathcal{T}$ results in the following expressions

$$\forall X(\lambda x.area(x) \wedge \forall Rroad(R) \wedge (\lambda x.\forall Y(contain(x, Y) \rightarrow \\ \mathcal{T}[class(building, exrelclass(intersectedby, varclass(road, R, [])))))](Y))(x))(X)$$

$$\forall X(\lambda x.area(x) \wedge \forall Rroad(R) \wedge (\lambda x.\forall Y(contain(x, Y) \rightarrow \\ (\lambda x.building(x) \wedge \mathcal{T}[exrelclass(intersectedby, varclass(road, R, []))])(x))(Y)))(X)$$

$$\forall X(\lambda x.area(x) \wedge \forall Rroad(R) \wedge (\lambda x.\exists Y(contain(x, Y) \wedge \\ (\lambda x.building(x) \wedge (\lambda x.intersectedby(x, R)))(x))(Y)))(X)$$

The expression is finally translated and we can apply λ-reduction:

$$\forall X(\lambda x.area(x) \wedge \forall Rroad(R) \wedge (\lambda x.\exists Y(contain(x, Y) \wedge \\ (\lambda x.building(x) \wedge intersectedby(x, R))(Y)))(X)$$

$$\forall X(\lambda x.area(x) \wedge \forall Rroad(R) \wedge (\lambda x.\exists Y(contain(x, Y) \wedge \\ building(Y) \wedge intersectedby(Y, R))(x))(X)$$

$$\forall X(\lambda x.area(x) \wedge \forall Rroad(R) \wedge \\ \exists Y(contain(x, Y) \wedge building(Y) \wedge intersectedby(Y, R))(X)$$

The final expression corresponding to the right-hand side can be seen below:

$$\forall X area(X) \wedge \forall R road(R) \wedge \\ \exists Y(contain(X,Y) \wedge building(Y) \wedge intersectedby(Y,R))$$

Going back to the right-hand side we have:

$$\mathcal{T}[exrelclass(contain, class(building, exrelclass(intersectedby, \\ varclass(road, R, [])))))](X)$$

Applying the strategy $\mathcal{T}$ results in

$$(\lambda x.\exists Y(contain(x,Y) \wedge \mathcal{T}[class(building, exrelclass(intersectedby, \\ varclass(road, R, [])))](Y)))(X)$$

$$(\lambda x.\exists Y(contain(x,Y) \wedge (\lambda x.building(x) \wedge \\ \mathcal{T}[exrelclass(intersectedby, varclass(road, R, []))])(x))(Y)))(X)$$

$$(\lambda x.\exists Y(contain(x,Y) \wedge (\lambda x.building(x) \wedge \\ (\lambda x.intersectedby(x,R))(x))(Y)))(X)$$

Applying λ-reduction yields:

$$(\lambda x.\exists Y(contain(x,Y) \wedge (\lambda x.building(x) \wedge intersectedby(x,R))(Y)))(X)$$

$$(\lambda x.\exists Y(contain(x,Y) \wedge building(Y) \wedge intersectedby(Y,R))(X)$$

$$\exists Y(contain(X,Y) \wedge building(Y) \wedge intersectedby(Y,R))$$

Which means the final expression is:

$$\forall X\ area(X) \wedge \forall R\, road(R) \wedge \exists Y(contain(X,Y) \wedge \\ building(Y) \wedge intersectedby(Y,R)) \rightarrow \exists Y(contain(X,Y) \\ \wedge building(Y) \wedge intersectedby(Y,R))$$

Which is clearly correct.

6.4 Full Translating Strategy

Let us recapitulate and look at all rules, which constitutes the full translating strategy. It is important to notice that these transformation rules do not ensure wellformedness. The translation strategy translates "blindly" and requires a separate wellformedness check at a later stage. A summary of all

rules can be seen below:

Strategy $\mathcal{T}$

$$\mathcal{T}[allmust(class(C_{id}, RC), Q)] = \\ \forall X(\lambda x.C_{id}(x) \wedge \mathcal{S}[and(RC, Q)][] \wedge \mathcal{T}[RC](x))(X) \rightarrow \mathcal{T}[Q](X)$$

$$\mathcal{T}[allmust(varclass(C_{id}, V_{id}, RC), Q)] = \\ \forall V_{id}(\lambda x.C_{id}(x) \wedge \mathcal{S}[and(RC, Q)][V_{id}] \wedge \mathcal{T}[RC](x))(V_{id}) \rightarrow \mathcal{T}[Q](V_{id})$$

$$\mathcal{T}[allmust(varclass(C_{id}, V_{id}, []), Q)] = \\ \forall V_{id} C_{id}(V_{id}) \wedge \mathcal{S}[and(RC, Q)][V_{id}] \rightarrow \mathcal{T}[Q](V_{id})$$

$$\mathcal{T}[nomay(class(C_{id}, RC), Q)] = \\ \neg\exists X(\lambda x.C_{id}(x) \wedge \mathcal{S}[and(RC, Q)][] \wedge \mathcal{T}[RC](x))(X) \wedge \mathcal{T}[Q](X)$$

$$\mathcal{T}[nomay(varclass(C_{id}, V_{id}, RC), Q)] = \\ \neg\exists V_{id}(\lambda x.C_{id}(x) \wedge \mathcal{S}[and(RC, Q)][V_{id}] \wedge \mathcal{T}[RC](x))(V_{id}) \wedge \mathcal{T}[Q](V_{id})$$

$$\mathcal{T}[nomay(varclass(C_{id}, V_{id}, []), Q)] = \\ \neg\exists V_{id} C_{id}(V_{id}) \wedge \mathcal{S}[and(RC, Q)][V_{id}] \wedge \mathcal{T}[Q](V_{id})$$

$$\mathcal{T}[exrelclass(R_{id}, class(C_{id}, RC))] = (\lambda x.\exists Y(R_{id}(x, Y) \wedge \mathcal{T}[class(C_{id}, RC)](Y)))$$
$$\mathcal{T}[exrelclass(R_{id}, varclass(C_{id}, V_{id}, []))] = (\lambda x.R_{id}(x, V_{id}))$$
$$\mathcal{T}[exrelclass(R_{id}, varclass(C_{id}, V_{id}, RC)] = (\lambda x.R_{id}(x, V_{id}) \wedge \mathcal{T}[RC](V_{id}))$$
$$\mathcal{T}[allrelclass(R_{id}, class(C_{id}, RC))] = (\lambda x.\forall Y(R_{id}(x, Y) \rightarrow \mathcal{T}[class(C_{id}, RC)](Y)))$$
$$\mathcal{T}[solelyrelclass(R_{id}, CE)] = (\lambda x.\forall Y(\mathcal{T}[CE](Y) \rightarrow R_{id}(x, Y)))$$
$$\mathcal{T}[numrelclass(eq, N, R_{id}, CE)] = (\lambda x.\exists Y_{=N}(R_{id}(x, Y) \wedge \mathcal{T}[CE](Y)))$$
$$\mathcal{T}[numrelclass(ge, N, R_{id}, CE)] = (\lambda x.\exists Y_{\geq N}(R_{id}(x, Y) \wedge \mathcal{T}[CE](Y)))$$
$$\mathcal{T}[numrelclass(le, N, R_{id}, CE)] = (\lambda x.\exists Y_{\leq N}(R_{id}(x, Y) \wedge \mathcal{T}[CE](Y)))$$

$$\mathcal{T}[class(C_{id}, RC)] = (\lambda x.C_{id}(x) \wedge \mathcal{T}[RC](x))$$
$$\mathcal{T}[class(C_{id}, [])] = (\lambda x.C_{id}(x))$$
$$\mathcal{T}[varclass(C_{id}, V_{id}, [])] = (\lambda x.C_{id}(V_{id}))$$
$$\mathcal{T}[varclass(C_{id}, V_{id}, RC)] = (\lambda x.C_{id}(V_{id}) \wedge \mathcal{T}[RC](V_{id}))$$

$$\mathcal{T}[unarypredicate(P_{id}, V_{id})] = (\lambda x.P_{id}(V_{id}))$$
$$\mathcal{T}[binarypredicate(P_{id}, V_{id1}, V_{id2})] = (\lambda x.P_{id}(V_{id1}, V_{id2}))$$

$$\mathcal{T}[or(P, Q)] = (\lambda x.\mathcal{T}[P](x) \vee \mathcal{T}[Q](x))$$
$$\mathcal{T}[and(P, Q)] = (\lambda x.\mathcal{T}[P](x) \wedge \mathcal{T}[Q](x))$$
$$\mathcal{T}[ornot(P, Q)] = (\lambda x.\mathcal{T}[P](x) \vee \neg\mathcal{T}[Q](x))$$
$$\mathcal{T}[andnot(P, Q)] = (\lambda x.\mathcal{T}[P](x) \wedge \neg\mathcal{T}[Q](x))$$

- Where Y is next available variable.

Strategy $\mathcal{S}$

The strategy $\mathcal{S}$ is defined below:

$\mathcal{S}[varclass(C_{id}, V_{id}, RC)][List] = \forall V_{id} C_{id}(V_{id}) \wedge \mathcal{S}[RC][List + V_{id}]$
iff V_{id} is not in list

$\mathcal{S}[varclass(C_{id}, V_{id}, RC)][List] = \mathcal{S}[RC][List]$
iff V_{id} is in list

$\mathcal{S}[attribute(A_{id}, eq, V_{id})][List] = \forall V_{id} \mathcal{S}[RC][List + V_{id}]$
iff V_{id} is user-defined and not in list

$\mathcal{S}[attribute(A_{id}, eq, V_{id})][List] = \mathcal{S}[RC][List]$
iff V_{id} is not or user-defined and in list

$\mathcal{S}[class(C_{id}, RC)][List] = \mathcal{S}[RC][List]$
$\mathcal{S}[attribute(A_{id}, Comp_{id}, V_{id})][\_] = []$
$\mathcal{S}[unarypredicate(P_{id}, V_{id})][\_] = []$
$\mathcal{S}[binarypredicate(P_{id}, V_{id1}, V_{id2})][\_] = []$
$\mathcal{S}[exrelclass(R_{id}, CE)][List] = \mathcal{S}[CE][List]$
$\mathcal{S}[allrelclass(R_{id}, CE)][List] = \mathcal{S}[CE][List]$
$\mathcal{S}[solelyrelclass(R_{id}, CE)][List] = \mathcal{S}[CE][List]$
$\mathcal{S}[numrelclass(Comp_{id}, N, R_{id}, CE)][List] = \mathcal{S}[CE][List]$
$\mathcal{S}[or(P, Q)][List] = \mathcal{S}[List]\mathcal{S}[List]$
$\mathcal{S}[and(P, Q)][List] = \mathcal{S}[List]\mathcal{S}[List]$
$\mathcal{S}[ornot(P, Q)][List] = \mathcal{S}[List]\mathcal{S}[List]$
$\mathcal{S}[andnot(P, Q)][List] = \mathcal{S}[List]\mathcal{S}[List]$
$\mathcal{S}[][] = []$

Chapter 7

Intermediate Steps

The translation strategy $\mathcal{T}$ from chapter 6 on page 53 results in a Predicate Logic expression. In order to translate this Predicate Logic into Extended DATALOG which will be the basis for SQL translation and finally into DATALOG$^\neg$ we need to make some logical rewritings. These rewritings occur in two steps as seen below:

Predicate Logic$\rightarrow$ Extended DATALOG $\rightarrow$ DATALOG$^\neg$

The Predicate Logic is first rewritten into Extended DATALOG. The Extended DATALOG can then be further translated into DATALOG$^\neg$. The specifics of the two kinds of DATALOG can be seen below:

Extended DATALOG**:** This is the logic representation which is fine-tuned for SQL translation. Basically we keep all operators and quantifiers which are available in SQL. Furthermore the numerical quantifiers are kept.

DATALOG$^\neg$**:** This is the purest logic representation, stripped of all quantifiers and is basically pure clauses including negation (negation-as-failure, see [Bra01]). This is meant to be an easy-to-read alternative, when one wants to see what the constraint actually expresses (which can be hard in SQL). Furthermore it is a well-defined interface with can be used by external applications. As previously mentioned only a subset of the HLCL namely HLCL without numerical quantifications and numerical attribute values can be translated into DATALOG$^\neg$.

The rewriting steps have been divided into two parts: The first 3 steps are the Predicate Logic to Extended DATALOG translation, and the last steps are the Extended DATALOG to DATALOG$^\neg$ translation. [Nil80] gives a 9-step procedure for rewriting the Predicate Logic into pure clauses. Theorems 7.1 to 7.7 are from this procedure, which form the basis of the logical rewriting steps seen below:

Steps to Reach Extended DATALOG

1. Eliminate Internal Universal Quantifiers
2. Eliminate Implication Symbols
3. Negate Expression.

Final Steps to Reach DATALOG$^\neg$

1. Reduce Scope of Negation
2. Distribute Conjunctions
3. Remove Outer Universal Quantifiers
4. Eliminate Existential Quantifiers
5. Eliminate $\vee$ symbols

In the following sub chapters we will look at these steps one by one, and manually translate one of the running examples in order to illustrate the principles.

7.1 Interface Definitions

In the following two sub chapters a formal definition of the Extended DATALOG and DATALOG$^\neg$ will be given.

7.1.1 Extended DATALOG

In this chapter a formal definition of our Extended DATALOG will be given. It is important to realize that Extended DATALOG is only a subset of DATALOG, the reason is that only certain structures are generated from the strategy $\mathcal{T}$.

⟨ext datalog⟩::= **all(**V_{id}**,**⟨ext datalog⟩**)** |
all(V_{id}**,and(**⟨subexp⟩**,**⟨subexp⟩**))**

⟨subexp⟩::= **and(**⟨class⟩**,**⟨quantifierexp⟩**)** | **or(**⟨class⟩**,**⟨quantiferexp⟩**)**
| ⟨class⟩ | ⟨predicate⟩ | ⟨quantifierexp⟩

⟨quantifierexp⟩::= **not(**⟨quantifierexp⟩**)** |
and(⟨relation⟩**,**⟨subexp⟩**)** | ⟨relation⟩ |
exists(V_{id}**,and(**⟨relation⟩**,**⟨subexp⟩**))** |
exists(V_{id}**,and(**⟨relation⟩**,not(**⟨subexp⟩**)))** |
exists(V_{id}**,and(**⟨subexp⟩**,not(**⟨relation⟩**)))** |
numexists(⟨num quant⟩**,**⟨integer⟩**,**V_{id}**,and(**⟨relation⟩**,**⟨subexp⟩**))**

⟨class⟩::= **class(**C_{id}**,**V_{id}**)**

⟨relation⟩::= **relation**(R$_{id}$,V$_{id}$,**V**$_{id}$)

⟨attribute⟩::= **attribute**(A$_{id}$,V$_{id}$,⟨num quant⟩,⟨value⟩)

⟨predicate⟩::= **unarypredicate**(P$_{id}$,V$_{id}$) | **binarypredicate**(P$_{id}$,V$_{id}$,V$_{id}$)

⟨num quant⟩::= **eq** | **le** | **ge**

⟨integer⟩::= 1 | 2 | 3 | ...

7.1.2 DATALOG¬

In this chapter a formal definition of our DATALOG¬ will be given. DATALOG¬ consists of clauses which each has a form as described below:

⟨datalog clause⟩::= ⟨result⟩ ← ⟨expression⟩

⟨expression⟩::= ⟨entity⟩ | **and**(⟨entity⟩,⟨expression⟩)

⟨entity⟩::=**not**(⟨entity⟩) |
class(C$_{id}$,V$_{id}$) |
attribute(A$_{id}$,V$_{id}$,**eq**,⟨value⟩) |
relation(R$_{id}$,V$_{id}$,V$_{id}$) |
function(F$_{id}$,V$_{id}$ − *list*) |
unarypredicate(P$_{id}$,V$_{id}$) |
binarypredicate(P$_{id}$,V$_{id}$,V$_{id}$) |

⟨result⟩::= *error* | **function**(F$_{id}$,V$_{id}$ − *list*)

7.2 From Predicate Logic to Extended DATALOG

In this sub chapter we will go through the steps to reach the Extended DATALOG from Predicate Logic.

Eliminate Internal Universal Quantifiers

Since SQL does not have an operator for universal quantification, we need to get rid of universal quantifications inside the predicate expression. The method we are using relies on the "closed world assumption", which states that if something is not explicitly stated true, then it is false. This is the case in a database, and theorem 7.1 allows us to replace the universal quantifier, with a double-negated existential quantifier. The result is that instead of saying *"for all X, Y must hold"*, then we say *"there does not exist a X where Y does not hold"*. The formal definition can be seen below:

$$\forall Aexp = \neg\exists\neg exp \tag{7.1}$$

Eliminate Implication Symbols

SQL does not have an implication operator either, so implications symbols should also be rewritten. Elimination of implication, both the outer implication, and any implications inside the expression, is done by applying the theorem below:

$$P \Rightarrow Q \equiv (\neg P) \vee Q \tag{7.2}$$

Negate Expression

The final step is to negate the expression. This is done because the HLCL sentence is formulated as a constraint that should hold for all entities within a given class. But the actual SQL constraint should query entities which break the constraint. Negation is done in differently depending on whether the constraint uses a *"allmust"* or *"nomay"* top construct. In constraints made from the *"allmust"* construct, negation and removal of the outer implication can be done at the same time, using the theorem seen below:

$$\neg(P \Rightarrow Q) \equiv P \wedge \neg(Q) \tag{7.3}$$

As an example look at the Predicate Logic expression below which shows the structure made from an *"allmust"* construct.

$$\forall V_{id}\, Exp1 \rightarrow Exp2$$

Negating the above expression by using the theorem 7.3 results in:

$$error \leftarrow \forall V_{id}\, Exp1 \wedge \neg Exp2$$

Constraints which are made from the *"nomay"* construct has an outer negation by nature. By negating the inside expression we get a "¬∃¬" structure which can be replaced shown in theorem 7.1.
As an example look at the Predicate Logic expression below which shows the structure made from a *"nomay"* construct.

$$\neg\exists V_{id}\, Exp1 \wedge Exp2$$

Negating the above expression and using theorem 7.1 results in:

$$error \leftarrow \forall V_{id}\, Exp1 \wedge Exp2$$

In most of this report we will leave out left-hand side of this expression, and leaving the "error" result implicit. The only time it is shown, is when we reach the DATALOG$^\neg$ interface.

Example 7.2.1

Let us look at an example illustrating the logical rewritings described above:

C14 `all building must type blockbuildings and touch solely building`

According to our translating strategy from chapter 6 on page 53 the above HLCL expression would be translated to the followed Predicate Logic.

$$\forall X building(X) \rightarrow type(X,'blockbuilding') \wedge \forall Y(touch(X,Y) \rightarrow building(Y))$$

First internal universal quantification is removed by applying the theorem 7.1:

$$\forall X building(X) \rightarrow type(X,'blockbuilding') \wedge \neg\exists Y\neg(touch(X,Y) \rightarrow building(Y))$$

Then theorem 7.3 can be applied to remove the inner implication:

$$\forall X building(X) \rightarrow type(X,'blockbuilding') \wedge \neg\exists Y(touch(X,Y) \wedge \neg building(Y))$$

Applying theorem 7.3 removes the outer implication, and negates the expression

$$error \leftarrow \forall X building(X) \wedge \neg(type(X,'blockbuilding') \wedge \neg\exists Y(touch(X,Y) \wedge \neg building(Y)))$$

The expression above is now in Extended DATALOG form and ready for SQL translation.

7.3 From Extended DATALOG to DATALOG¬

In this chapter the final steps to reach DATALOG¬ will be described.

Reduce Scope of Negation

The first step in order to reach DATALOG¬ is to reduce the scope of negation. This means that all negations covering a conjunction or disjunction should be moved "inwards" in the expression. The result should be an expression where only the literals are negated. This can be done by applying DeMorgan's theorem 7.4 and 7.5 seen below and the theorem of double negation, theorem 7.6.

$$\neg(P \vee Q) \equiv \neg P \wedge \neg Q \tag{7.4}$$

$$\neg(P \wedge Q) \equiv \neg P \vee \neg Q \tag{7.5}$$

$$\neg(\neg(P)) \equiv P \tag{7.6}$$

Distribute Conjunctions

The next step is to put the expression in disjunctive normal form. This means that expressions containing the disjunction operator inside a conjugation, should be distributed by applying the distributive law seen in theorem 7.7.

$$P \wedge (Q \vee R) \equiv (P \wedge Q) \vee (P \wedge R) \tag{7.7}$$

This results in an expression consisting of one of more disjuncts each which is a conjugation between one or more literals. This form is sometimes referred to as "disjunctive normal form".

Eliminate Outer Universal Quantifier

The next step is to remove the last universal quantifiers, since the inner universal quantifications are already removed in Extended DATALOG, we only have left to remove the out universal quantifiers. They are simply stripped away, so that a sentence seen below

$$\forall X, Y, Z class(X)....$$

Simply turns into:

$$class(X)....$$

It is important to realize the variables in the expression above are still universally quantified, it is just left implicit, because all variables in DATALOG$^{\neg}$ are implicitly universally quantified.

Eliminate Existential Quantifiers

The next step is to eliminate the existential quantifiers. We do not use method described in [Nil80][1], instead we realize that our existentially quantified variables only come in a certain pattern. E.g. in conjunction with a relational path as the one seen below:

$$...\exists Y relation(X, Y) \wedge class(Y)...$$

The above expression can simply be "pulled" out of the expression, and made into a separate function expression. This means that the expression above turns into a seperate function seen below:

$$f(X) = \forall Y relation(X, Y) \wedge class(Y)$$

[1] Which is briefly described in appendix A.6 on page 129

The existentially quantified subexpression is replace by the function name. Let us look at an example to illustrate the principle.

$$\forall X area(X) \wedge \neg\exists Y(Buildings(Y) \wedge \neg contain(X, Y))$$

Let us then remove the above quantifier by using the principle. We introduce a function f, which can be seen below:

$$f(X) = Buildings(Y) \wedge \neg contain(X, Y)$$

And the original sentence would turn into:

$$\forall X area(X) \wedge \neg f(X)$$

Eliminate $\vee$ Symbols

The final step is to eliminate disjunction symbols. This is simply done by breaking up a clauses containing disjunctions into several clauses, each keeping the left-hand side of the expression. Since the expression is in disjunctive normal form, the $\vee$ should be the outermost operators in the expression. So that a sentence as:

$$error \leftarrow area(X) \vee f(X)$$

Would turn into:

$$error \leftarrow area(X)$$
$$error \leftarrow f(X)$$

Example 7.3.1

Let us continue converting the example which we started in 7.2.1 on page 72.

$$\forall X building(X) \wedge \neg(type(X,' blockbuilding') \wedge \neg\exists Y(touch(X, Y) \wedge \neg building(Y)))$$

The first step is to reduce the scope of the negation, the result can be seen below.

$$\forall X building(X) \wedge \neg type(X,' blockbuilding') \vee \exists Y(touch(X, Y) \wedge \neg building(Y))$$

The next step is to distribute conjunctions according to theorem 7.7 on the facing page

$$\forall X building(X) \wedge \neg type(X,'blockbuilding') \vee building(X) \wedge \exists Y(touch(X,Y) \wedge \neg building(Y))$$

Then we remove outer universal quantifiers

$$building(X) \wedge \neg type(X,'blockbuilding') \vee building(X) \wedge \exists Y(touch(X,Y) \wedge \neg building(Y))$$

And remove existential quantifiers:

$$building(X) \wedge \neg type(X,'blockbuilding') \vee building(X) \wedge f_1(X)$$
$$f_1(X) \leftarrow touch(X,Y) \wedge \neg building(Y)$$

The last step is to break up disjunctions. These are eliminated by breaking up the expression in multiple clauses, resulting in:

$$building(X) \wedge f_1(X)$$
$$building(X) \wedge \neg type(X,'blockbuilding')$$
$$f_1(X) \leftarrow touch(X,Y) \wedge \neg building(Y)$$

The end result is a DATALOG$^\neg$ expression which can be seen below (where the error result is defined explicitly):

error $\leftarrow$ `building(X)` $\wedge$ `f1(X)`
error $\leftarrow$ `building(X)` $\wedge \neg$ `type(X,'blockbuilding')`
f$_1$(X) $\leftarrow$ `touch(X,Y)` $\wedge \neg$ `building(Y)`

7.4 Attributes in DATALOG$^\neg$

One of the major differences between "real" DATALOG and our DATALOG$^\neg$ is the difference in representing attributes. In our Predicate Logic and DATALOG$^\neg$ attributes are represented in a separate predicate, i.e. if we want to state that an area is of type residential we would have:

`area(X)` $\wedge$ `type(X,residential)`

This is not how one traditionally represents attributes in DATALOG$^\neg$, in fact in real DATALOG$^\neg$ every predicate corresponds exactly to a table in the database implementation, so that the class predicate's arity depends on the table. Furthermore the attribute is added directly inside the class predicate. Let us imagine that class "area" in fact is represented by a 3 column table in the database. The columns are labelled "areaID","Zipcode" and "type", and furthermore "areaID" is the primary key of the table. Then the resi-

dential area would be represented as:

```
area(X,_,'residential')
```

As seen there is no separate attribute predicate, and the notation is more compact, but also a lot more database specific.
Take the area table from before, let us imagine for that the two columns "areaID" and "zipcode" constitute the primary key. Then we need to compare both of these fields, using extra variables so that:

```
area(X,Y,_) and contain(X,Y,....)
```

Operators are also treated differently when we chose to represent attributes in the class predicate. Take a look at the HLCL fragment below:

```
area type residential and zipcode 90210
```

Because there is a conjunction between the two attributes, we can declare them in one single predicate:

```
area(X,'90210','residential')
```

But if the HLCL fragment was formulated like:

```
area type residential or zipcode 90210
```

Then we would have to split the DATALOG¬ declaration up into two class definitions as seen below:

```
area(X,'90210',_) or area(X,_,'residential')
```

These issues with attributes in DATALOG¬ are not "show-stoppers". We have the table structure represented in our database model, so all information which we need to make this form are available. Also the algorithms for making these compact are not complex. But looking back at our original purpose for having a DATALOG¬ interface, namely as an easy-to-read interface which could be used as basis for translation to other query languages. It is not optimal to do these last transformations, because the interface becomes much more relation-database specific, and not as easy to read.

Chapter 8

Database Representation

The HLCL system needs information about the target database which the HLCL constraints should apply to. Basically it needs 3 different kinds of specifications which are listed below:

Conceptual Model, E/R-Diagram This is an abstraction over the database, and is the basis from which the user formulates the HLCL constraints. In the conceptual model we operate only with Entities (Classes) and Relations (including ISA-relationships).

Database Model This is the specific implementation details of the database. We are using a relational database which operates with tables, attributes and primary keys. Furthermore there are some pre-defined SQL functions.

A mapping between the two models This is a specification of how the two models correspond. This map enables the user to formulate HLCL constraints in the context of the conceptual model, which will then be translated into working query in the context of a real database specified in the database model.

In order for HLCL to use these models and the mapping information they need to be represented in PROLOG. A suggestion for representing conceptual models in PROLOG can be found in [KZ98]. Their representation has a lot of information, such as arity on relations and attributes, which are needed to make rewritings into normal forms, but this is not used in the HLCL-system. Instead a minimal representation is chosen for the HLCL system, where only the essential information in the database representation is represented.

8.1 Formal Description

This chapter is meant to give the reader a quick overview of what information is contained in the database model. A brief RSL specification can be

seen below.[1]

A specification for the Database Representation

---

scheme DATABASEREPRESENTATION =
class
  type
    Name,
    Property,
    ValueInterval,
    Class = Name,
    Relation = Name × Class × Class,
    AttRelation = Class × Property × ValueInterval,
    Isa = Class $\overrightarrow{m}$ Class-**set**,
    ConceptualModel = Class-**set** × Relation-**set** × AttRelation-**set** × Isa-**set**,

    Column,
    SQLExp,
    Condition = SQLExp,
    Table = Name × Column*,
    Function = Name × Column* $\overrightarrow{m}$ SQLExp,
    DatabaseModel = Table-**set** × Function-**set**,

    Key = Column,
    IntervalDefinition,
    Classmap = Class $\overrightarrow{m}$ Table × Condition × Key*,
    Relationmap = Relation × Class × Class $\overrightarrow{m}$ Table × Key* × Key*,
    Valuemap = Class × Property $\overrightarrow{m}$ Table × Column,
    Typemap = ValueInterval $\overrightarrow{m}$ IntervalDefinition,
    Mapping = Classmap-**set** × Relationmap-**set** × Valuemap-**set** × Typemap-**set**
end

The specification consists solely of type declarations, where the three most important ones are: "ConceptualModel", "DatabaseModel" and "Mapping". These three subspecifications contain exactly the same information and has the same shortcomings and limitations as the actual database representation contained in the HLCL-system. The specification will not be explained further, instead the actual predicates used for representing the database will be

[1] The specification language RSL is described in [Gro93].

explained in the next chapter, and the specification will provide an overview of all information stored in the database representation.

8.2 Informal Description

In general the database representation is represented as a collection of predicates, in which each predicate represents one class, relationship, function, table or mapping. This representation takes up quite a lot of space, but it makes the conceptual model representation easier to understand and modify. The usage of these predicates will be defined in the following subchapters, using the conceptual model in figure 8.1 and the database implementation in figure 8.2 on the following page as reference.

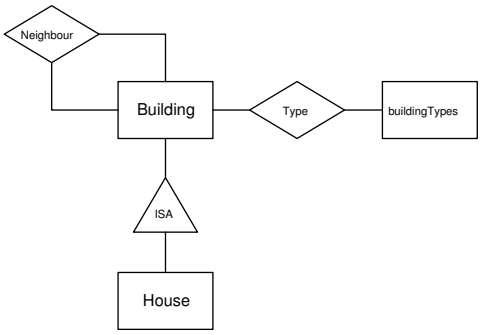

Figure 8.1: Conceptual Model Example

8.2.1 Conceptual Model

Each class in the conceptual model is represented by a unary "class" predicate, which holds the class name as the argument. The general form of a "class" predicate can be seen below:

```
class(Classname)
```

The conceptual model in figure 8.1 would have the following classes defined:

```
class(building).
class(house).
```

Notice how the class "residential" is not represented, since this is not a real class, but a property of the class "building".

AllBuildingTable

BuildingID	Type	Owner
2055	Residential	John
2056	Commercial	Jack
...	...	...

AllNeighboursTable

BuildingID1	BuildingID2	Zipcode
1055	2055	2200
1056	2056	2100
...	...	...

Figure 8.2: Database Tables corresponding to figure 8.1

Relations use the 3-ary "relation" predicate, and can be of two different types. Either relations can be a relation which relates two classes, or it can be a property assignment to a class, with a given value. In case it is a "real" relation, the first argument is the first class the relation connects, the second argument is the name of the relation, and the third argument is the second class which the relation connects. The structure can be seen below:

```
relation(Class,Relation,Class)
```

This is the case for the "neighbour" relation in figure 8.1 on the preceding page, where it would be represented as below:

```
relation(building,neighbour,building).
```

The second option is that a relation is a property of the class, which has the representation seen below:

```
relation(Class,Property,AllowedValues)
```

This is the case for the "type" relation in figure 8.1 on the previous page, where it would be represented as below:

```
relation(building,type,buildingTypes).
```

The "buildingTypes" is a list of values which is represented in the mapping part of the database model.
Finally we have a special kind of relationships, namely the ISA-structures that denote inheritance. ISA-structures are represented by the binary "isa"-predicate: The first argument is the parent class, and the second argument is a list of the immediate sub-classes.

```
isa(Class,Classlist).
```

The ISA-structures in figure 8.1 on page 81 would be represented as:

```
isa(building,[house]).
```

It is important to notice that the ISA-structures function as a partial order. When we specify "house", we do not need to specify the relations "neighbour" and "type", since we have specified that "house" is a subclass of "building", and "house" has these relations defined. For more information about how ISA-structures can be represented, the reader is referred to appendix A.4.1 on page 124

8.2.2 Database Model

The database model is a description of the actual low-level database. The main predicate used in the database model is the binary "table" predicate, which is used to describe the tables of the database. The first argument is the name of the table, the second argument is a list of the columns contained in that table.

```
table(Name,ListOfColumns)
```

The tables in figure 8.2 on the facing page would be represented as:

```
table(allBuildingTable,[buildingID,type,owner])
table(allNeighboursTable,
   [buildingID1,buildingID2,zipcode])
```

Furthermore we have a number of functions which can be applied to the database. The functions are defined with the 3-argument "function" predicate: The first argument is the name of the function, the second argument is a variable map (A list of variables mapping into attributes needed), and the third argument is the actual function as it would look in SQL. The structure of the "function" predicate can be seen below

```
function(Name,VarMap,SQLExpression).
```

As an example, a binary function Zdifference calculating the difference between two buildings would be defined as:

```
sqlfunction(zdifferencelargerthan5,[[A,buildingID],
[B,buildingID]],function(zdiff,[(A,buildingID),(B,buildingID)])).
```

As seen above the zdifference function needs two 'BuildingID' attributes, and it translates straighforward into a "zdiff(a,b)" function which should be made available in SQL.

8.2.3 Mapping

The mapping between the Conceptual Model and the Database Model should be as flexible as possible, so that low-level database specific changes do not affect the HLCL expressions. The basic idea is that every class and relation correspond to a table.
To map a class in the conceptual model to a table we use the 4-ary "classmap" predicate: The first argument is the class name, the second is the name of the table, the third argument is a condition to the rows which represent the class and the fourth and last argument is a list of attributes which constitutes the primary key for that class.

```
classmap(Classname,Tablename,Condition,PrimaryKeyList)
```

In our the conceptual model in figure 8.1 on page 81 the mapping would look like the following:

```
classmap(building,allBuildingTable,[],[buildingID])
```

In the above mapping the condition argument is an empty list, which means that all rows in the "AllBuildingsTables" will be selected. This might not always be the case, i.e. let us imagine that for the subclass "houses", we only wanted the rows in the "AllBuildingsTables" where the "buildingStyle" attribute was equal to 'house', then the condition argument should be a tuple with the attributes name and the value, as seen below:

```
classmap(house,allBuildingTable,
   [type,house],[buildingID])
```

The condition criteria can only be either an empty list or a binary tuple, which limits the condition to be of this primitive kind of selection criteria. More advanced selection criteria could be added fairly easily, but we have chosen not to pursue this further for the presented system.

The set of attributes which forms the primary key is needed for comparing classes. Other keys are not important, i.e. other attributes may also form a key, but for our usage they do not really matter, since the only key we are interested in is the one which is used by the classes and relations.
The above mapping allows the class and table to be differently named, and it furthermore opens up for the possibility that one table can hold more than one class.

Relations also map into tables, this is specified with the 6-ary "relmap" predicate. The first 3 arguments are the relation name and its connecting classes, the fourth argument is the name of the corresponding databasetable. The last two arguments is the lists of attributes which form the primary key for each connected class. The structure can be seen below

```
relmap(Class1,Relation,Class2,Table,
  Class1keys,Class2keys)
```

The lists in the last two arguments are the names of the keys in the relation table. This is because it is not certain that the attributes that form the key in the class has the same name in the relation table. The expression below is a good example of how the keys have different names in the relation and in the class.

```
relmap(building,neighbour,building,allNeighbourTable,
  [buildingID1],[buildingID2])
```

The "type" relation is not mapped, since this does not correspond to a table in the database. Instead it corresponds to a specific attribute in the "building" class, and this is defined by the binary "valuemap" predicate, which maps the property to an attribute. The first argument is the Class, the second argument is the relation name in the conceptual model, the third argument is the value interval, and the final and fourth argument is the actual column name in the database. The structure can be seen below:

```
valuemap(Class,Attribute,ValueInterval,ActualAttribute)
```

In our example we need to define the "type" relation, which should map to an attribute. The corresponding definition can be seen below

```
valuemap(building,type,buildingTypes,type)
```

Furthermore we need to map the allowed value interval, which building types can be within. This is done with the binary "type" predicate. The first argument is the name of the value interval given in the "relation" and

"valuemap" definition in the conceptual model, and the second argument is the list of values which are allowed. The structure can be seen below:

```
type(ValueInterval,ValueIntervalDefinition)
```

In the HLCL-system we support 3 types of valueinterval definitions: 1) A list of allowed values, 2) A string or 3) An integer. Further datatypes could built in at a later stage. In our case buildingtype uses the list definition:

```
type(buildingTypes,[residential,commercial,house])
```

One thing to notice is that there are actually a couple of more attributes in the database model, such as "zipcode" where we do not define a map. The reason is that this attribute is not defined in the conceptual model, and thereby not available in the HLCL-syntax, and it will therefore be ignored in the HLCL-system. This is a general concept of the HLCL-system: Only class/relations/properties defined in the conceptual model are available to the user, but the resulting integrity constraints will work on a database much larger, with a lot more class/relations/properties defined if needed. This adds to the flexibility of the database representation.

Wildcards

The final feature of the mapping is addition of wildcards. If one relation is valid for all classes, then instead of putting a "relation" predicate for all class combinations, a underscore operator ("_") working as wildcard can be used instead of the classes, ie. as seen below:

```
relmap(_,neighbour,_,allNeighboursTable,[ID1],[ID2])
```

This requires a bit of caution though, as the user should be careful not to define any relations twice.

Example 8.2.1

A summary of the database representation of the models in figure 8.1 on page 81 and 8.2 on page 82 can be seen below:

```
% Conceptual Model
class(building).
class(house).
relation(building,neighbour,building).
relation(building,type,buildingtypes).
isa(building,[house]).
```

```
% Database Model
table(allBuildingTable,[buildingID,type,Owner])
table(allNeighboursTable,[buildingID1,buildingID2,zipcode])

% Mapping
classmap(building,allBuildingTable,[],[buildingID])
classmap(house,allBuildingTable,[type,house],[buildingID])
valuemap(building,type,buildingtypes,type)
relmap(building,neighbour,building,allNeighboursTable,
   [BuildingID1],[BuildingID2])
type(buildingtypes,[residential,commercial,house])
```

8.3 Wellformedness

There are a number of wellformedness requirements that should hold for the database representation. The HLCL-system includes a function which checks that the database specification is wellformed. (The implementation details can be found in appendix B on page 131). The wellformedness criteria are listed below:

- All classes defined in the conceptual model should have a mapping into an existing table.
- All relations defined in the conceptual model should have a mapping into an existing table.
- The keys specified in the mappings should exist in the table which it is mapped to.

Furthermore there is a couple of wellformedness requirements which should also hold, but which cannot be checked automatically. These are listed below:

- All relations and their inverse relation should map into the same table, with the corresponding key-attributes switched.

8.4 Shortcomings of Representation

There are a number of shortcomings in our conceptual mode representation. These are listed below. All of these could be solved, but it would make the representation more "blurry", and it would make the overall idea harder to see in the implementation.

- Classes only have one key, meaning that all relations must use the same attributes/key for the same class. It is not possible to define

the fact that some relations use one key for a class, while others use another key for the class.

- The conditions criteria in the class mapping is very limited, one can only chose that an attribute should be a certain value. This could be expanded to involve combinations of attributes, various arithmic operations, such that a certain attribute should be bigger or smaller than a given value or yet another attribute.
- Attribute value intervals are limited, to be either a list, an integer or a string. This could be expanded to many different types.

Chapter 9

From Extended DATALOG to SQL

This chapter will explain the translation of Extended DATALOG to SQL, which consists of two parts: The first section will discuss issues and general problems with DATALOG to SQL translation, the second part will cover the general idea behind the translation and will explain by examples how it is done.

9.1 Issues

In general when translating DATALOG into SQL one needs to be aware of two potential problems: Differences in expressiveness and safety of DATALOG and SQL: DATALOG is a more expressive language than SQL, i.e. that it is possible to formulate constraints in DATALOG which cannot be formulated in SQL. Furthermore it should be possible to evaluate every query in SQL without ending up with an infinite relation. This subset of queries is called "safe".

One approach to ensure safety and correct expressiveness is to convert the DATALOG into an intermediate form called "Range Form", which is then converted to SQL. This approach is described in [Dec01] and [Dec02]. "Range Form" is a logical form which only has $\wedge$, $\vee$, $\neg$ and $\exists$ operators. Furthermore "Range Form" is proven safe, which means that every DATALOG expression which can be converted to "Range Form" can be evaluated in SQL.
An algorithm for translating DATALOG into range form can be found in [GT91]. This algorithm is quite complex, since it is made to modify every type of DATALOG expression.

But our translating strategy $\mathcal{T}$ generates the same pattern of DATALOG-expressions, hence it seems unnecessary to use the intermediate "range form". Instead we examine the pattern $\mathcal{T}$ generates, and make some cus-

tomized transformations to the expression, so that it can be translated directly into SQL. In the two following subchapters the issues will be described and it will be showed informally that our Extended DATALOG interface is both safe and non-recursive.

9.1.1 Expressiveness

DATALOG is a more expressive language than SQL, i.e. that it is possible to formulate constraints in DATALOG which cannot be formulated in SQL. For example the recursive DATALOG expression:

```
Ancestor(X,Y) :- Fatherof(X,Y).
Ancestor(X,Y) :- Ancestor(X,Z),Fatherof(Z,Y).
```

The above example is recursive, because the second clause contains a reference to its own definition. These clauses cannot be transformed into SQL statements[1].

It is easy to realize that The Extended DATALOG which we use cannot be recursive, since it only consists of one single clause returning error. There is no subpart of the clause which refers to the clause itself in any way. We can therefore conclude that our Extended DATALOG interface not more expressive than SQL, and translation between the two should be possible.

9.1.2 Safety

There is an easy way of checking if a DATALOG expression is safe, which is explained in [GMUW02]:

"Every variable that appears anywhere in the rule must appear in some non-negated, relational subgoal" [GMUW02] pp. 467.

Extended DATALOG is safe because every variable is always declared both a class and a relation, and at least one of them is not negated. Let us look at the basic example:

C2 `all area must contain building`

In Extended DATALOG this would be represented as seen below:

$$\forall X area(X) \wedge \neg \exists Y (contain(X, Y) \wedge building(Y))$$

As we can see both X and Y appear in a non-negated subgoal (actually

[1] It is actually possible to evaluate recursive queries in SQL version 3, but this has not been implemented by any commercial DBMS systems yet.

there are no negated subgoals at all in the expression). For most HLCL-expressions the first class, and the classes related to it will always be non-negated, thereby making a safe Extended DATALOG-expression. The only sentence types which have a negated related class are constructs using the ‘‘solely’’ keyword, an example can be seen below:

```
all area must contain solely building
```

The above HLCL-expression has the Extended DATALOG-expression seen below:

$$\forall X area(X) \wedge \neg\exists Y(contain(X,Y) \wedge \neg(lake(Y)))$$

In the above expression the Y variable is negated in the last class “lake”, but the relation “contain” has both variables X and Y and is non-negated. Therefore the Extended DATALOG-expression is still safe.
In general, looking at the Extended DATALOG expressions it can be realized that in order to have an unsafe expression, we would need to have an Extended DATALOG-expression on one of the two forms seen below:

$$\forall X area(X) \wedge \neg\exists Y(\neg(contain(X,Y)) \wedge \neg(lake(Y)))$$
$$\forall X \neg(area(X)) \wedge \neg\exists Y(\neg(contain(X,Y)) \wedge lake(Y))$$

In the top case, the variable Y are only present in negated subgoals, hence the expression is unsafe. But the Extended DATALOG-expression corresponds to the constraint expressing: *“There should exist an entity which is not contained in an area and which is not a building.”*. A construct like this can simply not be made in HLCL. Similarly with the second expression: The Extended DATALOG-expression corresponds to the constraint: *“Everything but areas may not contain lakes”*, this constraint is simply not possible either to formulate in HLCL. It is not possible by starting the HLCL expression with a negated class. This can lead us to conclude that Extended DATALOG is a safe subset of DATALOG, since it is not possible to formulate constraints HLCL which translates into unsafe DATALOG.

9.2 Translation Strategy

The general idea is that every class and relation correspond to a table, which is defined in the conceptual model. The SQL expressions are built up as nested SELECT-FROM-WHERE clauses[2]. The Extended DATALOG is already prepared for the SQL translation, and the translation from Ex-

[2]For a more comprehensive introduction to SQL the reader is referred to appendix A.3 on page 121

tended DATALOG to SQL can be done predicate by predicate. Therefore the overall strategy is to break the entire Extended DATALOG expression up, and translate each predicate by it self. In the following chapters we will look at how the individual Extended DATALOG fragments are being translated into SQL.

Classes and Relations

The very first class expression is translated into a simple table lookup, i.e. the fragment "class(X)...", would be translated into the following SQL:

```
SELECT *
FROM classtable a
WHERE ...
```

where "classtable" is the table corresponding to the class. The translator will introduce labels, which are lowercase single character letters, which will be used to refer to the class at a later time. In this top example the first class is allocated the label 'a'. Classexpressions within an expression look a bit different since they are inside a WHERE-clause already and consequently need an extra EXISTS statement. I.e. the fragment "...class(X)..." would be translated into:

```
EXISTS (SELECT *
FROM classtable b
WHERE ...)
```

Similarly a relation, i.e. "relation(X,Y)" would be translated into:

```
EXISTS (SELECT *
FROM relationtable b
WHERE ...)
```

Where "relationtable" is the table corresponding to the relation. In order to relate the classes and relations, a condition which set their primary keys equal is added. Consider the fragment below:

$$\forall X class_1(X) \wedge class_2(X)$$

This would be translated into the following SQL:

```
SELECT *
FROM class1table a
WHERE EXISTS(
  SELECT *
  FROM class2table b
  WHERE b.key = a.key)
```

It should be noted that the two class expressions are nested, the second class $class_2$ is translated using the EXISTS keyword in order to nest it in the WHERE clause. From the Extended DATALOG-expression we can see that the classes both use the variable X, hence they should be set equal. This is done by setting their primary keys equally after the second class-expression. In the above case only one attribute forms the key. If the key consisted of three attributes they would all be set equal and conjugated as seen below:

```
...
WHERE b.key1 = a.key1
AND b.key2 = a.key2
AND b.key3 = b.key3
```

Example 9.2.1
Let us look at the translation of a constraint with only classes and relations as seen below:

C2 `all area must contain building`

The corresponding SQL can be seen below:

```
SELECT *
FROM areatable a
WHERE NOT (EXISTS(
  SELECT *
  FROM contain b
  WHERE (b.id1 = a.areaID) AND EXISTS(
    SELECT *
    FROM buildingtable c
    WHERE (c.buildingID = b.id2))))
```

As seen, classes and relations are no longer referred to as their names in the conceptual model, instead the actual table names in the database model are used. The SQL query follows the structure in the Extended DATALOG directly: The tables in SQL are used in the same order as they appear in the Extended DATALOG. Comparison is done with the set of attributes which constitutes the primary key.

Attributes

Attribute fragments in SQL are much simpler to translate than relations. No EXISTS-expression is needed in the SQL, instead they are translated into a simple equation. As an example look at the fragment below

$$\forall X Class(X) \wedge attribute(X, eq, 'val')...$$

The above Extended DATALOG fragment would be translated into:

```
SELECT *
FROM classtable a
WHERE a.attribute = 'val'
...
```

Numerical quantified attributes use the '<=' and '>=' operators directly available in SQL so that an Extended DATALOG fragment:

$$\forall X Class(X) \wedge attribute(X, ge, 50)...$$

Would turn into an SQL-query which would look like the one seen below:

```
SELECT *
FROM classtable a
WHERE a.attribute >= 50
...
```

Example 9.2.2

Let us look at the translation of an constraint example which uses attributes. Consider the HLCL-expression below:

C4 `all building type industrial must beusedby company`

In the actual database implementation, "type" is not a relation between the class "building" and "industrial", but rather "type" is an attribute of building with value "industrial" - In Extended DATALOG this would be expressed as:

$$\forall X building(X) \wedge type(X, eq, industrial) \wedge \\ \neg\exists Y(beusedby(X, Y) \wedge company(Y))$$

The resulting SQL would look like the following:

```
SELECT *
FROM buildingtable a
WHERE a.type = 'industrial'
AND NOT EXISTS(
   SELECT *
   FROM beusedbytable b
   WHERE b.buildingID = a.buildingID
   AND EXISTS(
      SELECT *
      FROM company c
      WHERE c.companyID = b.companyID)))
```

Operators and Negations

In general operators and negations are translated directly into the equivalent operators in SQL, which can be seen below:

$\neg(Exp) \rightarrow$ NOT(Exp)
$Exp_1 \wedge Exp_2 \rightarrow Exp_1$ AND Exp_2
$Exp_1 \vee Exp_2 \rightarrow Exp_1$ OR Exp_2

One thing to notice is that not all conjunctions in the Extended DATALOG-expression are translated similar, since the conjunction is also used to "glue" path expressions together. Therefore will some conjunctions be translated into a simple "AND", while others will be translated as a condition in the previous SELECT structure. The situation is best explained by using a example. Imagine the HLCL fragment below:

```
...contain house and contain river
```

The above would in Predicate Logic look like

$$...\exists X(contain(X,Y) \wedge house(Y)) \underline{\wedge} \exists X(contain(X,Y) \wedge river(Y))$$

The underlined $\wedge$ should will translated into "AND" in SQL. But the two other conjunctions which are used to "glue" the relation and the class together, will be translated such that tablelookup corresponding to "house" will be made inside the "contain" table selection and likewise with "contain" and "river". The HLCL-system can tell the two different kinds of conjunction apart by the usage of the existential quantifiers. The rule is that only conjunctions between two (possible negated) existential quantifiers should be into "AND" in SQL without being put nested in the previous table lookup. In order to illustrate this principle take a look at the example below.

Example 9.2.3

To illustrate the above point, let us look at the translation of the following HLCL-expression:

```
all area must contain building and contain lake
```

Which in Extended DATALOG would have the following expression:

$$\forall X area(X) \wedge \\ \neg(\exists Y(contain(X,Y) \wedge building(X)) \wedge (\exists Y(contain(X,Y) \wedge lake(X)))$$

This would have the following SQL translation:

```
SELECT *
FROM areatable a
WHERE NOT (EXISTS(
   SELECT *
   FROM containtable b
   WHERE b.areaID = a.areaID
   AND EXISTS(
      SELECT *
      FROM buildingtable c
      WHERE c.buildingID = b.buildingID))
AND
   EXISTS(
   SELECT *
   FROM containtable b
   WHERE b.areaID = a.areaID
   AND EXISTS(
      SELECT *
      FROM laketable c
      WHERE c.lakeID = b.lakeID)))
```

In the SQL above we can see how the "glue"-conjunctions are nested in the previous 'WHERE' clause, but the conjunction between "building" and "contain" in the HLCL-expression corresponds to simply an "AND" which is not nested inside previous 'WHERE' clause.

Quantifiers

Existential and Universal quantifiers do not result in any SQL fragments in the translation. But they are indirectly used to ensure correct translation, by defining quantifier scope. As mentioned in the description of operators, they are used to indicate whether a conjugation is a "real" conjugation or a "glue"-conjugation.
Numerical Quantifiers on the other hand do affect SQL translation, where they modify the SELECT statement to invoke the COUNT construct in the following table lookup. Such that a numerical operator are translated as seen below:

$\exists_{=>N}$ Exp $\rightarrow$ ((EXISTS Exp) => N)
$\exists_{=<N}$ Exp $\rightarrow$ ((EXISTS Exp) =< N)
$\exists_{=N}$ Exp $\rightarrow$ ((EXISTS Exp) = N)

Furthermore it raises a flag, which tells that the next Extended DATALOG fragment should not use a "SELECT *" but instead a "SELECT count(*)" in the scoping. To illustrate the point look at the HLCL fragment seen below:

```
...contain at least 5 house
```

This would have an Extended DATALOG-expression which would look like:

$$...\exists_{=>5} Y(contain(X, Y) \wedge house(Y))$$

The above expression would be translated into the following SQL

```
...
((EXISTS(
   SELECT count(*)
   FROM contain b
   WHERE (b.id1 = a.areaID) AND EXISTS(
     SELECT *
     FROM housetable c
     WHERE c.buildingID = b.id2))) => 5)
```

As seen the only difference is the outer comparison, and that the first fragment, in the case the lookup in the "contain" table, uses an extra "count(*)" expression.

Variables

Translating user-defined variables is simply a matter of adding an extra condition in the table lookup. In general the translation strategy is that on the first encounter of a user defined variable, nothing extra SQL is added. Instead the variable and class is stored, such that it can be looked up at a later stage. At the next encounter of the user-defined variable, an extra condition is added in the 'WHERE' clause in the lookup. Take a look at the Extended DATALOG fragment seen below:

$$...class(X) \wedge relation(X, A)$$

Let us imagine that A is a user-defined variable, and that this is not the first encounter. The translator would at this stage already know that A was actually linked to the class, labelled a, and would therefore add an extra condition in the SQL translation as seen below:

```
...EXISTS( SELECT *
FROM classtable b
WHERE EXISTS( SELECT *
   FROM relationtable c
   WHERE c.key1 = b.key
   AND
     b.key2 = a.key))
```

As seen on the very last line, the ‘‘`b.key2 = a.key`’’ fragment, ensures that the A's are set equal.
Translations of variables used in attributes are done in a similar way, but instead of comparing the keys, the actual attribute is set to be the value of each other. So if we have an expression as seen below:

$$...Class1(Y) \wedge Attribute(Y, A)...Class2(X) \wedge Attribute(X, A)$$

And the table corresponding to the class1 was labelled a, and the table corresponding to class2 was labelled b, then we would simply need to add the following in SQL:

```
a.attribute = b.attribute
```

Example 9.2.4
To illustrate the above point, let us look at the translation of an HLCL-expression using variables:

C11
```
all area intersectedby road R must
contain building intersectedby road R
```

Which in Extended DATALOG would have the following expression:

$$\forall R, X road(R) \wedge area(X) \wedge intersectedby(X, R) \wedge \\ \neg\exists Y(contain(X, Y) \wedge building(X) \wedge intersectedby(X, R)$$

This would have the following SQL translation:

```
SELECT *
FROM roadtable a
WHERE EXISTS
   SELECT *
   FROM areatable b
   WHERE EXISTS(
      SELECT *
      FROM intersectedareatable c
      WHERE c.roadID = a.roadID
      AND c.areaID = b.areaID)
AND NOT EXISTS(
   SELECT *
   FROM containtable b
   WHERE b.areaID = a.areaID
   AND EXISTS
      SELECT *
      FROM buildingtable c
```

```
WHERE c.buildingID = b.buildingID
AND EXISTS(
   SELECT *
   FROM intersectedbuildingtable d
   WHERE d.buildingID = c.buildingID
   AND d.roadID = a.roadID))))
```

Notice how the very last line: ''d.roadID = a.roadID'' ensures that the two intersecting roads are the same.

Functions

Finally we have the user-defined predicates, which translates directly into SQL functions of the same name. The point is illustrated in the example seen below:

```
all area A must function(A)
```

This would have the following Extended DATALOG expression:

$$\forall A area(A) \wedge \neg \exists function(A)$$

Which would correspond to the SQL expression seen below:

```
SELECT *
FROM areatable a
WHERE NOT function(a.areaattribute)
```

The function is simply translated into the corresponding SQL function, which takes a specified attribute (areaattribute) from the areatable as argument. The atttribute specification is done in the database representation.

9.2.1 ISA-structures

Dealing with ISA-structures in SQL is straightforward. In most of the situations the fact that a class is a member of an ISA-structure does not change the query. But there are special cases, where an extra lookup is needed, this is when the attribute is not available in the referred class [3]. Let us revisit the example from the attribute section:

$$\forall XClass(X) \wedge attribute(X, eq, 'val')...$$

Let us suppose that "class" is actually a child of another class: "superclass", and this is actually where the attribute is located. Then the above Extended DATALOG fragment would be translated into:

[3] See also appendix A.4.1 on page 124

```
SELECT *
FROM classtable a
WHERE EXISTS(
   SELECT *
   FROM superclasstable b
   WHERE b.key = a.key AND b.attribute = 'val')
...
```

As seen translating the attribute results in an extra SELECT-statement, relating the class to the superclass and then using the superclass to set the attribute. Since classes in an ISA-structures has the same keys, relating the "class" and "superclass" can be done by setting the attributes forming their key to equal.
The general rule is: If the attribute is not present in the class, then we should browse through its parents to find that attribute. When the parent which has the attribute is found, the class and the parent should be set equal and the attribute should be set on the parent. The procedure is the same for immediate parents or a further related classes.

This approach is similar when dealing with user-defined predicates. One could imagine a situation as below:

$$\forall X Class(X) \wedge function(X)$$

Parallel to the above situation we could imagine that the function needed an attribute from "class", which was actually located in the "superclass" of "class". Then we would have to translate the above Extended DATALOG fragment to the SQL as seen below:

```
SELECT *
FROM classtable a
WHERE EXISTS(
   SELECT *
   FROM superclasstable b
   WHERE b.key = a.key AND NOT function(b.areaattribute)))
```

The approach is similar to the one for attributes, and the same line of argumentation can be extended to binary predicates.

ISA-structures and relations do not pose any further problems. This is due to the fact that our mapping between the conceptual model and database schema has been simplified, such that all relations always use the class' primary key to relate relations. Since all entities in a ISA-structures always has the primary keys, we do not need to add extra SELECT clause to search for the attributes, they are always present in all classes.

But if the mapping was to be extended, to support that different relations used different keys in the classes, ISA-structures could still be implemented using the strategy as sketched above.

Example 9.2.5
The final example in this chapter shows functions and an ISA-structure being translated:

C11 `all residentialArea A must havezipcode(A)`

Which in Extended DATALOG would have the following expression:

$$\forall A residentialArea(A) \wedge \neg havezipcode(A)$$

Let us imagine that the havezipcode(A) used an attribute "zipcode" which is not located in "residentialArea", but in the superclass "Area". This would have the following SQL translation:

```
SELECT *
FROM residentialArea a
WHERE EXISTS(
   SELECT *
   FROM area b
   WHERE b.areaID = a.areaID AND NOT function(b.zipcode)))
```

This concludes the description of the HLCL-system. In the next chapter various major implementation design descissions are discussed.

Chapter 10

Implementation

In this chapter major design decisions and overall ideas regarding the actual implementation of the HLCL system will be explained. For a more detailed and code specific explanation of the HLCL-system the reader is referred to appendix B on page 131.

10.1 PROLOG and Logic Programming

The HLCL-system is implemented in PROLOG, a declarative programming language which consists of sequences of horn-clauses[1]. PROLOG is well suited for recognizing and parsing languages, due to its resolution engine and built-in functionality such as Definte Clause Grammars (explained below). The declarative code is inherently high-level, because it focus on the logic of the computations rather than the "mechanics" of it. This results in code which is shorter and simpler to overview, since all code is directly connected to the algorithms and strategies presented.
The drawbacks is a poorer performance than traditional imperative languages such as C, and limited opportunity to program user-friendly functionality such as a GUI etc.[2].
There exists several PROLOG implementations, but we have chosen to use SWI-Prolog version 5.4.2. A free version can be found at http://www.swi-prolog.org. A more comprehensive introduction to PROLOG can be found in [Bra01] and [BBS01].

[1] Definition of Horn Clauses can be found in [Nil99]

[2] But this can be done with one of the numerous interfaces between PROLOG and Java/C/C++ (i.e. The JPL Package: Found at www.swi-prolog.org/packages/jpl)

10.2 Definite Clause Grammars

In the translation from HLCL to Predicate Logic, a "Definite Clause Grammar" (DCG) notation is used, which was first described in [PW80] [3]. DCG notation is a shorthand notation for describing grammars. Every DCG has an equivalent PROLOG predicate, and there is no magic involved - it is just "syntactic sugar", which makes the grammar easier to understand and modify. The general syntax for a DCG is:

```
s --> v1,v2...vn.
```

Which is equivalent of the PROLOG predicate:

```
s(P0,P) :- v1(P0,P1),v2(P1,P2),...,vn(Pn-1,P)
```

A grammar consists of a number of production rules as the one seen above. Furthermore it is possible to add extra PROLOG code which follows the expression in curly braces. One example of this can be seen in the code snippet below:

```
class(X,class(Class,X)) --> [Class], {is_class(Class)}.
```

The PROLOG code in the curly brackets add an extra condition - namely that Class should succeed in the "is_class" predicate.

Since DCG is just a shorthand notation, it "inherits" the same problems PROLOG has. When designing DCG grammars, one should watch out for left-recursive grammars. Since PROLOG is a topdown parser, left recursion will throw the application in a never ending-loop and crash the system.

10.3 λ-Calculus in PROLOG

To make the translation between HLCL and Predicate Logic one could choose to make a "full" parser, which in multiple passes generated a parse tree, which was then visited during the translation. Since the HLCL syntax is fairly limited we have chosen to simplify the procedure by not building a parse tree, but instead building the expression "on the fly". Translating from HLCL to DATALOG$^\neg$ without a parse tree is not straightforward, since the HLCL parts do not appear in the same order as the DATALOG$^\neg$ parts. Therefore we have the problem that we need to store partly-translated expressions during the translation.
This is the same reason why the translating strategy $\mathcal{T}$ uses λ-calculus. To solve this in PROLOG, one approach is to make a special notation for the λ-operator, this is suggested in [PS87], where they introduce a "^" operator,

[3] A more comprehensive introduction to DCG's can be found in [Bra01],[PS87] and [BWK+02]

and then program their own β-reduction engine in PROLOG.
We use another strategy which is explained in [PW80]. This strategy makes use of the "logical variable", such that unspecified parts of the translation are saved as variable, which are passed around in the translation. The "logical variables" are added as extra arguments in the DCG, which collects all of them in both a top-down and a bottom-up manner at the same time. In the sourcecode the variables 'X','P','P1' and 'P2' (where the X variable represents a variable, and the P variables represent a translated fragment) are used as the λ variables.

10.4 Variables

To represent variables in the HLCL sentences and in Extended DATALOG sentences, we had the choice of using either the PROLOG system variables (ungrounded notation, typically having names starting with an underscore, i.e. "_G123") or to introduce a special variable notation (ground notation). We have chosen to introduce a special notation. It has the disadvantage that PROLOG's resolution engine cannot be used - since PROLOG does not recognize them as variables. But it makes it much easier to make modifications made in the intermediate steps, when there are no unbound variables. Variables that are introduced in the translation use a nested notation, such as:

```
v(0)
v(v(0))
...
```

These variables are easy to generate and recognize in PROLOG. For user-defined variables a notation seen below is used, since capital letters can not be used:

```
var_a
var_b
...
```

This concludes the major design choises in the HLCL-system. For a more code near explanation of the HLCL-system the reader is referred to appendix B on page 131.

Chapter 11

Future Work

This chapter will contain proposals for further research. The proposals have been split into two sections; the first section describes various issues and improvement ideas to the current HLCL-system. The second section describes novel approaches and usages of the HLCL-system.

11.1 Improvements of Current System

11.1.1 Optimizations of Queries

At this stage of the development the HLCL-system is just a simple compiler, it translates constraints into correct SQL, but not necessarily the most optimal SQL. The HLCL-system could be improved to optimize queries on both the HLCL level and at the SQL level. For an example of optimizations on the HLCL level, take a look at the HLCL expression below:

```
All area must contain building and contain building
```

The HLCL system will not recognize that the two conjugated relational paths on the right-hand side are in fact the same, and it will translate the HLCL to make two identical lookups. A mechanism[1] recognizing repetitions could be built-in in future HLCL-systems.
Queries could also be optimized at the SQL level, an example is already given in appendix A.4.2 on page 126. In general the HLCL-system does not exploit the fact that relations or classes could map to the same table in the database schema. In a future HLCL-system functionality which recognizes that two classes or relations actually map down into the same table, could be implemented, thereby saving unnecessary lookups in the database[2].

[1] This mechanism is referred to as: "Common Subexpression Eliminiation" in compiler theory

[2] Most of these optimizations would also be recognized by the query optimizer present in most DBMS, which would make the nescessary optimizations before executing the query

11.1.2 Expressiveness

The HLCL syntax could be expanded to become more expressive: I.e. it could be looked into if arithmetic operations could be supported more elegantly than in the current solution where the user has to use user-defined predicates. In general the only arithmetic expressions supported, are the one supported by the numerical operators, and only between a fixed number and a attribute/relation. Allowing expressions as the above would make HLCL more expressive and thereby usable.

11.1.3 Optimization of Database Representation

The final improvement suggestion is make the database representation more extensive. A number of shortcomings has already been listed in chapter 8.4 on page 87, and the database representation could be expanded to solve these problems.

11.2 Other Applications of Current System

11.2.1 As a Basis for Yet Another Interface

The HLCL interface proposed in this report could be used as a basis for an even higher interface for formulating database constraints. The layer could be inspired by some of the approaches seen in chapter 5 on page 43.
One approach is to add another textual interface, which on the basis of the database model came with suggestions "on the fly", as the user was formulating the constraints in order to help further formulation. Alternatively it could guide the user through steps, where the user selected a fragment of the constraint step by step, the same procedure as seen in "Kaleidoscope" or "SQL FORMS"[3].
Another option is the exploit the fact that HLCL is tightly bonded to the E/R-model. One could imagine a graphical interface, i.e. the conceptual model where the user could compose an HLCL constraint by clicking on the E/R-model. This could be done either by graphical selection only (drag'n'drop), or by a multi-modal input mechanism consisting of both textual and graphical input.

11.2.2 Deducting HLCL Constraints

A novel approach could be to deduct HLCL constraints for an already given database. A datamining system could be used in order to examine a database with a defined database model. By examining the data-sets in the database it could try to deduct which HLCL constraints was valid for all or

[3]See chapter 5 on page 43

most of the data-sets.
One could then imagine that a database designer which was not completely sure about the constraints, would take a representative part of his data-sets and try to deduct which constraints were valid, and then applying these constraints to the entire database.

11.2.3 Logical Relationships Between Constraints

It could also be interesting to examine the logical relationships between constraints formulated in HLCL. By figuring out the logical relationships between HLCL constraints, an HLCL system could compute which constraints that would be a subset of other constraints, or if two or more constraints exclude each other. Furthermore the logical relationships might reveal relationships which could be used as rewrite rules, which a user formulating HLCL constraints could use in order to simplify and clean up a constraint collection.

Chapter 12

Conclusion

The aim of the present thesis has been to create a new language for formulating integrity constraints, which are at a higher level than constraints formulated in Predicate Logic. A new High Level Constraint Language called HLCL has been developed which shortens the gap between constraint specification and constraint formulation. HLCL has an easy-to-use syntax due to its resemblance to natural language which also makes HLCL intuitive to read and understand. The underlying semantic model is based on a well-known algebraic logic called "Peirce Algebra", thereby making the language formal, unambiguous and excellent for constraint specification and formulation.
As a case study a number of constraints from the Geographic Information Systems domain has been examined, and converted to HLCL. It has been shown that HLCL is sufficiently expressive to formulate most constraints present in the GIS domain.

The present thesis also contains a brief discussion of the efforts that have been made in the integrity constraint formulation field over the last twenty years, and particular attention has been given to the four most widely used constraint specification languages. It seems that none of them suits all our needs at the same time: COLAN has a syntax close to HLCL, but it is not as simple and close to natural language as HLCL. Description Logic is well-defined and can even reason with the defined constraints, but is not sufficiently expressive for our needs. EER has a well-defined semantic model, but is too complicated to use to fill the gap between natural language and Predicate Logic. OCL is probably the most widely used constraint language currently, but it seems to have been developed as a constraint *specification* language, not an actual constraint language, and its underlying semantic model is ambiguous and unclear. We therefore conclude that it seems there is a need for a new constraint language like HLCL which is both easy to use and has a formal semantic model.

An HLCL-system has been implemented in PROLOG, which translates HLCL into both a SQL and a DATALOG$^\neg$ interface. The SQL interface can be executed directly in a target database, revealing any data-sets which breaks the constraint. The DATALOG$^\neg$ interface gives an easy-to-read alternative to the SQL and can be used as a basis for translation into other query languages. The HLCL system has been documented and tested with constraints from the GIS domain. A formal translation specification of how HLCL is translated into DATALOG$^\neg$ and SQL has been provided, such that further developments of the HLCL-system and syntax are possible.

In order to translate the HLCL constraints the HLCL-system needs access to a database representation specification. This specification has been partitioned into three separate specifications: A specification of the Conceptual Model, a specification of the Database Schema and a specification of the mapping between the two. This allows the conceptual model and thereby the HLCL constraints to stay the same, even if the underlying database design is modified. Hence database schema design changes are readily accommodated by recompilation of the constraints.

Furthermore directions for future work are indicated in the present report. These spans from extensions of the current system, such as optimization of translated SQL queries, a wider database representation or extending the HLCL syntax with new constructs, etc, to novel usages of HLCL. HLCL might be used as basis for yet another interface, or as a language in a data-mining process.

It could also be interesting to see HLCL applied to other domains than GIS, i.e. one could imagine HLCL applied to the so-called "business rules", which can be viewed as "constraints for business processes". HLCL might help economists and logistic personnel formulate the rules and thereby be used to optimize business procedures.

The work carried out in the present thesis has resulted in two articles which are planned to be published later this year, namely [NJ05] which discusses the formal aspects of the HLCL and its translation into SQL and [Chr05] which discusses the issues related to specificating geographic data.

Bibliography

[AG97] Nabil R. Adam and Aryya Gangopadhyay. A form-based natural language front-end to a cim database. *IEEE Transactions on Knowledge and Data Engineering*, 9(2):238–250, 1997.

[AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1 edition, 1995.

[BBS94] C. Brink, K. Britz, and R. A. Schmidt. Peirce algebras. *Formal Aspects of Computing*, 6(3):339–358, 1994.

[BBS01] Patrick Blackburn, Johan Bos, and Kristina Striegnitz. Learn prolog now!, 2001. Available online at: http://www.coli.uni-sb.de/kris/learn-prolog-now.

[Ben86] Johan Van Benthem. *Essays in Logical Semantics*. Reidel Publ. Co., 1986.

[BG95] N. Bassiliades and P. M. D. Gray. Colan: a functional constraint language and its implementation. *Data Knowl. Eng.*, 14(3):203–249, 1995.

[BKS02] Bernhard Beckert, Uwe Keller, and Peter H. Schmitt. Translating the Object Constraint Language into first-order predicate logic. In *Proceedings, VERIFY, Workshop at Federated Logic Conferences (FLoC), Copenhagen, Denmark*, 2002.

[BMNPS02] Franz Baader, Deborah McGuinness, Danielle Nardi, and Peter F. Patel-Schneider. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, 2002.

[Bor96] Alexander Borgida. On the relative expressiveness of description logics and predicate logics. *Artificial Intelligence*, 82(1-2):353–367, 1996.

[Bra01] Ivan Bratko. *Prolog, Programming for Artificial Intelligence*. Addison Wesley, 2001.

[BS94] Thomas Balstrøm and Ole Jacobi Esben Munk Sørensen, editors. *GIS i Danmark*. Teknisk Forlag, 1 edition, 1994.

[BWK+02] Aljoscha Burchardt, Stephan Walter, Alexander Koller, Michael Kohlhase, Patrick Blackburn, and Johan Bos. *Computational Semantics*. MiLCA, Sarbrucken, 2002. Available online at: http://www.coli.uni-sb.de/cl/projects/milca/courses/comsem/.

[CERE90] Bogdan D. Czejdo, Ramez Elmasri, Marek Rusinkiewicz, and David W. Embley. A graphical data manipulation language for an extended entity-relationship model. *IEEE Computer*, 23(3):26–36, 1990.

[Cha90] Sang K. Cha. Kaleidoscope: a cooperative menu-guided query interface (sql version). In *Proceedings of the sixth conference on Artificial intelligence applications*, pages 304–310. IEEE Press, 1990.

[Chr04] Jesper Vinther Christensen. Specifying constraints for geographic information. -, July 2004.

[Chr05] Jesper Vinther Christensen. *Creating and Maintaining distributed GeoObjects in Loosely Coupled Systems*. PhD thesis, Technical University of Denmark, 2005. Forthcoming.

[CWD00] Miro Casanova, Thomas Wallet, and Maja D'Hondt. Ensuring quality of geographic data with UML and OCL. In Andy Evans, Stuart Kent, and Bran Selic, editors, *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings*, volume 1939 of *LNCS*, pages 225–239. Springer, 2000.

[Dec01] Hendrik Decker. Soundcheck for sql. In *Proceedings of the Third International Symposium on Practical Aspects of Declarative Languages*, pages 214–228. Springer-Verlag, 2001.

[Dec02] Hendrik Decker. *Translating advanced integrity checking technology to SQL*, pages 203–249. Idea Group Publishing, 2002.

[DHL01] Birgit Demuth, Heinrich Hussmann, and Sten Loecher. OCL as a specification language for business rules in database applications. In Martin Gogolla and Cris Kobryn, editors, *UML 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools. 4th International Conference, Toronto, Canada, October 2001, Proceedings*, volume 2185 of *LNCS*, pages 104–117. Springer, 2001.

[EG96] Suzanne M. Embury and Peter M. D. Gray. Compiling a declarative high-level language for semantic integrity constraints. In *Proceedings of the Sixth IFIP TC-2 Working Conference on Data Semantics*, pages 188–226. Chapman & Hall, Ltd., 1996.

[EW81] Ramez Elmasri and Gio Wiederhold. Gordas: A formal high-level query language for the entity-relationship model. In Peter P. Chen, editor, *Entity-Relationship Approach to Information Modeling and Analysis, Proceedings of the Second International Conference on the Entity-Relationship Approach (ER'81), Washington, DC, USA, October 12-14, 1981*, pages 49–72. North-Holland, 1981.

[GEHK99] Peter M. D. Gray, Suzanne M. Embury, Kit Y. Hui, and Graham J. L. Kemp. The evolving role of constraints in the functional data model. *J. Intell. Inf. Syst.*, 12(2-3):113–137, 1999.

[GH91] Martin Gogolla and Uwe Hohenstein. Towards a semantic view of an extended entity-relationship model. *ACM Trans. Database Syst.*, 16(3):369–416, 1991.

[GHP01] Peter M. D. Gray, Kit Y. Hui, and Alun Preece. An expressive constraint language for semantic web applications. *E-business and the intelligent Web: Papers from the IJCAI-01 Workshop*, pages 46–53, 2001.

[GMUW02] Hector Garcia-Molina, Jeffrey D. Ullman, and Jenniger Widom. *Database Systems, The complete book.* Pearson Prentince Hall, international edition, 2002.

[GR98] Martin Gogolla and Mark Richters. On constraints and queries in UML. In Martin Schader and Axel Korthaus, editors, *The Unified Modeling Language – Technical Aspects and Applications*, pages 109–121. Physica-Verlag, Heidelberg, 1998.

[Gro] Object Management Group. Uml 2.0 ocl specification. Available at online at http://www.omg.org/technology/ documents/modeling_spec_catalog.htm#UML.

[Gro93] The RAISE Language Group. *The RAISE specification language.* Prentice-Hall, Inc., 1993.

[GT91] Allen Van Gelder and Rodney W. Topor. Safety and translation of relational calculus. *ACM Trans. Database Syst.*, 16(2):235–278, 1991.

[JQ92] H. V. Jagadish and Xiaolei Qian. Integrity maintenance in object-oriented databases. In *Proceedings of the 18th International Conference on Very Large Data Bases*, pages 469–480. Morgan Kaufmann Publishers Inc., 1992.

[Kin04] Jeffrey C. King. Anaphora. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. -, Spring 2004.

[KZ98] Manuel Kolp and Estaban Zimanyi. Prolog-based algorithms for database design. In *Proceedings of the 6th International Conference on Practical Applications of Prolog (PAP'98)*, 1998.

[MR83] Victor M. Markowitz and Yoav Raz. Errol: An entity-relationship, role oriented, query language. In Carl G. Davis, Sushil Jajodia, Peter A. Ng, and Raymond T. Yeh, editors, *Proceedings of the 3rd Int. Conf. on Entity-Relationship Approach (ER'83)*, pages 329–345. North-Holland, 1983.

[Nil80] Nils J. Nilsson. *Principles of artificial intelligence*. Tioga Publishing co., 1980.

[Nil99] Jørgen Fischer Nilsson. *Data Logic - A gentle Intoduction to Logical Languages*. Department of Information Technology, 1999.

[NJ05] Jørgen Fischer Nilsson and Mads Johnsen. A high level logico-algebraic constraint checking language compiling into database queries. Forthcomming, 2005.

[Ora02] Oracle. Oracle spatial, user guide and reference, March 2002. -.

[PS87] Fernando C.N. Pereira and Stuart M. Shieber. *Prolog and natural-language analysis*. CSLI, 1987.

[PW80] Fernando Pereira and David Warren. Definite clause grammars for language analysis—a survey of the formalism and a comparison to augmented transition networks. *Artificial Intelligence*, 13:231–278, 1980.

[Red93] Swarup Reddi. Integrity constraint enforcement in the functional database language pfl. In *Advances in Databases, 11th British National Conference on Databases, BNCOD 11*, volume 696 of *Lecture Notes in Computer Science*, pages 238–257. Springer, 1993.

[Roe85] Wolfgang Roesner. Despath: An er manipulation language. In Peter P. Chen, editor, *Entity-Relationship Approach: The*

Use of ER Concept in Knowledge Representation, Proceedings of the Fourth International Conference on Entity-Relationship Approach, Chicago, Illinois, USA, 29-30 October 1985, pages 72–81. IEEE Computer Society and North-Holland, 1985.

[SC01] National Survery and Copenhagen Cadastre. Top10dk specification, version 3.2.0, May 2001. In danish.

[SK] K. Slonneger and B. L. Kurtz. Formal semantics of programming languages, - -. Available online: http://www.cs.uiowa.edu/∼slonnegr/plf/Book/.

[SK84] Allan Shepherd and Larry Kerschberg. Prism: a knowledge based system for semantic integrity specification and enforcement in database systems. *SIGMOD Rec.*, 14(2):307–315, 1984.

[Urb89] Susan Darling Urban. Alice: An assertion language for integrity constraint expression. In *Proceedings of Computer Software and Applications Conference*, pages 292–299. IEEE, 1989.

[Wik05a] Wikipedia. Lambda calculus. In , editor, *Wikipedia*. -, 2005.

[Wik05b] Wikipedia. Sql. In , editor, *Wikipedia*. -, 2005.

Appendix A

Concepts Explained

A.1 Constraints in the E/R-model

[GMUW02] suggests E/R diagram notations which expresses constraints. These graphical notations are not supported nor modelled in the conceptual model which the HLCL-system use. Instead these constraints should be formulated in HLCL rather on the E/R diagram it self. In the following it will be shown that all of these graphical constraints can be expressed easily in HLCL.

The diagram labelled A in figure A.1 shows a so-called "multiplicity" con-

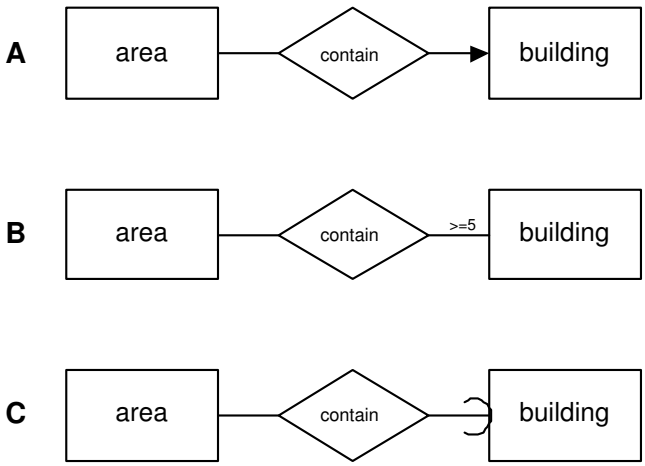

Figure A.1: E/R Diagram Constraint Examples

straint on the "contain" relation. The shown constraint is a "many-one" constraint, which constrain each area to be related through "contain" to at least one building. This can also be formulated in HLCL as seen below:

A `All area must contain building`

The diagram labelled B in figure A.1 on the preceding page imposes the same kind of constraint on the relation, but instead of containing at least one, the ">= 5" expresses that each area should contain at least 5 buildings. This could also be expressed in HLCL as seen below:

B `All area must contain at least 5 building`

Finally we have the diagram labelled C in A.1 on the previous page. The rounded arrowhead expresses referential integrity, which is similar to multiplicity constraints, yet more restrictive. The constraint expresses that the relation should be a "many-one", and furthermore that there should be exactly one building related to every area. This can also be expressed in HLCL as seen below:

C `All area must contain exactly 1 building`

We can conclude that the constraints which [GMUW02] suggests could be put on E/R-diagram, could in fact also be expressed in HLCL. Therefore do the conceptual model used in the HLCL-system not support these graphical constraints, but instead expect the user to formulate them in HLCL.

A.2 Anaphora and "Donkey-sentences"

"Anaphora" is a certain phenomenon in natural language, which [Kin04] has a good definition of seen below:

"Anaphora is sometimes characterized as the phenomena whereby the interpretation of an occurrence of one expression depends on the interpretation of an occurrence of another or whereby an occurrence of an expression has its referent supplied by an occurrence of some other expression in the same or another sentence."[Kin04]

Basically anaphoric phenomena are hard to translate because the meaning of the sentence depends on words located multiple places in the sentence. One of the famous anaphoric sentences example is the so-called "donkey sentence", which describes a common problem with quantifier scoping in anaphora. The sentence can be seen below:

"Every farmer that owns a donkey beats it"

Translating this into HLCL and Predicate Logic seems straightforward, the translation can be seen below:

```
all farmer own donkey must beat donkey
```

$$\forall x farmer(x) \wedge \exists y(donkey(y) \wedge own(x,y)) \rightarrow \exists y(donkey(y) \wedge beat(x,y))$$

But as we can see above, the HLCL and Predicate Logic expression does not capture our intended meaning. The above expressions fail to capture the fact that it is the same donkey which is owned and beaten, which the anaphoric "it" in the sentence describes. In order to express this relationship we need to introduce variables in HLCL in were the expression then would be:

```
all farmer own donkey D must beat donkey D
```

This would be translated into the Predicate Logic expression seen below:

$$\forall x farmer(x) \wedge \exists D(donkey(D) \wedge own(x,D)) \rightarrow beat(x,D)$$

But it can be seen that the scoping of the D variable is wrong, the scope should extend to reach the D in *beat(x,D)* predicate. The correct result is reached by extending the scope of the D quantifier, thereby converting it to a universal quantifier as seen below:

$$\forall x, D farmer(x) \wedge donkey(D) \wedge own(x,D) \rightarrow beat(x,D)$$

We can conclude that when we introduce variables in HLCL we also introduce anaphoric constructs with the variables. As a result user-defined variable names should be globally universally quantified instead of locally existentially quantified. Therefore in the HLCL system all userdefined variables are automatically converted to global universal variables, thereby solving the anaphoric issues, and making HLCL more expressive. A further introduction to "Donkey Sentences" can be found in [Ben86].

A.3 SQL

The target language for HLCL translation is SQL. SQL is an acronym for "Structured Query Language", pronounced "Sequel" and is the de facto standard query language for relational database systems today. It was originally developed by IBM, and it exists in a number of versions: SQL (1987), SQL2 (1992) and SQL3 (1999). Different DBMS vendors implement different parts of the versions, although most commercial DBMS support SQL2 fully, and some even extend it with their own functionality. However the core syntax remains the same across DBMS's and platforms. SQL is an abstraction over the database, where the user is relieved from algorithmic details. From the user perspective all data are stored in tables, where each data set is represented by a row in the table. The basic structure for retrieving data is the "SELECT-FROM-WHERE" construct which can be seen below:

```
SELECT [Columns, * = All columns in the table]
FROM [Tables]
WHERE [Condition]
```

These constructs can be nested so that another SELECT clause can begin in the WHERE clause, as seen below:

```
SELECT [Columns, * = All columns in the table]
FROM [Tables]
WHERE EXISTS(
   SELECT [Yet Another Column]
   FROM [Yet Another Table]
   WHERE [Yet Another Condition]
```

Conditions are used to relate columns, i.e. a condition which requires column to have the same value as another column would be "column = anothercolumn". In general the operators in SQL are straight forward, all the common arithmetic comparisons such as '=','<' and '>' are available, and so are the conjunction/disjunction/negation (AND/OR/NOT) operators and the existential quantifier operator (EXISTS). For a more comprehensive introduction to SQL the reader is referred to [Wik05b, GMUW02].

A.3.1 Spatial Data and SQL

In the following chapter it is described how the actual production database at KMS is built up. KMS uses Oracle DBMS version 9.2.0 with the Spatial Package installed. The Spatial Package contains functions for querying topological data, which are the classes and relations seen in figure A.2 on the facing page and A.3 on page 124. These tables are taken straight from the the documentation for Oracle Spatial which can be found in [Ora02].
In order to express constraints using the topological properties such as: "touch", "intersect" etc, we need to make use of the "SDO_RELATE" spatial function in Oracle SQL. In general "SDO_RELATE" has a form as seen below:

```
boolean::= SDO\_RELATE <geometry>,<geometry>,
'MASK = <mask>, QUERYTYPE = <querytype>') = <bool>;
```

Where <geometry> are geometry columns, and the mask is a combination of the topological operators given in table A.1 on page 130. In order to illustrate its usage look at the example below:

C2 `all area must contain building`

The actual SQL needed for the spatial database can be seen below:

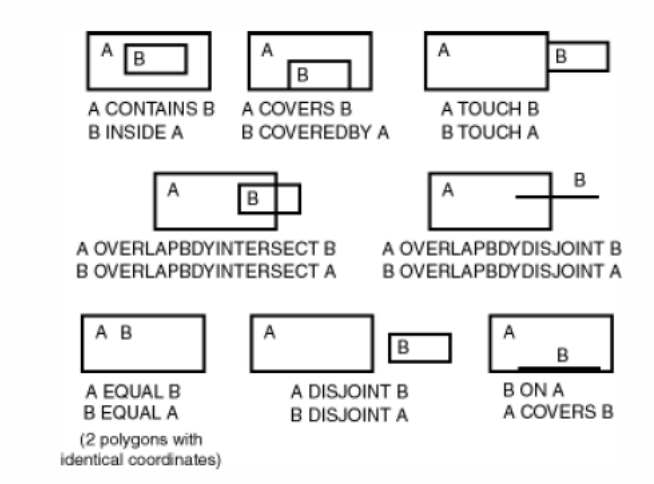

Figure A.2: Topological Relations

```
SELECT *
FROM Area, Building
WHERE
   SDO_RELATE(Area.geom, Building.geom,
   'mask=INSIDE+COVEREDBY querytype=WINDOW') = 'FALSE';
```

"SDO_RELATE" is not the only function the spatial database has: Another feature of spatial databases are to filter queries, so that you can specify a geographic area in which you want to search. This can limit the query evaluation time. But it is beyond the scope of this report to look into performance issues, and also it should be fairly simple for a system to add these primary filters. We have chosen not to implement these functions, instead we adopt the view that these functions should be computed in a view, which is treated just as tables in SQL. For an explanation of why we do not translate into functions please see appendix A.4.3 on page 127.

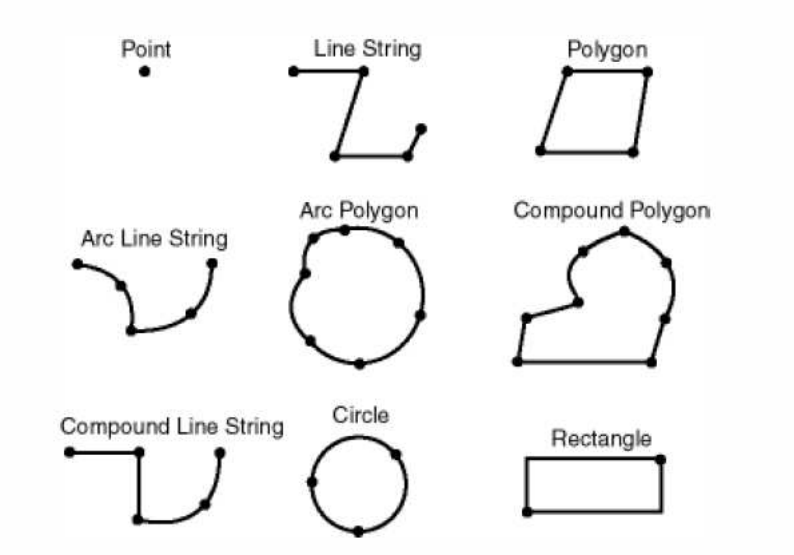

Figure A.3: Topological Classes

A.4 Problematic Representations

In this chapter the discussion of representing the database model started in chapter 8 on page 79 will be continued. In particular we will look at three special cases, namely representing ISA-structures, Weak Entity Sets and Computed Relations.

A.4.1 ISA-structures

Introducing ISA-structures make the translation into SQL a bit more complicated. ISA-structures open up for the possibility of that an attribute is placed at a parent instead at the child, yet the child still refers to the attribute as was attribute its own. How the ISA-structures are treated depends on the representation of them in the database tables. [GMUW02] describes 3 different ways of implementing ISA-structures in relational schema:

1. Using NULL values

2. Treating entities as objects belonging to a single class

3. Following the E/R viewpoint

The following subchapters will contain a walk through of each of these cases, using figure A.4 as the a basis example hierarchy.

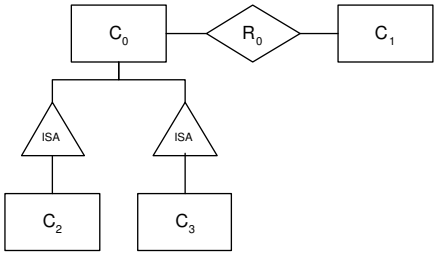

Figure A.4: An example ISA-structure

Using NULL Values

The first method of implementing ISA-structures in databases, is to use one common table for all the classes in the ISA-structure. This results in all attributes being available for all entities. If an entity does not have a value for an attribute, the value is simply set to NULL, hence the name of the method. In the case of the diagram in figure A.4, classes c_0, c_2 and $_3$ would use one common table.
If one chooses to implement ISA-structures by the above method no extra work for the HLCL-system is required, since all attributes relevant for the class is directly available in their own table, no extra SQL lookups are needed.

Treating the entities as objects belonging to a single class

Another method of implementing ISA-structures is to create a table for each subtree in the ISA-structure. This means that in the case of figure A.4 we would create a separate table for c_0, c_2 and c_3 which would each contain all relevant entities. Every class would have their parents attributes and the attributes specific for that class. Yet again, since all attributes relevant for the class is directly available in their own table, no extra SQL lookups are needed.

Following the E/R viewpoint

The final method of implementing ISA-structures is by "following the E/R viewpoint". Following the E/R Viewpoint means that each entity set in the E/R-diagram, should map into its own table, only having the primary key and the attributes specific for that class. This is the only problematic representation of ISA-structures. The solution is already given in chapter 9.2.1 on page 99: An extra SELECT-clause is added, selecting the parent which has the attribute in its table. By relating the class and its parent, and by setting the parents attribute to the given value, we have solved the problem of a child referring to an attribute which its parent have.

A.4.2 Mapping Weak Entity Sets

Weak entity sets are entity sets whose key is composed of attributes which belong to another entity set. In the most extreme case weak entity sets do not have any attributes of their own. Take a look at figure A.5, which shows a fragment of an E/R-diagram with weak entity sets. Imagine that the classes "student","course" and "grade" are all connected to other classes not shown in the E/R-diagram fragment. The basic problem is that we normally have a

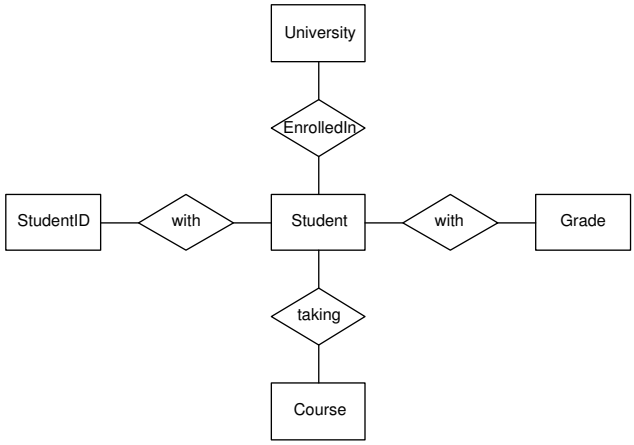

Figure A.5: A Conceptual Model Fragment having Weak Entity Sets

table for each class and relation in the conceptual model, but in weak entity sets there are no tables for the supporting relations: "with" and "in". The Exam entity is basically just a collection of a "studentid", a "courseid" and a "grade". [GMUW02] suggests that these should be modelled as one single table, which means the E/R-diagram would have the following database description:

```
table(examtable,[studentid,courseid,grade])
```

On the other hand we would like to have a conceptual model representation which corresponds exactly to the diagram notation seen in figure A.5. I.e. as the one seen below:

```
class(exam)
...
relation(exam,in,course)
relation(exam,with,grade)
...
```

So the challenge is to make a mapping which can map all the entities in the conceptual model to single table which exists in the database schema. There

are a number of solutions to this problem: In the current HLCL-system it is required that a table as seen above should have a key, a unique attribute identifying each exam, i.e. "examid". This key would allow us to map the supporting relations down into the table as seen below:

```
classmap(exam,examtable,[],[examid]).
relmap(student,taken,exam,[studentid],[examid]).
relmap(exam,with,grade,[examid],[grade]).
...
```

But even with the mapping above, the corresponding SQL would be far from optimal. Constraints involving the exam would make multiple lookups in the exam table, because the HLCL system does not realize the relations map into the same table. A similar approach could be to map the supporting relations as if they were attributes, but the HLCL-system see expressions as these as wellformed, because it does not allow paths made from attributes.

The preferred solution is to introduce yet another mapping predicate "weakmap", which can map weak entity sets in a table:

```
weakmap(student,taken,exam,[studentid],[studentid,courseid,grade])
...
```

Translating weak entity sets in SQL do not pose any major problems. It is simply a matter of that the HLCL should be aware that certain classes are weak entity sets. An extra predicate as seen above, would give that information and allow the HLCL system to translate these correctly. Unfortunately time did not allow us to exploit this issue further, but it could be designed in future HLCL-systems.

A.4.3 Computed Relations

As mentioned in appendix A.3.1 on page 122 some relations in the GIS domain are not actually tables in the database, but in fact computed relations / functions.
In the current HLCL system it is required that all relations in the E/R-model should correspond to a table, or an attribute in a table. So in order to represent computed functions as relations, the HLCL requires that a view made from the computed function, which in SQL is treated just as tables.
But another option would actually be to make a special mapping for computed relations in the database model. We have already defined functions in the database description, for the user-defined predicates. Similarly we could represent the topological functions in the same predicate, as seen below:

```
function(contain,[[a,buildingID],[b,buildingID]],
  sdorelate(a,b)).
```

Then we could introduce an extra 4-ary predicate "relfuncmap", which would map the relation to a SQL function defined in the database representation. Then we could imagine that the relation "contain" would have a definition as seen below:

```
relfuncmap(Area,Contain,Building,contain)
```

So from the representative point of view, functions could easily be added in the database representation. But it seems it is not possible from the SQL point of view. In A.3.1 on page 122 we have seen that an equivalent sentence exists using a function instead of a table for the simplest cases of HLCL sentences. But let us look at the more advanced sentence seen below:

```
all area must contain solely building
```

This would result in the following Predicate Logic expression:

$$\forall x area(x) \wedge \neg\exists y(contain(x, y) \wedge \neg building(y))$$

This should have a SQL structure as seen below:

```
SELECT *
FROM area a
WHERE NOT
   contain(a,X)
AND NOT(
   SELECT *
   FROM building
   WHERE b.id != X)
```

The structure seen above is not allowed in SQL. The problem is the variable 'X' cannot "look ahead". It is not specified what 'X' should be, instead the only thing we specify is that it should not be buildings, the query is "unsafe". It might be possible to construct this query by using built-in variable functionality in SQL somehow. But it is not trivial and it has not been pursued further in this thesis.

A.5 λ-Calculus

The translation strategy in chapter 6 on page 53 uses a notation called λ-calculus. A λ-expression can be seen as a single argument function which contains 2 parts: The first part is the actual function, which consists of a λ, a variable and the function expression, and the second part is the function argument. The syntax can be seen below:

$$(\lambda\langle\text{variable}\rangle.\langle\text{function}\rangle)(\langle\text{value}\rangle)$$

As an example look at the λ-expression below:

$$(\lambda\text{x.x+1})(3)$$

The example above is similar to writing "f(x) = x+1" and "x = 3". The above λ-expression can be solved by using β-reduction. In β-reduction the variable is substituted by the argument, thereby reducing the example to:

$$3+1$$

Which is 3 + 1 = 4. β-reduction left associative and can be continually applied. Another important rule states that the naming of the variables in a λ-expression is unimportant, i.e. $(\lambda\text{x.x+1})(3) = (\lambda\text{y.y+1})(3)$. The renaming of variables are done by α-reduction. These two rules combined are referred to as λ-reduction and this is the rule which will be used throughout this report. A more comprehensive introduction to λ-calculus can be found in [Wik05a, PS87].

A.6 Skolem Functions

For removing existential quantifiers [Nil80] uses a method in which one uses "skolem"-functions. The idea is that all existentially quantified variables can be replaced by a function. Let us imagine we have a Predicate Logic expression with an existentially quantified variable "V". This variable must be dependent on all other variables which are in the scope of "V", and nothing else. This means we can substitute "V", with a function taking the other variables as input and returning the new variable as output. This function called a skolem-function and is left unspecified. Let us look at an example seen below:

$$\forall X \exists Y contain(X, Y)$$

A skolem function would be introduced where Y = f_1(X), so that the expression is equivalent of the expression seen below:

$$\forall X contain(X, f_1(X))$$

We have chosen not to use this approach, since this would move the DATALOG$^\neg$ expression out of First Order Logic. Instead we use the method described in chapter 7.3 on page 74

DISJOINT	The boundaries and interiors do not intersect.
TOUCH	The boundaries intersect but the interiors do not intersect.
OVERLAPBDYDISJOINT	The interior of one object intersects boundary and interior of other object, but two boundaries do not intersect (example: a line originates outside a polygon and ends inside the polygon)
OVERLAPBDYINTERSECT	The boundaries and interiors of the two objects intersect
EQUAL	The two objects have the same boundary and interior
CONTAINS	The interior and boundary of one object is completely contained in the interior of other object
COVERS	The interior of one object is completely contained in interior of other object and their boundaries intersect
INSIDE	The opposite of CONTAINS. A INSIDE B implies B CONTAINS A.
COVEREDBY	The opposite of COVERS. A COVEREDBY B implies B COVERS A.
ANYINTERACT	The objects are non-disjoint

Table A.1: The topological operators in Oracle Spatial

Appendix B

Detailed Implementation

This chapter will contain the specific implementation details. This is meant as a very code near explanation, it does therefore not elaborate on the general ideas. For major design decisions the reader is referred to chapter 10 on page 103. The actual source code can be seen in appendix E on page 191.

The general approach to implementing the HLCL-system is the divide-and-conquer method, this leads to a very modular application with very strict interfaces, where sub-parts of the application functions independently and can be understood separately.
We have tried to reflect this in the source code documentation, where each sub-part of the application is described in its own chapter, thereby giving the reader a possibility to understand the application bottom-up.
The main HLCL-system code is presented in in appendix B.1 on page 133 to B.7 on page 143. A number of predicates which "pretty print" the HLCL expression both to the console, and in LaTeX are explained in Appendix B.8.1 on page 144. Furthermore there are a number of predicates which automates tests which are described in Appendix B.8.2 on page 144.

B.0.1 Notation

To help getting an overview of the code, each subpart of the application uses a graphic notation as seen in figure B.1 on the following page. The notation consists of a frame which has the subpart name, and a series of boxes containing the major predicates used in that subpart. These boxes should be viewed in order: The topmost predicate is the main predicate, any boxes which are beneath and left indented are predicates called by the top-predicate. That means that in figure B.1 on the next page "main_predicate" is the main application, which calls "sub_predicate1" and "sub_predicate2", where "sub_predicate2" calls "sub_sub_predicate". In order to give the reader a better overview the predicate list is not exhaustive; it only lists major predicates and leaves auxiliary predicates out.

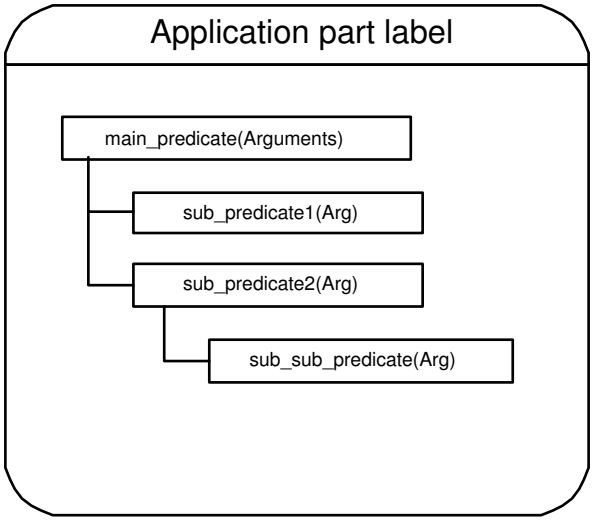

Figure B.1: Notation

B.0.2 Coding Convention

In order for the PROLOG code to be as easy to read and maintain as possible, the code conforms to the code guidelines given below:

- Every predicate name consists of lower-case letters and underscore (_) as a separator, i.e.: "fix_scoping".
- Every variable name starts with upper-case letters and uses no spacing, i.e.: "VarMap".
- Arguments in predicates appear in the following order: First input arguments, then any internally used arguments, and finally output arguments.
- Usage of the SWI-Prolog built-in predicates is kept to a minimum. The ones that are used are the typical list operations such as: `member/2`, `append/3`, etc.
- Every predicate definition will be preceded by a comment, which states the arguments preceeded by + for input, - for output, and ? for both[1]. The names of these arguments follow a type structure which can be seen in table B.1 on the facing page.

[1]In this chapter we will use the term "input" for variables which should be initiated before the predicate is applied, and "output" for variables which are initiated as the predicate succeeds.

B.0.3 Interfaces

A list of the most commonly used interfaces / datatypes can be seen in table B.1. All interfaces except the input HLCL is in a term-notation, instead of the direct notation. For instance is conjugation represented as "and(arg1,arge2)" rather than the straightforward infix notation: "arg1 and arg2", similarly with classes which are represented as "class(area,v(0))" rather than "area(v(0))". This is due to the fact that terms are easier to handle in PROLOG, and we can avoid using the "..=" operator which is potentially hazardous. Instead I/O predicates have been implemented which, at the final stage, can convert interfaces from term-notation to their correct straightforward notation.

Another important datatype are the "ClassVarMap" and "ClassLabelMap".

Name	**Type**	**Example**
Hexp	HLCL Expression	[all,area,must,contain,building]
Pexp	Predicate Logic Expression	all(v(0),imp(class(area,v(0)),...
Sexp	SQL Expression	selectstat(*,area,and(equals(area....
Dexp	DATALOG¬ Expression	[leftimp(function...],[...],...
Class	Class Expression	class(area,v(0))
Rel	Relation Expression	relation(contains,v(0),v(v(0))))
Var	Variable Expression	v(v(v(0)))
VarClassMap	Variable Class Map	[[class,area,v(0)],[attribute,type,var_a]]
VarLabelMap	Variable Label Map	[[v(0),class(area,v(0)),a], [var_a,relation(contain,v(0),var_a),c]]
CountMap	Variable Count Map	[[2,v(0)],[4,var_a]]
...List	List of one of the above	i.e. VarList: [v(0),var_a]

Table B.1: Naming Convention of Interfaces in Documentation

"ClassVarMap" links a variable to a class or an attribute and is used in the HLCL to Predicate Logic expression translation and wellformedness checking. "ClassLabelMap" links a variable to a Class or a Relation and a label, and is used in Extended DATALOG to SQL translation. Both maps consist of a list with 3 dimensional tuples. Therefore the two maps shares various lookup and modification predicates which are described in appendix B.9 on page 144.

B.1 Overview - HLCL to SQL/DATALOG¬

The HLCL system consists of one top-predicate: *"hlcl_to_sqldat"*, which takes an HLCL expression on a list form and returns both DATALOG and SQL versions of that expression. If the HLCL expression is not wellformed the *"hlcl_to_sqldat"* will not succeed, thereby returning no output. Translating

the HLCL expression is a five step process which consists of the following subpredicates:

hlcl_to_pred Transforms the HLCL expression into Predicate Logic. This predicate is explained in appendix: B.2.

check_wellformedness Checks that the HLCL expression is valid. This predicate is explained in appendix: B.3.

perform_intermediate_steps Transforms the well-formed Predicate Logic expression to Extended DATALOG. This predicate is explained in appendix: B.4.

pred_to_sql Transforms the Extended DATALOG into a SQL expression. This predicate is explained in appendix B.5

pred_to_dat Transforms the Extended DATALOG into DATALOG$^\neg$. This predicate is explained in appendix B.6

B.2 HLCL to Predicate LogicTranslation

The HLCL to Predicate Logicexpression translation is done by the binary top predicate: *"hlcl_to_pred"*. The input is an HLCL-expression on list form, and the output is the corresponding Predicate Logic expression on term-form. The actual translation is a three step procedure seen below:

apply_macro_functionality The first step substitutes the class-definitions. The predicate takes an HLCL expression as input and returns an HLCL expression without class definitions. This predicate traverses the HLCL-expression for any class definitions, and substitutes them by the full class expression. The predicate works recursively, e.g. before a substitution is made, the class expression is scanned for further definitions which is substituted further etc.

hlcl The second step translates the HLCL list form to a Predicate Logic expression on term form. This is done by HLCL which is a Definite Clause Grammar. The input is an HLCL-expression and the output is (not necessarily wellformed) Predicate Logic on term form. The DCG corresponds exactly to the BNF syntax given in chapter 3.5 on page 29. The DCG converts the HLCL on list form to the Predicate Logic expression on term-form. The rules for this translation is the translation strategy as seen in chapter 6 on page 53. In the implementation of the HLCL-system we have chosen not to implement all of the advanced translation strategy in the DCG, since it would make the DCG very cluttered, and thereby hard to modify. Instead the DCG translates the simple rules, and the scoping issues is solved in the next

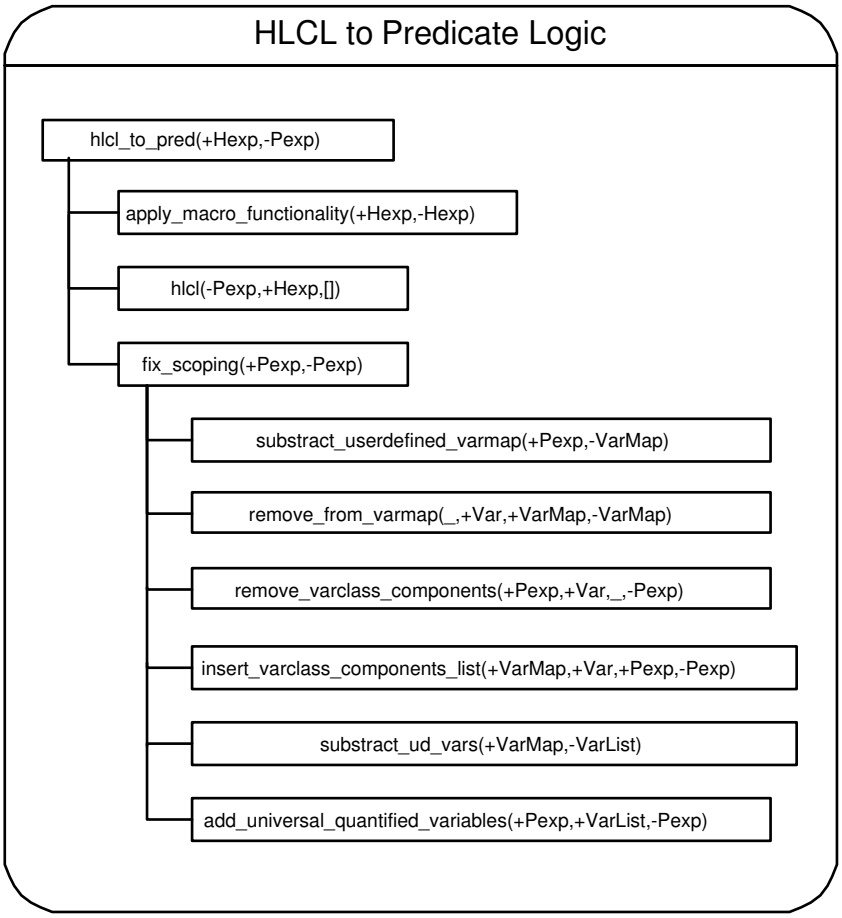

Figure B.2: HLCL to Predicate Logic Translation

step. Furthermore a minimal wellformedness checking is performed, checking if relations and classes used in the expression in fact exists in the conceptual model.

fix_scoping The third step fixes the scoping issues with user-defined variables. The input is a Predicate Logic expression and the output is a Predicate Logic expression with correctly scoped variables and no dual class definitions. If there are no user-defined variables defined in the Predicate Logic expression, then no modifications will be done to the expression. First we need to figure out which user-defined variables are wrongly scoped, this is done by the first two predicates, Second, we need to make sure that there is only one "varclass' component for each user-defined variable which is located immediately after the first class expression, this is done by the next two predicates. And third

we need to correctly scope the user-defined variables, this is done by the last predicate. The scoping issues are solved by the following five subpredicates:

substract_userdefined_varmap First we need to find all user-defined variables. This predicate takes a Predicate Logic expression as input and returns a VarClassMap containing a map of all user-defined variables, and their class and type (class/attribute). The predicate simply traverses the Predicate Logic expression, and every time it meets an "varclass" component or an attribute component with a user-defined variable as value the details are appended to the map.

remove_from_varmap The next step is to remove the first class variable from the map, the reason is that the first variable will by nature always be scoped and located correctly. This predicate takes a VarClassMap and the outer variable as input and the output is a VarClassMap. If the map contains an entry with the outer variable, then that entry is removed from the map, otherwise the map is returned as it was given as input.

remove_varclass_components The next step is to remove all "varclass" components from the Predicate Logic expression. Each userdefined variable is translated into a varclass component by the DCG, and all of these should be removed. This predicate takes a Predicate Logic expression and the VarClassMap from above and returns a Predicate Logic expression. The predicate works by traversing the Predicate Logic expression, and if it finds a varclass component which is in the VarClassMap it is removed from the expression.

insert_varclass_components_list The next step is to insert one of each "varclass" component right after the very first class expression. This predicate takes a Predicate Logic expression and the VarClassMap and returns a Predicate Logic expression. This predicate adds the removed varclass components, right after the very first class expression, this way the classes will archeive the correct scoping.

add_universal_quantified_variable Finally we need to scope the user-defined variables correctly. This predicate takes the variable map from above and the Predicate Logic expression as input and returns a correct Predicate Logic expression as output. The predicate takes each element in the varclassmap and adds an outer universal quantifier quantifiying that variable over the expression.

B.3 Wellformedness Checking

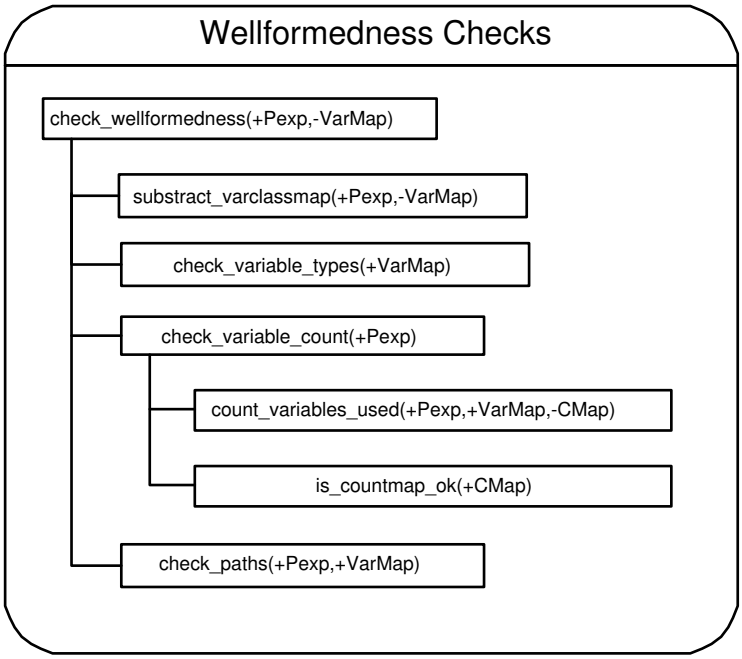

Figure B.3: Wellformedness Checking

The overall predicate checking wellformedness is called *"check_wellformedness"*. The input is a Predicate Logic expression on term form, and there is no output from this predicate. Instead the predicate succeeds if the Predicate Logic expression is wellformed. No modifications are made to the Predicate Logic expression in this predicate. One should note that a minimal wellformedness checking is already done in the DCG in the *"hlcl_to_pred"* predicate, namely WFF4, 5 & 6, but we still need to check the rest of the wellformedness requirements given in chapter 3.5 on page 31. The wellformedness checking is split up, such that each wellformedness check in the list in chapter 3.5 on page 31 corresponds to a predicate. This results in a four-step procedure as seen in figure B.3.

substract_varclassmap First we need to subtract all variables, their type and their class, this map will be used of the rest of the wellformedness checking sub predicates. The input of this predicate is a Predicate Logic expression, and the output is a VarClassMap. This predicate works by traversing the Predicate Logic expression and adding the information every time a "varclass" component is encountered, the class and variable is added to the map.

check_variable_types This predicate checks that the user-defined variables are referring to the same class (WFF3). The predicate takes the VarClassMap as input, and has no output. Instead the predicate succeeds if no variables in the VarClassMap are referring to different classes. The predicate works by traversing the VarClassMap, and for every element in the map, it search the rest of the VarClassMap making sure that there is not an entry where the same variable refers to another type or name of class.

check_variable_count This predicate checks that user-defined variables are used at least twice (WFF2). The input is the Predicate Logic expression, and there is no output, instead the predicate succeeds if the variables are used correctly. The predicate uses two predicates seen below:

count_variables_used This predicate counts the number of times every user-defined variable is used in the Predicate Logic expression. This predicate takes the Predicate Logic expression, the VarClassMap and returns a CountMap as output. The predicate works by traversing the Predicate Logic expression and every time a relation or a user-defined predicate is met, the variables are added to the CountMap. The count map is a list of tuples, where each tuple is the variable name and a number, when added this number is either incremented, or new tuple containing the variable and '1' is added to the list. See table B.1 on page 133 for an example of a CountMap.

We only increment the CountMap when a variable is used in either a relation or a userdefined predicate. One could think that it would be easier to increment when a variable was used in a "varclass" component, but this is not possible since the all "varclass" components except one for each variable were removed in the HLCL to Predicate Logic expression translation.

is_countmap_ok The next step is to check if every element in the VarCountMap is used at least twice. This predicate takes the VarCountMap as input and returns nothing for output, instead the predicate succeeds if the VarCountMap is correct. The predicate traverses through every element in the VarCountMap to ensure that the count in there is at least 2. If every element has at least two the predicate succeeds.

check_paths Finally we need to check that the Predicate Logic expression uses valid paths in the conceptual model (WFF1). This predicate takes the Predicate Logic expression as and VarClassMap as input and returns nothing as output. Instead it succeeds if all paths used in the Predicate Logic expression are correct. The predicate traverses the

Predicate Logic expression and every time a relation/attribute/usedefined predicate is met, the variables in their arguments is looked up in the varclass map, and the predicate checks to see if the path exists in the conceptual model.

B.4 Intermediate Steps

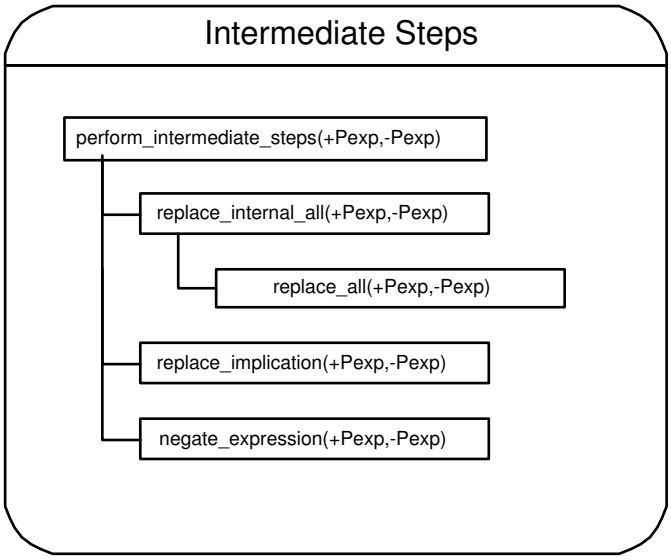

Figure B.4: Intermediate steps

The intermediate steps converts the Predicate Logic expression to a Extended DATALOG expression. The input is the Predicate Logic expression in term-form, and the output is Extended DATALOG in term-form. There are three steps in the intermediate translation, which is handled by the following three subpredicates.

replace_internal_all The first step is to replace all internal universal quantifications. This predicate takes an Predicate Logic expression with internal universal quantifications and returns a Predicate Logic expression without internal universal quantifications. The predicate works by first traversing through the outer universal quantifications, which are to be left untouched, and then call *"replace_all"* on the inner Predicate Logic expression.

replace_all This predicate replaces all universal quantifications with a double-negated existential quantifier, furthermore it replaces the implication inside the universal quantification with a conjugation. The input of this predicate is a Predicate Logic expression

and the output is a Predicate Logic expression without any universal quantifications.

replace_implication The second step is to replace the outer implication from "all-must" constructs. This predicate takes a Predicate Logic expression as input and returns a Predicate Logic expression without any implications. The predicate works by replacing the implication with a conjugation, thereby both removing the implication and negating the expression, as described in chapter 7 on page 69. This predicate does not have any affect on expressions made up by a "no-may" construct.

negate_expression The third and final step is to negate any "no-may" expressions. This predicate takes a Predicate Logic expression as input and returns a Predicate Logic expression. This predicate finds the first negated existential quantifier and replaces it with a universal quantifier, thereby negating the expression and returning it as output.

B.5 Extended DATALOG to SQL

The Extended DATALOG into SQL translation is implemented differently than the other parts of the HLCL-system. Instead of being broken down into multiple predicates each doing a small step of the translation, we basically only have one predicate: *"subpred_to_sql"* which instead case out in numerous different cases. *"subpred_to_sql"* predicate which takes 7 arguments seen which can be seen below. Their typical name can be seen below:

1: - The input Extended DATALOG expression

2: - The output SQL expression

3: Inside A SQL fragment which needs to be put in the "WHERE" clause of the translated SQL

4: VarMapIn A ClassLabelMap for all variables within the current scope before the translation.

5: VarMapOut A ClassLabelMap for all variables within the current scope after the translation.

6: Varclassmap A map containing all the classes and relations which has been translated into tables.

7: CountFlag A flag indicating if the next select statement should use '*' or 'count(*)'.

In order to differentiate between the maps, we will in the following call the maps used in 4th and 5th argument the "relational map", whereas the map

in the 6th argument is referred to as the "classmap". The actual translation cases out on many different cases, which can be seen in table B.2 on page 145. Most of the cases uses a couple of subfunctions in order to build the SQL query. The most common ones can be seen below:

is_exists_expression This function can tell if a DATALOG¬ subexpression starts with an existential quantifier. The input is an Extended DATALOG expression on termform, and there is not output. This function is used to determine wether a conjugation is a "real" AND or a "glue" AND.

conceptual_to_table This function translates a class into a table and a condition. The input is a classexpression, and the output is a table-name and a (possibly empty) condition.

conceptual_relation_to_table This function translates a relation into a table and a condition. The input is a relation and its connected classes, and the output is a table-name and a (possibly empty) condition.

condition_to_sql This function translates the condition from the two above functions into SQL. The input is a condition and the output is an SQL fragment on term form.

set_equal This function is used to generate the SQL fragment which sets two entities equal. The input is two classexpression or relation expressions, and the output is a SQL fragment which makes the condition that the two entitites should be equal.

make_negation_expression This function is the exact oposite *"set_equal"*. Instead of generating the SQL fragment which sets two entities equal, it generates a SQL fragment which sets the two not to be equal. The input is two classexpressions, and the output is a SQL fragment which makes the condition that the two entitites should be not equal.

B.6 Extended DATALOG to DATALOG¬

Converting Extended DATALOG into DATALOG¬ is done in *"pred_to_datalog"*. The input is an Extended DATALOG expression and the output is DATALOG¬ expression on term form. The translation is split up in steps which corresponds to the steps formally covered in chapter 7.3 on page 73.

reduce_scope_of_negation The first step is to reduce the scope of negation. The input is a Predicate Logic expression and the output is a Predicate Logic expression where only literals are negated. The predicate works by traversing the expression, and every time it meets a negated conjunction or disjunction, the negation is moved inwards according to de-morgans rules.

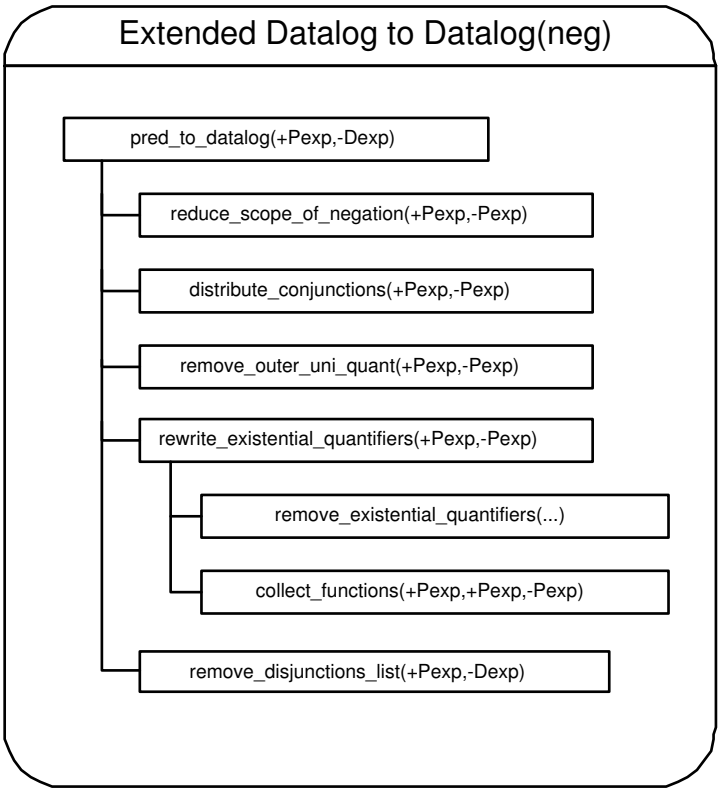

Figure B.5: Extended DATALOG to DATALOG$^\neg$ Translation

distribute_conjunctions The next step is to distribute conjunctions. The input is a Extended DATALOG expression with only negated literals and the output is a Extended DATALOG expression on disjunctive normal form. The predicate works by traversing down through the expression finding structures of conjugated disjuncts, and unfolds them as described in chapter 7.3 on page 73.

remove_outer_universal_quantifiers The next step is to remove outer universal quantifiers. The input is a Extended DATALOG expression on disjunctive normal form, and the output is a Extended DATALOG expression on disjunctive normal form without any universal quantifiers. The predicate works by traverses through the first universal quantifiers (which is the only place where they can be located, since inner universal quantifiers were removed in the intermediate steps) and returns the expression inside, thereby removing the universal quantifiers.

rewrite_existential_quantifiers The next step is to rewrite existential

quantifiers. This predicate takes a Extended DATALOG expression on disjunctive normal form and returns a list of DATALOG¬ clauses. This predicate works by using two other predicates, first *"remove_existential_quantifiers"* is called breaks up the Extended DATALOG expression into a number of clauses, and then *"collect_functions"* which merges the DATALOG¬ clause with the list of clauses.

remove_existential_quantifiers This predicate takes a Predicate Logic expression as input and returns a DATALOG¬ clause which corresponds to the original Predicate Logic expression and a list of extra clauses. The predicate works by traversing the expression, and every time a existential quantifier is met, it is being substidized with a function. The expression inside the existentially quantified expression is set equal to the function and put in the clause list.

collect_functions The second step is to append the DATALOG¬ expression corresponding to the original Predicate Logic expression to the list of function clauses. This predicate puts the "error←" on the original clause, and appends it to the list, resulting in one list with all clauses.

remove_disjunctions_list Finally we need to break up disjunctions. This predicate takes a DATALOG¬ clause list as input and returns a DATALOG¬ clause list as output. The predicate applies remove_disjunction on every clause, appending all returning clauses together.

remove_disjuntions This predicate splits up clauses which contain disjunctions. The predicate take a single DATALOG¬ clause as input and returns a (possibly single-element) list of clauses. The predicate looks for any outer disjunctions, and if found, the expression is split up into two DATALOG¬ sentences each resulting in the function or error.

B.7 Settings

Conceptual Model

The conceptual model representation is in already described in detail in chapter: 8 on page 79, and will not be further explained here. Besides the conceptual model representation a predicate for checking wellformedness of the database model is supplied, and will be explained below:

check_conceptual_model this 0-argument predicate checks if the conceptual model lives up to the wellformed requirements given in chapter 8.3 on page 87.

User-defined Variables

Besides the actual conceptual model, the user can also define the names of their own user-defined variables. By default these are set to "var_a" ... "var_e", but they can be declared with the *"userDefVariable"* predicate.

Definitions

Class definitions as described in chapter 3.3.2 on page 25 can be introduced by the binary *"definition"* predicate, where the first argument is a label of the defined class, and the second is the corresponding class expression, given in the terms of the conceptual model.

B.8 Other Predicates

In the following sub-appendixes predicates not directly related to the HLCL-system will be described. These are typically I/O and validation predicates.

B.8.1 I/O Predicates

For pretty-printing our interfaces: HLCL, Predicate Logic expression, DATALOG and SQL a couple of output predicates has been defined.
These predicates converts the internal term representation of the interfaces to an output form which is easy to read. The names of the print predicate are self explanatory. The details of these predicates will not be further explained, they are simply a collection of *"print"* and *"nl"* predicates, and are straightforward to understand. There exists pretty-printing predicates for both the terminal (plain-text) and for LaTeX.

B.8.2 Test Predicates

In order to do a lot of tests easily and verify the correctness of the HLCL-systems, two predicates for automating tests have been implemented. The predicates are called *"run_tests"* and *"run_tests_tex"*, and they simply call the predicates in the order described in chapter: B.1 on page 133, where they print out the intermediate results. *"run_tests"* makes use of the plain-text pretty printing facilities, and *"run_tests_tex"* makes use of the LaTeX pretty-printing facilities. Test funktioner af definitoner

B.9 Auxiliary Predicates

The auxialliary predicates are used of most part of the HLCL-system. The names are fairly self explanatory, and the predicates themselves are typically very simple. Therefore the reader is referred to the user comments in the code.

Table B.2: The cases in Extended DATALOG to SQL Translation

#	Case	Description
1	`Conjunction ‘‘real’’ And`	The conjugation is translated to "AND" in SQL and given maps are passed on to the translation of the sub-expressions on the left- and right-hand side.
2	`Disjunction Or`	The disjunction is translated to "AND" in SQL and given maps are passed on to the translation of the sub-expressions on the left- and right-hand side.
3	`‘‘Glue’’ Conjunction`	The "glue" conjunction is translated to "AND" in SQL similar to case 1. But the VarMaps from the translated left-hand side of the conjunction is passed to the transaltion of the right-hand side of the expression. Furtermore is the translated SQL from the right-hand side nested in the 'WHERE'-clause of the translated right-hand side.
4	`Negation`	The negation translates into a 'NOT' in SQL, and all VarMaps are passed on to the translation of the sub-expression.
5	`Existential Quantifier`	Existential quanfitifier is not translated into anything in SQL. Instead the class which corresponds to the existential quantified variable is found and added it to the relational map.
6	`Nummerical Quantifier`	Nummerical quanfitifier is translated to the outer comparison and the flag for nummerical relation is raised. Furthermore the class which corresponds to the nummerical quantified variable is found and added it to the relational map.

Continued on next page

Table B.2 – continued from previous page

#	Case	Description
7	`Universal Quantifer`	Universal quanfitifier is not translated into anything in SQL. Instead the class which corresponds to the universally quantified variable is found and added it to the relational map.
8	`Universal Quantifier, quantifying a attribute value`	As case 7.
9	`The very first class in the extended datalog expression`	Translates into SQL without an 'EXIST' infront, and the label is set to "a"
10	`The very first varclass in the extended datalog expression`	As case 9.
11	`A class in a relational path`	Translates into a 'SELECT' statement, with an 'EXISTS' in front and the table which corresponds to the class key attributes is set equal to the previous relation
12	`A varclass in the relational path`	As 11
13	`A varclass in a relational path, before the relation is met`	Translates into a 'SELECT' statement, with an 'EXISTS' in front.
14	`A relation relating the same variables`	Translates into a 'SELECT' clause where both key-sets in the relation table is set equal to the class.
15	`A relation relating already translated classes`	Translates into a 'SELECT' statement, and relates both key-sets in the relation table to the keys of the table corresponding to the classes.
16	`A relation relating a translated class to a non-translated class`	Translates into a 'SELECT' statement, setting the first key-set in the relation equal to the keys in the table corresponding to the first class
17	`A relation right after a nummerical quantifier`	Same as 16, but a 'count(*)' instead of a '*' is used in the select statement. Resets the nummerical flag

Continued on next page

Table B.2 – continued from previous page

#	Case	Description
18	`The first time an attribute with a user-defined variable as value, which is located at a parent in a ISA-structures is met.`	Translates into an extra 'SELECT' statement which relates the parent to the class, and the parents label is added to the classmap
19	`The next time an attribute with a user-defined variable as value, which is located at a parent in a ISA-structures is met`	Translated into a SQL expression which set the current attribute equal to the one which is looked up in the VarMap
20	`The first time an attribute, which is located at a parent in a ISA-structures is met`	Translates into an extra 'SELECT' statement which relates the parent to the class, the attribute in the parent is set to be the attributes value, and the parents label is added to the classmap
21	`The next time an attribute, which is located at a parent in a ISA-structures is met.`	Same as 20
22	`A nummerically quantified attribute which is located at a parent in a ISA-structures`	Same as 20, but instead of setting the attributes equal the comparison is done.
23	`All other attributes not meeting any of the above descriptions`	The attribute is set the the given value in SQL
24	`A unary predicate which has its argument in a parent in a ISA-structures`	Translates into a 'SELECT' statement relating the class and its parent together, using the parent as an argument in the predicate
25	`All other unary predicates`	Translates to a SQL function. The argument is looked up in the relation map.

Continued on next page

Table B.2 – continued from previous page

#	Case	Description
26	`Binary predicate which has both arguments in parents in an ISA-structures`	As 24, but with two extra SELECT statements.
27	`Binary predicate which its first arguments in parent in an ISA-structures`	As 24, but with one extra SELECT statement relation the first argument.
28	`Binary predicate which its second arguments in parent in an ISA-structures`	As 24, but with one extra SELECT statement relation the second argument.
29	`All other binary predicates`	Translates to a SQL. The arguments are looked up in the relation map.

Appendix C

Userguide and CD Contents

C.1 CD Contents

The attached CD contains the report and HLCL system, and it has the following file structure:

```
read.me                 This description

\doc\report.pdf         This report
\doc\report.ps          This report

\src\hlcl.pl            The HLCL System
\src\tests.pl           The HLCL Test Cases
\src\usersettings.pl    User Settings incl. Database Model
```

C.2 User Guide

This guide concerns the more specific technical issues related to installing and using the HLCL-system. For an introduction to the actual HLCL language, the reader is referred to chapter 3 on page 13.

C.2.1 Installing the HLCL System

In order for the HLCL system to run, one needs to have PROLOG installed. SWI-Prolog can be downloaded free of charge from the following URL: "http://www.swi-prolog.org/". The HLCL-system is tested with version 5.4.2, so this is the version we recommend to download. To install PROLOG follow the installation instructions found at the same page.
The HLCL-system can run directly from the CD, although if one needs to make changes to the user-settings, it should be copied on to a hard drive.

To start the HLCL-system open the file "hlcl.pl" by doubleclicking on it, this should result in the SWI-prolog starting up with the following text:

```
%  usersettings compiled 0.00 sec, 12,464 bytes
%   usersettings compiled 0.00 sec, 0 bytes
%  helper compiled 0.00 sec, 2,896 bytes
% e:\hlcl.pl compiled 0.01 sec, 104,296 bytes
Welcome to SWI-Prolog (Multi-threaded, Version 5.4.2)
Copyright (c) 1990-2003 University of Amsterdam.
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.

For help, use ?- help(Topic). or ?- apropos(Word).

1 ?-
```

Now the HLCL-system is loaded and ready for translating HLCL expressions.

C.2.2 Running the HLCL System

The user can either run the given testsuite, which is done by the following command

```
:-[tests].
```

This will print out all testcases to the screen. If the user wants to try to compile a query himself, then the toppredicate *"hlcl_to_datalog"* should be used. It takes HLCL as input on a list form, ie:

```
:-hlcl_to_datalog([all,area,must,contain,building],SQL,DATALOG).
```

It is possible to modify usersettings and database representaion by openening the file: "user.pl". A detailed explanation for how to represent the database in PROLOG is given in chapter 8 on page 79. It is recommended that the database model is checked before actually using it - this can be done with the following command:

```
:-check_conceptual_model.
```

Which will return a "conceptual model is OK", if everything is in order.

Appendix D

Test Cases

This chapter will describe the tests which were performed on the HLCL-system. In general two kinds of tests were performed: First a number of HLCL-expressions were translated into SQL and DATALOG$^{\neg}$. Second, some of translated SQL queries were tested in a test-database.
For testing the SQL queries, a test-database was implemented in MySQL 4.1.7[1]. The subset of SQL which our queries use functions similar in MySQL and in Oracle which is used by KMS, but MySQL is easier to install and this was chosen as a test platform instead of the huge and costly Oracle DBMS. Only a representative part of the test queries were tested in the database, the rest of the queries were manually inspected.

D.1 Overview

In the following table each of the test cases will be described. Testing every single HLCL constraint is not possible, due to the vast amount of constraints. Instead a representative part of HLCL constraints has been chosen, which tries to combine as many as the constructs and implications as possible.

[1] An open-source database which can be downloaded free of charge at: www.mysql.com

Table D.1: Test Cases for the HLCL-system

Test #	Constraint	Rationale	Result
1	`all lake must area`	The simplest sentence type	√
2	`no area may building`	The simplest sentence type negated	√
3	`all house must house`	The simplest sentence relating entities with keys formed by multiple attributes	√
4	`no area must building type industrial`	The simplest sentence type with paths on righthandsside	√
5	`no area touch area must building type industrial`	The simplest sentence type with paths on both sides	√
6	`all area must contain building`	One simple path expression	√
7	`all building must touch building`	Another simple path expression	√
8	`all area must have house`	Another simple path expression using multiple attributes as key	√
9	`no area may contain lake`	Negated path expression	√
10	`all building must type industrial`	Path which is just an attribute	√
11	`all building type industrial must touch building`	Longer path on righthandsside	√
12	`all building type industrial must touch building type residential`	Longer path on righthandsside	√
13	`no building type industrial may touch building type residential`	Longer path on righthandsside	√
14	`no building may containedin area contain lake`	Longer path on righthandsside	√

Continued on next page

Table D.1 – continued from previous page

Test #	Constraint	Rationale	Result
15	`all building type industrial must beusedby company`	Longer path on lefthandsside and righthandsside	√
16	`no building may containedin area have house`	Longer path expression using multiple attributes as key	√
17	`all area must contain building or contain lake`	Using the or operator	√
18	`all area must contain building and contain lake`	Using the and operator	√
19	`all area must contain building ornot contain lake`	Using the ornot operator	√
20	`all area must contain building andnot contain lake`	Using the andnot operator	√
21	`all area must contain building touch building or contain lake`	Using the or operator and a path	√
22	`no area may contain all building`	Using the all keyword	√
23	`no area may contain all building touch building`	Using the all keyword and a path	√
24	`all area must contain solely building`	Using the solely keyword	√
25	`all industrialArea must contain solely building type industrial`	Using the solely keyword and a path	√

Continued on next page

Table D.1 – continued from previous page

Test #	Constraint	Rationale	Result
26	`all building must type blockbuilding and touch solely building`	Using the solely keyword and an operator	√
27	`all building X must touch building X`	Using variable	√
28	`all area contain building X must contain building X`	Using variables and paths	√
29	`all building type X must touch building type X`	Using variable for attributes	√
30	`all area intersectedby road X must contain building intersectedby road X`	Using variable and long pahts	√
31	`all building X must touch building X and touch building X`	Referencing to the same variable more than twice	√
32	`all building X touch building Y must zdifferencelargerthan5 X Y`	Using variable and userdefined predicates	√
33	`all building X type Z and touch building Y type Z must zdifferencelargerthan5 X Y`	Using variable, predicates and paths	√
34	`all residentialArea must contain at most 5 building`	Using nummerical quantifiers	√
35	`all residentialArea must contain at least 37 building`	Using nummerical quantifiers	√

Continued on next page

Table D.1 – continued from previous page

Test #	Constraint	Rationale	Result
36	`all residentialArea must contain excactly 1 building type industrial`	Using nummerical quantifiers and paths	√
37	`all residentialArea must zipcode at most 2000`	Using nummerical quantifiers for attributes	√
38	`all residentialArea must zipcode 3520`	Using a child in an ISA-structures	√
39	`all residentialArea zipcode X must touch residentialArea zipcode X`	Setting an attribute value variable for a child ISA-structures, where the attribute actually in parent	√
40	`all residentialArea zipcode X must touch area zipcode X`	Setting an attribute value variable for a child ISA-structures, where the attribute actually in parent	√
41	`all area zipcode X must touch residentialArea zipcode X`	Setting an attribute value variable for a child ISA-structures, where the attribute actually in parent	√
42	`all residentialArea X must zipcodefunc X`	Setting an attribute value for a child ISA-structures, where the attribute actually in parent and using a userdefined predicate	√
43	`all residentialArea X touch area Y must binfunc X Y`	Setting an attribute value for a child ISA-structures, where the attribute actually in parent and using a userdefined predicate	√
44	`all area must ( contain building )`	Use of optional brackets	√
45	`all area must ( contain building type industrial )`	Use of optional brackets	√

Continued on next page

Table D.1 – continued from previous page

Test #	Constraint	Rationale	Result
46	`all definedbuilding must touch definedbuilding`	Testing usage of class definitions	√
47	`no definedarea must intersectedby road`	Testing usage of recursive class definitions	√
48		An empty expression. Should fail!	√
49	`all ( area must ( contain building type industrial )`	Wrong use of optional brackets. Should fail!	√
50	`all area and building must contain building`	Use of operators on righthansside. Should fail!	√
51	`all area X must contain building`	Use of only one userdefined variable. Should fail!	√

D.2 Actual Tests

Testcase 1

The simplest sentence type

HLCL `all lake must area`

Pred. Logic $\forall A\ (lake(A) \rightarrow area(A))$

Ext. Dat $\forall A\ (lake(A) \wedge \neg(area(A)))$

SQL:

```
SELECT *
FROM lake a
WHERE NOT (EXISTS(
SELECT *
FROM area b
WHERE (b.areaID = a.lakeID)))
```

DATALOG$^{\neg}$:

error $\leftarrow$ lake(A) $\wedge$ $\neg$area(A)

Testcase 2

The simplest sentence type negated

HLCL `no area may building`

Pred. Logic $\neg(\exists A\ (area(A) \wedge building(A)))$

Ext. Dat $\forall A\ (area(A) \wedge building(A))$

SQL:

```
SELECT *
FROM area a
WHERE EXISTS(
SELECT *
FROM building b
WHERE (b.buildingID = a.areaID))
```

DATALOG$^{\neg}$:

error $\leftarrow$ area(A) $\wedge$ building(A)

Testcase 3

The simplest sentence relating entities with keys formed by multiple attributes

HLCL `all house must house`

Pred. Logic $\forall A\ (house(A) \rightarrow house(A))$

Ext. Dat $\forall A\ (house(A) \wedge \neg(house(A)))$

SQL:

```
SELECT *
FROM house a
WHERE NOT (EXISTS(
SELECT *
FROM house b
WHERE (b.houseID = a.houseID) AND (b.owner = a.owner)))
```

DATALOG$^{\neg}$:

$error \leftarrow house(A) \wedge \neg house(A)$

Testcase 4

The simplest sentence type with paths on righthandsside

HLCL `no area must building type industrial`

Pred. Logic $\neg(\exists A\ (area(A) \wedge building(A) \wedge type(A,type(A,industrial)))$

Ext. Dat $\forall A\ (area(A) \wedge building(A) \wedge type(A,type(A,industrial))$

SQL:

```
SELECT *
FROM area a
WHERE EXISTS(
SELECT *
FROM building b
WHERE (b.buildingID = a.areaID) AND (b.type) = 'industrial')
```

DATALOG$^{\neg}$:

$error \leftarrow area(A) \wedge building(A) \wedge type(A,industrial)$

Testcase 5

The simplest sentence type with paths on both sides

HLCL `no area touch area must building type industrial`

Pred. Logic $\neg(\exists A\ (area(A) \wedge \exists B\ (touch(A,B) \wedge area(B)) \wedge building(A) \wedge type(A,type(A,industrial)))$

Ext. Dat $\forall A\ (area(A) \wedge \exists B\ (touch(A,B) \wedge area(B)) \wedge building(A) \wedge type(A,type(A,industrial))$

SQL:

```
SELECT *
FROM area a
WHERE EXISTS(
SELECT *
FROM contain b
WHERE (b.geoID1 = a.areaID) AND EXISTS(
SELECT *
FROM area c
WHERE (c.areaID = b.geoID2) AND EXISTS(
SELECT *
FROM building d
WHERE (d.buildingID = a.areaID) AND (d.type) = 'industrial')))
```

DATALOG$^{\neg}$:

error $\leftarrow$ area(A) $\wedge$ f1(A) $\wedge$ building(A) $\wedge$ type(A,industrial)
f1(A) $\leftarrow$ touch(A,B) $\wedge$ area(B)

Testcase 6

One simple path expression

HLCL `all area must contain building`

Pred. Logic $\forall A\ (area(A) \rightarrow \exists B\ (contain(A,B) \wedge building(B)))$

Ext. Dat $\forall A\ (area(A) \wedge \neg(\exists B\ (contain(A,B) \wedge building(B))))$

SQL:

```
SELECT *
FROM area a
WHERE NOT (EXISTS(
SELECT *
FROM contain b
WHERE (b.geoID1 = a.areaID) AND EXISTS(
SELECT *
FROM building c
WHERE (c.buildingID = b.geoID2))))
```

DATALOG$^{\neg}$:
error $\leftarrow$ area(A) $\wedge$ $\neg$f1(A)
f1(A) $\leftarrow$ contain(A,B) $\wedge$ building(B)

Testcase 7

Another simple path expression

HLCL `all building must touch building`

Pred. Logic $\forall A$ *(building(A)* $\rightarrow$ $\exists B$ *(touch(A,B)* $\wedge$ *building(B)))*

Ext. Dat $\forall A$ *(building(A)* $\wedge$ $\neg$*(*$\exists B$ *(touch(A,B)* $\wedge$ *building(B))))*

SQL:

```
SELECT *
FROM building a
WHERE NOT (EXISTS(
SELECT *
FROM contain b
WHERE (b.geoID1 = a.buildingID) AND EXISTS(
SELECT *
FROM building c
WHERE (c.buildingID = b.geoID2))))
```

DATALOG$^{\neg}$:
error $\leftarrow$ building(A) $\wedge$ $\neg$f1(A)
f1(A) $\leftarrow$ touch(A,B) $\wedge$ building(B)

Testcase 8

Another simple path expression using multiple attributes as key

HLCL `all area must have house`

Pred. Logic $\forall A\ (area(A) \rightarrow \exists B\ (have(A,B) \wedge house(B)))$

Ext. Dat $\forall A\ (area(A) \wedge \neg(\exists B\ (have(A,B) \wedge house(B))))$

SQL:

```
SELECT *
FROM area a
WHERE NOT (EXISTS(
SELECT *
FROM have b
WHERE (b.areaID = a.areaID) AND EXISTS(
SELECT *
FROM house c
WHERE (c.houseID = b.houseID) AND (c.owner = b.owner))))
```

DATALOG$^{\neg}$:

error $\leftarrow$ area(A) $\wedge$ $\neg$f1(A)
f1(A) $\leftarrow$ have(A,B) $\wedge$ house(B)

Testcase 9

Negated path expression

HLCL `no area may contain lake`

Pred. Logic $\neg(\exists A\ (area(A) \wedge \exists B\ (contain(A,B) \wedge lake(B))))$

Ext. Dat $\forall A\ (area(A) \wedge \exists B\ (contain(A,B) \wedge lake(B)))$

SQL:

```
SELECT *
FROM area a
WHERE EXISTS(
SELECT *
FROM contain b
```

```
WHERE (b.geoID1 = a.areaID) AND EXISTS(
SELECT *
FROM lake c
WHERE (c.lakeID = b.geoID2)))
```

DATALOG$^{\neg}$:
error ← area(A) ∧ f1(A)
f1(A) ← contain(A,B) ∧ lake(B)

Testcase 10

Path which is just an attribute

HLCL `all building must type industrial`

Pred. Logic $\forall A$ *(building(A)* $\rightarrow$ *type(A,type(A,industrial))*

Ext. Dat $\forall A$ *(building(A)* $\wedge \neg$*(type(A,type(A,industrial)))*

SQL:

```
SELECT *
FROM building a
WHERE NOT ((a.type) = 'industrial')
```

DATALOG$^{\neg}$:
error ← building(A) ∧ ¬type(A,industrial)

Testcase 11

Longer path on righthandsside

HLCL `all building type industrial must touch building`

Pred. Logic $\forall A$ *(building(A)* $\wedge$ *type(A,type(A,industrial)* $\rightarrow \exists B$ *(touch(A,B)* $\wedge$ *building(B)))*

Ext. Dat $\forall A$ *(building(A)* $\wedge$ *type(A,type(A,industrial)* $\wedge \neg$*(*$\exists B$ *(touch(A,B)* $\wedge$ *building(B))))*

SQL:

```
SELECT *
FROM building a
WHERE (a.type) = 'industrial' AND NOT (EXISTS(
SELECT *
FROM contain b
WHERE (b.geoID1 = a.buildingID) AND EXISTS(
SELECT *
FROM building c
WHERE (c.buildingID = b.geoID2))))
```

DATALOG$^{\neg}$:
error $\leftarrow$ building(A) $\wedge$ type(A,industrial) $\wedge$ $\neg$f1(A)
f1(A) $\leftarrow$ touch(A,B) $\wedge$ building(B)

Testcase 12

Longer path on righthandsside

HLCL `all building type industrial must touch building type residential`

Pred. Logic $\forall A$ *(building(A)* $\wedge$ *type(A,type(A,industrial)* $\rightarrow$ $\exists B$ *(touch(A,B)* $\wedge$ *building(B)* $\wedge$ *type(B,type(B,residential)))*

Ext. Dat $\forall A$ *(building(A)* $\wedge$ *type(A,type(A,industrial)* $\wedge$ $\neg$*(*$\exists B$ *(touch(A,B)* $\wedge$ *building(B)* $\wedge$ *type(B,type(B,residential))))*

SQL:

```
SELECT *
FROM building a
WHERE (a.type) = 'industrial' AND NOT (EXISTS(
SELECT *
FROM contain b
WHERE (b.geoID1 = a.buildingID) AND EXISTS(
SELECT *
FROM building c
WHERE (c.buildingID = b.geoID2) AND (c.type) = 'residential')))
```

DATALOG$^{\neg}$:
error $\leftarrow$ building(A) $\wedge$ type(A,industrial) $\wedge$ $\neg$f1(A)
f1(A) $\leftarrow$ touch(A,B) $\wedge$ building(B) $\wedge$ type(B,residential)

Testcase 13

Longer path on righthandsside

HLCL `no building type industrial may touch building type residential`

Pred. Logic $\neg(\exists A$ *(building(A)* $\wedge$ *type(A,type(A,industrial)* $\wedge \exists B$ *(touch(A,B)* $\wedge$ *building(B)* $\wedge$ *type(B,type(B,residential))))*

Ext. Dat $\forall A$ *(building(A)* $\wedge$ *type(A,type(A,industrial)* $\wedge \exists B$ *(touch(A,B)* $\wedge$ *building(B)* $\wedge$ *type(B,type(B,residential)))*

SQL:

```
SELECT *
FROM building a
WHERE (a.type) = 'industrial' AND EXISTS(
SELECT *
FROM contain b
WHERE (b.geoID1 = a.buildingID) AND EXISTS(
SELECT *
FROM building c
WHERE (c.buildingID = b.geoID2) AND (c.type) = 'residential'))
```

DATALOG$^{\neg}$:

error $\leftarrow$ building(A) $\wedge$ type(A,industrial) $\wedge$ f1(A)
f1(A) $\leftarrow$ touch(A,B) $\wedge$ building(B) $\wedge$ type(B,residential)

Testcase 14

Longer path on righthandsside

HLCL `no building may containedin area contain lake`

Pred. Logic $\neg(\exists A$ *(building(A)* $\wedge \exists B$ *(containedin(A,B)* $\wedge$ *area(B)* $\wedge \exists C$ *(contain(B,C)* $\wedge$ *lake(C)))))*

Ext. Dat $\forall A$ *(building(A)* $\wedge \exists B$ *(containedin(A,B)* $\wedge$ *area(B)* $\wedge \exists C$ *(contain(B,C)* $\wedge$ *lake(C))))*

SQL:

```
SELECT *
FROM building a
WHERE EXISTS(
SELECT *
FROM contain b
WHERE (b.geoID2 = a.buildingID) AND EXISTS(
SELECT *
FROM area c
WHERE (c.areaID = b.geoID1) AND EXISTS(
SELECT *
FROM contain d
WHERE (d.geoID1 = c.areaID) AND EXISTS(
SELECT *
FROM lake e
WHERE (e.lakeID = d.geoID2)))))
```

DATALOG$^{\neg}$:

error $\leftarrow$ building(A) $\wedge$ f1(A)
f1(A) $\leftarrow$ containedin(A,B) $\wedge$ area(B) $\wedge$ f2(B)
f2(B) $\leftarrow$ contain(B,C) $\wedge$ lake(C)

Testcase 15

Longer path on lefthandsside and righthandsside

HLCL `all building type industrial must beusedby company`

Pred. Logic $\forall A$ *(building(A)* $\wedge$ *type(A,type(A,industrial)* $\rightarrow$ $\exists B$ *(beusedby(A,B)* $\wedge$ *company(B)))*

Ext. Dat $\forall A$ *(building(A)* $\wedge$ *type(A,type(A,industrial)* $\wedge$ $\neg$*(*$\exists B$ *(beusedby(A,B)* $\wedge$ *company(B))))*

SQL:

```
SELECT *
FROM building a
WHERE (a.type) = 'industrial' AND NOT (EXISTS(
SELECT *
FROM relBuildingCompany b
WHERE (b.buildingID = a.buildingID) AND EXISTS(
SELECT *
FROM company c
WHERE (c.companyID = b.companyID))))
```

DATALOG$^{\neg}$:
error ← building(A) ∧ type(A,industrial) ∧ ¬f1(A)
f1(A) ← beusedby(A,B) ∧ company(B)

Testcase 16

Longer path expression using multiple attributes as key

HLCL `no building may containedin area have house`

Pred. Logic ¬*(∃A (building(A) ∧ ∃B (containedin(A,B) ∧ area(B) ∧ ∃C (have(B,C) ∧ house(C)))))*

Ext. Dat ∀*A (building(A) ∧ ∃B (containedin(A,B) ∧ area(B) ∧ ∃C (have(B,C) ∧ house(C))))*

SQL:

```
SELECT *
FROM building a
WHERE EXISTS(
SELECT *
FROM contain b
WHERE (b.geoID2 = a.buildingID) AND EXISTS(
SELECT *
FROM area c
WHERE (c.areaID = b.geoID1) AND EXISTS(
SELECT *
FROM have d
WHERE (d.areaID = c.areaID) AND EXISTS(
SELECT *
FROM house e
WHERE (e.houseID = d.houseID) AND (e.owner = d.owner)))))
```

DATALOG$^{\neg}$:
error ← building(A) ∧ f1(A)
f1(A) ← containedin(A,B) ∧ area(B) ∧ f2(B)
f2(B) ← have(B,C) ∧ house(C)

Testcase 17

Using the or operator

HLCL `all area must contain building or contain lake`

Pred. Logic $\forall A$ *(area(A)* $\rightarrow \exists B$ *(contain(A,B)* $\wedge$ *building(B))* $\vee \exists B$ *(contain(A,B)* $\wedge$ *lake(B)))*

Ext. Dat $\forall A$ *(area(A)* $\wedge \neg(\exists B$ *(contain(A,B)* $\wedge$ *building(B))* $\vee \exists B$ *(contain(A,B)* $\wedge$ *lake(B))))*

SQL:

```
SELECT *
FROM area a
WHERE NOT (EXISTS(
SELECT *
FROM contain b
WHERE (b.geoID1 = a.areaID) AND EXISTS(
SELECT *
FROM building c
WHERE (c.buildingID = b.geoID2))) OR EXISTS(
SELECT *
FROM contain b
WHERE (b.geoID1 = a.areaID) AND EXISTS(
SELECT *
FROM lake c
WHERE (c.lakeID = b.geoID2))))
```

DATALOG$^{\neg}$:

error $\leftarrow$ area(A) $\wedge \neg$f1(A) $\wedge \neg$f2(A)
f1(A) $\leftarrow$ contain(A,B) $\wedge$ building(B)
f2(A) $\leftarrow$ contain(A,B) $\wedge$ lake(B)

Testcase 18

Using the and operator

HLCL `all area must contain building and contain lake`

Pred. Logic $\forall A$ *(area(A)* $\rightarrow \exists B$ *(contain(A,B)* $\wedge$ *building(B))* $\wedge \exists B$ *(contain(A,B)* $\wedge$ *lake(B)))*

Ext. Dat $\forall A$ *(area(A)* $\wedge \neg(\exists B$ *(contain(A,B)* $\wedge$ *building(B))* $\wedge \exists B$ *(contain(A,B)* $\wedge$ *lake(B))))*

SQL:

```
SELECT *
FROM area a
WHERE NOT (EXISTS(
SELECT *
FROM contain b
WHERE (b.geoID1 = a.areaID) AND EXISTS(
SELECT *
FROM building c
WHERE (c.buildingID = b.geoID2))) AND EXISTS(
SELECT *
FROM contain b
WHERE (b.geoID1 = a.areaID) AND EXISTS(
SELECT *
FROM lake c
WHERE (c.lakeID = b.geoID2))))
```

DATALOG$^{\neg}$:

error $\leftarrow$ area(A) $\wedge$ $\neg$f1(A)
error $\leftarrow$ area(A) $\wedge$ $\neg$f2(A)
f1(A) $\leftarrow$ contain(A,B) $\wedge$ building(B)
f2(A) $\leftarrow$ contain(A,B) $\wedge$ lake(B)

Testcase 19

Using the ornot operator

HLCL `all area must contain building ornot contain lake`

Pred. Logic $\forall A$ *(area(A)* $\rightarrow$ $\exists B$ *(contain(A,B)* $\wedge$ *building(B))* $\vee$ $\neg$*(*$\exists B$ *(contain(A,B)* $\wedge$ *lake(B))))*

Ext. Dat $\forall A$ *(area(A)* $\wedge$ $\neg$*(*$\exists B$ *(contain(A,B)* $\wedge$ *building(B))* $\vee$ $\neg$*(*$\exists B$ *(contain(A,B)* $\wedge$ *lake(B)))))*

SQL:

```
SELECT *
FROM area a
WHERE NOT (EXISTS(
SELECT *
```

```
FROM contain b
WHERE (b.geoID1 = a.areaID) AND EXISTS(
SELECT *
FROM building c
WHERE (c.buildingID = b.geoID2))) OR NOT (EXISTS(
SELECT *
FROM contain b
WHERE (b.geoID1 = a.areaID) AND EXISTS(
SELECT *
FROM lake c
WHERE (c.lakeID = b.geoID2)))))
```

DATALOG$^{\neg}$:

error $\leftarrow$ area(A) $\wedge$ $\neg$f1(A) $\wedge$ f2(A)
f1(A) $\leftarrow$ contain(A,B) $\wedge$ building(B)
f2(A) $\leftarrow$ contain(A,B) $\wedge$ lake(B)

Testcase 20

Using the andnot operator

HLCL `all area must contain building andnot contain lake`

Pred. Logic $\forall A$ *(area(A)* $\rightarrow$ $\exists B$ *(contain(A,B)* $\wedge$ *building(B))* $\wedge$ $\neg$*(*$\exists B$ *(contain(A,B)* $\wedge$ *lake(B))))*

Ext. Dat $\forall A$ *(area(A)* $\wedge$ $\neg$*(*$\exists B$ *(contain(A,B)* $\wedge$ *building(B))* $\wedge$ $\neg$*(*$\exists B$ *(contain(A,B)* $\wedge$ *lake(B)))))*

SQL:

```
SELECT *
FROM area a
WHERE NOT (EXISTS(
SELECT *
FROM contain b
WHERE (b.geoID1 = a.areaID) AND EXISTS(
SELECT *
FROM building c
WHERE (c.buildingID = b.geoID2))) AND NOT (EXISTS(
SELECT *
FROM contain b
WHERE (b.geoID1 = a.areaID) AND EXISTS(
```

```
SELECT *
FROM lake c
WHERE (c.lakeID = b.geoID2)))))
```

DATALOG$^{\neg}$:
error $\leftarrow$ area(A) $\wedge$ $\neg$f1(A)
error $\leftarrow$ area(A) $\wedge$ f2(A)
f1(A) $\leftarrow$ contain(A,B) $\wedge$ building(B)
f2(A) $\leftarrow$ contain(A,B) $\wedge$ lake(B)

Testcase 21

Using the or operator and a path

HLCL `all area must contain building touch building or contain lake`

Pred. Logic $\forall A$ *(area(A)* $\rightarrow$ $\exists B$ *(contain(A,B)* $\wedge$ *building(B)* $\wedge$ $\exists C$ *(touch(B,C)* $\wedge$ *building(C)))* $\vee$ $\exists B$ *(contain(A,B)* $\wedge$ *lake(B)))*

Ext. Dat $\forall A$ *(area(A)* $\wedge$ $\neg$*(*$\exists B$ *(contain(A,B)* $\wedge$ *building(B)* $\wedge$ $\exists C$ *(touch(B,C)* $\wedge$ *building(C)))* $\vee$ $\exists B$ *(contain(A,B)* $\wedge$ *lake(B))))*

SQL:

```
SELECT *
FROM area a
WHERE NOT (EXISTS(
SELECT *
FROM contain b
WHERE (b.geoID1 = a.areaID) AND EXISTS(
SELECT *
FROM building c
WHERE (c.buildingID = b.geoID2) AND EXISTS(
SELECT *
FROM contain d
WHERE (d.geoID1 = c.buildingID) AND EXISTS(
SELECT *
FROM building e
WHERE (e.buildingID = d.geoID2))))) OR EXISTS(
SELECT *
FROM contain b
WHERE (b.geoID1 = a.areaID) AND EXISTS(
SELECT *
```

```
FROM lake c
WHERE (c.lakeID = b.geoID2))))
```

DATALOG$^{\neg}$:

error ← area(A) ∧ ¬f1(A) ∧ ¬f3(A)
f1(A) ← contain(A,B) ∧ building(B) ∧ f2(B)
f2(B) ← touch(B,C) ∧ building(C)
f3(A) ← contain(A,B) ∧ lake(B)

Testcase 22

Using the all keyword

HLCL `no area may contain all building`

Pred. Logic $\neg(\exists A\ (area(A) \wedge \forall B\ (building(B) \rightarrow contain(A,B))))$

Ext. Dat $\forall A\ (area(A) \wedge \neg(\exists B\ (building(B) \wedge \neg(contain(A,B)))))$

SQL:

```
SELECT *
FROM area a
WHERE NOT (EXISTS(
SELECT *
FROM building b
WHERE NOT (EXISTS(
SELECT *
FROM contain c
WHERE (c.geoID1 = a.areaID) AND (c.geoID2 = b.buildingID)))))
```

DATALOG$^{\neg}$:

error ← area(A) ∧ ¬f1(A)
f1(A) ← building(B) ∧ ¬contain(A,B)

Testcase 23

Using the all keyword and a path

HLCL `no area may contain all building touch building`

Pred. Logic $\neg(\exists A\ (area(A) \wedge \forall B\ (building(B) \wedge \exists C\ (touch(B,C) \wedge building(C)) \rightarrow contain(A,B))))$

Ext. Dat $\forall A$ *(area(A)* $\wedge \neg(\exists B$ *(building(B)* $\wedge \exists C$ *(touch(B,C)* $\wedge$ *building(C))* $\wedge \neg$*(contain(A,B)))))*

SQL:

```
SELECT *
FROM area a
WHERE NOT (EXISTS(
SELECT *
FROM building b
WHERE EXISTS(
SELECT *
FROM contain c
WHERE (c.geoID1 = b.buildingID) AND EXISTS(
SELECT *
FROM building d
WHERE (d.buildingID = c.geoID2) AND NOT (EXISTS(
SELECT *
FROM contain e
WHERE (e.geoID1 = a.areaID) AND (e.geoID2 = b.buildingID)))))))
```

DATALOG$^\neg$:

error $\leftarrow$ area(A) $\wedge \neg$f1(A)
f1(A) $\leftarrow$ building(B) $\wedge$ f2(B) $\wedge \neg$contain(A,B)
f2(B) $\leftarrow$ touch(B,C) $\wedge$ building(C)

Testcase 24

Using the solely keyword

HLCL `all area must contain solely building`

Pred. Logic $\forall A$ *(area(A)* $\rightarrow \forall B$ *(contain(A,B)* $\rightarrow$ *building(B)))*

Ext. Dat $\forall A$ *(area(A)* $\wedge \exists B$ *(contain(A,B)* $\wedge \neg$*(building(B))))*

SQL:

```
SELECT *
FROM area a
WHERE EXISTS(
SELECT *
```

```
FROM contain b
WHERE (b.geoID1 = a.areaID) AND NOT (EXISTS(
SELECT *
FROM building c
WHERE (c.buildingID = b.geoID2))))
```

DATALOG$^{\neg}$:
error ← area(A) ∧ f1(A)
f1(A) ← contain(A,B) ∧ ¬building(B)

Testcase 25

Using the solely keyword and a path

HLCL `all industrialArea must contain solely building type industrial`

Pred. Logic $\forall A$ *(industrialArea(A)* $\rightarrow \forall B$ *(contain(A,B)* $\rightarrow$ *building(B)* $\wedge$ *type(B,type(B,industrial)))*

Ext. Dat $\forall A$ *(industrialArea(A)* $\wedge \exists B$ *(contain(A,B)* $\wedge \neg$*(building(B)* $\wedge$ *type(B,type(B,industrial))))*

SQL:

```
SELECT *
FROM industrialArea a
WHERE EXISTS(
SELECT *
FROM contain b
WHERE (b.geoID1 = a.areaID) AND NOT (EXISTS(
SELECT *
FROM building c
WHERE (c.buildingID = b.geoID2) AND (c.type) = 'industrial')))
```

DATALOG$^{\neg}$:
error ← industrialArea(A) ∧ f1(A)
error ← industrialArea(A) ∧ f2(A)
f1(A) ← contain(A,B) ∧ ¬building(B)
f2(A) ← contain(A,B) ∧ ¬type(B,industrial)

Testcase 26

Using the solely keyword and an operator

HLCL `all building must type blockbuilding and touch solely building`

Pred. Logic $\forall A$ *(building(A)* $\rightarrow$ *type(A,type(A,blockbuilding)* $\wedge \forall B$ *(touch(A,B)* $\rightarrow$ *building(B)))*

Ext. Dat $\forall A$ *(building(A)* $\wedge \neg$*(type(A,type(A,blockbuilding)* $\wedge \neg$*(*$\exists B$ *(touch(A,B)* $\wedge \neg$*(building(B))))))*

SQL:

```
SELECT *
FROM building a
WHERE NOT ((a.type) = 'blockbuilding' AND NOT (EXISTS(
SELECT *
FROM contain b
WHERE (b.geoID1 = a.buildingID) AND NOT (EXISTS(
SELECT *
FROM building c
WHERE (c.buildingID = b.geoID2))))))
```

DATALOG$^{\neg}$:

error $\leftarrow$ building(A) $\wedge \neg$type(A,blockbuilding)
error $\leftarrow$ building(A) $\wedge$ f1(A)
f1(A) $\leftarrow$ touch(A,B) $\wedge \neg$building(B)

Testcase 27

Using variable

HLCL `all building X must touch building X`

Pred. Logic $\forall X$ *(building(X)* $\rightarrow$ *touch(X,X))*

Ext. Dat $\forall X$ *(building(X)* $\wedge \neg$*(touch(X,X)))*

SQL:

```
SELECT *
FROM building a
WHERE NOT (EXISTS(
SELECT *
FROM contain b
WHERE (b.geoID1 = a.buildingID) AND (b.geoID2 = a.buildingID)))
```

DATALOG$^{\neg}$:
error $\leftarrow$ building(X) $\wedge$ $\neg$touch(X,X)

Testcase 28

Using variables and paths

HLCL `all area contain building X must contain building X`

Pred. Logic $\forall X\ (\forall A\ (area(A) \wedge building(X) \wedge contain(A,X) \rightarrow contain(A,X)))$

Ext. Dat $\forall X\ (\forall A\ (area(A) \wedge building(X) \wedge contain(A,X) \wedge \neg(contain(A,X))))$

SQL:

```
SELECT *
FROM area a
WHERE EXISTS(
SELECT *
FROM building b
WHERE EXISTS(
SELECT *
FROM contain c
WHERE (c.geoID1 = a.areaID) AND (c.geoID2 = b.buildingID) AND NOT (EXISTS(
SELECT *
FROM contain d
WHERE (d.geoID1 = a.areaID) AND (d.geoID2 = b.buildingID)))))
```

DATALOG$^{\neg}$:
error $\leftarrow$ area(A) $\wedge$ building(X) $\wedge$ contain(A,X) $\wedge$ $\neg$contain(A,X)

Testcase 29

Using variable for attributes

HLCL `all building type X must touch building type X`

Pred. Logic $\forall X\ (\forall A\ (building(A) \wedge type(A,X) \rightarrow \exists B\ (touch(A,B) \wedge building(B) \wedge type(B,X))))$

Ext. Dat $\forall X\ (\forall A\ (building(A) \wedge type(A,X) \wedge \neg(\exists B\ (touch(A,B) \wedge building(B) \wedge type(B,X)))))$

SQL:

```
SELECT *
FROM building a
WHERE NOT (EXISTS(
SELECT *
FROM contain b
WHERE (b.geoID1 = a.buildingID) AND EXISTS(
SELECT *
FROM building c
WHERE (c.buildingID = b.geoID2) AND (c.type) = a.type)))
```

DATALOG$^{\neg}$:

error $\leftarrow$ building(A) $\wedge$ type(A,X) $\wedge \neg$f1(X,A)
f1(X,A) $\leftarrow$ touch(A,B) $\wedge$ building(B) $\wedge$ type(B,X)

Testcase 30

Using variable and long pahts

HLCL `all area intersectedby road X must contain building intersectedby road X`

Pred. Logic $\forall X\ (\forall A\ (area(A) \wedge road(X) \wedge intersectedby(A,X) \rightarrow \exists B\ (contain(A,B) \wedge building(B) \wedge intersectedby(B,X))))$

Ext. Dat $\forall X\ (\forall A\ (area(A) \wedge road(X) \wedge intersectedby(A,X) \wedge \neg(\exists B\ (contain(A,B) \wedge building(B) \wedge intersectedby(B,X)))))$

SQL:

```
SELECT *
FROM area a
WHERE EXISTS(
SELECT *
```

```
FROM road b
WHERE EXISTS(
SELECT *
FROM intersect c
WHERE (c.geoID2 = a.areaID) AND (c.geoID1 = b.roadsegmentID)
AND NOT (EXISTS(
SELECT *
FROM contain d
WHERE (d.geoID1 = a.areaID) AND EXISTS(
SELECT *
FROM building e
WHERE (e.buildingID = d.geoID2) AND EXISTS(
SELECT *
FROM intersect f
WHERE (f.geoID2 = e.buildingID) AND
(f.geoID1 = b.roadsegmentID)))))))
```

DATALOG$^{\neg}$:

error $\leftarrow$ area(A) $\wedge$ road(X) $\wedge$ intersectedby(A,X) $\wedge$ $\neg$f1(A)
f1(A) $\leftarrow$ contain(A,B) $\wedge$ building(B) $\wedge$ intersectedby(B,X)

Testcase 31

Referencing to the same variable more than twice

HLCL `all building X must touch building X and touch building X`

Pred. Logic $\forall X$ *(building(X)* $\rightarrow$ *touch(X,X)* $\wedge$ *touch(X,X))*

Ext. Dat $\forall X$ *(building(X)* $\wedge$ $\neg$*(touch(X,X)* $\wedge$ *touch(X,X)))*

SQL:

```
SELECT *
FROM building a
WHERE NOT (EXISTS(
SELECT *
FROM contain b
WHERE (b.geoID1 = a.buildingID) AND
(b.geoID2 = a.buildingID) AND EXISTS(
SELECT *
FROM contain c
WHERE (c.geoID1 = a.buildingID) AND
(c.geoID2 = a.buildingID))))
```

DATALOG$^{\neg}$:
error ← building(X) ∧ ¬touch(X,X)
error ← building(X) ∧ ¬touch(X,X)

Testcase 32

Using variable and userdefined predicates

HLCL `all building X touch building Y must zdifferencelargerthan5 X Y`

Pred. Logic $\forall Y\ (\forall X\ (building(X) \wedge building(Y) \wedge touch(X,Y) \rightarrow zdifferencelargerthan5(X,Y)))$

Ext. Dat $\forall Y\ (\forall X\ (building(X) \wedge building(Y) \wedge touch(X,Y) \wedge \neg(zdifferencelargerthan5(X,Y))))$

SQL:

```
SELECT *
FROM building a
WHERE EXISTS(
SELECT *
FROM building b
WHERE (b.buildingID != a.buildingID) AND EXISTS(
SELECT *
FROM contain c
WHERE (c.geoID1 = a.buildingID) AND (c.geoID2 = b.buildingID)
AND NOT ('zdiff'(a.buildingID,b.buildingID))))
```

DATALOG$^{\neg}$:
error ← building(X) ∧ building(Y) ∧ touch(X,Y) ∧ ¬zdifferencelargerthan5(X,Y)

Testcase 33

Using variable, predicates and paths

HLCL `all building X type Z and touch building Y type Z must zdifferencelargerthan5 X Y`

Pred. Logic $\forall Z\ (\forall Y\ (\forall X\ (building(X) \wedge building(Y) \wedge type(X,Z) \wedge touch(X,Y) \wedge type(Y,Z) \rightarrow zdifferencelargerthan5(X,Y))))$

Ext. Dat $\forall Z\ (\forall Y\ (\forall X\ (building(X) \wedge building(Y) \wedge type(X,Z) \wedge touch(X,Y) \wedge type(Y,Z) \wedge \neg(zdifferencelargerthan5(X,Y)))))$

SQL:

```
SELECT *
FROM building a
WHERE EXISTS(
SELECT *
FROM building b
WHERE (b.buildingID != a.buildingID) AND EXISTS(
SELECT *
FROM contain c
WHERE (c.geoID1 = a.buildingID) AND (c.geoID2 = b.buildingID)
AND (b.type) = a.type AND NOT ('zdiff'(a.buildingID,b.buildingID))))
```

DATALOG$^\neg$:

error $\leftarrow$ building(X) $\wedge$ building(Y) $\wedge$ type(X,Z) $\wedge$ touch(X,Y) $\wedge$ type(Y,Z) $\wedge$ $\neg$zdifferencelargerthan5(X,Y)

Testcase 34

Using nummerical quantifiers

HLCL `all residentialArea must contain at most 5 building`

Pred. Logic $\forall A\ (residentialArea(A) \rightarrow \exists_{le5}\ B\ (contain(A,B) \wedge building(B)))$

Ext. Dat $\forall A\ (residentialArea(A) \wedge \neg(\exists_{le5}\ B\ (contain(A,B) \wedge building(B))))$

SQL:

```
SELECT *
FROM residentialArea a
WHERE NOT ((EXISTS(
SELECT COUNT(*)
FROM contain b
WHERE (b.geoID1 = a.areaID) AND EXISTS(
SELECT *
FROM building c
WHERE (c.buildingID = b.geoID2)))) <= 5)
```

DATALOG$^\neg$:

N/A - Cannot express nummerical quantification in datalog

Testcase 35

Using nummerical quantifiers

HLCL `all residentialArea must contain at least 37 building`

Pred. Logic $\forall A$ *(residentialArea(A)* $\rightarrow$ $\exists_{ge37}$ *B (contain(A,B)* $\wedge$ *building(B)))*

Ext. Dat $\forall A$ *(residentialArea(A)* $\wedge$ $\neg(\exists_{ge37}$ *B (contain(A,B)* $\wedge$ *building(B))))*

SQL:

```
SELECT *
FROM residentialArea a
WHERE NOT ((EXISTS(
SELECT COUNT(*)
FROM contain b
WHERE (b.geoID1 = a.areaID) AND EXISTS(
SELECT *
FROM building c
WHERE (c.buildingID = b.geoID2)))) => 37)
```

DATALOG$^{\neg}$:
N/A - Cannot express nummerical quantification in datalog

Testcase 36

Using nummerical quantifiers and paths

HLCL `all residentialArea must contain excactly 1 building type industrial`

Failed

Testcase 37

Using nummerical quantifiers for attributes

HLCL `all residentialArea must zipcode at most 2000`

Pred. Logic $\forall A$ *(residentialArea(A)* $\rightarrow$ *zipcode(A,zipcode(A,2000))*

Ext. Dat $\forall A$ *(residentialArea(A)* $\wedge$ $\neg$*(zipcode(A,zipcode(A,2000)))*

SQL:

```
SELECT *
FROM residentialArea a
WHERE NOT (SELECT *
FROM area b
WHERE (b.areaID = a.areaID) AND (a.zipcode) <= 2000)
```

DATALOG$^{\neg}$:
N/A - Cannot express nummerical quantification in datalog

Testcase 38

Using a child in an ISA-structures

HLCL `all residentialArea must zipcode 3520`

Pred. Logic $\forall A$ *(residentialArea(A)* $\rightarrow$ *zipcode(A,zipcode(A,3520))*

Ext. Dat $\forall A$ *(residentialArea(A)* $\wedge \neg$*(zipcode(A,zipcode(A,3520)))*

SQL:

```
SELECT *
FROM residentialArea a
WHERE NOT (SELECT *
FROM area b
WHERE (b.areaID = a.areaID) AND (a.zipcode) = 3520)
```

DATALOG$^{\neg}$:
error $\leftarrow$ residentialArea(A) $\wedge \neg$zipcode(A,3520)

Testcase 39

Setting an attribute value variable for a child ISA-structures, where the attribute actually in parent

HLCL `all residentialArea zipcode X must touch residentialArea zipcode X`

Pred. Logic $\forall X$ *(*$\forall A$ *(residentialArea(A)* $\wedge$ *zipcode(A,X)* $\rightarrow$ $\exists B$ *(touch(A,B)* $\wedge$ *residentialArea(B)* $\wedge$ *zipcode(B,X))))*

Ext. Dat $\forall X\ (\forall A\ (residentialArea(A) \wedge zipcode(A,X) \wedge \neg(\exists B\ (touch(A,B) \wedge residentialArea(B) \wedge zipcode(B,X)))))$

SQL:

```
SELECT *
FROM residentialArea a
WHERE SELECT *
FROM area b
WHERE (b.areaID = a.areaID) AND NOT (EXISTS(
SELECT *
FROM contain c
WHERE (c.geoID1 = a.areaID) AND EXISTS(
SELECT *
FROM residentialArea d
WHERE (d.areaID = c.geoID2) AND SELECT *
FROM area e
WHERE (e.areaID = d.areaID) AND (e.zipcode) = b.zipcode)))
```

DATALOG$^\neg$:

error ← residentialArea(A) ∧ zipcode(A,X) ∧ ¬f1(X,A)
f1(X,A) ← touch(A,B) ∧ residentialArea(B) ∧ zipcode(B,X)

Testcase 40

Setting an attribute value variable for a child ISA-structures, where the attribute actually in parent

HLCL `all residentialArea zipcode X must touch area zipcode X`

Pred. Logic $\forall X\ (\forall A\ (residentialArea(A) \wedge zipcode(A,X) \rightarrow \exists B\ (touch(A,B) \wedge area(B) \wedge zipcode(B,X))))$

Ext. Dat $\forall X\ (\forall A\ (residentialArea(A) \wedge zipcode(A,X) \wedge \neg(\exists B\ (touch(A,B) \wedge area(B) \wedge zipcode(B,X)))))$

SQL:

```
SELECT *
FROM residentialArea a
WHERE SELECT *
FROM area b
```

```
WHERE (b.areaID = a.areaID) AND NOT (EXISTS(
SELECT *
FROM contain c
WHERE (c.geoID1 = a.areaID) AND EXISTS(
SELECT *
FROM area d
WHERE (d.areaID = c.geoID2) AND (d.zipcode) = b.zipcode)))
```

DATALOG$^\neg$:
error $\leftarrow$ residentialArea(A) $\wedge$ zipcode(A,X) $\wedge$ $\neg$f1(X,A)
f1(X,A) $\leftarrow$ touch(A,B) $\wedge$ area(B) $\wedge$ zipcode(B,X)

Testcase 41

Setting an attribute value variable for a child ISA-structures, where the attribute actually in parent

HLCL `all area zipcode X must touch residentialArea zipcode X`

Pred. Logic $\forall X\ (\forall A\ (area(A) \wedge zipcode(A,X) \rightarrow \exists B\ (touch(A,B) \wedge residentialArea(B) \wedge zipcode(B,X))))$

Ext. Dat $\forall X\ (\forall A\ (area(A) \wedge zipcode(A,X) \wedge \neg(\exists B\ (touch(A,B) \wedge residentialArea(B) \wedge zipcode(B,X)))))$

SQL:

```
SELECT *
FROM area a
WHERE NOT (EXISTS(
SELECT *
FROM contain b
WHERE (b.geoID1 = a.areaID) AND EXISTS(
SELECT *
FROM residentialArea c
WHERE (c.areaID = b.geoID2) AND SELECT *
FROM area d
WHERE (d.areaID = c.areaID) AND (d.zipcode) = a.zipcode)))
```

DATALOG$^\neg$:
error $\leftarrow$ area(A) $\wedge$ zipcode(A,X) $\wedge$ $\neg$f1(X,A)
f1(X,A) $\leftarrow$ touch(A,B) $\wedge$ residentialArea(B) $\wedge$ zipcode(B,X)

Testcase 42

Setting an attribute value for a child ISA-structures, where the attribute actually in parent and using a userdefined predicate

HLCL `all residentialArea X must zipcodefunc X`

Pred. Logic $\forall X$ *(residentialArea(X)* $\rightarrow$ *zipcodefunc(X))*

Ext. Dat $\forall X$ *(residentialArea(X)* $\wedge \neg$*(zipcodefunc(X)))*

SQL:

```
SELECT *
FROM residentialArea a
WHERE NOT (SELECT *
FROM area b
WHERE (a.areaID = b.areaID) AND 'zipcodefunc'(b.zipcode))
```

DATALOG$^{\neg}$:

error $\leftarrow$ residentialArea(X) $\wedge \neg$zipcodefunc(X)

Testcase 43

Setting an attribute value for a child ISA-structures, where the attribute actually in parent and using a userdefined predicate

HLCL `all residentialArea X touch area Y must binfunc X Y`

Pred. Logic $\forall Y$ *(*$\forall X$ *(residentialArea(X)* $\wedge$ *area(Y)* $\wedge$ *touch(X,Y)* $\rightarrow$ *binfunc(X,Y)))*

Ext. Dat $\forall Y$ *(*$\forall X$ *(residentialArea(X)* $\wedge$ *area(Y)* $\wedge$ *touch(X,Y)* $\wedge \neg$*(binfunc(X,Y))))*

SQL:

```
SELECT *
FROM residentialArea a
WHERE EXISTS(
SELECT *
FROM area b
WHERE EXISTS(
SELECT *
```

```
FROM contain c
WHERE (c.geoID1 = a.areaID) AND
(c.geoID2 = b.areaID) AND NOT (SELECT *
FROM area d
WHERE (d.areaID = a.areaID) AND
'binfunc'(d.zipcode,b.areaID))))
```

DATALOG$^{\neg}$:

error $\leftarrow$ residentialArea(X) $\wedge$ area(Y) $\wedge$ touch(X,Y) $\wedge \neg$binfunc(X,Y)

Testcase 44

Use of optional brackets

HLCL `all area must ( contain building )`

Pred. Logic $\forall A$ *(area(A)* $\rightarrow \exists B$ *(contain(A,B)* $\wedge$ *building(B)))*

Ext. Dat $\forall A$ *(area(A)* $\wedge \neg(\exists B$ *(contain(A,B)* $\wedge$ *building(B))))*

SQL:

```
SELECT *
FROM area a
WHERE NOT (EXISTS(
SELECT *
FROM contain b
WHERE (b.geoID1 = a.areaID) AND EXISTS(
SELECT *
FROM building c
WHERE (c.buildingID = b.geoID2))))
```

DATALOG$^{\neg}$:

error $\leftarrow$ area(A) $\wedge \neg$f1(A)

f1(A) $\leftarrow$ contain(A,B) $\wedge$ building(B)

Testcase 45

Use of optional brackets

HLCL `all area must ( contain building type industrial )`

Pred. Logic $\forall A$ *(area(A)* $\rightarrow \exists B$ *(contain(A,B)* $\wedge$ *building(B)* $\wedge$ *type(B,type(B,industrial)))*

Ext. Dat $\forall A$ *(area(A)* $\wedge \neg($*(*$\exists B$ *(contain(A,B)* $\wedge$ *building(B)* $\wedge$ *type(B,type(B,industrial))))*

SQL:

```
SELECT *
FROM area a
WHERE NOT (EXISTS(
SELECT *
FROM contain b
WHERE (b.geoID1 = a.areaID) AND EXISTS(
SELECT *
FROM building c
WHERE (c.buildingID = b.geoID2) AND (c.type) = 'industrial')))
```

DATALOG$^{\neg}$:

error $\leftarrow$ area(A) $\wedge \neg$f1(A)

f1(A) $\leftarrow$ contain(A,B) $\wedge$ building(B) $\wedge$ type(B,industrial)

Testcase 46

Testing usage of class definitions

HLCL `all definedbuilding must touch definedbuilding`

Pred. Logic $\forall A$ *(building(A)* $\wedge$ *type(A,type(A,residential)* $\rightarrow \exists B$ *(touch(A,B)* $\wedge$ *building(B)* $\wedge$ *type(B,type(B,residential)))*

Ext. Dat $\forall A$ *(building(A)* $\wedge$ *type(A,type(A,residential)* $\wedge \neg($*(*$\exists B$ *(touch(A,B)* $\wedge$ *building(B)* $\wedge$ *type(B,type(B,residential))))*

SQL:

```
SELECT *
FROM building a
WHERE (a.type) = 'residential' AND NOT (EXISTS(
SELECT *
FROM contain b
WHERE (b.geoID1 = a.buildingID) AND EXISTS(
SELECT *
FROM building c
WHERE (c.buildingID = b.geoID2) AND (c.type) = 'residential')))
```

DATALOG$^{\neg}$:
error ← building(A) ∧ type(A,residential) ∧ ¬f1(A)
f1(A) ← touch(A,B) ∧ building(B) ∧ type(B,residential)

Testcase 47

Testing usage of recursive class definitions

HLCL `no definedarea must intersectedby road`

Pred. Logic ¬*(∃A (area(A) ∧ ∃B (contain(A,B) ∧ building(B) ∧ type(B,type(B,residential)) ∧ ∃B (intersectedby(A,B) ∧ road(B))))*

Ext. Dat ∀*A (area(A) ∧ ∃B (contain(A,B) ∧ building(B) ∧ type(B,type(B,residential)) ∧ ∃B (intersectedby(A,B) ∧ road(B)))*

SQL:

```
SELECT *
FROM area a
WHERE EXISTS(
SELECT *
FROM contain b
WHERE (b.geoID1 = a.areaID) AND EXISTS(
SELECT *
FROM building c
WHERE (c.buildingID = b.geoID2) AND (c.type) = 'residential' AND EXISTS(
SELECT *
FROM intersect d
WHERE (d.geoID2 = a.areaID) AND EXISTS(
SELECT *
FROM road e
WHERE (e.roadsegmentID = c.buildingID)))))
```

DATALOG$^{\neg}$:
error ← area(A) ∧ f1(A) ∧ f2(A)
f1(A) ← contain(A,B) ∧ building(B) ∧ type(B,residential)
f2(A) ← intersectedby(A,B) ∧ road(B)

Testcase 48

An empty expression. Should fail!

HLCL

Failed

Testcase 49

Wrong use of optional brackets. Should fail!

HLCL `all ( area must ( contain building type industrial )`

Failed

Testcase 50

Use of operators on righthansside. Should fail!

HLCL `all area and building must contain building`

Failed

Testcase 51

Use of only one userdefined variable. Should fail!

HLCL `all area X must contain building`

Failed

Appendix E

Sourcecode

E.1 HLCL-system

/**

Title : *A Highlevel* **interface to** *GIS*
Author : *Mads Johnsen*
Last Modified : 29/03/2005

Build : 0067 / *Final Build*
Notes : *This* **.pl file contains the actual** $HLCL-$ **system**

**/

Imports

←[**usersettings**]. %Conceptual Model, Variables and Class Definitions
←[**helper**]. %Functions allowing for WFF Checking of the above

---

hlcl_to_sqldat(+*HLCLExp*, −*SQLExp*, −*DatExp*)

Takes an HLCL Expression and returns the SQL and Datalog equivalent of that query

hlcl_to_sqldatalog(*HLCL*, *SQL*, *DAT*) ←
  hlcl_to_pred(*HLCL*, *Pred*),
  check_wellformedness(*Pred*),
  perform_intermediate_steps(*Pred*, *IntDat*),
  subpred_to_sql(*IntDat*, *SQL*, [], [], _, [], **nocount**),
  pred_to_datalog(*IntDat*, *DAT*),
  nl, **print_sql**(*SQL*), **nl**, **nl**, **print_dat_list**(*DAT*).

/**

From HLCL → Predicate Logic

**/

---

hlcl_to_pred(+*HLCLExp*, −*PredExp*)

Converts an HLCL expression in a list form to a Predicate expression

hlcl_to_pred(*HLCL*, *Pred*) ←
  apply_macro_functionality(*HLCL*, *HLCLDef*),
  hlcl(*Intermediate*, *HLCLDef*, []),
  fix_scoping(*Intermediate*, *Pred*).

---

apply_macro_functionality(+*HLCLExp*, −*HLCLExp*)

Scans the HLCL Expression for definitions and substitutes them by their class definition.

apply_macro_functionality([ ], [ ]).
apply_macro_functionality([*Head*|*Tail*], *Res*) ←
  definition([*Head*], *Exp*),
  apply_macro_functionality(*Exp*, *Mexp*),
  append(*Mexp*, *Rest*, *Res*),
  apply_macro_functionality(*Tail*, *Rest*), **!**.

apply_macro_functionality([*Head*|*Tail*], [*Head*|*Rest*]) ←
  apply_macro_functionality(*Tail*, *Rest*).

---

hlcl(−*PredExp*, +*HLCLExp*, +*List*)

Converts an HLCL expression in a list form to a Predicate expression. The third argument is a difference list used by the DCG, and usually initiated as an empty list.

hlcl(*S*) ⟶ **q**(**v**(0), *P1*, *P2*, *S*), **class**(**v**(0), *P1*), **imp**, **sl**(**v**(0), *P2*).

hlcl(*S*) ⟶ **q**(*X*, *P1*, *P2*, *S*), **varclass**(*X*, *P1*), **imp**, **sl**(*X*, *P2*).

q(*X*, *P1*, *P2*, **all**(*X*, **imp**(*P1*, *P2*))) ⟶ [**all**].
q(*X*, *P1*, *P2*, **neg**(**exists**(*X*, **and**(*P1*, *P2*)))) ⟶ [**no**].

sl(*X*, *P*) ⟶ ['('], **sl**(*X*, *P*), [')'].
sl(*X*, *P*) ⟶ **class**(*X*, *P*).
sl(_, *P*) ⟶ **pred**(*P*).
sl(*X*, *P*) ⟶ **rc**(*X*, *P*).
sl(*X*, *P*) ⟶ **rc**(*X*, *P1*), **operator**(*P1*, *P2*, *P*), **sl**(*X*, *P2*).
sl(*X*, *P*) ⟶ **ac**(*X*, *P*).
sl(*X*, *P*) ⟶ **ac**(*X*, *P1*), **operator**(*P1*, *P2*, *P*), **sl**(*X*, *P2*).

pred(**unarypredicate**(*Pred*, *Var*)) ⟶
  [*Pred*], **variable**(*Var*),
  {**sqlfunction**(*Pred*, [[_, _]], _)}.
pred(**binarypredicate**(*Pred*, *Var1*, *Var2*)) ⟶
  [*Pred*], **variable**(*Var1*), **variable**(*Var2*),
  {**sqlfunction**(*Pred*, [[_, _], [_, _]], _)}.

class(*X*, **class**(*Class*, *X*)) ⟶
  [*Class*],
  {**is_class**(*Class*)}.
class(*X*, **and**(**class**(*Class*, *X*), *P*)) ⟶
  [*Class*], **sl**(*X*, *P*),
  {**is_class**(*Class*)}.

```
varclass(Var, varclass(Class, Var)) ⟶
    [Class], variable(Var),
    {is_class(Class)}.
varclass(Var, and(varclass(Class, Var), P)) ⟶
    [Class], variable(Var), sl(Var, P),
    {is_class(Class)}.

ac(X, attribute(Attribute, eq, Var, X)) ⟶
    [Attribute], variable(Var),
    {is_attribute(_, Attribute)}.
ac(X, attribute(Attribute, eq, Value, X)) ⟶
    [Attribute, Value],
    {is_attribute(_, Attribute)}.
ac(X, attribute(Attribute, Comp, Value, X)) ⟶
    [Attribute], numericalquantifier(Comp), [Value],
    {is_attribute(_, Attribute)}.

rc(X, exists(v(X), and(relation(Rel, X, v(X)), Term2))) ⟶
    [Rel], class(v(X), Term2),
    {is_relation(Rel)}.
rc(X, numexists(Q, No, v(X), and(relation(Rel, X, v(X)), Term2))) ⟶
    [Rel], numericalquantifier(Q), integerno(No), class(v(X), Term2),
    {is_relation(Rel)}.
rc(X, and(relation(Rel, X, Var), Term2)) ⟶
    [Rel], varclass(Var, Term2),
    {is_relation(Rel)}.
rc(X, all(v(X), imp(Term2, relation(Rel, X, v(X))))) ⟶
    [Rel, all], class(v(X), Term2),
    {is_relation(Rel)}.
rc(X, all(v(X), imp(relation(Rel, X, v(X)), Term2))) ⟶
    [Rel, solely], class(v(X), Term2),
    {is_relation(Rel)}.

/ * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
      Lexicon
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * /

imp ⟶ [must].
imp ⟶ [may].

operator(P1, P2, and(P1, P2)) ⟶ [and].
operator(P1, P2, or(P1, P2)) ⟶ [or].
operator(P1, P2, and(P1, neg(P2))) ⟶ [andnot].
operator(P1, P2, or(P1, neg(P2))) ⟶ [ornot].
```

numericalquantifier(**eq**) $\longrightarrow$ [**exactly**].
numericalquantifier(**ge**) $\longrightarrow$ [**at**, **least**].
numericalquantifier(**le**) $\longrightarrow$ [**at**, **most**].

integerno(*Int*) $\longrightarrow$ [*Int*], {**integer**(*Int*)}.

variable(*Var*) $\longrightarrow$ [*Var*], {**userDefVariable**(*Var*)}.

---

fix_scoping(+*PredExp*, −*PredExp*)

Fixes the scoping issues that userdefined variables introduces.

fix_scoping(**neg**(**exists**(*Var*, *Exp*)), *Result*) ←
  substract_userdefined_varmaps(*Exp*, *PrevVarMap*),
  remove_from_varmap(_, *Var*, *PrevVarMap*, *VarMap*),
  remove_varclass_components(*Exp*, *Var*, _, *FixedExp*),
  insert_varclass_components_list(*VarMap*, *Var*, *FixedExp*, *BetterFixedExp*),
  add_universal_quantified_variable(**neg**(**exists**(*Var*, *BetterFixedExp*)), *VarMap*, *Result*).

fix_scoping(**all**(*Var*, *Exp*), *Result*) ←
  substract_userdefined_varmaps(*Exp*, *PrevVarMap*),
  remove_from_varmap(_, *Var*, *PrevVarMap*, *VarMap*),
  remove_varclass_components(*Exp*, *Var*, _, *FixedExp*),
  insert_varclass_components_list(*VarMap*, *Var*, *FixedExp*, *BetterFixedExp*),
  add_universal_quantified_variable(**all**(*Var*, *BetterFixedExp*), *VarMap*, *Result*).

---

remove_varclass_components(+*PredExp*, −*PredExp*).

Removes class expressions which are initated with a userdefined variable, e.g. varclass components of the expression

remove_varclass_components(**and**(**varclass**(*Class*, *Var*), *Exp*), *Var*, *Out*,
and(**varclass**(*Class*, *Var*), *Texp*)) ←
  remove_varclass_components(*Exp*, [ ], *Out*, *Texp*), **!**.

remove_varclass_components(**and**(*Exp*, **varclass**(*Class*, *Var*)), *Var*, [ ],
and(*Texp*, **varclass**(*Class*, *Var*))) ←
  remove_varclass_components(*Exp*, *Var*, *Var*, *Texp*), **!**.

remove_varclass_components(**and**(**varclass**(_, _), *Exp*), *Var*, *Out*, *Texp*) ←
  remove_varclass_components(*Exp*, *Var*, *Out*, *Texp*), **!**.

remove_varclass_components(**and**(*Exp*, **varclass**(_, _)), *Var*, *Out*, *Texp*) ←
  remove_varclass_components(*Exp*, *Var*, *Out*, *Texp*), **!**.

remove_varclass_components(**imp**(**varclass**(*Class*, *Var*), *Exp*), *Var*, *Out*,

imp(**varclass**(*Class*, *Var*), *Texp*)) ←
  remove_varclass_components(*Exp*, [ ], *Out*, *Texp*), **!**.

remove_varclass_components(**imp**(*Exp*, **varclass**(*Class*, *Var*)), *Var*, [ ],
imp(*Texp*, **varclass**(*Class*, *Var*))) ←
  remove_varclass_components(*Exp*, *Var*, *Var*, *Texp*), **!**.

remove_varclass_components(**imp**(**varclass**(_, _), *Exp*), *Var*, *Out*, *Texp*) ←
  remove_varclass_components(*Exp*, *Var*, *Out*, *Texp*), **!**.

remove_varclass_components(**imp**(*Exp*, **varclass**(_, _)), *Var*, *Out*, *Texp*) ←
  remove_varclass_components(*Exp*, *Var*, *Out*, *Texp*), **!**.

remove_varclass_components(**class**(*C*, *V*), *Var*, *Var*, **class**(*C*, *V*)).
remove_varclass_components(**varclass**(*C*, *V*), *Var*, *Var*, **varclass**(*C*, *V*)).
remove_varclass_components(**relation**(*C*, *X*, *V*), *Var*, *Var*, **relation**(*C*, *X*, *V*)).
remove_varclass_components(**attribute**(*C*, *Comp*, *Val*, *Var1*), *Var*, *Var*,
attribute(*C*, *Comp*, *Val*, *Var1*)).
remove_varclass_components(**unarypredicate**(*P*, *V*), *Var*, *Var*,
unarypredicate(*P*, *V*)).
remove_varclass_components(**binarypredicate**(*P*, *V1*, *V2*), *Var*, *Var*,
binarypredicate(*P*, *V1*, *V2*)).
remove_varclass_components(**neg**(*Exp*), *Var*, *Var*, **neg**(*Exp*)).
remove_varclass_components(**and**(*Exp1*, *Exp2*), *VarIn*, *VarOut*, **and**(*Texp1*, *Texp2*)) ←
  remove_varclass_components(*Exp1*, *VarIn*, *VarInt*, *Texp1*),
  remove_varclass_components(*Exp2*, *VarInt*, *VarOut*, *Texp2*).
remove_varclass_components(**imp**(*Exp1*, *Exp2*), *VarIn*, *VarOut*, **imp**(*Texp1*, *Texp2*)) ←
  remove_varclass_components(*Exp1*, *VarIn*, *VarInt*, *Texp1*),
  remove_varclass_components(*Exp2*, *VarInt*, *VarOut*, *Texp2*).
remove_varclass_components(**or**(*Exp1*, *Exp2*), *VarIn*, *VarOut*, **or**(*Texp1*, *Texp2*)) ←
  remove_varclass_components(*Exp1*, *VarIn*, *VarInt*, *Texp1*),
  remove_varclass_components(*Exp2*, *VarInt*, *VarOut*, *Texp2*).
remove_varclass_components(**all**(*Exp1*, *Exp2*), *VarIn*, *VarOut*, **all**(*Exp1*, *Texp2*)) ←
  remove_varclass_components(*Exp2*, *VarIn*, *VarOut*, *Texp2*).
remove_varclass_components(**exists**(*Exp1*, *Exp2*), *VarIn*, *VarOut*, **exists**(*Exp1*, *Texp2*)) ←
  remove_varclass_components(*Exp2*, *VarIn*, *VarOut*, *Texp2*).
remove_varclass_components(**numexists**(*Exp1*, *A*, *B*, *Exp2*), *VarIn*, *VarOut*,
numexists(*Exp1*, *A*, *B*, *Texp2*)) ←
  remove_varclass_components(*Exp2*, *VarIn*, *VarOut*, *Texp2*).

---

insert_varclass_components_list

$(+VarClassMap, +Var, +PredExp, -PredExp)$

Repeatly calls insert_varclass_components for each varclass element in

the given classmap.

insert_varclass_components_list([], _, *Exp*, *Exp*).
insert_varclass_components_list([[**class**, *Class*, *Var*]|*Rest*], *OuterVar*, *Exp*, *Res*) ←
  insert_varclass_components(**varclass**(*Class*, *Var*), *OuterVar*, *Exp*, *Inter*),
  insert_varclass_components_list(*Rest*, *OuterVar*, *Inter*, *Res*).
insert_varclass_components_list([[**attribute**, _, _]|*Rest*], *OuterVar*, *Exp*, *Res*) ←
  insert_varclass_components_list(*Rest*, *OuterVar*, *Exp*, *Res*).

---

insert_varclass_components(+*VarClassExp*, +*Var*, +*Exp*, −*Exp*)

Inserts the classes which uses userdefined variables right next to the first class. The classmap has all userdefined variables, the Variable marks the first class

insert_varclass_components(*VC*, *Outer*, **imp**(*Exp1*, *Exp2*), **imp**(*Res*, *Exp2*)) ←
  insert_varclass_components(*VC*, *Outer*, *Exp1*, *Res*).
insert_varclass_components(*VC*, *Outer*, **and**(*Exp1*, *Exp2*), **and**(*Res*, *Exp2*)) ←
  insert_varclass_components(*VC*, *Outer*, *Exp1*, *Res*).
insert_varclass_components(**varclass**(*C*, *V*), *Outer*, **class**(*Class*, *Outer*),
and(**class**(*Class*, *Outer*), **varclass**(*C*, *V*))).
insert_varclass_components(**varclass**(*C*, *V*), *Outer*, **varclass**(*Class*, *Outer*),
and(**varclass**(*Class*, *Outer*), **varclass**(*C*, *V*))).

---

add_universal_quantified_variable(+*PredExp*, +*VarMap*, −*PredExp*).

Adds universally quantified variables which are in the VarMap to the predicate expression.

add_universal_quantified_variable(*Exp*, [], *Exp*).

add_universal_quantified_variable(*Exp*, [[_, _, *Var*]|*Tail*], **all**(*Var*, *Res*)) ←
  add_universal_quantified_variable(*Exp*, *Tail*, *Res*).

/ *

Wellformednesschecks

* /

---

check_wellformedness(+*PredExp*, −*VarList*)

Succeeds if the predicate expression is well-formed.

check_wellformedness(*Exp*) ←

substract_varclassmap(*Exp*, *Map*),
substract_ud_varmap(*Map*, *UDMap*),
check_variable_types(*UDMap*),
check_variable_count(*Exp*),
check_paths(*Exp*, *Map*).

---

check_variable_types(+*VarClassMap*)

Calls check_different on every element in the given VarClassMap. Suceeds if check different succeds on all combinations of the elements.

check_variable_types([]).
check_variable_types([[*Type*, *Class*, *Var*]|*Rest*]) ←
check_different([*Type*, *Class*, *Var*], *Rest*),
check_variable_types(*Rest*).

---

check_variable_types(+*VarClassEntity*, +*VarClassMap*)

Suceeds if no variables in the VarClassMap refers to different classes or attributes

check_different(_, []).
check_different([*Type*, *Class*, *Var1*], [[*Type*, *Class*, *Var1*]|*Rest*]) ←
check_different([*Type*, *Class*, *Var1*], *Rest*), **!**.
check_different([*Type*, *Class*, *Var1*], [[_, *Class*, *Var2*]|*Rest*]) ←
check_different([*Type*, *Class*, *Var1*], *Rest*),
Var1 \= *Var2*, **!**.
check_different([*Type*, *Class1*, *Var1*], [[_, *Class2*, *Var2*]|*Rest*]) ←
check_different([*Type*, *Class1*, *Var1*], *Rest*),
Class1 \= *Class2*, *Var1* \= *Var2*, **!**.
check_different([*Type*, *Class1*, *Var1*], [[*Type2*, _, *Var2*]|*Rest*]) ←
check_different([*Type*, *Class1*, *Var1*], *Rest*),
Type \= *Type2*, *Var1* \= *Var2*, **!**.

---

check_variable_count(+*Expression*)

Uses count_variable_used to create a VarCountMap, which is then checked with is_countmap_ok. This predicate suceeds if every userdefined variable is used at least twice in the hlcl expression

check_variable_count(*Exp*) ←
count_variable_used(*Exp*, [], *Map*),
is_countmap_ok(*Map*).

---

is_countmap_ok(+*VarCountMap*)

Suceeds if every entitity in the VarCountMap is used at least twice.

is_countmap_ok([ ]).
is_countmap_ok([[*Int*, _]|*Rest*]) ←
  Int **is max**(2, *Int*),
  is_countmap_ok(*Rest*).

---

count_variable_used(+*PredExp*, +*VarClassMap*, −*VarClassMap*)

Collects all used userdefined variables in a VarClassNoMap

count_variable_used(**and**(*Exp1*, *Exp2*), *MapIn*, *MapOut*) ←
  count_variable_used(*Exp1*, *MapIn*, *MapInt*),
  count_variable_used(*Exp2*, *MapInt*, *MapOut*).
count_variable_used(**or**(*Exp1*, *Exp2*), *MapIn*, *MapOut*) ←
  count_variable_used(*Exp1*, *MapIn*, *MapInt*),
  count_variable_used(*Exp2*, *MapInt*, *MapOut*).
count_variable_used(**imp**(*Exp1*, *Exp2*), *MapIn*, *MapOut*) ←
  count_variable_used(*Exp1*, *MapIn*, *MapInt*),
  count_variable_used(*Exp2*, *MapInt*, *MapOut*).
count_variable_used(**all**(_, *Exp*), *MapIn*, *MapOut*) ←
  count_variable_used(*Exp*, *MapIn*, *MapOut*).
count_variable_used(**exists**(_, *Exp*), *MapIn*, *MapOut*) ←
  count_variable_used(*Exp*, *MapIn*, *MapOut*).
count_variable_used(**numexists**(_, _, _, *Exp*), *MapIn*, *MapOut*) ←
  count_variable_used(*Exp*, *MapIn*, *MapOut*).
count_variable_used(**neg**(*Exp*), *MapIn*, *MapOut*) ←
  count_variable_used(*Exp*, *MapIn*, *MapOut*).
count_variable_used(**relation**(_, *Var1*, *Var2*), *MapIn*, *MapOut*) ←
  add_to_map(*Var1*, *MapIn*, *MapInt*),
  add_to_map(*Var2*, *MapInt*, *MapOut*).
count_variable_used(**binarypredicate**(_, *Var1*, *Var2*), *MapIn*, *MapOut*) ←
  add_to_map(*Var1*, *MapIn*, *MapInt*),
  add_to_map(*Var2*, *MapInt*, *MapOut*).
count_variable_used(**unarypredicate**(_, *Var*), *MapIn*, *MapOut*) ←
  add_to_map(*Var*, *MapIn*, *MapInt*),
  add_to_map(*Var*, *MapInt*, *MapOut*).
count_variable_used(**attribute**(_, _, *Val*, _), *MapIn*, *MapOut*) ←
  is_ud_variable(*Val*),
  add_to_map(*Val*, *MapIn*, *MapOut*), **!**.
count_variable_used(**attribute**(_, _, _, _), *Map*, *Map*).
count_variable_used(**class**(_, _), *Map*, *Map*).

count_variable_used(**varclass**(_, _), *Map*, *Map*).

---

add_to_map(+*Var*, +*VarNoMap*, −*VarNoMap*)

Adds the Var to the given VarNoMap and returns the VarNoMap.

add_to_map(*Val*, [ ], [[1, *Val*]]) ←
  is_ud_variable(*Val*), **!**.
add_to_map(_, [ ], [ ]).
add_to_map(*Val*, [[*Int*, *Val*]|*Rest*], [[*NewInt*, *Val*]|*Rest*]) ←
  NewInt **is** *Int* + 1, **!**.
add_to_map(*Val*, [[*Int*, *AnotherVal*]|*Rest*], [[*Int*, *AnotherVal*]|*Result*]) ←
  add_to_map(*Val*, *Rest*, *Result*).

---

check_paths(+*PredExp*, +*VarList*)

Checks relations and predicates expressions corresponds to the given conceptual model.

check_paths(**neg**(*X*), *Res*) ←
  check_paths(*X*, *Res*).

check_paths(**or**(*X*, *Y*), *Res*) ←
  check_paths(*X*, *Res*),
  check_paths(*Y*, *Res*).

check_paths(**and**(*X*, *Y*), *Res*) ←
  check_paths(*X*, *Res*),
  check_paths(*Y*, *Res*).

check_paths(**imp**(*X*, *Y*), *Res*) ←
  check_paths(*X*, *Res*),
  check_paths(*Y*, *Res*).

check_paths(**exists**(_, *X*), *Res*) ←
  check_paths(*X*, *Res*).
check_paths(**numexists**(_, _, _, *X*), *Res*) ←
  check_paths(*X*, *Res*).
check_paths(**all**(_, *X*), *Res*) ←
  check_paths(*X*, *Res*).
check_paths(**relation**(*Relation*, *Var1*, *Var2*), *List*) ←
  is_related(*Class1*, *Relation*, *Class2*, _),
  lookup_class_label(*Class1*, _, *Var1*, *List*),
  lookup_class_label(*Class2*, _, *Var2*, *List*).
check_paths(**relation**(*Relation*, *Var1*, *Var2*), *List*) ←

relation(*Class1*, *Relation*, *Class2*),
lookup_class_label(*Class1*, _, *Var1*, *List*),
lookup_class_label(*Class2*, _, *Var2*, *List*).
check_paths(**unarypredicate**(*Predicate*, *Var1*), *List*) ←
sqlfunction(*Predicate*, [[_, *Att1*]], _),
lookup_class_label(*Class1*, _, *Var1*, *List*),
has_attribute(*Class1*, *Att1*).
check_paths(**binarypredicate**(*Predicate*, *Var1*, *Var2*), *List*) ←
sqlfunction(*Predicate*, [[_, *Att1*], [_, *Att2*]], _),
lookup_class_label(*Class1*, _, *Var1*, *List*),
lookup_class_label(*Class2*, _, *Var2*, *List*),
has_attribute(*Class1*, *Att1*), **has_attribute**(*Class2*, *Att2*).
check_paths(**class**(_, _), _).
check_paths(**varclass**(_, _), _).
check_paths(**attribute**(*Att*, _, *Value*, *Var*), *Map*) ←
lookup_class_label(*Class*, _, *Var*, *Map*),
is_attribute(*Class*, *Att*),
is_ud_variable(*Value*).
check_paths(**attribute**(*Att*, _, *Value*, *Var*), *Map*) ←
lookup_class_label(*Class*, _, *Var*, *Map*),
is_related(*Class*, *Att*, _, _),
is_ud_variable(*Value*).
check_paths(**attribute**(*Att*, _, *Value*, *Var*), *Map*) ←
lookup_class_label(*Class*, _, *Var*, *Map*),
is_attribute(*Class*, *Att*),
is_attribute_within_boundries(*Class*, *Att*, *Value*).
check_paths(**attribute**(*Att*, _, *Value*, *Var*), *Map*) ←
lookup_class_label(*Class*, _, *Var*, *Map*),
is_related(*Class*, *Att*, _, *ActualClass*),
is_attribute_within_boundries(*ActualClass*, *Att*, *Value*).

/**

Intermediate Steps

**/

---

perform_intermediate_steps(+*PredExp*, −*PredExp*)

Translates the predicate expression into Datalog(not,or,exists).

perform_intermediate_steps(*Expression*, *Result*) ←
replace_internal_all(*Expression*, *R1*),
replace_implication(*R1*, *R2*),
negate_expression(*R2*, *Result*).

---

replace_internal_all(+*PredExp*, −*PredExp*)

Replaces internal universal quantifiers with a double negated existential quantifier. This predicate just traverses through the outer universal quantifiers and uses replace_all for the actual removal.

replace_internal_all(**all**(A, *Exp*), **all**(A, *Res*)) ←
  replace_internal_all(*Exp*, *Res*), **!**.
replace_internal_all(*Exp*, *Res*) ←
  replace_all(*Exp*, *Res*).

---

replace_all(+*PredExp*, −*PredExp*)

Replaces internal universal quantifiers with a double negated existential quantifier

replace_all(**all**(A, **imp**(B, C)), **neg**(**exists**(A, **and**(B, **neg**(C))))) ← **!**.
replace_all(**imp**(A, B), **imp**(C, D)) ←
  replace_all(A, C),
  replace_all(B, D), **!**.
replace_all(**or**(A, B), **or**(C, D)) ←
  replace_all(A, C),
  replace_all(B, D), **!**.
replace_all(**and**(A, B), **and**(C, D)) ←
  replace_all(A, C),
  replace_all(B, D), **!**.
replace_all(**exists**(A, B), **exists**(A, D)) ←
  replace_all(B, D), **!**.
replace_all(**numexists**(Q, N, A, B), **numexists**(Q, N, C, D)) ←
  replace_all(A, C),
  replace_all(B, D), **!**.
replace_all(**neg**(A), **neg**(B)) ←
  replace_all(A, B), **!**.
replace_all(A, A).

---

replace_implication(+*PredExp*, −*PredExp*)

Removes the outer implication and negates the whole expression.

replace_implication(**imp**(A, **neg**(B)), **and**(C, D)) ←
  replace_implication(A, C),
  replace_implication(B, D), **!**.
replace_implication(**imp**(A, B), **and**(C, **neg**(D))) ←
  replace_implication(A, C),

```
   replace_implication(B, D), !.
replace_implication(or(A, B), or(C, D)) ←
   replace_implication(A, C),
   replace_implication(B, D), !.
replace_implication(and(A, B), and(C, D)) ←
   replace_implication(A, C),
   replace_implication(B, D), !.
replace_implication(exists(A, B), exists(C, D)) ←
   replace_implication(A, C),
   replace_implication(B, D), !.
replace_implication(numexists(Q, N, A, B), exists(Q, N, C, D)) ←
   replace_implication(A, C),
   replace_implication(B, D), !.
replace_implication(all(A, B), all(C, D)) ←
   replace_implication(A, C),
   replace_implication(B, D), !.
replace_implication(neg(A), neg(B)) ←
   replace_implication(A, B), !.
replace_implication(A, A).
```

---

negate_expression(+PredExp, −PredExp)

Negates the expression if it is of type no-may, all-must type expressions are already negated in replace_implication

```
negate_expression(all(Var, Exp), all(Var, Texp)) ←
   negate_expression(Exp, Texp), !.
negate_expression(neg(exists(Var, Exp)), all(Var, Exp)) ← !.
negate_expression(Exp, Exp).
```

```
/*******************************************
      Datalog → SQL
*******************************************/
```

---

subpred_to_sql(+PredExp, −SQLExp)

Translates the predicate expression into SQL. The different cases are commented seperatly below

```
%CASE: Conjunction "real" And
subpred_to_sql(and(Exp1, Exp2), and(Texp1, Texp2), Inside,
VarMapIn, VarMapOut, RelLUMap, CountFlag) ←
   is_exist_expression(Exp1), is_exist_expression(Exp2),
```

subpred_to_sql(*Exp1*, *Texp1*, *Inside*, *VarMapIn*, *VarMapOut1*, *RelLUMap*, *CountFlag*),
subpred_to_sql(*Exp2*, *Texp2*, *Inside*, *VarMapIn*, *VarMapOut2*, *RelLUMap*, *CountFlag*),
append(*VarMapOut1*, *VarMapOut2*, *VarMapOut*).

%CASE: Disjunction Or
subpred_to_sql(**or**(*Exp1*, *Exp2*), **or**(*Texp1*, *Texp2*), *Inside*,
VarMapIn, *VarMapOut*, *RelLUMap*, *CountFlag*) ←
subpred_to_sql(*Exp1*, *Texp1*, *Inside*, *VarMapIn*, *VarMapOut1*, *RelLUMap*, *CountFlag*),
subpred_to_sql(*Exp2*, *Texp2*, *Inside*, *VarMapIn*, *VarMapOut2*, *RelLUMap*, *CountFlag*),
append(*VarMapOut1*, *VarMapOut2*, *VarMapOut*).

%CASE: "Glue" Conjunction
subpred_to_sql(**and**(*Exp1*, *Exp2*), *Texp1*, *Inside*, *VarMapIn*, *VarMapOut*, *RelLUMap*,
CountFlag) ←
subpred_to_sql(*Exp1*, *Texp1*, *Texp2*, *VarMapIn*, *VarMapOut1*, *RelLUMap*, *CountFlag*),
subpred_to_sql(*Exp2*, *Texp2*, *Inside*, *VarMapOut1*, *VarMapOut*, *RelLUMap*, *CountFlag*).

%CASE: Normal Existential Quantifier
subpred_to_sql(**exists**(*Var*, *Exp*), *Texp*, *Inside*,
VarMapIn, *VarMapOut*, *RelLUMap*, _) ←
find_class(*Var*, *Exp*, **class**(*Class*, *Var*)),
append(*RelLUMap*, [[*Var*, *Class*]], *NewRelLUMap*),
subpred_to_sql(*Exp*, *Texp*, *Inside*, *VarMapIn*, *VarMapOut*, *NewRelLUMap*, **nocount**).

%CASE: Nummerical Quantifier
subpred_to_sql(**exists**(*Comp*, *No*, *Var*, *Exp*), **comparison**(*Comp*, *Texp*, *No*),
Inside, *VarMapIn*, *VarMapOut*,
RelLUMap, _) ←
find_class(*Var*, *Exp*, **class**(*Class*, *Var*)),
append(*RelLUMap*, [[*Var*, *Class*]], *NewRelLUMap*),
subpred_to_sql(*Exp*, *Texp*, *Inside*, *VarMapIn*, *VarMapOut*, *NewRelLUMap*, **count**).

%CASE: Universal Quantifer
subpred_to_sql(**all**(*Var*, *Exp*), *Texp*, *Inside*, *VarMapIn*, *VarMapOut*, *RelLUMap*, _) ←
find_class(*Var*, *Exp*, **class**(*Class*, *Var*)),
append(*RelLUMap*, [[*Var*, *Class*]], *NewRelLUMap*),
subpred_to_sql(*Exp*, *Texp*, *Inside*, *VarMapIn*, *VarMapOut*, *NewRelLUMap*, **nocount**), **!**.

%CASE: Universal Quantifier, quantifying a attribute value
subpred_to_sql(**all**(_, *Exp*), *Texp*, *Inside*, *VarMapIn*, *VarMapOut*, *RelLUMap*, *CF*) ←
subpred_to_sql(*Exp*, *Texp*, *Inside*, *VarMapIn*, *VarMapOut*, *RelLUMap*, *CF*).

%CASE: Negation
subpred_to_sql(**neg**(*Exp*), **notstat**(*TExp*), *Inside*, *VarMapIn*,
VarMapOut, *RelLUMap*, *CF*) ←
subpred_to_sql(*Exp*, *TExp*, *Inside*, *VarMapIn*, *VarMapOut*, *RelLUMap*, *CF*).

%CASE: The very first class in the extended datalog expression

subpred_to_sql(**class**(*Class*, *Var*),
selectstat('*', *ClassTable*, **a**, **and**(*CondExp*, *InsideExp*)),
InsideExp, [ ], [[*Var*, **class**(*Class*, *Var*), **a**]], _, _) ←
conceptual_to_table(*Class*, *ClassTable*, *Condition*),
condition_to_sql(**a**, *Condition*, *CondExp*), **!**.

%CASE: The very first varclass in the extended datalog expression
subpred_to_sql(**varclass**(*Class*, *Var*),
selectstat('*', *ClassTable*, **a**, **and**(*CondExp*, *InsideExp*)),
InsideExp, [ ], [[*Var*, **class**(*Class*, *Var*), **a**]], _, _) ←
conceptual_to_table(*Class*, *ClassTable*, *Condition*),
condition_to_sql(**a**, *Condition*, *CondExp*), **!**.

%CASE: A class in a relational path
subpred_to_sql(**class**(*Class*, *Var*),
exists(**selectstat**('*', *ClassTable*, *CLabel*, **and**(**and**(*CondExp*, *EQ2*), *InsideExp*))),
InsideExp, *VarMap*, *NewVarMap*, _, _) ←
conceptual_to_table(*Class*, *ClassTable*, *Condition*),
condition_to_sql(*CLabel*, *Condition*, *CondExp*),
lookup_class_label(*Exp*, *Var*, *ELabel*, *VarMap*),
get_next_label(*VarMap*, *CLabel*),
exchange_expression(*Var*, **class**(*Class*, *Var*), *CLabel*, *Exp*, *ELabel*, *VarMap*, *NewVarMap*),
set_equal(**class**(*Class*, *Var*), *CLabel*, *Exp*, *ELabel*, *EQ2*).

%CASE: A class in a relational path, before the relation is met
subpred_to_sql(**class**(*Class*, *Var*),
exists(**selectstat**('*', *ClassTable*, *CLabel*, **and**(*CondExp*, *InsideExp*))),
InsideExp, *VarMap*, *NewVarMap*, _, _) ←
conceptual_to_table(*Class*, *ClassTable*, *Condition*),
get_next_label(*VarMap*, *CLabel*),
condition_to_sql(*CLabel*, *Condition*, *CondExp*),
append(*VarMap*, [[*Var*, **class**(*Class*, *Var*), *CLabel*]], *NewVarMap*), **!**.

%CASE: A varclass in a relational path, before the relation is met
subpred_to_sql(**varclass**(*Class*, *Var*),
exists(**selectstat**('*', *ClassTable*, *CLabel*, **and**(*CondExp*, **and**(*NegExp*, *InsideExp*)))),
InsideExp, *VarMap*, *NewVarMap*, _, _) ←
conceptual_to_table(*Class*, *ClassTable*, *Condition*),
get_next_label(*VarMap*, *CLabel*),
condition_to_sql(*CLabel*, *Condition*, *CondExp*),
make_negation_expression(*Class*, *CLabel*, *VarMap*, *NegExp*),
append(*VarMap*, [[*Var*, **class**(*Class*, *Var*), *CLabel*]], *NewVarMap*), **!**.

%CASE: A varclass in the relational path
subpred_to_sql(**varclass**(*Class*, *Var*),
exists(**selectstat**('*', *ClassTable*, *CLabel*, **and**(**and**(*CondExp*, *EQ2*), *InsideExp*))),

InsideExp, *VarMap*, *NewVarMap*, _, _) ←
conceptual_to_table(*Class*, *ClassTable*, *Condition*),
condition_to_sql(*CLabel*, *Condition*, *CondExp*),
lookup_class_label(*Exp*, *Var*, *ELabel*, *VarMap*),
get_next_label_simple(*ELabel*, *CLabel*),
exchange_expression(*Var*, **class**(*Class*, *Var*), *CLabel*, *Exp*, *ELabel*, *VarMap*, *NewVarMap*),
set_equal(**class**(*Class*, *Var*), *CLabel*, *Exp*, *ELabel*, *EQ2*).

%CASE: A relation relating the same variables
subpred_to_sql(**relation**(*Relation*, *Var*, *Var*),
exists(**selectstat**('*', *RelationTable*, *RLabel*, **and**(*EQ1*, *InsideExp*))),
InsideExp, *VarMap*, *NewVarMap*, *LookUpInMe*, _) ←
lookup_class(*Class*, *Var*, *LookUpInMe*),
lookup_class_label(**class**(*Class*, *Var*), *Var*, *ELabel*, *VarMap*),
conceptual_relation_to_table(*Class*, *Relation*, *Class*, *RelationTable*, [ ]),
get_next_label(*VarMap*, *RLabel*),
append(*VarMap*, [[**dummy**, **relation**(*Relation*, *Var*, *Var*), *RLabel*]], *NewVarMap*),
set_equal(**relation**(*Relation*, *Var*, *Var*), *RLabel*, **class**(*Class*, *Var*), *ELabel*, *EQ1*), **!**.

%CASE: A relation relating already translated classes
subpred_to_sql(**relation**(*Relation*, *Var1*, *Var2*),
exists(**selectstat**('*', *RelationTable*, *RLabel*, **and**(**and**(*EQ1*, *EQ2*), *InsideExp*))),
InsideExp, *VarMap*, *NewVarMap*, *LookUpInMe*, _) ←
lookup_class(*Class2*, *Var2*, *LookUpInMe*),
lookup_class_label(**class**(*Class*, *Var1*), *Var*, *ELabel*, *VarMap*),
lookup_class_label(**class**(*Class2*, *Var2*), *Var2*, *ELabel2*, *VarMap*),
conceptual_relation_to_table(*Class*, *Relation*, *Class2*, *RelationTable*, [ ]),
get_next_label(*VarMap*, *RLabel*),
append(*VarMap*, [[**dummy**, **relation**(*Relation*, *Var1*, *Var2*), *RLabel*]], *NewVarMap*),
set_equal(**relation**(*Relation*, *Var1*, *Var2*), *RLabel*, **class**(*Class*, *Var*), *ELabel*, *EQ1*),
set_equal(**relation**(*Relation*, *Var1*, *Var2*), *RLabel*, **class**(*Class2*, *Var2*), *ELabel2*, *EQ2*), **!**.

%CASE: A relation relating a translated class to a non−translated class
subpred_to_sql(**relation**(*Relation*, *Var1*, *Var2*),
exists(**selectstat**('*', *RelationTable*, *RLabel*, **and**(*EQ1*, *InsideExp*))),
InsideExp, *VarMap*, *NewVarMap*, *LookUpInMe*, **nocount**) ←
lookup_class(*Class2*, *Var2*, *LookUpInMe*),
lookup_class_label(**class**(*Class*, *Var*), *Var*, *ELabel*, *VarMap*),
conceptual_relation_to_table(*Class*, *Relation*, *Class2*, *RelationTable*, [ ]),
get_next_label(*VarMap*, *RLabel*),
set_equal(**relation**(*Relation*, *Var1*, *Var2*), *RLabel*, **class**(*Class*, *Var*), *ELabel*, *EQ1*),
append(*VarMap*, [[*Var2*, **relation**(*Relation*, *Var1*, *Var2*), *RLabel*]], *NewVarMap*).

%CASE: A relation right after a nummerical quantifier
subpred_to_sql(**relation**(*Relation*, *Var1*, *Var2*),
exists(**selectstat**('COUNT(*)', *RelationTable*, *RLabel*, **and**(*EQ1*, *InsideExp*))),

InsideExp, *VarMap*, *NewVarMap*, *LookUpInMe*, **count**) ←
lookup_class(*Class2*, *Var2*, *LookUpInMe*),
lookup_class_label(**class**(*Class*, *Var*), *Var*, *ELabel*, *VarMap*),
conceptual_relation_to_table(*Class*, *Relation*, *Class2*, *RelationTable*, [ ]),
get_next_label(*VarMap*, *RLabel*),
set_equal(**relation**(*Relation*, *Var1*, *Var2*), *RLabel*, **class**(*Class*, *Var*), *ELabel*, *EQ1*),
append(*VarMap*, [[*Var2*, **relation**(*Relation*, *Var1*, *Var2*), *RLabel*]], *NewVarMap*).

%CASE: The first time an attribute with a userdefined variable as value,
% which is located at a parent in a ISA−Hierachy is met.
subpred_to_sql(**attribute**(*Att*, **eq**, *Val*, *Var*),
selectstat('*', *ClassTable*, *CLabel*, **and**(*EQ1*, **and**(*CondExp*,
and(**comparison**(**eq**, (*CLabel*, *Att*), (*AttLabel*, *Att*)), *Inside*)))),
Inside, *VarMap*, *VarMap*, _, _) ←
is_ud_variable(*Val*),
lookup_class_label(**class**(*Class*, *Var*), *Var*, *ELabel*, *VarMap*),
lookup_class_label(*Att*, *Val*, *AttLabel*, *VarMap*),
is_related(*Class*, *Att*, _, *SuperClass*),
conceptual_to_table(*SuperClass*, *ClassTable*, *Condition*),
condition_to_sql(*CLabel*, *Condition*, *CondExp*),
get_next_label(*VarMap*, *CLabel*),
set_equal(**class**(*SuperClass*, *Var*), *CLabel*, **class**(*Class*, *Var*), *ELabel*, *EQ1*), **!**.

%CASE: The next time an attribute with a userdefined variable as value,
% which is located at a parent in a ISA−Hierachy is met.
subpred_to_sql(**attribute**(*Att*, **eq**, *Val*, *Var*),
and(**comparison**(**eq**, (*ELabel*, *Att*), (*AttLabel*, *Att*)), *Inside*),
Inside, *VarMap*, *VarMap*, _, _) ←
is_ud_variable(*Val*),
lookup_class_label(**class**(_, *Var*), *Var*, *ELabel*, *VarMap*),
lookup_class_label(*Att*, *Val*, *AttLabel*, *VarMap*), **!**.

%CASE: The first time an attribute, which is located at a parent in a ISA−Hierachy is met.
subpred_to_sql(**attribute**(*Att*, **eq**, *Val*, *Var*),
selectstat('*', *ClassTable*, *CLabel*, **and**(*EQ1*, **and**(*CondExp*, *Inside*))),
Inside, *VarMap*, *NewVarMap*, _, _) ←
lookup_class_label(**class**(*Class*, *Var*), *Var*, *ELabel*, *VarMap*),
is_ud_variable(*Val*),
is_related(*Class*, *Att*, _, *SuperClass*),
conceptual_to_table(*SuperClass*, *ClassTable*, *Condition*),
condition_to_sql(*CLabel*, *Condition*, *CondExp*),
get_next_label(*VarMap*, *CLabel*),
set_equal(**class**(*SuperClass*, *Var*), *CLabel*, **class**(*Class*, *Var*), *ELabel*, *EQ1*),
append(*VarMap*, [[*Val*, *Att*, *CLabel*]], *NewVarMap*), **!**.

%CASE: The next time an attribute, which is located at a parent in a ISA−Hierachy is met.

```
subpred_to_sql(attribute(Att, eq, Val, Var),
Inside, Inside, VarMap, NewVarMap, _, _) ←
   lookup_class_label(class(_, Var), Var, ELabel, VarMap),
   is_ud_variable(Val),
   append(VarMap, [[Val, Att, ELabel]], NewVarMap), !.

%CASE: A nummerically quantified attribute which is located at a parent in a ISA−Hierachy.
subpred_to_sql(attribute(Att, Comp, Val, Var),
and(selectstat('*', ClassTable, CLabel, and(EQ1, and(CondExp,
comparison(Comp, (ELabel, Att), Val)))), Inside),
Inside, VarMap, VarMap, _, _) ←
   lookup_class_label(class(Class, Var), Var, ELabel, VarMap),
   is_related(Class, Att, _, SuperClass),
   conceptual_to_table(SuperClass, ClassTable, Condition),
   condition_to_sql(CLabel, Condition, CondExp),
   get_next_label(VarMap, CLabel),
   set_equal(class(SuperClass, Var), CLabel, class(Class, Var), ELabel, EQ1), !.

%CASE: All other attributes not meeting any of the above descriptions.
subpred_to_sql(attribute(Att, Comp, Val, Var),
and(comparison(Comp, (ELabel, Att), Val), Inside),
Inside, VarMap, VarMap, _, _) ←
   lookup_class_label(class(_, _), Var, ELabel, VarMap).

%CASE: A unary predicate which has its argument in a parent in a ISA−hierachy.
subpred_to_sql(unarypredicate(Name, Var),
and(selectstat('*', ClassTable, CLabel, and(EQ1, and(CondExp, Result))), Inside),
Inside, VarMap, VarMap, _, _) ←
   lookup_class_label(class(Class, Var), Var, ELabel, VarMap),
   get_next_label(VarMap, CLabel),
   sqlfunction(Name, [[CLabel, Att]], Result),
   is_related(Class, Att, _, SuperClass),
   conceptual_to_table(SuperClass, ClassTable, Condition),
   condition_to_sql(CLabel, Condition, CondExp),
   set_equal(class(Class, Var), ELabel, class(SuperClass, Var), CLabel, EQ1), !.

%CASE: All other unary predicates.
subpred_to_sql(unarypredicate(Name, Var),
and(Result, Inside),
Inside, VarMap, VarMap, _, _) ←
   lookup_class_label(class(_, Var), Var, Label, VarMap),
   sqlfunction(Name, [[Label, _]], Result).

%CASE: Binary predicate which has both arguments in parents in an ISA−hierachy
subpred_to_sql(binarypredicate(Name, Var, Var2),
and(selectstat('*', ClassTable, CLabel, and(EQ1, and(CondExp,
```

selectstat('*', *ClassTable2*, *CLabel2*, **and**(*EQ2*, **and**(*CondExp2*, *Result*))))))), *Inside*),
Inside, *VarMap*, *VarMap*, _, _) ←
lookup_class_label(**class**(*Class*, *Var*), *Var*, *ELabel*, *VarMap*),
get_next_label(*VarMap*, *CLabel*),
lookup_class_label(**class**(*Class2*, *Var2*), *Var2*, *ELabel2*, *VarMap*),
get_next_label_simple(*CLabel*, *CLabel2*),
sqlfunction(*Name*, [[*CLabel*, *Att*], [*CLabel2*, *Att2*]], *Result*),
is_related(*Class*, *Att*, _, *SuperClass*),
is_related(*Class2*, *Att2*, _, *SuperClass2*),
conceptual_to_table(*SuperClass*, *ClassTable*, *Condition*),
condition_to_sql(*CLabel*, *Condition*, *CondExp*),
conceptual_to_table(*SuperClass2*, *ClassTable2*, *Condition2*),
condition_to_sql(*CLabel2*, *Condition2*, *CondExp2*),
set_equal(**class**(*SuperClass*, *Var*), *CLabel*, **class**(*Class*, *Var*), *ELabel*, *EQ1*),
set_equal(**class**(*SuperClass2*, *Var2*), *CLabel2*, **class**(*Class2*, *Var2*), *ELabel2*, *EQ2*), **!**.

%CASE: Binary predicate which its first arguments in parent in an ISA−hierachy
subpred_to_sql(**binarypredicate**(*Name*, *Var*, *Var2*),
and(**selectstat**('*', *ClassTable*, *CLabel*, **and**(*EQ1*, **and**(*CondExp*, *Result*))), *Inside*),
Inside, *VarMap*, *VarMap*, _, _) ←
lookup_class_label(**class**(*Class*, *Var*), *Var*, *ELabel*, *VarMap*),
get_next_label(*VarMap*, *CLabel*),
lookup_class_label(**class**(_, *Var2*), *Var2*, *Label2*, *VarMap*),
sqlfunction(*Name*, [[*CLabel*, *Att*], [*Label2*, _]], *Result*),
is_related(*Class*, *Att*, _, *SuperClass*),
conceptual_to_table(*SuperClass*, *ClassTable*, *Condition*),
condition_to_sql(*CLabel*, *Condition*, *CondExp*),
set_equal(**class**(*SuperClass*, *Var*), *CLabel*, **class**(*Class*, *Var*), *ELabel*, *EQ1*), **!**.

%CASE: Binary predicate which its second arguments in parent in an ISA−hierachy
subpred_to_sql(**binarypredicate**(*Name*, *Var2*, *Var*),
and(**selectstat**('*', *ClassTable*, *CLabel*, **and**(*EQ1*, **and**(*CondExp*, *Result*))), *Inside*),
Inside, *VarMap*, *VarMap*, _, _) ←
lookup_class_label(**class**(*Class*, *Var*), *Var*, *ELabel*, *VarMap*),
get_next_label(*VarMap*, *CLabel*),
lookup_class_label(**class**(_, *Var2*), *Var2*, *Label2*, *VarMap*),
sqlfunction(*Name*, [[*Label2*, _], [*CLabel*, *Att*]], *Result*),
is_related(*Class*, *Att*, _, *SuperClass*),
conceptual_to_table(*SuperClass*, *ClassTable*, *Condition*),
condition_to_sql(*CLabel*, *Condition*, *CondExp*),
set_equal(**class**(*SuperClass*, *Var*), *CLabel*, **class**(*Class*, *Var*), *ELabel*, *EQ1*), **!**.

%CASE: All other binary predicates.
subpred_to_sql(**binarypredicate**(*Name*, *Var1*, *Var2*),
and(*Result*, *Inside*), *Inside*, *VarMap*, *VarMap*, _, _) ←

lookup_class_label(**class**(_, *Var1*), *Var1*, *Label1*, *VarMap*),
lookup_class_label(**class**(_, *Var2*), *Var2*, *Label2*, *VarMap*),
sqlfunction(*Name*, [[*Label1*, _], [*Label2*, _]], *Result*).

---

make_negation_expression(+*Classexp1*, +*Label1*, *VarMap*, −*EqExp*)

Looks up in the VarMap to see if a class is already defined which has the same type and userdefined variable. In that case a sql expression setting the two classes different is returned. This is used when we declare different variables of same time.

make_negation_expression(_, _, [], []) ← **!**.

make_negation_expression(*Class*, *CLabel*, [[*Var*, **class**(*Class*, *Var*), *Olabel*]|*Rest*], **and**(*Exp*, *RestExp*)) ←
is_ud_variable(*Var*),
get_class_column_names(*Class*, *CID*),
make_not_equal_exp(*CLabel*, *CID*, *Olabel*, *CID*, *Exp*),
make_negation_expression(*Class*, *CLabel*, *Rest*, *RestExp*), **!**.

make_negation_expression(*Class*, *CLabel*, [_|*Rest*], *RestExp*) ←
make_negation_expression(*Class*, *CLabel*, *Rest*, *RestExp*), **!**.

---

make_not_equal_exp(+*Entity1*, +*AttList1*, +*Entity2*, +*AttList2*, −*EqExp*)

Only Used by the make_negation_expressoin predicate, constructs the actual SQL comparison expression

make_not_equal_exp(*Ent1*, [*Att1*], *Ent2*, [*Att2*], **notequals**((*Ent1*, *Att1*), (*Ent2*, *Att2*))) ← **!**.

make_not_equal_exp(*Ent1*, [*Att1*|*Tail1*], *Ent2*, [*Att2*|*Tail2*], **and**(**notequals**((*Ent1*, *Att1*), (*Ent2*, *Att2*)), *Result*)) ←
make_not_equal_exp(*Ent1*, *Tail1*, *Ent2*, *Tail2*, *Result*).

---

set_equal(+*Exp*, +*Exp*, −*EqExp*)

Sets the class or the relation (both tables in SQL) equal, no matter how many keys they have in common.

set_equal(**relation**(*Relation*, *ClassVar*, *ClassVar*), *RLabel*, **class**(*Class*, *ClassVar*), *CLabel*, **and**(*Result1*, *Result2*)) ←
get_class_column_names(*Class*, *CID*),
get_relation_column_names(*Relation*, *RID1*, *RID2*),

make_equal_exp(*RLabel*, *RID1*, *CLabel*, *CID*, *Result1*),
make_equal_exp(*RLabel*, *RID2*, *CLabel*, *CID*, *Result2*), **!**.

set_equal(**relation**(*Relation*, *ClassVar*, _), *RLabel*,
class(*Class*, *ClassVar*), *CLabel*, *Result*) ←
get_class_column_names(*Class*, *CID*),
get_relation_column_names(*Relation*, *RID*, _),
make_equal_exp(*RLabel*, *RID*, *CLabel*, *CID*, *Result*).

set_equal(**relation**(*Relation*, _, *ClassVar*), *RLabel*,
class(*Class*, *ClassVar*), *CLabel*, *Result*) ←
get_class_column_names(*Class*, *CID*),
get_relation_column_names(*Relation*, _, *RID*),
make_equal_exp(*RLabel*, *RID*, *CLabel*, *CID*, *Result*).

set_equal(**class**(*Class*, *ClassVar*), *CLabel*,
relation(*Relation*, _, *ClassVar*), *RLabel*, *Result*) ←
get_class_column_names(*Class*, *CID*),
get_relation_column_names(*Relation*, _, *RID*),
make_equal_exp(*CLabel*, *CID*, *RLabel*, *RID*, *Result*).

set_equal(**class**(*Class*, *ClassVar*), *CLabel*,
relation(*Relation*, *ClassVar*, _), *RLabel*, *Result*) ←
get_class_column_names(*Class*, *CID*),
get_relation_column_names(*Relation*, *RID*, _),
make_equal_exp(*CLabel*, *CID*, *RLabel*, *RID*, *Result*).

set_equal(**class**(*Class*, *ClassVar*), *CLabel*,
class(*Class1*, *ClassVar*), *CLabel1*, *Result*) ←
get_class_column_names(*Class*, *CID*),
get_class_column_names(*Class1*, *CID1*),
make_equal_exp(*CLabel*, *CID*, *CLabel1*, *CID1*, *Result*).

---

make_equal_exp(+*Entity1*, +*AttList1*, +*Entity2*, +*AttList2*, −*EqExp*)

Only used by the set_equal predicate, constructs the actual SQL equal expression

make_equal_exp(*Ent1*, [*Att1*], *Ent2*, [*Att2*], **equals**((*Ent1*, *Att1*), (*Ent2*, *Att2*))) ← **!**.

make_equal_exp(*Ent1*, [*Att1*|*Tail1*], *Ent2*, [*Att2*|*Tail2*],
and(**equals**((*Ent1*, *Att1*), (*Ent2*, *Att2*)), *Result*)) ←
make_equal_exp(*Ent1*, *Tail1*, *Ent2*, *Tail2*, *Result*).

---

get_next_label_simple(+*Char*, −*Char*)

Given a char (label) it returns the next char in the alfabet.

get_next_label_simple(*Label*, *Res*) ←
  char_code(*Label*, *Int*),
  NextInt **is** *Int* + 1,
  char_code(*Res*, *NextInt*).

---

get_next_label(+*LabelList*, −*Label*)

Given a char-list (label) it returns the next char in the alfabet not in the list.

get_next_label([ ], **a**) ← **!**.
get_next_label(*List*, *Res*) ←
  find_biggest(*List*, *Int*),
  NextInt **is** *Int* + 1,
  char_code(*Res*, *NextInt*), **!**.

---

find_biggest(+*LabelList*, −*Label*)

Given VarLabelMap it returns the next char (label) in the alfabet not in the list.

find_biggest([ ], 0).
find_biggest([[_, _, *Label*]|*Rest*], *Result*) ←
  char_code(*Label*, *Int*),
  find_biggest(*Rest*, *Res*),
  Result **is max**(*Int*, *Res*).

---

conceptual_to_table

(+Class,-Tablename,-Condition)
Given a class it returns the corresponding table in the database and any conditions.

conceptual_to_table(*Class*, *Table*, *Condition*) ←
  classmap(*Class*, *Table*, *Condition*, _).

---

conceptual_relation_to_table

(+*Class1* , +*Relation*, +*Class2*, −*Tablename*, −*Condition*)

Given a relation and its connecting classes it returns the corresponding table in the database and any conditions.

conceptual_relation_to_table(*Class1*, *Relation*, *Class2*, *Table*, []) ←
  relmap(*Class1*, *Relation*, *Class2*, *Table*, _, _).

---

condition_to_sql(+Label, +Condition, −SQLSubExp)

Given a label and a condition (from conceptual model) the corresponding comparison in SQL is returned

condition_to_sql(_, [], []).
condition_to_sql(*Label*, [*Att*, *Val*], **equals**((*Label*, *Att*), *Val*)).

---

is_exist_expression(+PredExp)

Suceeds if the expression is starting with a (negated) existential/numerrical quantifier.

is_exist_expression(**exists**(_, _)).
is_exist_expression(**neg**(**exists**(_, _))).
is_exist_expression(**exists**(_, _, _, _)).
is_exist_expression(**neg**(**exists**(_, _, _, _))).

/ *

Intermediate → **pure** *Datalog*(**neg**)

* /

---

pred_to_datalog(+PredExp, −DatExp)

Converts a Predicate expression into a "pure" Datalog(Neg) expression

pred_to_datalog(*Pexp*,
`['N/A␣␣-␣Cannot␣express␣nummerical␣quantification␣in␣datalog']`) ←
  contain_nummerical_quantifier(*Pexp*), **!**.

pred_to_datalog(*Pexp*, *Res*) ←
  reduce_scope_of_negation(*Pexp*, *Pexp1*),
  distribute_conjunctions(*Pexp1*, *Pexp2*),
  remove_outer_universal_quantifiers(*Pexp2*, *Pexp3*),
  rewrite_existential_quantifiers(*Pexp3*, *Pexp4*),
  remove_disjunctions_list(*Pexp4*, *Res*).

---

contain_nummerical_quantifier(+*PredExp*)

Suceeds if the expression contains a nummerical quantification.

```
contain_nummerical_quantifier(exists(_, _, _, _)).
contain_nummerical_quantifier(attribute(_, le, _, _)).
contain_nummerical_quantifier(attribute(_, ge, _, _)).
contain_nummerical_quantifier(all(_, Exp)) ←
   contain_nummerical_quantifier(Exp).
contain_nummerical_quantifier(exists(_, Exp)) ←
   contain_nummerical_quantifier(Exp).
contain_nummerical_quantifier(neg(Exp)) ←
   contain_nummerical_quantifier(Exp).
contain_nummerical_quantifier(and(Exp, _)) ←
   contain_nummerical_quantifier(Exp).
contain_nummerical_quantifier(and(_, Exp)) ←
   contain_nummerical_quantifier(Exp).
contain_nummerical_quantifier(or(Exp, _)) ←
   contain_nummerical_quantifier(Exp).
contain_nummerical_quantifier(or(_, Exp)) ←
   contain_nummerical_quantifier(Exp).
```

---

remove_disjunctions_list(+*PredExpList*, −*PredExpList*)

Applies remove_disjunctions on every clause in the clause list

```
remove_disjunctions_list([], []).

remove_disjunctions_list([leftimp(Val, Exp)|Tail], FinalResult) ←
   remove_disjunctions(Val, Exp, Res),
   remove_disjunctions_list(Tail, Result),
   append(Res, Result, FinalResult).
```

---

remove_disjunctions(+*Value*, −*PredExp*, +*PredExp*)

Breaks up disjunctions and returns them in a clause list.

```
remove_disjunctions(Val, or(Exp1, Exp2), Res) ←
   remove_disjunctions(Val, Exp1, Texp1),
   remove_disjunctions(Val, Exp2, Texp2),
   append(Texp1, Texp2, Res), !.

remove_disjunctions(Val, Exp, [leftimp(Val, Exp)]).
```

---

distribute_conjunctions(+*PredExp*, −*PredExp*)

Distributes conjunctions and returns the predicate expression.

distribute_conjunctions(**and**(*Exp1*, **or**(*Exp2*, *Exp3*)), **or**(*Texp1*, *Texp2*)) ←
  distribute_conjunctions(**and**(*Exp1*, *Exp2*), *Texp1*),
  distribute_conjunctions(**and**(*Exp1*, *Exp3*), *Texp2*), **!**.

distribute_conjunctions(**and**(**or**(*Exp1*, *Exp2*), *Exp3*), **or**(*Texp1*, *Texp2*)) ←
  distribute_conjunctions(**and**(*Exp1*, *Exp3*), *Texp1*),
  distribute_conjunctions(**and**(*Exp2*, *Exp3*), *Texp2*), **!**.

distribute_conjunctions(**or**(*Exp1*, *Exp2*), **or**(*Texp1*, *Texp2*)) ←
  distribute_conjunctions(*Exp1*, *Texp1*),
  distribute_conjunctions(*Exp2*, *Texp2*).

distribute_conjunctions(**and**(*Exp1*, *Exp2*), **or**(**and**(*Texp11*, *Texp2*),
and(*Texp12*, *Texp2*))) ←
  distribute_conjunctions(*Exp1*, **or**(*Texp11*, *Texp12*)),
  distribute_conjunctions(*Exp2*, *Texp2*), **!**.

distribute_conjunctions(**and**(*Exp1*, *Exp2*), **or**(**and**(*Texp1*, *Texp21*), **and**(*Texp1*, *Texp22*))) ←
  distribute_conjunctions(*Exp1*, *Texp1*),
  distribute_conjunctions(*Exp2*, **or**(*Texp21*, *Texp22*)), **!**.

distribute_conjunctions(**and**(*Exp1*, *Exp2*), **or**(**and**(*Texp1*, **exists**(*Var*, *Texp21*)),
and(*Texp1*, **exists**(*Var*, *Texp22*)))) ←
  distribute_conjunctions(*Exp1*, *Texp1*),
  distribute_conjunctions(*Exp2*, **exists**(*Var*, **or**(*Texp21*, *Texp22*))), **!**.

distribute_conjunctions(**and**(*Exp1*, *Exp2*), **and**(*Texp1*, *Texp2*)) ←
  distribute_conjunctions(*Exp1*, *Texp1*),
  distribute_conjunctions(*Exp2*, *Texp2*).

distribute_conjunctions(**neg**(*Exp1*), **neg**(*Texp1*)) ←
  distribute_conjunctions(*Exp1*, *Texp1*).
distribute_conjunctions(**all**(*Var*, *Exp1*), **all**(*Var*, *Texp1*)) ←
  distribute_conjunctions(*Exp1*, *Texp1*).
distribute_conjunctions(**exists**(*Var*, *Exp1*), **exists**(*Var*, *Texp1*)) ←
  distribute_conjunctions(*Exp1*, *Texp1*).
distribute_conjunctions(**leftimp**(*Var*, *Exp1*), **leftimp**(*Var*, *Texp1*)) ←
  distribute_conjunctions(*Exp1*, *Texp1*).

distribute_conjunctions(**relation**(*Relation*, *Var1*, *Var2*), **relation**(*Relation*, *Var1*, *Var2*)).
distribute_conjunctions(**class**(*Class*, *Var*), **class**(*Class*, *Var*)).
distribute_conjunctions(**varclass**(*Class*, *Var*), **varclass**(*Class*, *Var*)).
distribute_conjunctions(**unarypredicate**(*Predicate*, *Var*),
unarypredicate(*Predicate*, *Var*)).
distribute_conjunctions(**binarypredicate**(*Predicate*, *Var1*, *Var2*),
binarypredicate(*Predicate*, *Var1*, *Var2*)).

distribute_conjunctions(**attribute**(*Att*, *C*, *Val*, *Var*), **attribute**(*Att*, *C*, *Val*, *Var*)).
distribute_conjunctions(**function**(*Class*, *Var*), **function**(*Class*, *Var*)).

---

reduce_scope_of_negation(+*PredExp*, −*PredExp*)

Moves negation inwards in the expression, using DemMorgan and double-negation theorems.

reduce_scope_of_negation(**neg**(**and**(*Exp1*, *Exp2*)), **or**(*Texp1*, *Texp2*)) ←
  reduce_scope_of_negation(**neg**(*Exp1*), *Texp1*),
  reduce_scope_of_negation(**neg**(*Exp2*), *Texp2*), **!**.

reduce_scope_of_negation(**neg**(**or**(*Exp1*, *Exp2*)), **and**(*Texp1*, *Texp2*)) ←
  reduce_scope_of_negation(**neg**(*Exp1*), *Texp1*),
  reduce_scope_of_negation(**neg**(*Exp2*), *Texp2*), **!**.

reduce_scope_of_negation(**neg**(**neg**(*Exp*)), *Texp1*) ←
  reduce_scope_of_negation(*Exp*, *Texp1*), **!**.

reduce_scope_of_negation(**neg**(**exists**(*Var*, *Exp*)), **neg**(**exists**(*Var*, *Texp1*))) ←
  reduce_scope_of_negation(*Exp*, *Texp1*), **!**.

reduce_scope_of_negation(**neg**(*Exp*), **neg**(*Texp1*)) ←
  reduce_scope_of_negation(*Exp*, *Texp1*).

reduce_scope_of_negation(**and**(*Exp1*, *Exp2*), **and**(*Texp1*, *Texp2*)) ←
  reduce_scope_of_negation(*Exp1*, *Texp1*),
  reduce_scope_of_negation(*Exp2*, *Texp2*).
reduce_scope_of_negation(**or**(*Exp1*, *Exp2*), **or**(*Texp1*, *Texp2*)) ←
  reduce_scope_of_negation(*Exp1*, *Texp1*),
  reduce_scope_of_negation(*Exp2*, *Texp2*).
reduce_scope_of_negation(**all**(*Var*, *Exp*), **all**(*Var*, *Texp*)) ←
  reduce_scope_of_negation(*Exp*, *Texp*).
reduce_scope_of_negation(**exists**(*Var*, *Exp*), **exists**(*Var*, *Texp*)) ←
  reduce_scope_of_negation(*Exp*, *Texp*).

reduce_scope_of_negation(**relation**(*Relation*, *Var1*, *Var2*),
relation(*Relation*, *Var1*, *Var2*)).
reduce_scope_of_negation(**class**(*Class*, *Var*), **class**(*Class*, *Var*)).
reduce_scope_of_negation(**varclass**(*Class*, *Var*), **varclass**(*Class*, *Var*)).
reduce_scope_of_negation(**unarypredicate**(*Predicate*, *Var*),
unarypredicate(*Predicate*, *Var*)).
reduce_scope_of_negation(**binarypredicate**(*Predicate*, *Var1*, *Var2*),
binarypredicate(*Predicate*, *Var1*, *Var2*)).
reduce_scope_of_negation(**attribute**(*Att*, *C*, *Val*, *Var*), **attribute**(*Att*, *C*, *Val*, *Var*)).

---

rewrite_existential_quantifiers(+*PredExp*, −*PredExp*)

A function replacing existential quantifiers by functions.

rewrite_existential_quantifiers(*Exp*, *Res*) ←
  remove_existential_quantifiers(*Exp*, [ ], _, 0, _, *List*, *Dexp*),
  collect_functions(*Dexp*, *List*, *Res*).

---

collect_functions(+*PredExp*, +*PredExpList*, −*PredExpList*)

appends the Predicate expression as an element in the list, returning a new list

collect_functions(*Exp1*, *List*, *Res*) ←
  append([**leftimp**(**error**, *Exp1*)], *List*, *Res*).

---

remove_existential_quantifiers

(+PredExp,+VarListIn,-VarListOut,+FuncID1,-FuncID2,-Predexp1,-PredExpList)
Removes existential quantifiers, by breaking existentially quantified parts out in separate clauses.
The VarList's are used to keep variables which should be used in the function argument. The FuncID's are used to lookup which function number should be used,
and the result is both one clause, corresponding to the first clause, and a number of extra clauses

remove_existential_quantifiers(**exists**(_, **and**(**relation**(*Name*, *Var*, *Anothervar*), *Exp*)),
AttIn,
AttIn,
FuncID,
ResID,
ExtraFunctions, **function**(*NewFuncID*, *AttList*)) ←
  NewFuncID **is** *FuncID* + 1,
  find_attribute_variables(*Var*, *AttIn*, *AttList*),
  remove_existential_quantifiers(**and**(**relation**(*Name*, *Var*, *Anothervar*), *Exp*), [ ], _,
  NewFuncID, *ResID*, *ExtraExtraFunction*, *IRes*),
  append([**leftimp**(**function**(*NewFuncID*, *AttList*), *IRes*)],
  ExtraExtraFunction, *ExtraFunctions*), **!**.

remove_existential_quantifiers(**exists**(_, **and**(*Exp*, **neg**(**relation**(*Name*, *Var*, *Anothervar*)))),
AttIn,
AttIn,
FuncID,

```
ResID,
ExtraFunctions, function(NewFuncID, AttList)) ←
    NewFuncID is FuncID + 1,
    find_attribute_variables( Var, AttIn, AttList),
    remove_existential_quantifiers(and(Exp, neg(relation(Name, Var, Anothervar))), [], _,
    NewFuncID, ResID, ExtraExtraFunction, IRes),
    append([leftimp(function(NewFuncID, AttList), IRes)],
    ExtraExtraFunction, ExtraFunctions), !.

remove_existential_quantifiers(and(Exp1, Exp2), AttIn, AttOut, FuncID,
FuncID2, Functions, and( Texp1, Texp2)) ←
    remove_existential_quantifiers(Exp1, AttIn, AttInt, FuncID, FuncID1,
    Functions1, Texp1),
    remove_existential_quantifiers(Exp2, AttInt, AttOut, FuncID1, FuncID2,
    Functions2, Texp2),
    append(Functions1, Functions2, Functions).

remove_existential_quantifiers(or(Exp1, Exp2), AttIn, AttOut, FuncID, FuncID2,
Functions, or( Texp1, Texp2)) ←
    remove_existential_quantifiers(Exp1, AttIn, AttInt, FuncID, FuncID1,
    Functions1, Texp1),
    remove_existential_quantifiers(Exp2, AttInt, AttOut, FuncID1, FuncID2,
    Functions2, Texp2),
    append(Functions1, Functions2, Functions).

remove_existential_quantifiers(neg(Exp1), AttIn, AttOut, ID1, ID2, Functions,
neg( Texp1)) ←
    remove_existential_quantifiers(Exp1, AttIn, AttOut, ID1, ID2, Functions, Texp1).

remove_existential_quantifiers(relation(Relation, Var1, Var2), Att, Att, ID, ID, [],
relation(Relation, Var1, Var2)).
remove_existential_quantifiers(class(Class, Var), Att, Att, ID, ID, [],
class( Class, Var)).
remove_existential_quantifiers(varclass( Class, Var), Att, Att, ID, ID, [],
varclass( Class, Var)).
remove_existential_quantifiers(unarypredicate(Predicate, Var), Att, Att, ID, ID, [],
unarypredicate(Predicate, Var)).
remove_existential_quantifiers(binarypredicate(Predicate, Var1, Var2), Att, Att,
ID, ID, [],
binarypredicate(Predicate, Var1, Var2)).
remove_existential_quantifiers(attribute(Att, C, Val, Var), AttIn, AttNew, ID, ID, [],
attribute(Att, C, Val, Var)) ←
    is_ud_variable( Val), append(AttIn, [[Var, Val]], AttNew), !.
remove_existential_quantifiers(attribute(Att, C, Val, Var), AttIn, AttIn, ID, ID, [],
attribute(Att, C, Val, Var)).
```

---

find_attribute_variables(+*Var*, +*VarMap*, −*VarList*)

Given a VarMap and a Variable it returns a list of the variable and any variables in the VarMap

find_attribute_variables(*Var*, [ ], [*Var*]).

find_attribute_variables(*Var*, [[*Var*, *Val*]|*Rest*], [*Val*|*Res*]) ←
  find_attribute_variables(*Var*, *Rest*, *Res*), **!**.

find_attribute_variables(*Var*, [[_, _]|*Rest*], *Res*) ←
  find_attribute_variables(*Var*, *Rest*, *Res*).

---

remove_outer_universal_quantifers(+*PredExp*, −*PredExp*)

Removes outer universal quantifiers.

remove_outer_universal_quantifiers(**all**(_, *Exp*), *Result*) ←
  remove_outer_universal_quantifiers(*Exp*, *Result*), **!**.
remove_outer_universal_quantifiers(*Exp*, *Exp*).

/ **

Aux. Functions

* */

%%
% VarMaps Aux. Functions
%%

---

substract_varclassmap(+*PredExp*, −*VarList*)

Substracts all variables and their classes from a predicate logic expression and at them to a list which is output.

substract_varclassmap(**neg**(*X*), *Res*) ←
  substract_varclassmap(*X*, *Res*).

substract_varclassmap(**or**(*X*, *Y*), *Res*) ←
  substract_varclassmap(*X*, *Res1*),
  substract_varclassmap(*Y*, *Res2*),
  merge_set(*Res1*, *Res2*, *Res*).

substract_varclassmap(**and**(*X*, *Y*), *Res*) ←
  substract_varclassmap(*X*, *Res1*),
  substract_varclassmap(*Y*, *Res2*),
  merge_set(*Res1*, *Res2*, *Res*).

substract_varclassmap(**imp**(*X*, *Y*), *Res*) ←
  substract_varclassmap(*X*, *Res1*),
  substract_varclassmap(*Y*, *Res2*),
  merge_set(*Res1*, *Res2*, *Res*).

substract_varclassmap(**exists**(_, *X*), *Res*) ←
  substract_varclassmap(*X*, *Res*).

substract_varclassmap(**numexists**(_, _, _, *X*), *Res*) ←
  substract_varclassmap(*X*, *Res*).

substract_varclassmap(**all**(_, *X*), *Res*) ←
  substract_varclassmap(*X*, *Res*).

substract_varclassmap(**varclass**(*Class*, *Var*), [[**class**, *Class*, *Var*]]).

substract_varclassmap(**class**(*Class*, *Var*), [[**class**, *Class*, *Var*]]).

substract_varclassmap(**attribute**(*Att*, **eq**, *Var*, _), [[**attribute**, *Att*, *Var*]]) ←
  is_ud_variable(*Var*), **!**.
substract_varclassmap(**attribute**(_, _, _, _), []).
substract_varclassmap(**relation**(_, _, _), []).
substract_varclassmap(**unarypredicate**(_, _), []).
substract_varclassmap(**binarypredicate**(_, _, _), []).

---

substract_userdefined_varmaps(+*PredExp*, −*VarList*).

Substracts all userdefined variables in the Predicate expression and returns them in the variable list

substract_userdefined_varmaps(*Exp*, *Res*) ←
  substract_varclassmap(*Exp*, *Map*),
  substract_ud_varmap(*Map*, *Res*).

---

substract_ud_varmap(+*VarMap1*, −*VarMap2*).

Returns every element from VarMap1 which has a userdefined variable in VarMap2

substract_ud_varmap([], []).
substract_ud_varmap([[*Type*, *Class*, *Var*]|*Rest*], [[*Type*, *Class*, *Var*]|*Res*]) ←
  is_ud_variable(*Var*),
  substract_ud_varmap(*Rest*, *Res*), **!**.
substract_ud_varmap([[_, _, _]|*Rest*], *Res*) ←
  substract_ud_varmap(*Rest*, *Res*).

---

substract_ud_vars(+*VarMap*, −*VarList*).

Returns every Variable from VarMap which has a userdefined variable in the list VarList.

substract_ud_vars([], []).
substract_ud_vars([[_, _, *Var*]|*Rest*], [*Var*|*Res*]) ←
  is_ud_variable(*Var*),
  substract_ud_vars(*Rest*, *Res*), **!**.
substract_ud_vars([[_, _, _]|*Rest*], *Res*) ←
  substract_ud_vars(*Rest*, *Res*).

---

remove_from_varmap(+*Class*, +*Var*, +*VarMap1*, −*VarMap2*).

Removes the [[class,Class,Var]] element from VarMap1 and returns it in VarMap2

remove_from_varmap(_, _, [], []).
remove_from_varmap(*Class*, *Var*, [[**class**, *Class*, *Var*]|*Rest*], *Rest*) ← **!**.
remove_from_varmap(*Class*, *Var*, [[*T2*, *C2*, *V2*]|*Rest*], [[*T2*, *C2*, *V2*]|*Result*]) ←
  remove_from_varmap(*Class*, *Var*, *Rest*, *Result*).

---

is_in_varmap(+*Class*, +*Var*, +*VarMap*).

Suceeds if the element [class,Class,Var] is in the VarMap.

is_in_varmap(*Class*, *Var*, [[**class**, *Class*, *Var*]|_]).
is_in_varmap(*Class*, *Var*, [[_, _, _]|*Rest*]) ←
  is_in_varmap(*Class*, *Var*, *Rest*).

---

lookup_class(?*Class*, ?*Var*, +*VarMap*)

looks up a class or a variable in the given variable list

lookup_class(*Class*, *Var*, [[*Var*, *Class*]]).
lookup_class(*Class*, *Var*, [[*Var*, *Class*]|_]).
lookup_class(*Class*, *Var*, [[_, _]|*Rest*]) ←
  lookup_class(*Class*, *Var*, *Rest*).

---

lookup_class_label(?*Class*, ?*Var*, ?*Label*, +*VarLabelMap*)

looks up a class or a variable in the given VarClassMap / VarLabelMap.

lookup_class_label(*Class*, *Var*, *Label*, [[*Var*, *Class*, *Label*]]).
lookup_class_label(*Class*, *Var*, *Label*, [[*Var*, *Class*, *Label*]|_]).
lookup_class_label(*Class*, *Var*, *Label*, [[_, _, _]|*Rest*]) ←
  lookup_class_label(*Class*, *Var*, *Label*, *Rest*).

---

exchange_expression(Var, To, TLabel, From, FLabel, VarMap, NVarMap)

Exchanges the [Var,To,TLabel] to [Var,From,FLabel] in the VarMap and returns it in NVarMap.

exchange_expression(*Var*, *To*, *ToLabel*, *From*, *FromLabel*, [[*Var*, *From*, *FromLabel*]|*Rest*], [[*Var*, *To*, *ToLabel*]|*Rest*]).

exchange_expression(*Var*, *To*, *ToLabel*, *From*, *FromLabel*, [[*A*, *B*, *C*]|*Rest*], [[*A*, *B*, *C*]|*Result*]) ←
  exchange_expression(*Var*, *To*, *ToLabel*, *From*, *FromLabel*, *Rest*, *Result*).

%%
% Conceptual Model Aux. Functions
%%

---

is_relation(+Expression)

Succeeds if the expression is a relation in the conceptual model

is_relation(*Relation*) ←
  relation(_, *Relation*, _).

---

is_related(+Expression)

checks isa as well Succeeds if the expression is a relation in the conceptual model

is_related(*Class1*, *Relation*, *Class2*, *SuperClass*) ←
  isa(*SuperClass*, *List*),
  member(*Class1*, *List*),
  relation(*SuperClass*, *Relation*, *Class2*).

is_related(*Class1*, *Relation*, *Class2*, *SuperClass1*) ←
  isa(*SuperClass*, *List*),
  member(*Class1*, *List*),
  is_related(*SuperClass*, *Relation*, *Class2*, *SuperClass1*).

is_related(*Class1*, *Relation*, *Class2*, *SuperClass*) ←
  isa(*SuperClass*, *List*),
  member(*Class2*, *List*),
  relation(*Class1*, *Relation*, *SuperClass*).

is_related(*Class1*, *Relation*, *Class2*, *SuperClass1*) ←
  isa(*SuperClass*, *List*),
  member(*Class2*, *List*),
  is_related(*Class1*, *Relation*, *SuperClass*, *SuperClass1*).

---

is_class(+*Expression*)

Succeeds if the expression is a class in the conceptual model

is_class(*Class*) ←
  class(*Class*).

---

has_attribute(*Class*, *Attribute*)

checks in isa hierachies as well-.

has_attribute(*Class*, *ActualAtt*) ←
  classmap(*Class*, *Table*, _, _),
  table(*Table*, *Rows*),
  member(*ActualAtt*, *Rows*), **!**.
has_attribute(*Class*, *Attribute*) ←
  isa(*AnotherClass*, *Children*),
  member(*Class*, *Children*),
  has_attribute(*AnotherClass*, *Attribute*).

---

is_attribute(+*Expression*)

Succeeds if the expression is a attribute in the conceptual model

is_attribute(*Class*, *Attribute*) ←
  valuemap(*Class*, *Attribute*, _, *ActualAtt*),
  classmap(*Class*, *Table*, _, _),
  table(*Table*, *Rows*),
  member(*ActualAtt*, *Rows*).

---

is_attribute_within_boundries(+*Expression*)

Succeeds if the attribute value is within the defined boundries

is_attribute_within_boundries(*Class*, *Attribute*, _) ←

valuemap(*Class*, *Attribute*, *Type*, _),
type(*Type*, **stringtype**).

is_attribute_within_boundries(*Class*, *Attribute*, *Value*) ←
valuemap(*Class*, *Attribute*, *Type*, _),
type(*Type*, **int**), **integer**(*Value*).

is_attribute_within_boundries(*Class*, *Attribute*, *Value*) ←
valuemap(*Class*, *Attribute*, *Type*, _),
type(*Type*, *RealType*), **member**(*Value*, *RealType*).

---

is_predicate(+*Expression*)

Succeeds if the expression is a predicate in the conceptual model

is_predicate(*Predicate*) ←
sqlfunction(*Predicate*, _, _).

---

get_relation_column_names(+*Expression*, −*Columnlist*, −*Columnlist*)

Returns the columns for a given relation

get_relation_column_names(*Relation*, *Names1*, *Names2*) ←
relmap(_, *Relation*, _, _, *Names1*, *Names2*).

---

get_class_column_names(+*Expression*, −*Columnlist*,)

Returns the columns for a given class

get_class_column_names(*Class*, *Names*) ← **classmap**(*Class*, _, _, *Names*).

---

is_ud_variable(+*Var*).

Succeeds if Var is a userdefined variable.

is_ud_variable(X) ← **userDefVariable**(X).

---

is_sys_variable(+*Var*).

Succeeds if Var is a system variable.

is_sys_variable(**v**(0)).
is_sys_variable(**v**(X)) ← **is_sys_variable**(X).

---

find_class(+*Var*, +*PredExp1*, −*ClassExp*).

Finds the expression class(X) in C and returns class. Note it always returns class, even on a varclass

find_class(*Var*, **neg**(*Exp*), *Res*) ←
  find_class(*Var*, *Exp*, *Res*).
find_class(*Var*, **or**(_, *Exp*), *Res*) ←
  find_class(*Var*, *Exp*, *Res*).
find_class(*Var*, **or**(*Exp*, _), *Res*) ←
  find_class(*Var*, *Exp*, *Res*).
find_class(*Var*, **and**(_, *Exp*), *Res*) ←
  find_class(*Var*, *Exp*, *Res*).
find_class(*Var*, **and**(*Exp*, _), *Res*) ←
  find_class(*Var*, *Exp*, *Res*).
find_class(*Var*, **imp**(_, *Exp*), *Res*) ←
  find_class(*Var*, *Exp*, *Res*).
find_class(*Var*, **imp**(*Exp*, _), *Res*) ←
  find_class(*Var*, *Exp*, *Res*).
find_class(*Var*, **all**(_, *Exp*), *Res*) ←
  find_class(*Var*, *Exp*, *Res*).
find_class(*Var*, **class**(*Class*, *Var*), **class**(*Class*, *Var*)).
find_class(*Var*, **class**(*Class*, *Var*, *VList*), **class**(*Class*, *Var*, *VList*)).
find_class(*Var*, **varclass**(*Class*, *Var*), **class**(*Class*, *Var*)).

/**

I/*O* − **functions** (**mostly output actually**)

**/

---

print_hlcl(+*HLCLExp*).

"Pretty prints" the HLCL expression to the Console.

print_hlcl([]).
print_hlcl([*Head*|*Tail*]) ←
  print(*Head*), **print**('␣'), **print_hlcl**(*Tail*).

print_hlcl_tex([]).
print_hlcl_tex([*Head*|*Tail*]) ←
  print_var(*Head*), **print**('␣'), **print_hlcl_tex**(*Tail*), **!**.
print_hlcl_tex([*Head*|*Tail*]) ←
  print(*Head*), **print**('␣'), **print_hlcl_tex**(*Tail*).

---

print_sql(+*SQLExp*).

"Pretty prints" the SQL expression to the Console.

```
print_sql(selectstat(X, Y, L, Z)) ←
   print('SELECT␣'), print(X), print('\n'), print('FROM␣'),
   print(Y), print('␣'), print(L), print('\n'), print('WHERE␣'), print_sql(Z), !.
print_sql(notstat(X)) ←
   print('NOT␣('), print_sql(X), print(')'), !.
print_sql(neg(X)) ←
   print('NOT␣('), print_sql(X), print(')'), !.
print_sql(and([], Y)) ← print_sql(Y), !.
print_sql(and(X, [])) ← print_sql(X), !.
print_sql(and(X, Y)) ←
   print_sql(X), print('␣AND␣'), print_sql(Y), !.
print_sql(or(X, Y)) ←
   print_sql(X), print('␣OR␣'), print_sql(Y), !.
print_sql(exists(X)) ←
   print('EXISTS(␣\n'), print_sql(X), print(')'), !.
print_sql(equals(X, Y)) ←
   print('('), print_sql(X), print('␣=␣'), print_sql(Y), print(')'), !.
print_sql(notequals(X, Y)) ←
   print('('), print_sql(X), print('␣!=␣'), print_sql(Y), print(')'), !.
print_sql(comparison(eq, X, Y)) ←
   print('('), print_sql(X), print(')'), print('␣=␣'), print_sql(Y), !.
print_sql(comparison(le, X, Y)) ←
   print('('), print_sql(X), print(')'), print('␣<=␣'), print_sql(Y), !.
print_sql(comparison(ge, X, Y)) ←
   print('('), print_sql(X), print(')'), print('␣=>␣'), print_sql(Y), !.
print_sql(function(Name, [(Label, Att)])) ←
   print_sql(Name), print('('), print(Label), print('.'), print(Att), print(')'), !.
print_sql(function(Name, [(Label, Att), (Label2, Att2)])) ←
   print_sql(Name), print('('), print(Label), print('.'), print(Att), print(','),
   print(Label2), print('.'), print(Att2), print(')'), !.
print_sql((Class, Attribute)) ←
   print(Class), print('.'), print(Attribute), !.
print_sql(Int) ← integer(Int), print(Int), !.
print_sql(String) ← print('\''),print(String),print('\'').
```

---

print_pred(+*PredExp*).

"Pretty prints" the predicate expression to the Console.

```
print_pred(all(X, Y)) ←
```

print('ALL␣'), **print_pred**(*X*), **print**('␣('), **print_pred**(*Y*), **print**(')'), **!**.
print_pred(**exists**(*X*, *Y*)) ←
print('EXISTS␣'), **print_pred**(*X*), **print**('␣('), **print_pred**(*Y*), **print**(')'), **!**.
print_pred(**numexists**(*Type*, *No*, *Var*, *Exp*)) ←
print('EXISTS␣'), **print**(*Type*), **print**('␣'), **print**(*No*), **print**('␣'), **print_pred**(*Var*),
print('␣('), **print_pred**(*Exp*), **print**(')'), **!**.
print_pred(**exists**(*Type*, *No*, *Var*, *Exp*)) ←
print('EXISTS␣'), **print**(*Type*), **print**('␣'), **print**(*No*), **print**('␣'), **print_pred**(*Var*),
print('␣('), **print_pred**(*Exp*), **print**(')'), **!**.
print_pred(**and**(*X*, *Y*)) ←
print_pred(*X*), **print**('␣AND␣'), **print_pred**(*Y*), **!**.
print_pred(**or**(*X*, *Y*)) ←
print_pred(*X*), **print**('␣OR␣'), **print_pred**(*Y*), **!**.
print_pred(**imp**(*X*, *Y*)) ←
print_pred(*X*), **print**('␣=>␣'), **print_pred**(*Y*), **!**.
print_pred(**neg**(*X*)) ← **print**('?('), **print_pred**(*X*), **print**(')'), **!**.
print_pred(**attribute**(*Attribute*, _, *Value*, *Var*)) ←
print(*Attribute*), **print**('('), **print_pred**(*Var*), **print**(','), **print**(*Value*), **print**(')').
print_pred(**class**(*Class*, *Var*)) ←
print(*Class*), **print**('('), **print_pred**(*Var*), **print**(')').
print_pred(**varclass**(*Class*, *Var*)) ←
print(*Class*), **print**('('), **print_pred**(*Var*), **print**(')').
print_pred(**relation**(*Class*, *Var1*, *Var2*)) ←
print(*Class*), **print**('('), **print_pred**(*Var1*), **print**(','), **print_pred**(*Var2*), **print**(')').
print_pred(**unarypredicate**(*Predicate*, *Var*)) ←
print(*Predicate*), **print**('('), **print_pred**(*Var*), **print**(')').
print_pred(**binarypredicate**(*Predicate*, *Var1*, *Var2*)) ←
print(*Predicate*), **print**('('), **print_pred**(*Var1*), **print**(','), **print_pred**(*Var2*), **print**(')').
print_pred(*Var*) ← **print_var**(*Var*).

---

print_pred(+*PredExp*).

"Pretty prints" the predicate expression to the Console.

print_pred_tex(**all**(*X*, *Y*)) ←
print('$\\forall$'), **print_pred_tex**(*X*), **print**('␣('), **print_pred_tex**(*Y*), **print**(')'), **!**.
print_pred_tex(**exists**(*X*, *Y*)) ←
print('$\\exists$'), **print_pred_tex**(*X*), **print**('␣('), **print_pred_tex**(*Y*), **print**(')'), **!**.
print_pred_tex(**numexists**(*Type*, *No*, *Var*, *Exp*)) ←
print('$\\exists_{'), **print**(*Type*), **print**('␣'), **print**(*No*), **print**('}$␣'),
print_pred_tex(*Var*), **print**('␣('), **print_pred_tex**(*Exp*), **print**(')'), **!**.
print_pred_tex(**exists**(*Type*, *No*, *Var*, *Exp*)) ←
print('$\\exists_{'), **print**(*Type*), **print**('␣'), **print**(*No*), **print**('}$␣'),
print_pred_tex(*Var*), **print**('␣('), **print_pred_tex**(*Exp*), **print**(')'), **!**.

```
print_pred_tex(and(X, Y)) ←
   print_pred_tex(X), print('␣$\\wedge$␣'), print_pred_tex(Y), !.
print_pred_tex(or(X, Y)) ←
   print_pred_tex(X), print('␣$\\vee$␣'), print_pred_tex(Y), !.
print_pred_tex(imp(X, Y)) ←
   print_pred_tex(X), print('␣$\\rightarrow$␣'), print_pred_tex(Y), !.
print_pred_tex(neg(X)) ← print('$\\neg$('), print_pred_tex(X), print(')'), !.
print_pred_tex(attribute(Attribute, _, Value, Var)) ←
   print(Attribute), print('('), print_pred_tex(Var), print(','),
   print_var(Value), print(')').
print_pred_tex(attribute(Attribute, _, Value, Var)) ←
   print(Attribute), print('('), print_pred_tex(Var), print(','), print(Value), print(')').
print_pred_tex(class(Class, Var)) ←
   print(Class), print('('), print_pred_tex(Var), print(')').
print_pred_tex(varclass(Class, Var)) ←
   print(Class), print('('), print_pred_tex(Var), print(')').
print_pred_tex(relation(Class, Var1, Var2)) ←
   print(Class), print('('), print_pred_tex(Var1), print(','), print_pred_tex(Var2),
   print(')').
print_pred_tex(unarypredicate(Predicate, Var)) ←
   print(Predicate), print('('), print_pred_tex(Var), print(')').
print_pred_tex(binarypredicate(Predicate, Var1, Var2)) ←
   print(Predicate), print('('), print_pred_tex(Var1), print(','),
   print_pred_tex(Var2), print(')').
print_pred_tex(Var) ← print_var(Var).
```

---

print_var(+Var)

"Pretty Prints" the variable to the console.

```
print_var(Var) ← var_no(Var, Int), char_code(Char, Int), print(Char), !.
print_var(UDVar) ← sub_atom(UDVar, 0, 4, 1, var_), sub_atom(UDVar, 4, 1, 0, Res),
   char_code(Res, Int), NewInt is Int - 9, char_code(Var, NewInt), print(Var), !.
```

---

var_no(+Var, -Int)

Returns the ascii integer corresponding to the variable.

```
var_no(v(var_a), 65) ← !.
var_no(v(0), 65) ← !.
var_no(v(X), No) ←
   var_no(X, Int),
   integer(Int), No is Int + 1, !.
```

---

print_dat_list(+*DatListExp*)

"Pretty prints" a series of datalog(neg) clauses

print_dat_list([]).
print_dat_list([*Head*|*Tail*]) ← **print_dat**(*Head*), **nl**, **print_dat_list**(*Tail*).

---

print_dat_list(+*DatListExp*)

"Pretty prints" a series of datalog(neg) clauses

print_dat_list_tex([]).
print_dat_list_tex([*Head*|*Tail*]) ← **write**('\\hspace*{1cm}'),
print_dat_tex(*Head*), **write**('\\\\'), **nl**, **print_dat_list_tex**(*Tail*).

---

print_dat(+*DatExp*).

"Pretty prints" the datalog(neg) expression to the Console.

print_dat(**and**(*X*, *Y*)) ←
print_dat(*X*), **print**('␣AND␣'), **print_dat**(*Y*), **!**.
print_dat(**or**(*X*, *Y*)) ←
print_dat(*X*), **print**('␣OR␣'), **print_dat**(*Y*), **!**.
print_dat(**leftimp**(*X*, *Y*)) ←
print_dat(*X*), **print**('␣<--␣'), **print_dat**(*Y*), **!**.
print_dat(**neg**(*X*)) ← **print**('?'), **print_dat**(*X*), **!**.
print_dat(**function**(*No*, *Var*)) ← **print**('f'), **print**(*No*), **print**('('), **print_list**(*Var*),
print(')'), **!**.
print_dat(**error**) ← **print**('error'), **!**.
print_dat(**attribute**(*Attribute*, **eq**, *Value*, *Var*)) ←
is_ud_variable(*Value*), **print**(*Attribute*), **print**('('), **print_dat**(*Var*), **print**(','),
print_dat(*Value*), **print**(')'), **!**.
print_dat(**attribute**(*Attribute*, **eq**, *Value*, *Var*)) ←
print(*Attribute*), **print**('('), **print_dat**(*Var*), **print**(','), **print**(*Value*), **print**(')'), **!**.
print_dat(**class**(*Class*, *Var*)) ←
print(*Class*), **print**('('), **print_list**(*Var*), **print**(')'), **!**.
print_dat(**varclass**(*Class*, *Var*)) ←
print(*Class*), **print**('('), **print_dat**(*Var*), **print**(')'), **!**.
print_dat(**relation**(*Class*, *Var1*, *Var2*)) ←
print(*Class*), **print**('('), **print_dat**(*Var1*), **print**(','), **print_dat**(*Var2*), **print**(')'), **!**.
print_dat(**unarypredicate**(*Predicate*, *Var*)) ←
print(*Predicate*), **print**('('), **print_dat**(*Var*), **print**(')'), **!**.
print_dat(**binarypredicate**(*Predicate*, *Var1*, *Var2*)) ←
print(*Predicate*), **print**('('), **print_dat**(*Var1*), **print**(','), **print_dat**(*Var2*), **print**(')'), **!**.

print_dat(*Var*) ← **print_var**(*Var*), **!**.
print_dat(*X*) ← **print**(*X*).

---

print_dat_tex(+*DatExp*).

"Pretty prints" for TeX the datalog(neg) expression to the Console.

print_dat_tex(**and**(*X*, *Y*)) ←
  print_dat_tex(*X*), **print**('␣$\\wedge$␣'), **print_dat_tex**(*Y*), **!**.
print_dat_tex(**leftimp**(*X*, *Y*)) ←
  print_dat_tex(*X*), **print**('␣$\\leftarrow$␣'), **print_dat_tex**(*Y*), **!**.
print_dat_tex(**neg**(*X*)) ← **print**('$\\neg$'), **print_dat_tex**(*X*), **!**.
print_dat_tex(**function**(*No*, *Var*)) ← **print**('f'), **print**(*No*), **print**('('),
  print_list(*Var*), **print**(')'), **!**.
print_dat_tex(**error**) ← **print**('error'), **!**.
print_dat_tex(**attribute**(*Attribute*, **eq**, *Value*, *Var*)) ←
  is_ud_variable(*Value*), **print**(*Attribute*), **print**('('), **print_dat_tex**(*Var*),
  print(','), **print_dat_tex**(*Value*), **print**(')'), **!**.
print_dat_tex(**attribute**(*Attribute*, **eq**, *Value*, *Var*)) ←
  print(*Attribute*), **print**('('), **print_dat_tex**(*Var*), **print**(','),
  print(*Value*), **print**(')'), **!**.
print_dat_tex(**class**(*Class*, *Var*)) ←
  print(*Class*), **print**('('), **print_list**(*Var*), **print**(')'), **!**.
print_dat_tex(**varclass**(*Class*, *Var*)) ←
  print(*Class*), **print**('('), **print_dat_tex**(*Var*), **print**(')'), **!**.
print_dat_tex(**relation**(*Class*, *Var1*, *Var2*)) ←
  print(*Class*), **print**('('), **print_dat_tex**(*Var1*), **print**(','),
  print_dat_tex(*Var2*), **print**(')'), **!**.
print_dat_tex(**unarypredicate**(*Predicate*, *Var*)) ←
  print(*Predicate*), **print**('('), **print_dat_tex**(*Var*), **print**(')'), **!**.
print_dat_tex(**binarypredicate**(*Predicate*, *Var1*, *Var2*)) ←
  print(*Predicate*), **print**('('), **print_dat_tex**(*Var1*), **print**(','),
  print_dat_tex(*Var2*), **print**(')'), **!**.
print_dat_tex(*Var*) ← **print_var**(*Var*), **!**.
print_dat_tex(*X*) ← **print**(*X*).

---

print_list(+*List*)

prints a list to the console, ie. [a,b] -¿ a,b

print_list([]) ← **!**.
print_list([*Head*|[]]) ← **print_dat**(*Head*), **!**.
print_list([*Head*|*Tail*]) ← **print_dat**(*Head*), **print**(','), **print_list**(*Tail*), **!**.
print_list(*NoList*) ← **print_dat**(*NoList*).

E.2 User Settings

/**

Title : *A Highlevel* **interface to** *GIS*
Author : *Mads Johnsen*
Last Modified : 15/02/2005

Notes : *Conceptual Model*, **must be imported in application**

**/

/**

User Settings – Database Model

**/

Conceptual model

class(**building**).
class(**area**).
class(**residentialArea**).
class(**industrialArea**).
class(**road**).
class(**company**).
class(**lake**).
class(**house**).

relation(**area**, **touch**, **area**).
relation(**area**, **have**, **house**).
relation(**area**, **contain**, **lake**).
relation(**area**, **zipcode**, **fourdigits**).
relation(**area**, **intersectedby**, **road**).
relation(**area**, **contain**, **building**).
relation(**residentialArea**, **n_ofRes**, **residents**).
relation(**residentialArea**, **type**, **residentialAreaType**).
relation(**industrialArea**, **type**, **industrialAreaType**).

relation(**building**, **touch**, **building**).
relation(**building**, **containedin**, **area**).
relation(**building**, **beusedby**, **company**).
relation(**building**, **type**, **buildingtype**).
relation(**building**, **height**, **height**).
relation(**building**, **numoffloors**, **floors**).

relation(**road**, **isType**, **roadtype**).
relation(**road**, **connected**, **road**).
relation(**road**, **roadname**, **roadname**).

relation(**road**, **intersect**, **building**).
relation(**building**, **intersectedby**, **road**).

isa(**area**, [**residentialArea**, **industrialArea**]).
%isa(residentialArea,[superResidentialArea]).

Database Description

table(**house**, [**houseID**, **geometry**, **owner**]).
table(**area**, [**areaID**, **geometry**, **zipcode**]).
table(**industrialArea**, [**areaID**, **industrial_type**]).
table(**residentialArea**, [**areaID**, **num_of_residents**, **residential_type**]).
table(**building**, [**buildingID**, **height**, **numoffloors**, **geometry**, **building_type**]).
table(**company**, [**companyID**, **company_name**]).
table(**lake**, [**lakeID**, **geometry**]).
table(**road**, [**roadsegmentID**, **geometry**, **road_type**, **rname**]).
table(**relBuildingCompany**, [**buildingID**, **companyID**]).
table(**contain**, [**geoID1**, **geoID2**]).
table(**have**, [**areaID**, **houseID**, **owner**]).
table(**intersect**, [**geoID1**, **geoID2**]).
table(**touch**, [**geoID1**, **geoID2**]).
table(**connected**, [**geoID1**, **geoID2**]).

sqlfunction(**zdifferencelargerthan5**, [[A, **buildingID**], [B, **buildingID**]],
function(**zdiff**, [(A, **buildingID**), (B, **buildingID**)])).
sqlfunction(**zipcodefunc**, [[A, **zipcode**]], **function**(**zipcodefunc**, [(A, **zipcode**)])).
sqlfunction(**binfunc**, [[A, **zipcode**], [B, **zipcode**]],
function(**binfunc**, [(A, **zipcode**), (B, **areaID**)])).

Mapping

classmap(**building**, **building**, [ ], [**buildingID**]).
classmap(**area**, **area**, [ ], [**areaID**]).
classmap(**house**, **house**, [ ], [**houseID**, **owner**]).
classmap(**residentialArea**, **residentialArea**, [ ], [**areaID**]).
classmap(**industrialArea**, **industrialArea**, [ ], [**areaID**]).
classmap(**road**, **road**, [ ], [**roadsegmentID**]).
classmap(**company**, **company**, [ ], [**companyID**]).
classmap(**lake**, **lake**, [ ], [**lakeID**]).

relmap(**building**, **beusedby**, **company**, **relBuildingCompany**, [**buildingID**], [**companyID**]).
relmap(**area**, **have**, **house**, **have**, [**areaID**], [**houseID**, **owner**]).
relmap(**_**, **contain**, **_**, **contain**, [**geoID1**], [**geoID2**]).

```
relmap(_, containedin, _, contain, [geoID2], [geoID1]).
relmap(_, intersect, _, intersect, [geoID1], [geoID2]).
relmap(_, intersectedby, _, intersect, [geoID2], [geoID1]).
relmap(_, touch, _, contain, [geoID1], [geoID2]).
relmap(_, connected, _, contain, [geoID1], [geoID2]).

valuemap(area, zipcode, fourdigits, zipcode).
valuemap(building, numoffloors, floors, numoffloors).
valuemap(building, height, height, height).
valuemap(road, roadname, roadname, rname).
valuemap(road, isType, roadtype, road_type).
valuemap(building, type, buildingtype, building_type).
valuemap(residentialArea, n_ofRes, residents, num_of_residents).
valuemap(residentialArea, type, residentialAreaType, residential_type).
valuemap(industrialArea, type, industrialAreaType, industrial_type).

type(residentialAreaType, stringtype).
type(industrialAreaType, stringtype).
type(roadname, stringtype).
type(floors, int).
type(height, int).
type(integertype, int).
type(fourdigits, int).
type(roadtype, [minor, major, highway]).
type(buildingtype, [residential, industrial, blockbuilding]).
type(residents, int).

/**********************************************
    User Settings - Variable settings
**********************************************/

userDefVariable(var_a).
userDefVariable(var_b).
userDefVariable(var_c).
userDefVariable(var_d).
userDefVariable(var_e).

/**********************************************
    User Settings - Class Definitions
**********************************************/

definition([resareas], [area, type, residential]).
definition([resbuildareas], [area, type, residential, and, contain, building]).
definition([mads], [area, mads, resareas, residential, and, contain, building]).
definition([definedbuilding], [building, type, residential]).
```

definition([**definedarea**], [**area**, **contain**, **definedbuilding**]).

E.3 Help Functions

/ *

Title : *A Highlevel* **interface to** *GIS*
Author : *Mads Johnsen*
Last Modified : 15/02/2005

Notes : ''*Helper Functions*'' **to userdefined settings** :
Database **model**, **class definitions etc**.

* */

Import User Settings Here

←[**usersettings**]. %Conceptual Model, Variables and Class Definitions

/ *

Functions

* */

---

check_conceptual_model()

Checks to see if the given conceptual model is wellformed. Either error statements or a
"conceptual model is OK" will be printed on screen.

%all classes defined should have a mapping into an existing table
check_conceptual_model ←
class(*Class*), **not**(**classmap**(*Class*, _, _, _)),
write('No␣classmap␣defined␣for␣class␣'), **write**(*Class*), **nl**, **!**.

check_conceptual_model ← %Check that conditions are infact an attribute
class(*Class*), **classmap**(*Class*, *Table*, _, _), **not**(**table**(*Table*, _)),
write('The␣table␣'), **write**(*Table*), **write**('␣does␣not␣exist!'), **nl**, **!**.

%all relations defined should have a mapping into an existing table
check_conceptual_model ←
relation(*C1*, *R*, *C2*), **not**(**check_relation**(*C1*, *R*, *C2*, **ok**)), **write**('relation␣'),
write(*C1*), **write**('␣'), **write**(*R*), **write**('␣'),
write(*C2*), **write**('␣is␣not␣welldefined!'), **!**.

%No of keyconstraints is the same in relation and classes
check_conceptual_model ←
relmap(_, *Relation*, _, _, *RList*, _), **relation**(*Class*, *Relation*, _),
classmap(*Class*, _, _, *CList*), **length**(*RList*, *No*), **not**(**length**(*CList*, *No*)),
write('relation␣'), **write**(*Relation*), **write**('␣and␣class␣'), **write**(*Class*),
write('␣does␣not␣have␣same␣no␣of␣key-attributes'), **!**.

```
%No of keyconstraints is the same in relation and classes
check_conceptual_model ←
   relmap(_, Relation, _, _, _, RList), relation(_, Relation, Class),
   classmap(Class, _, _, CList),          length(RList, No), not(length(CList, No)),
   write('relation '), write(Relation), write(' and class '),
   write(Class), write(' does not have same no of key-attributes'), !.

check_conceptual_model ←     %No of keyconstraints is the same in relation and classes
   write('Conceptual model OK'), nl.
```

---

check_relation(+*ClassExp*, +*RelationExp*, +*ClassExp*, −*Result*)

checks the given Relation to see if it wellformed and welldefined in
the conceptual model. If it is correct,
a "ok" is returned, otherwise a "nok" is returned.

```
check_relation(C1, Att, Values, ok) ←
   classmap(C1, Table, _, _), valuemap(C1, Att, Values, ColumnLabel),
   table(Table, AllAtts), type(Values, _), member(ColumnLabel, AllAtts), !.
check_relation(C1, Relation, C2, ok) ←
   relmap(C1, Relation, C2, Table, _, _), table(Table, _), !.
check_relation(_, _, _, nok).
```

E.4 Test Functions

```
/*******************************************
    Title : A Highlevel interface to GIS
    Author : Mads Johnsen
    Last Modified : 29/03/2005

    Notes : Testfunctions and Testcases
*******************************************/
←[hlcl].
/*******************************************
    TESTS & Test Functions
*******************************************/
```

---

run_tests(+TestMap)

Runs the tests in the TestMap and prints results in console

```
run_tests([]).
run_tests([[Head, Description]|Tail]) ←
hlcl_to_pred(Head, X), check_wellformedness(X),
perform_intermediate_steps(X, Y),
pred_to_datalog(Y, Q), subpred_to_sql(Y, Z, [], [], _, [], nocount),
print('****************************************\n\n'),
print_hlcl(Head), print('\n\n'),
print_hlcl(Description), print('\n\n'),
print_pred(X), print('\n'),                        %pred. logic
print_pred(Y), print('\n\n'),              %Intermediate Steps
print('DATALOG:\n\n'), print_dat_list(Q), print('\n\n'),
print('SQL:\n\n'), print_sql(Z), print('\n\n'),
print('\n\n'),
run_tests(Tail), !.

run_tests([[Head, [Description]]|Tail]) ←
hlcl_to_pred(Head, X), check_wellformedness(X),
print('****************************************\n\n'),
print_hlcl(Head), print('\n\n'),
print(Description), print('\n\n'),
print('Internal␣Error:␣Intermediate␣Steps␣could␣not␣be␣made!\n\n'),
run_tests(Tail), !.

run_tests([[Head, Description]|Tail]) ←
hlcl_to_pred(Head, X),
print('****************************************\n\n'),
print_hlcl(Head), print('\n\n'),
```

print(*Description*), **print**('\n\n'),
print_pred(*X*), **print**('HLCL␣is␣not␣wellformed!\n\n'),
run_tests(*Tail*), **!**.

run_tests([[*Head*, *Description*]|*Tail*]) ←
print('**\n\n'),
print_hlcl(*Head*), **print**('\n\n'),
print(*Description*), **print**('\n\n'),
print('HLCL␣could␣not␣be␣parsed!\n\n'),
run_tests(*Tail*), **!**.

---

run_tests_tex(+*TestMap*)

Runs the tests in the TestMap and prints results in console in TeX format

run_tests_tex([], _).
run_tests_tex([[*Head*, [*Description*]]|*Tail*], *Int*) ←
hlcl_to_pred(*Head*, *Pred*), **check_wellformedness**(*Pred*),
perform_intermediate_steps(*Pred*, *PredInt*),
subpred_to_sql(*PredInt*, *SQL*, [], [], _, [], **nocount**), **pred_to_datalog**(*PredInt*, *Dat*),
print('\\subsubsection{Testcase␣'), **print**(*Int*), **print**('}'), **nl**,
print('\\emph{'), **print**(*Description*), **print**('}\\\\'), **nl**,
print('\\begin{description}\\item[{\\sc␣hlcl}]'),
print('\\texttt{'), **print_hlcl_tex**(*Head*), **print**('}\\end{description}'), **nl**,
print('\\begin{description}\\item[Pred.␣Logic]'),
print('\\emph{'), **print_pred_tex**(*Pred*), **print**('}\\end{description}'), **nl**,
print('\\begin{description}\\item[Ext.␣Dat]'),
print('\\emph{'), **print_pred_tex**(*PredInt*), **print**('}\\end{description}'), **nl**,
print('\\textbf{SQL:}\\\\'),
print('\\begin{verbatim}'), **nl**, **print_sql**(*SQL*), **nl**, **print**('\\end{verbatim}'),
nl, **nl**, **nl**,
print('{\\sc␣datalog}\\ensuremath{^{\\neg}}:\\\\'), **nl**,
print_dat_list_tex(*Dat*), **nl**,
NextInt **is** *Int* + 1, **run_tests_tex**(*Tail*, *NextInt*), **!**.

run_tests_tex([[*Head*, [*Description*]]|*Next*], *Int*) ←
print('\\subsubsection{Testcase␣'), **print**(*Int*), **print**('}'), **nl**,
print('\\emph{'), **print**(*Description*), **print**('}\\\\'), **nl**,
print('\\begin{description}\\item[{\\sc␣hlcl}]'),
print('\\texttt{'), **print_hlcl_tex**(*Head*), **print**('}\\end{description}'), **nl**,
print('\\textbf{Failed}'), **nl**, **nl**, *NextInt* **is** *Int* + 1, **run_tests_tex**(*Next*, *NextInt*).

print_test_table([], _).
print_test_table([[*Head*, [*Description*]]|*Tail*], *Int*) ←
print(*Int*), **print**('&␣\\texttt{'), **print_hlcl_tex**(*Head*), **print**('}␣&␣'),

print(*Description*),
print(`'&'`), **print**(`'$\\surd$'`), **print**(`'\\\\'`), **nl**, **print**(`'\\hline'`), **nl**,
NextInt **is** *Int* + 1,
print_test_table(*Tail*, *NextInt*).