Uniqueness Constraints in Object-Relational Databases and Description Logics

by

Vitaliy Leonidovich Khizder

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 1999

© Vitaliy Leonidovich Khizder 1999

National Library of Canada
Acquisitions and Bibliographic Services
395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale du Canada
Acquisitions et services bibliographiques
395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-51204-5

Canada

The University of Waterloo requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

Abstract

We address problems that arise in many areas of information technology. In particular, our work considers how to effectively represent semantic constraints commonly used in object-relational database systems and then develops efficient algorithms to reason about such constraints. We expand data dependencies that are commonly used by relational models and combine them with constraints arising in object-relational models to form a theory that allows one to reason about both kinds of constraints. We then present procedures that can efficiently reason in such a theory. The procedures can be used to help solve problems relating to both object-relational and relational databases. A fundamental reason that we are able to derive such procedures relates to the variety of *uniqueness constraints* incorporated into the theory, which strictly generalize the notions of keys and functional dependencies that are inherent parts of relational and object-relational models.

We investigate the interaction between such constraints and other data dependencies, including inheritance, typing and equational constraints. In addition, the problem is explored in the context of *description logics* (DLs), which are a family of knowledge representation schemas that have found myriad applications in information systems technology. From this perspective, we introduce a new **fd** concept constructor for capturing uniqueness constraints within the DL framework, show how various DL dialects that include this new constructor can be used to address problems in information technology and present efficient decision procedures for subsumption checking in these dialects.

Among other contributions, we further analyze our procedures with respect to their generality, incremental abilities, and other characteristics.

Acknowledgements

I would like to sincerely thank my supervisor, Grant Weddell, for his incredible guidance and support, always having time for me and readily providing his technical expertise, extraordinary patience in ALL situations, great sense of humor, invaluable ability to calm me down whenever it had to be done, and many other qualities that amaze me to this day and that have made him a better supervisor than I could ever hope for. Not only has the successful completion of this work become possible due to his supervision, he is the first person to thank for making my program at the University of Waterloo a very enjoyable experience.

I am very grateful to my other committee members, Ken Salem, David Toman, Ric Holt, and Stan Burris, for their efforts, time and valuable suggestions. I would like to also thank my external examiner, Professor Maurizio Lenzerini, for his thorough examination of this work and his support.

In addition, I would like to express my gratitude to Alex Borgida and Grant Weddell for the original inspiration of this work.

I would like to thank many people I have met during my stay at Waterloo for their help, enjoyable discussions and just some good times. In particular, I want to thank Dmitri Kuznetsov, Peter Yang, Tim Snider, Konstantin Chumak, Oleg Shatalov, Alexei Kaltchenko, Petrus Chan, Khuzaima Daudjee, Vlado Keselj, Huizhu Liu, Ian Davis, Paul Ward, and my officemates, Stella Lai and Qing Zhang.

I owe special thanks to one of my best friends, Francis G. Wolff, not only for his priceless friendship but also for his invaluable help at getting me into the University of Waterloo in the first place.

Above all, I would like to express my love and deepest thanks to my parents, Leonid and Margarita Khizder, and my sister, Svetlana, who have loved and supported me throughout my life.

Contents

1 Introduction **1**

1.1 General Picture 1

1.2 Motivation 2

1.3 Contributions 6

1.4 Thesis Outline 8

2 Related Work **10**

2.1 Semantic Reasoning 10

2.1.1 Constraints in the Relational Model 10

2.1.2 Constraints in Object-Relational Models 13

2.2 Description Logics 14

2.2.1 Expressiveness vs. Tractability 15

2.2.2 Terminologies and the Logical Implication Problem 18

2.2.3 Description Logics in Databases 20

2.3 Other Related Work 21

3 Definitions **23**

3.1 Descriptions and their Semantics 23

3.2 Terminology 25

3.3 On Views 26

3.4 On Path Functions 27

3.5 Logical Implication 30
3.6 Description Graphs 32

4 Reasoning with Symmetric Uniqueness Constraints **33**
4.1 Problem Definition 33
4.2 Mapping to Atomic Terminologies 34
4.3 Further Applications 35
4.4 Acceptor Trees and Procedure *Prop* 38
4.5 Analysis 42
4.5.1 Termination 43
4.5.2 Soundness 45
4.5.3 Completeness 50
4.5.4 Runtime Complexity 59
4.6 Generality 63
4.7 On Schema Analysis: Diagnosing Object Normal Form 64

5 On General Logical Implication Problems **68**
5.1 Problem Definition 68
5.1.1 Descriptions 68
5.1.2 An Example Terminology 69
5.1.3 Applications in Query Optimization 71
5.2 Mapping to Molecular Terminologies 74
5.3 Inference Axioms 75
5.4 Description Graphs and Paths 79
5.5 Procedures 82
5.5.1 Procedure *Merge* 82
5.5.2 Procedure *Exp* 84
5.5.3 Procedure *Subsumes* 88
5.6 Analytic results 89
5.6.1 Termination 90

5.6.2 Soundness 99
5.6.3 Completeness 126
5.6.4 Runtime complexity 138
5.7 Integration 146
5.7.1 Inter Procedure Reuse 146
5.7.2 Intra Procedure Reuse 146

6 Conclusion **149**
6.1 Summary 149
6.2 Extensions 151
6.2.1 Procedure *Accepts* 151
6.2.2 THING 152
6.2.3 Concept Conjunction 153
6.2.4 Equational Restrictions in Terminology Constraints 155
6.2.5 Roles 158
6.2.6 Extending Uniqueness Constraints 159
6.2.7 Summary of Extensions 160
6.3 Research Directions 162

A Simplifying Terminology for the Membership Problem **164**

B Simplifying Terminology for the General Logical Implication Problem **176**

Bibliography **185**

List of Tables

Table 3.1: Normalizing attribute descriptions. 28

Table 4.1: Atomic simplification of terminology. 35

Table 5.1: Molecular simplification of terminology 75

Table A.1: Atomic simplification of terminology in Chapter 4 (revisited). 165

Table B.1: Molecular simplification of terminology in Chapter 5 (revisited). 177

List of Figures

Figure 3.1: The UNIV terminology. 26

Figure 4.1: The atomic UNIV terminology 36

Figure 4.2: Evaluating *Prop*(*Init*(PROFESSOR, {Name}), UNIV) 41

Figure 4.3: Construction of the interpretation **I** in the proof of completeness. 58

Figure 4.4: The tree used for the algorithm that determines whether a concept C with primitive attributes A_1 through A_n is in ONF. 66

Figure 5.1: Example of a terminology representing a schema. 70

Figure 5.2: The graph resulting after invocation of *Exp*(*Init*(QView), COMPANY') 87

Figure 5.3: Execution of *Subsumes*((**fd** QView e dm d), *Exp*(*Init*(QView)), COMPANY'). 90

Figure 5.4: Graph structures in the proof of termination. 97

Figure 5.5: Some steps in the first construction of the proof of Lemma 5.7 120

Figure 5.6: Some steps in the second construction of the proof of Lemma 5.7. 121

Figure 5.7: An i-th step in construction of the graph in the proof of completeness. 130

Chapter 1

Introduction

1.1 General Picture

Over the past three decades, the need to provide the ability to conveniently capture, manipulate and query data by application programs has led to the development of a pervasive database technology based primarily on an underlying *relational model* [Cod70]. However, two problems remain as the complexity and performance requirements of these programs have grown. There continues to be an increasing demand for richer models that provide additional flexibility for describing and manipulating data, and for underlying query engines capable of more efficient processing of data that can in turn be encoded with a more diverse collection of data structures.

Indeed, there are several ways that the relational model has been extended to address the first of these problems. So-called *object-relational* models, that combine the advantages of *object-oriented* and relational models, are probably the most important example of these. There are at least two basic reasons for the emergence of object-relational models: (a) a richer structural capability that combines the notions of "relation" and "domain" into a more general notion of "class", that allows for relating classes in terms of an inheritance taxonomy, and that recognizes the fundamental notion of object identity; and (b) the development of object-oriented languages by the programming language community that have undergone further adaptation and enhancement in the context of data persistence. Some issues that are resolved by object-relational models include: (a) the necessity of introducing extra properties to serve as surrogate identifiers for "objects" in tables, and (b) the need for *indirect* references to capture relationships between objects.

A major direction in addressing the second problem consists of capturing and reasoning about *data dependencies*. This has proven to be an important capability in information systems technology, with many applications in database design, data integration, query optimization, plan generation, and so on. Historically, data dependencies have been captured by means of

constraints. Relational databases, in particular, have a long history of using *functional* [Cod72] and *inclusion dependencies* [Fag81] in query optimization and in relational schema synthesis and evaluation. However, while *typing* and *inheritance* constraints in *object-relational databases* (ORDBs) appear to be sufficient manifestations of inclusion dependencies in practical applications, the incorporation of functional dependencies or more general kinds of *uniqueness constraints* is still far from its potential. This is exactly the problem addressed in this thesis.

This work can also be viewed in the context of *description logics* (DLs) which are a family of knowledge representation schemas that have found myriad applications in information systems technology [Bor95]. The applications derive from using DLs as a formal means of capturing such artifacts as database schema, views and queries. DLs have their foundations in the field of artificial intelligence, and have been studied for a long time in that context. However, only recently have they been recognized as valuable tools in capturing database constraints.

While a number of DLs have now been explored in the context of databases, very few dialects have considered language constructs that can capture even the notion of a key. Notably, however, a *concept constructor* for functional dependencies has been introduced in a DL called Classic/FD [BW97]. This thesis constitutes a more thorough exploration of this constructor and its interaction with other "foundational" constructors relating, for example, to the above-mentioned facility of ORDBs for expressing typing and inheritance constraints. Thus, on one side, we contribute to the integration of DLs with databases, and on the other, we use the benefits provided by the clear and concise semantics of DLs to study uniqueness constraints in the object-relational context.

1.2 Motivation

To illustrate the utility of uniqueness constraints in an ORDB environment, consider the problem of duplicate elimination in query optimization. In its simplest form, the **distinct** keyword can be safely removed from queries of the form

select distinct $x.A_1, x.A_2, \ldots, x.A_k, \ldots$[1]
from $C\ x, \ldots$
...

if *attributes* A_1 through A_k are known to uniquely determine objects in the class (or *concept*) C. With the identity function, **Id**, one can employ the notation used in [Wed89] to capture such a constraint by writing

[1] We assume that the reader is familiar with the SQL/OQL syntax that we use throughout the thesis for our query examples.

$$C(\{A_1, A_2, \ldots, A_k\} \rightarrow \{\mathbf{Id}\}).$$

This form of uniqueness constraint in an ORDB corresponds to the notion of a *key* constraint in the relational model. However, unlike the relational model, attribute values of objects in the ORDB model are also objects, possibly with their own values. This gives rise to so-called *path functions* that "navigate" through a number of objects and their attributes. For now, one can think of a path function as a sequence of attributes separated by dots. For example, "Boss.Name.Last" could be a path function representing last names of bosses of objects in an EMPLOYEE class. If path functions can be used in place of attributes, uniqueness constraints become more expressive [Wed92]. For example, path functions enable one to capture and reason about such common sense facts as "no student can be enrolled in two different courses that meet at the same time" [BW94].

For some path functions Pf_1 through Pf_k, a key constraint of the form

$$C(\{Pf_1, Pf_2, \ldots, Pf_k\} \rightarrow \{\mathbf{Id}\})$$

justifies removing the **distinct** keyword from queries of the form

select distinct $x.Pf_1, x.Pf_2, \ldots, x.Pf_k, \ldots$
from $C\ x, \ldots$
...

Intuitively, the constraint expresses the fact that there are no two distinct objects in C that "agree" on the values of the path functions Pf_1 through Pf_k; that is, it is *not* the case that the same object is obtained by navigating through the path function Pf_i starting from any two objects, for all $1 \leq i \leq k$. Thus, for example, the **distinct** keyword can be safely removed from

select distinct e.Name.Last, e.Name.First
from EMPLOYEE e, ...
...

if it is true that no two employees have the same first and last names; that is if the constraint

EMPLOYEE({Name.Last, Name.First} → {**Id**}).

is true.

More generally, the constraint

$$C(\{Pf_1, Pf_2, \ldots, Pf_k\} \rightarrow \{Pf_{k+1}, \ldots, Pf_m\})$$

expresses the fact that if any pair of (not necessarily distinct) objects in C agree on the path functions Pf_1 through Pf_k, they must then also agree on the path functions Pf_{k+1} through Pf_m. Such constraints were first defined in [Wed89], and are called *path functional dependencies* (PFDs). A PFD of the form above enables one to remove the **distinct** keyword from a more general family of queries with the form

select distinct $x.Pf_{k+1}$, $x.Pf_{k+2}$, ..., $x.Pf_m$, ...
from C x, ...
where $x.Pf_1 = \text{Param}_1$ **and** ... **and** $x.Pf_k = \text{Param}_k$
...

since there can be at most one result for such queries. In addition, the PFD can also help in obtaining efficient *access plans* for queries that have the structure

select distinct $x.Pf_1$, $x.Pf_2$, ..., $x.Pf_m$
from C x, ...
...

Indeed, one needs only to ensure that values $x.Pf_1$ through $x.Pf_k$ are distinct, since otherwise, the uniqueness constraint would imply that all m values are the same.

Our examples have so far involved uniqueness constraints that are *symmetric* in the sense that the constraints apply to pairs of objects occurring in a common class. This thesis also considers more general *asymmetric uniqueness constraints* that apply to pairs of objects from possibly *distinct* classes. These constraints allow one to capture such facts as "if an employee and a person have the same social security number, they must be the same person" for a company that ensures the statement holds for its employee data but not necessarily for other customer or client data. As will become evident in Chapters 3 and 5, asymmetric uniqueness constraints naturally arise in our framework due to our decision to express constraints in a manner consistent with the idea of concept construction that is fundamental in description logics. In fact, asymmetric uniqueness constraints nicely abstract *inter-relational functional dependencies* [GM91], as well as a form of *coupled functional dependencies* [CK85] and *union functional dependencies* [CV83] in the relational model.

In addition to their use in reasoning about duplicate elimination [Wed92], there are many other problems in query optimization and in *semantic query optimization* [HZ80, Kin81] that benefit from an ability to reason about uniqueness constraints:

- Determining minimal covers of selection and join conditions [ASU79];
- The automatic insertion of "cut" operators [DW89, Men85, MW91, Wed92];

- Order optimization [SSM96]; and
- Enabling the use of so-called path indices [BW97, Wed92].

In addition, there are a number of applications in schema design and evaluation [Ber76, BDB79, TF82]. (Later on, we present an application of our work to a problem in schema evaluation; in particular, we present an efficient algorithm for diagnosing a kind of *object normal form* originally proposed in [Bis89].)

In addition to typing, inheritance and uniqueness constraints, ORDBs also benefit from so-called *equational constraints*. Among their applications, such constraints allow one to capture selection and join conditions occurring inside queries, and are natural representations of restrictions occurring in view definitions [BBM89, BW94, BJN94, BW97].

Overall, this work attempts to combine all four kinds of constraints into the common framework of description logics, and to present efficient procedures to reason about their interaction.

DLs have helped to address many problems in the area of information systems [Bor95]. Among their advantages are a clear and concise semantics and their intuitive capacity for constructing descriptions of sets of objects that commonly occur. Another advantage is a growing body of algorithmic results that relate to the problem of deducing so-called *subsumption relationships* between descriptions. For example, the existence of *efficient* subsumption algorithms has enabled Classic, a DL dialect, to be used for "industrial grade" applications [WWB93]. Among many other applications, DLs have been used for database schema design [BS92], for merging multiple information sources [LSK95], as an abstraction of queries [BGN89] and in query optimization [BJN94].

In this thesis, we use DLs as the framework in which we consider typing, inheritance, equational and uniqueness constraints. Our main objective is to develop efficient procedures for determining *logical consequence*; that is, determining whether a given collection of constraints logically implies another. (We shall refer to the underlying class of problems as *logical implication problems* in the remainder of the thesis.) If indeed this can be resolved efficiently, optimizations that involve such constraints (such as "distinct keyword elimination" outlined above) become practical. In addition, since we use (almost) the same DL to represent schema, query, and view definition languages, such procedures can also be employed to decide relatively general *query containment* problems; that is, in view of a given database schema, to determine if all results of one query must also be returned by another. This in turn suggests a number of additional applications: automatic classification (or taxonomic reasoning), multiple query evaluation, common subexpression analysis and information integration [BS92, MG93, BJN94]. Other researchers have considered the advantages of combining *data definition languages* (DDLs) and *data manipulation languages* (DMLs) as well [BGN89, BBM89]. ([Bor95] provides a more general survey of applications of DLs in databases.)

Note that the DL languages used in this thesis are more limited in their expressiveness than common database query languages such as SQL (a property shared by all tractable DL languages). However, we can still use DLs to reason about the parts of the queries that they can capture [BJN94], a strategy that appears to not "spoil" the utility of DL subsumption algorithms [Bor95].

Finally, although we have given a broad survey of applications for our work, further examples will be presented throughout the remainder of the thesis that will provide additional motivation for our results. As we go along, our definitions of new constructs and, more generally, our discussion of the material in depth enables us to present more detailed examples and to outline applications of this research with greater precision.

1.3 Contributions

Recall from our motivating comments that PFDs have the form

$$C(\{Pf_1, Pf_2, \ldots, Pf_k\} \rightarrow \{Pf_{k+1}, \ldots, Pf_m\})$$

for some class, or concept, C and path functions Pf_i ($1 \leq i \leq m$). The case of a *relational functional dependency* (or *relational uniqueness constraint*) is obtained when each Pf_i corresponds to a single attribute. There are efficient procedures to solve the PFD logical implication problem when every path function on the right-hand-side of the arrow is a "prefix" of some path function on the left-hand-side of the arrow. Such constraints are called *key path functional dependencies* [IW94].

This thesis presents two procedures that allow a more general class of functional dependencies in the database schema. The first procedure solves the logical implication problem in which a schema may consist of typing constraints, inheritance constraints and a variety of PFD constraints that satisfy a regularity condition. We refer to instances of the latter as *symmetric regular uniqueness constraints*; such constraints satisfy a syntactic condition that every path function on the right-hand-side of the arrow, *possibly missing the last attribute*, is a "prefix" of some path function on the left-hand-side of the arrow, and *are a strict generalization of both relational and key path functional dependencies.*

Moreover, we show that this class of uniqueness constraints is "boundary" in the sense that if one allows a functional dependency on the right-hand-side to differ from a prefix of a functional dependency on the left-hand-side by more than one attribute, the resulting logical implication problem becomes equivalent to the more general problem in which a schema may contain *arbitrary* PFDs. (While the complexity of this problem remains open at this time, it *is* known that chase-like procedures analogous to the ones presented in this work can require

exponential time.) Finally, we present proofs of soundness as well as completeness of the procedure, and show that it terminates and runs in time corresponding to the product of the size of schema and the size of *subsumption question* (i.e. the size of deduced constraint).

Our second procedure generalizes the first in a number of ways. First, it allows *asymmetric* uniqueness constraints in the schema. Second, it works with so-called *extended functional dependencies* that allow one to reason about uniqueness constraints on views and queries. And third, the procedure allows equational constraints to also occur in the schema. We show that this procedure is sound and complete, and that it also terminates, but that runtime complexity involves an additional factor over the complexity of our first procedure that roughly corresponds to the size of subsumption question.

Another contribution of this work relates to the integration of description logics and object-relational databases. In particular, we consider the procedures developed in this work and the underlying DL languages to be important contributions to the existing computational theory in description logics. They are among the first of their kind that are able to efficiently accommodate the kinds of uniqueness constraints that occur in virtually every object-relational model and that are of considerable relevance to a large variety of problems in query optimization, view integration and so on.

Also note that, apart from the integration of DLs and ORDBs, the languages presented in this thesis contribute to the study of the important tradeoff between expressiveness and tractability of DL languages, an issue considered in [LB87] and further explored by many researchers. This work also illustrates some advantages of the DL framework: an ability to *choose* appropriate language constructors according to the requirements of the anticipated application, and an ability to introduce new constructors (such as our constructor for uniqueness constraints) to address requirements the application may have that would otherwise necessitate the use of less efficient constructors. In particular, [Bor95, BW97] show how uniqueness constraints can indeed be captured by using existing constructors; however, the presence of these constructors in a DL language would cause logical implication problems to "fall off the computational cliff".

Some other contributions of our work are as follows.

- Both implication problems are solved in the presence of cyclic schemas. (Generally, cyclic schemas are known to greatly increase the difficulty of reasoning about subsumption.)
- Our procedures are incremental in the sense that they can reuse earlier computations when applied to a sequence of similar logical implication problems (a circumstance that we will see often occurs in practice).
- The thesis extends the work on a graphical analogue of descriptions, called *description graphs*, that were first introduced in [BP94].
- We consider a number of ways of mapping a set of constraints that constitute, for example, a database schema to an equivalent set of constraints satisfying desirable syntactic properties.

In particular, we present mappings that simplify a collection of constraints by factoring the use of conjunction, by "normalizing" attribute descriptions, and by simplifying typing constraints. (We believe such mappings and our approach to verifying their correctness can be of use to other procedures that must reason about similar DL languages.)

Finally, note that while this work is presented in the context of ORDBs that deal with complex objects, it is also applicable to the relational model which can clearly be regarded as a special case of object-relational models (e.g. by treating tuples as objects, tables as classes, etc.). Moreover, constraints considered in the thesis include generalizations of most of the constraints that are considered foundational in the relational world. In fact, a number of motivating examples presented later in the thesis are described in the context of the relational model to both underline the above statement and to provide a good illustration of the use of this work in a setting familiar to the relational database community.

1.4 Thesis Outline

The rest of the thesis is organized as follows. Chapter 2 gives an overview of relevant literature. It first considers the research that explores uniqueness, typing and equational constraints in both relational and object-relational models, and then examines research in description logics and other areas. The chapter illustrates how our work fits into the general picture and provides sufficient references for further exploration of the presented areas by the interested reader.

Chapter 3 introduces notation used throughout the rest of the thesis and formally defines the problem of logical implication. It first introduces the essential concepts that underlie description logics, and then defines a core grammar for DL expressions, or *descriptions*, that is adopted in later chapters. The standard model theory for descriptions is presented, and the use of descriptions themselves for capturing database schemas is illustrated. Then, we review some properties of path functions, and introduce the notion of a *description graph* (the computational artifact manipulated by our procedures).

Chapter 4 presents the first of our decision procedures. Recall that the procedure solves a logical implication problem in which a schema may consist of typing constraints, inheritance constraints and a variety of PFD constraints that satisfy a regularity condition. Definitions specific to Chapter 4 are first introduced, followed by the first of our above-mentioned mappings for "normalizing" an input set of constraints in preparation for use by the procedure. The procedure is then defined and its properties analyzed. Following this, we consider the generality of *regular uniqueness constraints*. This chapter also presents a number of examples that serve two purposes: they motivate the results of the chapter, and they help a reader to obtain a more thorough appreciation for what can be accomplished by using the procedure.

Chapter 5 considers our second, more general but slower procedure. Analogously to Chapter 4, relevant definitions are first introduced, followed by the second of our mappings for "normalizing" constraints. Next, we present the second of our decision procedures, including a thorough analysis of its computational properties. Recall that this procedure solves a more general class of logical implication problems. Finally, we discuss the incremental capabilities inherent in both procedures. Again, as in Chapter 4, a number of examples are included to motivate the procedures, and to help with their appreciation.

Finally, Chapter 6 summarizes the main results of the thesis and explores a number of possible extensions of our procedures. It concludes with considerations of future research directions.

Chapter 2

Related Work

We begin with a review of work on so-called *semantic reasoning* that considers constraints in relational and ORDB models. We then give an overview of work in the area of *description logics*, and its applications to problems in information technology. The final section is a brief overview of other related work on functional constraints.

2.1 Semantic Reasoning

2.1.1 Constraints in the Relational Model

There has been a large variety of constraints that have been proposed for capturing data dependencies for the relational model. Perhaps one of the most expressive classes of such constraints, called *algebraic dependencies*, were first introduced in [YP82]. An algebraic dependency is a pair of algebraic queries, and holds for a given database when the first query returns a subset of the second. Such constraints have a close resemblance to so-called *subsumption constraints* in DLs (introduced in the next section), and are able to capture the two most important classes of relational constraints: *relational functional dependencies* (FDs) (first introduced in [Cod72]), and *inclusion dependencies* (INDs) [Fag81].

An FD has the form

$$R: A_1, \ldots, A_m \rightarrow A_{m+1}, \ldots, A_k$$

where R is a relation name and each A_i is an attribute defined on R. For example, one can express the fact that "in an EMPLOYEE relation, an employee age is functionally determined by an employee name" by the FD

EMPLOYEE: Name → Age,

where Name and Age are attributes defined on EMPLOYEE.

An inclusion dependency (IND) has the form

$$R[A_1, \ldots, A_k] \subseteq S[B_1, \ldots, B_k],$$

where R and S are (not necessarily distinct) relation names and each A_i (resp. B_i) is an attribute defined on R (resp. S). The constraint is satisfied by relations R and S if and only if, for any tuple (row) r in R, there is a tuple s in S such that the value of A_i in r is the same as the value of B_i in s, for $1 \leq i \leq k$. In the special case that $k = 1$, the inclusion dependency is called a *unary inclusion dependency* (UIND). For example, the (unary) inclusion dependencies

DEPARTMENT[Head-Of] ⊆ MANAGER[Man-ID], and
MANAGER[Man-ID] ⊆ EMPLOYEE[Emp-ID]

express the facts that "heads of departments are managers", and that "every manager is an employee", respectively. The former UIND is an example of a so-called *foreign key constraint*, perhaps the most common type of INDs used in relational databases. The latter UIND is an example of a particular kind of *referential integrity constraint* that represents inheritance.

A number of results have been discovered about the behavior of FDs and INDs in the relational model. A linear time decision procedure for the FD implication problem is presented in [BB79]. It answers the question of whether an FD holds for a relation that satisfies a number of other (not necessarily distinct) FDs. The procedure is essentially an efficient implementation of the axiomatization of FDs in [Arm74]. An extension of FDs called *inter-relational* functional dependencies (that can be captured using asymmetric uniqueness constraints introduced in this thesis) has a cubic decision procedure [GM91].

While there is also a polynomial (and even linear) decision procedure for the UIND implication problem, the (general) IND implication problem is PSPACE-complete [CFP82, CFP84]. Because of this, researchers have tried to come up with other useful subclasses of INDs. For example, *typed* and *acyclic* INDs have been considered. An IND is *typed* if the sequences of attribute names on the right- and the left-hand-sides of the constraint are the same; that is, a typed IND has the form

$$R[A_1, \ldots, A_k] \subseteq S[A_1, \ldots, A_k],$$

where R and S are relation names and A_i $(1 \leq i \leq k)$ are attribute names. On the other hand, a *set* of INDs is *acyclic* if, informally, no tuple in a relation puts any restriction on any other tuple in the

relation (possibly via a combination of the INDs). It has been established that, while the implication problem for typed INDs can be solved in polynomial time [CV83], the implication problem for acyclic INDs is NP-complete [CK86].

Combinations of FDs and INDs generally complicate things. The implication problem for FDs and typed INDs turns out to be undecidable [CK85]. The problem is decidable in exponential time when we combine FDs with acyclic INDs [CK86], and it is NP-hard for acyclic *and* typed INDs together with FDs [CK86]. The good news, however, is that the implication problem for FDs and UINDs is decidable in linear time when infinite databases are allowed and in cubic time when only finite databases are allowed [KCV83, CKV90]. Moreover, the result is "boundary" in the sense that allowing even two attributes instead of one inside INDs (i.e. allowing *binary* INDs) makes both infinite and finite implication problems undecidable [Mit83].

In general, if only finite databases are allowed, an implication problem is referred to as the *finite logical implication problem*. Otherwise, it is called the *infinite* (or *general*) *implication problem*, or simply the *implication problem*. For many (decidable) implication problems, such as IND or FD implication problems, finite and infinite implication problems coincide. However, it is not the case for implication problems that combine FDs and INDs [CFP82, CFP84]. Moreover, it is also shown in [CFP82, CFP84] that there is no k-ary complete axiomatization (i.e. every rule has at most k antecedents) for both finite and infinite logical implication problems with FDs and INDs combined. The authors show that the same results hold when INDs are restricted to UINDs in the case of the finite implication problem. (There is, however, a binary axiomatization for the infinite implication problem for FDs and UINDs [KCV83].) Non-existence of a k-ary complete axiomatization for finite and infinite problems is also shown for the case when FDs, INDs, *and* so-called *repeating dependencies* (RDs) are allowed. RDs, as well as the generalized types of INDs in [Mit83], were essentially one of the first attempts to express equational constraints in the relational model. Also note that [Mit83] presented a (non-k-ary) complete axiomatization for the general implication problem for FDs and INDs. However, no complete, recursively enumerable axiomatization exists for the finite problem.

Thus, much of this earlier work suggests that there is an inherent difficulty with finite implication problems as well as infinite implication problems for the sets of constraints that even "slightly" extend the set of FDs and UINDs. Our work deals exclusively with the infinite implication problem and considers extensions of the above set in the object-relational environment. An overview of some recent results on functional and other dependencies in the object-relational models is presented below.

2.1.2 Constraints in Object-Relational Models

In ORDBs, UINDs are essentially replaced by inheritance and typing constraints, while an ability to express equational constraints in ORDBs then allows one to capture general INDs. Moreover, the natural extension of attributes in the relational model to path functions in object-relational models leads to *path functional dependencies* (PFDs), a strict generalization of functional dependencies.

PFDs were first introduced in [Wed89, Wed92]. The papers consider databases to be directed labeled graphs with nodes denoting objects in a database and edges denoting their attribute values. They show that allowing path functions inside functional dependencies increases the expressiveness of the constraints. They also present an axiomatization for the general PFD implication problem and give an exponential time decision procedure for what we call *simple key PFDs* that have **Id** on the right-hand-side of the dependency (recall that key PFDs only require the right-hand-side to be a "prefix" of a functional dependency on the left-hand-side). The author also presents some evidence that a chase-like procedure to solve an arbitrary PFD implication problem would take at least exponential time. Finally, a limited form of inheritance constraints are also considered in conjunction with PFDs.

[IW94] further explores PFDs, establishing that finite and general implication problems for PFDs are distinct. It also shows that the general implication problem is decidable for arbitrary PFDs and presents a polynomial time procedure for the key PFD general implication problem.

Around the same time, the interaction between PFDs and equational constraints is considered in [BW94]. The authors introduce so-called *path equation constraints* (PEs) that have the form

$$C(Pf_1 = Pf_2),$$

where C is a class and Pf_1 and Pf_2 are path functions. Informally, the constraint is *satisfied* by a database if and only if for every object o in class C, path functions Pf_1 and Pf_2 "lead" from o to the same object.

A finite axiomatization for the arbitrary equational and PFD implication problem is presented in [BW94]. The implication problem for PEs alone is shown to be undecidable; however, a polynomial time decision procedure is presented for the case when PE constraints satisfy a special *stratification* condition (which intuitively limits the "depth" of equational constraints) and when all PFDs are simple key PFDs.

[Bom96] further develops the work in [BW94], presenting a decision procedure for a constraint theory that also includes inheritance, set-valued properties in typing constraints, and an extension of simple key PFDs to key PFDs.

2.2 Description Logics

Description logics are a class of knowledge representation languages that are used to build and reason about sets of individual objects that have the same properties. Such sets are usually called *concepts*, *descriptions*, or *concept descriptions*, and are built out of other *descriptions* that specify the characteristics of the objects in the concept. DLs also capture relationships between objects, and as pointed out in [Bor95], functional relationships are often considered between objects (in particular, when DLs are applied to DBs). Such relationships are called *attributes*. On the other hand, more general set-valued relationships are called *roles*. To illustrate, consider the following description.

(and STUDENT
(all Emergency-Contact STUDENT)
(same-as Office (**comp** Emergency-Contact Office))

The description is composed out of a number of "terms", or member descriptions, using the top-level *conjunction constructor* **and**. STUDENT is a *primitive concept* term describing some "basic" set of objects in a domain. (**all** Emergency-Contact STUDENT) is a *value restriction* description, also known as *type restriction* or *universal quantification description*, that intuitively represents everyone who listed a student as their emergency contact. Finally, the **"same-as"** part, called *equality restriction*, describes everyone who listed their officemate as the emergency contact. Note that it uses *attribute composition* (the **comp** constructor) for combining *attributes*, in this case Emergency-Contact and Office, rather than concept descriptions. Therefore, the top-level description represents "all students who listed other students that are their officemates as the emergency contact".

In general, description logics examine various languages composed of different concept and role/attribute constructors such as those presented above, and study the interaction among them. Other concept constructors that have been explored in the literature are able to capture:

- *disjunction* of concepts (**or**);
- *negation* of primitive or arbitrary concepts (**not**);
- *unqualified existential quantification* on roles that asserts that there is a *role filler* for an object (**some**);
- *qualified existential quantification* on roles that asserts that there is a role filler that also belongs to a given concept (in this case, **some** has an additional argument to specify the concept);

- two concepts that denote all objects in the domain (called THING) or the empty set (called NOTHING);
- number restrictions on the number of role fillers (**at-least** and **at-most**); and
- *enumeration* of objects (**one-of**).

In addition to the above-mentioned role or attribute composition (**comp**), some other role constructors that are explored in the literature include:

- role *intersection* (**androle**);
- *inverse* of a role (**inverse**);
- an *identity* attribute that "points" to the identity of the object (**Id**); and
- *restriction* of roles to point to specific concepts (**restrict**).

By far the most important and studied relationship between descriptions is *subsumption*, which is the DL term for the subset relationship. Much of the research in the DL community explores what we call the *subsumption question* (or *subsumption problem*) of whether one description is subsumed by another in a given DL language. Conversely, the phrase *subsumption constraint* is used to denote an *assertion* that one description is subsumed by another. (Such constraints are further discussed in Subsection 2.2.2 when we introduce the notion of a terminology.)

2.2.1 Expressiveness vs. Tractability

A fundamental tradeoff between expressiveness and tractability of description logics is recognized in [BL84, LB87]. Generally, the more expressive the language is, the more difficult it becomes to reason with that language. As an example, [BL84, LB87] consider two languages called *FL* and FL^-. The first consists of **and**, **all**, and (unqualified) **some** concept constructors, and the **restrict** role constructor. The second has the same concept constructors but no role constructors. The authors show that while the problem of answering subsumption questions is quadratic in FL^-, it is intractable for the *FL* language. In fact, FL^- has become a very influential language due to its polynomial behavior and has been a subset of almost all subsequent DL languages.

Unfortunately, a number of more expressive languages turned out to be intractable. The addition of a "restricted" disjunction constructor, or addition of the (full) negation to FL^- leads to intractability [LS91, DLN91]. [Neb88] reveals the intractability of the DL languages used in the BACK and KANDOR systems. In particular, the paper shows that while extending FL^- either with number restrictions on roles (i.e. adding **at-least** and **at-most** constructors) or with role conjunction (**androle**) does not cause problems, but that adding both makes the subsumption

problem intractable. More generally, the paper establishes that extending FL^- with number restrictions and "the possibility to express that there are a certain number of role-fillers of a certain concept" causes the subsumption problem to be co-NP-hard [Neb88].

[Sch89] shows that the DL language used in a system called KL-ONE, as well as its sublanguage (*ALR*) is undecidable due to the presence of roles inside equality restrictions. In addition to the **same-as** constructor, *ALR* is defined to only have **and** and **all** concept constructors, and the paper also allows role composition inside equality restrictions. A DL ALR^* is also shown to be undecidable even though it restricts all roles, with the possible exception of the first, to be attributes on either the left- or right-hand-side of an equality restriction.

[BGN89] presents the CANDIDE system. As noted in [BBM89], however, the DL used in CANDIDE is intractable due to the arguments presented in [Neb88]. Nevertheless, we mention [BGN89] as one of the (earlier) examples of work that considers applications of a DL language in a database environment. The paper is also one of the first to make a strong argument for the "unification" of data manipulation and data definition languages. It recognizes that merging the two languages produces a uniform view of the data, query and view objects, and allows the database schema to be automatically used for query processing. The paper also introduces defined classes (or views).

Further work in the same direction is presented in [BBM89]. The authors introduce the CLASSIC DL that incorporates the following constructors: **and**, **all**, **at-least**, **at-most**, **same-as** with attributes, **one-of** and some others.[2] [BBM89] also presents a number of advantages of DLs such as CLASSIC in database management, including:

- capturing data schema and updates (as well as automatic propagation of some consequences of updates);
- representing incomplete knowledge by capturing *some* properties of an individual;
- an ability to handle partial answers to queries;
- integrity checking;
- automatic classification of concepts and objects;
- flexible use of defined concepts; and
- integration of DDL and DML.

Also introduced is a valuable notion of a **test** concept constructor which is essentially considered as a primitive concept for the purposes of the subsumption but allows a smooth integration of the DL reasoning systems into the environments where not all of the data dependencies can be captured by a given DL. Unfortunately, while the authors hoped for a polynomial time subsumption algorithm, [LS91] found the language to be intractable due to the enumeration

[2] Equality restriction and type restriction constraints are called *co-reference* and *role-value constraints* respectively in the paper.

constructor. As a result, the semantics of this constructor is modified in [BP94], and it is then proven that the subsumption problem in CLASSIC becomes polynomial.

More specifically, [BP94] first considers a subset of CLASSIC, called Basic CLASSIC, with intersection, typing constraints, number restrictions on roles, and equational constraints on attribute composition. A polynomial time decision procedure for the subsumption problem in Basic CLASSIC is presented along with the notion of *description graphs*. Description graphs, which are extensively used in our work as well, are also discussed separately as an important tool to solve the subsumption problem. Next, the addition of the enumeration concept with special semantics and other concepts of CLASSIC are discussed.

Additional "positive" results are obtained by [DLN91]. The authors consider two description languages, PL_1 and PL_2, that have polynomial time procedures for their subsumption questions. Both PL_1 and PL_2 add inverse roles to FL^-. Furthermore, PL_1 adds number restrictions and the negation of primitive concepts, whereas PL_2 adds role conjunction and role composition constructors. However, the paper also shows that further extensions of these languages with a number of constructors considered in the paper would lead to intractability.

The notion of defined concepts, or views, considered by [BGN89, BBM89] is carefully studied in a series of papers on so-called *taxonomic reasoning*, which is the problem of automatic classification of the newly defined concepts inside a concept hierarchy (that can represent, for example, a database schema). For instance, [BS92] explores the problem of such classification along with maintaining minimality (eliminating redundancy) and consistency (eliminating contradictory concepts) of the conceptual schema. (Similarly, [BL89] proposes a model that maintains consistency and minimality in the Entity-Relationship model.) In particular, [BS92] chose the concept definition language (FL^*) that contains **and**, **all**, a rather restricted form of the **not** constructor (specifically, atom negation to capture disjointedness of concepts), NOTHING, and number restrictions. The paper allows acyclic view definitions and presents a quadratic time algorithm for the corresponding subsumption problem. In addition, a subsumption algorithm is also given for FL^*_{inv} which is FL^* augmented with inverse roles. (As examples of works that studied taxonomic reasoning problem using notation which is more closely related to the standard notation of the *first order predicate calculus* (FOPC) consider [DD89, MG93].)

To further improve expressiveness of DLs without losing tractability, [LS91] considers the idea of allowing a more general language for one of the descriptions being compared. In particular, if one considers the problem of whether a description D_1 subsumes a description D_2, one might consider allowing the language in which the subsumer (D_1) is expressed to be more general than the language that expresses the subsumee (D_2). As a motivating example, the authors prove that the pair of languages AL and (the more general) QL have a polynomial time decision procedure for the subsumption problem. The AL language is the extension of FL^- with negation of primitive concepts, whereas the QL language extends AL with role conjunction, qualified existential quantification on roles and enumeration. The result is contrasted with the fact that the subsumption question for a pair of QL language concepts is NP-hard.

In addition, the paper considers the general use of DL languages for both capturing data in so-called *knowledge bases* and for expressing queries on the knowledge base, and it studies the following question: given a number of assertions about objects in a knowledge base and their connections via roles, does a certain object belong to a description. When *AL* is then used as a schema language and a subsumer is expressed in *QL*, it is again shown that the question can be answered in polynomial time.

In general, the problem of determining whether a set of assertions *logically implies* another assertion is a natural generalization of the subsumption question to the systems that also contain so-called *terminologies* that are the DL counterpart for the database schema as discussed in the next two subsections.

2.2.2 Terminologies and the Logical Implication Problem

A *terminology* is simply a collection of subsumption constraints. In the presence of a terminology, the subsumption question is naturally generalized to the logical implication problem of whether a terminology logically implies a subsumption constraint; or in other words, whether any *world* (or a *database instance* in the database environment) that satisfies constraints in a terminology also satisfies a subsumption constraint. (Sometimes we refer to this subsumption constraint as the *posed* or *stated* question.)

In general, the addition of a terminology converts the subsumption question into a potentially more complicated logical implication problem. To recognize this potential for an increase in complexity, consider a *cyclic* terminology in which a concept "depends" on itself [BDN94]. In this case, the corresponding logical implication problem cannot be replaced by a subsumption question obtained by simply "macro expanding" nested terminology. (In fact, terminologies are often cyclic in practice, and as we will see later, the presence of the new **fd** constructor would make cyclic terminologies even more widespread.) The fact that the subsumption problem is generally easier than the logical implication problem under the *same* conditions (such as the same DL language), makes it not surprising that the *query language*, i.e. the language in which the posed question is expressed, can be (and is often) made more expressive than the *schema language*, i.e. the language in which the terminology constraints are expressed, without "falling off the computational cliff". From the practical perspective, this situation is favorable since one would want to *at least* extract the information which is stored in the database.

Note how seemingly opposite proposals are suggested in the literature. On the one hand, some authors argue for the unification of the DML and DDL [BGN89, BBM89]; and on the other hand, the separation of the DDL, DML, subsumee and subsumer languages is argued to provide better expressiveness without losing tractability [LS91, BDN94]. Our approach in that of the "golden mean" where we combine the positive sides of the two proposals. On the one hand, our

languages for the proposed terminologies and stated questions are similar enough to provide a uniform view of the data, query and view objects, and to automatically use the database schema constraints for query processing. (One way this is illustrated in some of our examples is by the trick of abstracting a query as an increment to a terminology.) On the other hand, our schema language is in fact less expressive than our view definition and subsumee languages (in Chapter 5) that are in turn less expressive than the subsumer language.

Constraints in a terminology often have the form of a primitive concept being subsumed by a description. These constraints capture *necessary* conditions that objects must satisfy to belong to the concept. Other constraints that represent both *necessary and sufficient* conditions are called *concept definitions*. Based on practical applications, [BDN94] recognized the important distinctions between the two and considered the advantages of dividing a terminology into a *schema* that contains a set of constraints of the first type and a set of constraints (called *views*) of the second type. The paper makes a strong argument about improved conceptual clarity under the suggested model and considers some examples of its computational benefits. Semantics of the *terminological cycles* is also clarified, and it is shown that cycles are less "harmful" in terms of added complexity in the schema than in the view definition part of the terminology. Our thesis uses exactly this model to capture database schemas and views[3]. We allow cycles in our schemas but disallow them in views. Moreover, just as suggested in the paper, our schema language is not as expressive as our view definition language which also allows equational constraints.

The division of a terminology into a schema and views is exploited, for example, in [BJN94]. The authors define a schema language (*SL*) to capture schemas and a query language (*QL*)[4] to capture views and queries. The query language *QL* consists of the concept constructors **and**, **same-as**, THING, singleton set constructor, and existential quantification over a path; and role constructors **inverse**, **restrict**, **comp**, and **Id**. (Note that no typing constraints (**all**) are allowed since they would lead to intractability even with empty schema.) The schema language *SL* only contains (concept) constructors **all**, a single-valued restriction of a role (i.e. specifying that a role is an attribute), and the unqualified existential quantification (**some**). The logical implication problem is then shown to be polynomially decidable. The authors also observe that the addition of either a limited form of negation or disjunction to *QL* leads to intractability of the logical implication problem. The same is true of the addition of the disjunction of only primitive concepts to *SL*.

[GL95] is another example of work that considers the logical implication problem. In this case, the authors present a very expressive DL, called *CATS*, that in particular works with domains that contain tuples and sets rather than just objects. (The logical implication problem for *CATS* is shown to be EXPTIME-complete.) The importance of this paper for us is that it is one of the first papers to study an influence of some form of functional dependencies in DLs. In

[3] The view part is empty in the DL considered in Chapter 4. Chapter 5, on the other hand, studies views as well.

[4] Not to be confused with the *QL* language in [LS91].

particular, the paper considers a concept constructor that essentially captures simple keys (right-hand-side is **Id**) with attributes on the left-hand-side. The same kind of FDs are considered in [CGL95]. There, the authors propose an extended object-oriented model, called *CVL*, which is designed to express most other object-oriented models. The paper makes a strong case for capturing more of the application semantics at the schema level. Our work, in turn, achieves that by enhancing the kinds of uniqueness constraints one can capture.

A study of logical implication problems that contrasts the influences of acyclic and cyclic schemas is presented in [Cal96]. (Only terminologies without view definitions are considered.) The paper shows that cycles in schemas "greatly increase the complexity of reasoning". In particular, it considers FL^- and *AL* family of languages, and among those, only FL^- turns out to have polynomial time complexity in the presence of cyclic schemas.

2.2.3 Description Logics in Databases

The ORDB notions of a *relation* or *class* is clearly analogous to the DL notion of a *concept*. Moreover, just as the logical implication problem is the most important problem in DLs that contain terminologies, "deciding implication problems is a central computational task both in database schema design and query processing" [CK86, CKV90]. Therefore, it is not surprising that many researchers have found that ORDBs can benefit from DL research and, as we have already seen, a number of results that apply DL research to problems in ORDBs have been reported [BGN89, BBM89, DD89, BS92, BP94, BDN94, BJN94, GL95, CGL95].

An excellent general survey of applications of DLs in the database area is presented in [Bor95]. The author considers a variety of works that argue for representing various semantic data models and DB schemas by relatively limited (but tractable) DLs. The author also discusses a number of advantages of using DLs in DBs including capturing *additional data semantics*, verifying *schema consistency*, reducing *redundancy in schema representation*, improved *type checking*, and dynamic view handling (or *taxonomic reasoning*). In addition, the paper considers research on applications of DLs in *federated databases* as a "richly expressive medium" for capturing semantics of multiple *co-operative information systems*; that is, the use of DLs for *schema integration*. Other uses of DLs on essentially all aspects of a database system are discussed: utilization of DLs as view definition, constraint and query languages, and so on. Advantages of using DLs in environments where users do not know the exact structure of the data or not sure what query to formulate (such as in *data mining*) are also considered. In particular, the benefits provided by DLs in *multiple query evaluation*, *iterative query refinement*, *generalization of queries* to provide answers when the original answer is empty, and dealing with *materialized views* are discussed. Benefits of DLs relating to maintaining and reasoning about *incomplete information* are also presented (such benefits become especially clear when compared to the use of *null* values in databases for similar purposes). The advantage of employing DLs for another

traditionally difficult database problem of *view updates* is discussed as well. Further benefits of DLs presented in the paper include their ability to solve problems in *configuration management* and to return *descriptive answers* when query output would be too lengthy otherwise (or when a user is not completely familiar with the schema). Finally, the paper also gives a strong argument for the notation used in DLs by showing that the DL notation is much more succinct and easier to reason about than *first order predicate calculus* (FOPC) for the kinds of expressions considered by DLs.

Finally, [BW97] provides a good framework (adopted in our work) for studying FDs in DLs. However, FDs presented in the paper are defined only to deal with path functions that are single attributes or **Id**, and no proofs are available for the claims made in the paper. In addition to solving the above problems in Chapter 5, we also "fix" a number of other problems found in the paper.

In comparison to many of the works above, our work only deals with attributes that are total functions (its extension to roles is discussed in Chapter 6). Therefore, unqualified existential quantification (**some** constructor) is not needed or considered in our framework. On the other hand, we deal with the new **fd** constructor (in both terminology and posed questions) and study its interaction with other constructors of FL^- as well as non-trivial equational (**same-as**) and role composition (**comp**) constructors that are very useful in ORDBs. By "non-trivial" we mean that these constructors are known to add complexity to other languages. For example, adding role composition to PL_1 raises the complexity of the subsumption problem from polynomial time to NP-hard [DLN91].

2.3 Other Related Work

There is a number of other papers that study the types of constraints considered in this thesis with some variations. For example, [AM86] explores so-called *intersection and union extended generalization constraints* which are extensions of typing constraints that include conjunction and disjunction of "entity types" (classes). In a similar manner, typing constraints that involve path functions [WC90] (or more generally, set valued "path descriptions") are extended in [IN93] to typing constraints on disjunction of classes. [BWW98] considers various *path constraints* in object models that could be considered as various forms of equational constraints.

Also, a number of recent works on extending functional dependencies can be considered as another indication of the significance of the FD theory. There is a renewed interest in FDs as means of capturing and reasoning about data semantics in advanced models. For example, [HD99] presents sound and complete axiomatization of *nested functional dependencies* which are defined for nested relational model. Another recent paper [Wij99] studies *temporal functional*

dependencies on complex objects in temporal databases and contains an overview of other recent works in the area. Temporal functional dependency theory is an alternative extension of FD theory which is orthogonal to ours, and the two theories could possibly be combined into a temporal PFD theory.

Finally, the work on so-called *congruence closure* algorithms [NO77, NO80, DST80] is also related to our own work, in particular to the kind of graph modifications that we perform when dealing with FD constraints inside description graphs.

Chapter 3

Definitions

3.1 Descriptions and their Semantics

All description logics usually deal with at least two kinds of descriptions: *concept descriptions D* that denote subsets of some domain Δ, and *attribute descriptions Pf*[5] that denote functions over Δ. In general, such descriptions are "constructed" from *primitive concepts* C_1, C_2, ... and *primitive attributes* A_1, A_2, ... using *concept* and *attribute constructors*. Note that the term "concept" is used in DLs as a preferred synonym for the term "class" or "relation" in ORDBs. For example, the concept description

(**and** PERSON
  (**all** Name STRING)
  (**all** Enum NUMBER)
  (**fd** PROFESSOR Name Enum)
  (**fd** PROFESSOR Enum **Id**))

uses primitive concepts PERSON, PROFESSOR, NUMBER, and STRING, primitive attributes Name and Enum, concept constructors **and**, **all** and **fd**, and attribute constructor **Id** to capture the set of "all people with distinct integer employee numbers and distinct names that are strings of characters". This example illustrates part of what can be expressed in the DL languages we consider in this thesis. Later on, we formally define subsumption constraints that allow us to then associate a primitive concept PROFESSOR with this description.

The occurrences of PERSON, **all** and **fd** in this example would correspond to inheritance, typing and various key constraints, respectively, in a hypothetical "class" definition for an ORDB. In particular, the two **fd** constructors illustrate how one might capture the notions of

[5] While *Pf* stands for "path function", it will become clear later why we use the term "attribute description" in this context.

candidate and primary key constraints. Thus, this example begins to demonstrate how description logics can be used to capture a database schema. The concept and attribute constructors in this example are shared by almost all dialects of DLs except for the **fd** concept constructor. (This constructor is peculiar to a recent extension of CLASSIC, called Classic/FD, that is proposed in [BW97].)

Formally, the following grammar presents the descriptions that we are interested in:

D ::=	C	*(primitive concept)*
	| V	*(view name)*
	| (**all** Pf D)	*(attribute value restriction)*
	| (**fd** C Pfs Pf)	*(functional dependency)*
	| (**and** D D Ds)	*(description intersection)*
	| (**same-as** Pf Pf)	*(equational restriction)*
Pf ::=	A	*(primitive attribute)*
	| **Id**	*(identity)*
	| (**comp** Pf Pf)	*(attribute composition)*
Pfs ::=	ε | Pfs Pf	*(attribute description sequence)*
Ds ::=	ε | Ds D	*(concept description sequence)*

Note that the grammar would be a subset of the Classic/FD DL grammar if we allowed only primitive attributes and **Id** instead of arbitrary attribute descriptions inside the **fd** descriptions.

Semantically, given an underlying domain Δ, an interpretation $\mathbf{I} = \langle \Delta, \cdot^{\mathbf{I}} \rangle$ first assigns a (not necessarily distinct) subset of Δ to each primitive concept and view, and a subset of $\Delta \times \Delta$ encoding a *total* function[6] over Δ to each primitive attribute. (We assume that the set of distinct primitive attributes is recursively enumerable.) The interpretation of the constructed concept and attribute descriptions is then defined by the following rules.

- $(\textbf{all}\ Pf\ D)^{\mathbf{I}} = \{\ x \in \Delta \mid Pf^{\mathbf{I}}(x) \in D^{\mathbf{I}}\ \}$
- $(\textbf{fd}\ C\ Pf_1\ \ldots\ Pf_m\ Pf)^{\mathbf{I}} = \{\ x \in \Delta \mid \forall y \in C^{\mathbf{I}}\colon [Pf_1^{\mathbf{I}}(x) = Pf_1^{\mathbf{I}}(y) \wedge \ldots \wedge Pf_m^{\mathbf{I}}(x) = Pf_m^{\mathbf{I}}(y)] \rightarrow Pf^{\mathbf{I}}(x) = Pf^{\mathbf{I}}(y)\ \}\ (m \geq 0)$
- $(\textbf{and}\ D_1\ D_2\ \ldots\ D_m)^{\mathbf{I}} = D_1^{\mathbf{I}} \cap \ldots \cap D_m^{\mathbf{I}}\ (m \geq 2)$
- $(\textbf{same-as}\ Pf_1\ Pf_2)^{\mathbf{I}} = \{\ x \in \Delta \mid Pf_1^{\mathbf{I}}(x) = Pf_2^{\mathbf{I}}(x)\ \}$
- $\mathbf{Id}^{\mathbf{I}} = \{\ (x, x) \mid x \in \Delta\ \}$
- $(\textbf{comp}\ Pf_1\ Pf_2)^{\mathbf{I}} = \{\ (x, y) \mid x \in \Delta \wedge Pf_2^{\mathbf{I}}(Pf_1^{\mathbf{I}}(x)) = y\ \}$

[6] DL languages are somewhat divided over the issue of whether or not attribute descriptions should denote partial or total functions. We follow [BP94] in opting for the latter case in order to avoid complications that would otherwise arise relating the semantics of some of our constructors [BDN94].

We are now ready to introduce constraints in DLs. Given a primitive concept C and a description D, an interpretation **I** is said to *satisfy* the *subsumption constraint* $C < D$, or constraint $C < D$ is *satisfied* by an interpretation **I**, if and only if $C^{\mathbf{I}}$ is a subset of $D^{\mathbf{I}}$. Thus, for example, to express the fact that "all objects of concept PROFESSOR are also objects of concept PERSON (inheritance), and have an employee number attribute Enum with its value from a concept NUMBER (typing)", one would use the subsumption constraint

PROFESSOR < (**and** PERSON (**all** Enum NUMBER)).

Every interpretation that satisfies this must then satisfy the corresponding inheritance and typing constraints. Note that the above does *not* imply a stronger assertion that every person with an employee number is a professor: the constraint expresses only some *necessary* conditions for an object to be a professor. This one-sided characteristic allows one to concentrate on particular properties of a concept one at a time. For example, given the above constraint, both constraints

PROFESSOR < PERSON and PROFESSOR < (**all** Enum NUMBER)

hold. In case of *defined concepts*, or *views*, however, one would want to capture both necessary and *sufficient* conditions. Then a *view definition* $V \equiv D$ should be used. Formally, given a view name V and a description D, an interpretation **I** is said to *satisfy* the constraint $V \equiv D$, or $V \equiv D$ is *satisfied* by **I**, if and only if $V^{\mathbf{I}}$ has exactly the same members as $D^{\mathbf{I}}$.

3.2 Terminology

A database schema can now be characterized in terms of a *terminology*. Following [BDN94], we define a terminology T as a sequence of subsumption constraints $(C_1 < D_1, \ldots, C_k < D_k)$ and a sequence of view definitions $(V_1 \equiv D_1', V_2 \equiv D_2', \ldots, V_m \equiv D_m')$[7], where $k \geq 0$ and $m \geq 0$. An interpretation **I** *satisfies* a terminology T if and only if it satisfies every constraint in T; that is, if **I** satisfies every subsumption constraint and every view definition. Thus, interpretation **I** represents a database of objects (from the domain Δ) that are grouped into a concept hierarchy based on their membership in $C_i^{\mathbf{I}}$ and that satisfy the appropriate typing, inheritance, uniqueness and equational constraints set by the database schema represented by T. On top of that, view definitions in T determine the sets of all objects in Δ that satisfy the appropriate constraints imposed by the right-hand-sides of the view definitions.

As in [BW97], we assume that every terminology satisfies the following conditions:

[7] We assume that $V_1, V_2, \ldots, V_m$ are the only view names occurring inside the terminology. In other words, there are no "undefined" views.

PROFESSOR < (**and** PERSON
(**all** Name STRING)
(**all** Enum NUMBER)
(**all** Boss PROFESSOR)
(**all** Dept DEPARTMENT)
(**fd** PROFESSOR Name Enum)
(**fd** PROFESSOR Enum **Id**))

DEPARTMENT < (**and**
(**all** Name STRING)
(**fd** DEPARTMENT Name **Id**))

Figure 3.1: The UNIV terminology.

(a) $V_i \neq V_j$ if $i \neq j$, and $C_i \neq V_j$ for all i and j.
(b) D_i is a description satisfying the grammar above except that the **same-as** constructor cannot occur in D_i for $1 \leq i \leq k$.
(c) V_i cannot occur in D_j for $1 \leq j \leq k$ and $1 \leq i \leq m$.
(d) Views are non-recursive: V_j cannot occur in D_i' for $1 \leq i \leq m$ and $i \leq j \leq m$.

To illustrate how a terminology can be used to capture a database schema, consider a part of a hypothetical administrative database schema in Figure 3.1 that presents what we shall refer to as the UNIV terminology. The terminology contains two subsumption constraints that expand on our earlier example concept of a professor. Observe that the typing and inheritance constraints occurring in the figure can be captured by both object-relational and relational models. In the latter case, for example, one can use a unary inclusion dependency (foreign key constraint) to capture the restriction on the Dept attribute for the corresponding table of professors. (Note that we allow recursive schemas; in this case, boss of a professor is a professor.)

3.3 On Views

Since views are non-recursive (recall restriction (d) above on terminologies), it becomes possible to unfold an arbitrary description by repeatedly substituting view names with their respective definitions. An *unfolded view description* $V_{[u]}$ of a view V relative to some terminology T containing a view definition of V is a description obtained from V by exhaustively substituting all view names, starting with V, with their corresponding (right-hand-side) descriptions in the view definitions in T. More generally, an *unfolded description* $D_{[u]}$ of a description D relative to some terminology T is defined as a description obtained from D by substituting all view names in D with their unfolded view descriptions. (The particular choice of terminology T will usually be understood from context.)

For example, consider a terminology T that contains the view definitions

$V_1 \equiv D_1$,
$V_2 \equiv D_2$, and
$V_3 \equiv$ (**and** V_2 (**all** A V_1)),

where V_i are view names, D_i are descriptions that do not contain any view names and A is an attribute. Then, the unfolded view description $V_{3[u]}$ is (**and** D_2 (**all** A D_1)), and the unfolded description (**and** V_1 V_3)$_{[u]}$ is (**and** D_1 (**and** D_2 (**all** A D_1))).

A straightforward consequence of our definitions is that unfolding a description does not change its interpretation. More formally, for any interpretation **I** that satisfies a terminology T, $V^{\mathbf{I}} = V_{[u]}{}^{\mathbf{I}}$ for any view V in T, and more generally, $D^{\mathbf{I}} = D_{[u]}{}^{\mathbf{I}}$ for any description D that may only contain view names defined in T.

3.4 On Path Functions

To simplify our presentation, we define a "normalized form" for an arbitrary attribute description. In particular, we refer to any attribute description generated by the grammar

Pf ::=	**Id**	(*identity*)
	! (**comp** A Pf)	(*attribute composition*)

as a *path function*. Such attribute descriptions always have the form

(**comp** A_1 (**comp** A_2 ... (**comp** A_k **Id**) ...))[8]

for some $k \geq 0$ and primitive attributes A_1 thought A_k.

Table 3.1 defines three rewrite rules that can be exhaustively applied to an arbitrary attribute description to obtain a path function. Note that this mapping preserves interpretations of attribute descriptions, a fact that we formally express and prove later on when we outline more general mappings that simplify terminologies. Thus, while it is easier and more intuitive to express examples using attribute descriptions, our procedures will assume that such descriptions are first mapped to equivalent path functions.

[8] Unless specified otherwise, A and B (possibly with subscripts, superscripts, and/or prime symbols) will denote primitive attributes in the rest of this work.

1. Replace "(**comp Id** *Pf*)" by "*Pf*".
2. Replace "(**comp** *Pf A*)" by "(**comp** *Pf* (**comp** *A* **Id**))".
3. Replace "(**comp** (**comp** Pf_1 Pf_2) Pf_3)" by "(**comp** Pf_1 (**comp** Pf_2 Pf_3))".

Table 3.1: Normalizing attribute descriptions.

The *length* of a path function *Pf*, written *Len*(*Pf*), is defined as the number of occurrences of the **comp** constructor inside *Pf*. In particular, if *Len*(*Pf*) is 0, then *Pf* is **Id**. The *composition* of an attribute description Pf_1 with an attribute description Pf_2, denoted $Pf_1 \circ Pf_2$, is defined as the path function resulting from an exhaustive application of the rules in Table 3.1 to "(**comp** Pf_1 Pf_2)". Note that $Pf \circ$ **Id** therefore denotes the path function equivalent to an attribute description *Pf*.

Finally, the following claim establishes a number of useful properties of attribute descriptions and the composition operator.

Claim 3.1 *Let A and B (possibly with subscripts) denote primitive attributes, Pf (possibly with subscripts) denote attribute descriptions, and D denote a description. Then the following properties hold for any interpretation* **I**.

(1) *Let* $A_1, A_2, \ldots, A_k$ $(k \geq 0)$ *be all primitive attributes of* Pf_1 *textually appearing in* Pf_1 *in that order, and let* $B_1, B_2, \ldots, B_m$ $(m \geq 0)$ *be all primitive attributes of* Pf_2 *textually appearing in* Pf_2 *in that order. Then,*

$$Pf_1 \circ Pf_2 = (\textbf{comp}\ A_1\ (\textbf{comp}\ A_2 \ldots (\textbf{comp}\ A_k\ (\textbf{comp}\ B_1\ (\textbf{comp}\ B_2 \ldots (\textbf{comp}\ B_m\ \textbf{Id}) \ldots)) \ldots)) \ldots)).$$

(2) *The composition operator,* $\circ$ *, is associative.*

(3) $Pf_1 \circ \textbf{Id} \circ Pf_2 \circ \textbf{Id} \circ \ldots \circ \textbf{Id} \circ Pf_k = Pf_1 \circ Pf_2 \circ \ldots \circ Pf_k$ *for any* $k \geq 1$.

(4) $(Pf_1 \circ Pf_2 \circ \ldots \circ Pf_k)^{\mathbf{I}} = \{(x, y) \mid x \in \Delta \wedge Pf_k^{\mathbf{I}}(\ldots Pf_2^{\mathbf{I}}(Pf_1^{\mathbf{I}}(x)) \ldots) = y\}$ *for any* $k \geq 1$.

(5) *If* $Pf_1^{\mathbf{I}} = Pf_2^{\mathbf{I}}$, *then replacing* Pf_1 *by* Pf_2 *in any description D does not modify* $D^{\mathbf{I}}$.

Proof.

Property (1). Note that none of the three rewrite rules in Table 3.1 modifies the textual order of the attributes or removes or adds primitive attributes. Also, by the propositions that we present in the later chapters and their proofs in the Appendices A and B, the result of the exhaustive application of those rules produces an attribute description that satisfies the grammar for path functions. Thus, if $m = k = 0$, the only possible result of applying the rewrite rules to (**comp** Pf_1 Pf_2) is **Id**. On the other hand, note that the composition constructor in "(**comp** Pf_1 Pf_2)" also does not change the textual order of attributes or adds or deletes any primitive attributes. Therefore, if m or k is greater than 0, the result of applying the rules to (**comp** Pf_1 Pf_2) must be

(**comp** A_1 (**comp** A_2 ... (**comp** A_k (**comp** B_1 (**comp** B_2 ... (**comp** B_m **Id**) ...))) ...)).

Property (2). Consider some attribute descriptions Pf_1, Pf_2 and Pf_3, and let A_1, A_2, ..., A_k be all primitive attributes of Pf_1 textually appearing in Pf_1 in that order; let A_1', A_2', ..., A_l' be all primitive attributes of Pf_2 textually appearing in Pf_2 in that order; and finally, let A_1'', A_2'', ..., A_m'' be all primitive attributes of Pf_3 textually appearing in Pf_3 in that order. First assume that k, l and m are greater than 0. Then, by property (1), $Pf_1 \circ Pf_2$ must be

(**comp** A_1 (**comp** A_2 ... (**comp** A_k (**comp** A_1' (**comp** A_2' ... (**comp** A_l' **Id**) ...))) ...)),

$Pf_2 \circ Pf_3$ must be

(**comp** A_1' (**comp** A_2' ... (**comp** A_l' (**comp** A_1'' (**comp** A_2'' ... (**comp** A_m'' **Id**) ...))) ...)),

and finally, both $(Pf_1 \circ Pf_2) \circ Pf_3$ and $Pf_1 \circ (Pf_2 \circ Pf_3)$ must be

(**comp** A_1 ... (**comp** A_k (**comp** A_1' ... (**comp** A_l' (**comp** A_1'' ... (**comp** A_m'' **Id**) ...)) ...)) ...).

By completely analogous arguments, the equality of $(Pf_1 \circ Pf_2) \circ Pf_3$ and $Pf_1 \circ (Pf_2 \circ Pf_3)$ can be established for cases when some of k, l, or m are 0.

Note that this property implies that we do *not* have to use the parentheses to indicate the order of applications of the composition operator.

Property (3). This property is a straightforward consequence of property (1) since both $Pf_1 \circ$ **Id** $\circ Pf_2 \circ$ **Id** $\circ$ ... $\circ$ **Id** $\circ Pf_k$ and $Pf_1 \circ Pf_2 \circ$ ... $\circ Pf_k$ have the same order of the same primitive attributes, and thus, would be (textually) equal.

Property (4). The proof is by induction on k. For the base case of $k = 1$, the property is trivially satisfied. Now, assume that the property holds for all $k \leq l$ for some $l \geq 1$ and consider $k = l + 1$. Since $(\textbf{comp}\ Pf_1\ Pf_2)^{\mathbf{I}} = \{(x, y) \mid x \in \Delta \wedge Pf_2^{\mathbf{I}}(Pf_1^{\mathbf{I}}(x)) = y\}$, and since by the proofs in the Appendices A and B, application of the rewrite rules of Table 3.1 does not change the interpretation of the attribute descriptions,

$$(Pf_1 \circ Pf_2)^{\mathbf{I}} = (\textbf{comp}\ Pf_1\ Pf_2)^{\mathbf{I}} = \{(x, y) \mid x \in \Delta \wedge Pf_2^{\mathbf{I}}(Pf_1^{\mathbf{I}}(x)) = y\}.$$

Thus, $(Pf_1 \circ Pf_2 \circ \ldots \circ Pf_{l+1})^{\mathbf{I}} = \{(x, y) \mid x \in \Delta \wedge Pf_{l+1}^{\mathbf{I}}((Pf_1 \circ Pf_2 \circ \ldots \circ Pf_l)^{\mathbf{I}}(x)) = y\}$. Using the inductive assumption now, we conclude that

$$(Pf_1 \circ Pf_2 \circ \ldots \circ Pf_{l+1})^{\mathbf{I}} = \{(x, y) \mid x \in \Delta \wedge Pf_{l+1}^{\mathbf{I}}(\ldots Pf_2^{\mathbf{I}}(Pf_1^{\mathbf{I}}(x)) \ldots) = y\}.$$

Property (5). This property is a straightforward consequence of the definition of the interpretation function.

■

One of the consequences of Claim 3.1 is that an arbitrary path function

$$(\textbf{comp}\ A_1\ (\textbf{comp}\ A_2 \ldots (\textbf{comp}\ A_k\ \textbf{Id}) \ldots))$$

can be also expressed by $A_1 \circ A_2 \circ \ldots \circ A_k$.

3.5 Logical Implication

Recall that one of our main concerns in this work is an efficient way of solving the logical implication problem; that is, whether a number of constraints imply another constraint. Thus, for a given terminology, we would like to deduce whether a particular constraint "follows" from it. While the terminology must meet the requirements stated above, we wish to generalize the notion of the constraint that we can ask about to the form $D_1 < D_2$ where both D_1 and D_2 are arbitrary descriptions. Like our earlier definitions, constraint $D_1 < D_2$ is said to be *satisfied* by an interpretation $\mathbf{I}$ if and only if $D_1^{\mathbf{I}}$ is a subset of $D_2^{\mathbf{I}}$.

In order to finally complete the formal characterization of the logical implication problem, we define the logical implication itself: a terminology T *logically implies* a constraint

$D_1 < D_2$, written $T \models D_1 < D_2$, if and only if for any possible interpretation **I** that satisfies T, **I** also satisfies $D_1 < D_2$[9].

Recall from Chapter 2 that most of the research that considers efficient decision procedures which solve logical implication problems with uniqueness constraints in the context of models with complex objects require all such constraints to be "key dependencies". In terms of our DL language described in the beginning of the chapter, such constraints would have the form

$$C < (\textbf{fd}\ C\ Pf_1\ \ldots\ Pf_k\ Pf)$$

where Pf is a "prefix" of some path function Pf_i ($1 \le i \le k$). Our procedure in Chapter 4 extends such constraints and allows Pf to be either a prefix of Pf_i or a prefix composed with another attribute.

More formally, consider an **fd** description $D = (\textbf{fd}\ C\ Pf_1\ \ldots\ Pf_k\ Pf)$ for some concept C, path functions Pf_1 through Pf_k and Pf. We call D *regular* if there exists a path function Pf_i for some $1 \le i \le k$ such that $Pf_i = A_1 \circ A_2 \circ \ldots \circ A_m \circ \textbf{Id}$ (for some $m \ge 0$ and primitive attributes A_1 through A_m), and Pf either has the form $A_1 \circ A_2 \circ \ldots \circ A_t \circ \textbf{Id}$ or $A_1 \circ A_2 \circ \ldots \circ A_t \circ A$ for some $0 \le t \le m$ and a primitive attribute A. Note that regular descriptions can be used to express not only key but also relational functional dependencies.

Consider a subsumption constraint of the form

$$C_1 < (\textbf{fd}\ C_2\ Pf_1\ \ldots\ Pf_k\ Pf).$$

If C_1 and C_2 are the *same* concepts, the constraint is called a *symmetric uniqueness constraint*. Otherwise, it is called an *asymmetric uniqueness constraint*. Further, if the right-hand-side of the constraint is a regular **fd** description, the constraint is called a *regular uniqueness constraint*.

Procedure in Chapter 4 allows only symmetric regular uniqueness constraints to occur in a terminology and considers a logical implication problem with posed question that can be an arbitrary symmetric uniqueness constraint. Chapter 5 also discusses terminologies that only contain regular **fd** descriptions. However, it explores a more general logical implication problem of whether a terminology logically implies a constraint $D_1 < D_2$, where all **fd** descriptions inside D_1 are also regular. On the other hand, **fd** descriptions allowed inside D_2 are not necessarily regular and are further generalized to what we call ***extended fd descriptions***. Such descriptions have the form

$$(\textbf{fd}\ D\ Pf_1\ \ldots\ Pf_m\ Pf)$$

[9] Again, we assume that there are no "undefined" views, i.e. D_1 and D_2 do not contain any view names that are not defined in T.

for some description D that in turn must satisfy our grammar presented above. (Note that D cannot contain extended **fd** descriptions inside.) As one might expect, interpretation of the extended **fd** description is defined as follows:

- $(\textbf{fd}\ D\ Pf_1\ \ldots\ Pf_m\ Pf)^{\mathrm{I}} = \{\ x \in \Delta \mid \forall y \in D^{\mathrm{I}}\colon [Pf_1^{\mathrm{I}}(x) = Pf_1^{\mathrm{I}}(y) \wedge \ldots \wedge Pf_m^{\mathrm{I}}(x) = Pf_m^{\mathrm{I}}(y)] \rightarrow Pf^{\mathrm{I}}(x) = Pf^{\mathrm{I}}(y)\ \}$.

In particular, these more general descriptions allow us to reason about uniqueness constraints on views. They further allow us to "ask" about uniqueness constraints that hold on queries that are captured as views.

3.6 Description Graphs

The procedures presented in this work use *description graphs* to solve their respective logical implication problems. A *description graph* $G = \langle N, E, Refs \rangle$ is a triple that consists of a set N of nodes, a bag E of directed edges labeled with either primitive attribute names or **Id**, and a set *Refs* of *references* to nodes. There are a variety of labels that can be attached to each node, each containing a set of descriptions. (Unless stated otherwise, whenever we add new nodes to a graph, we assume that these labels are initially empty.)

Elements in E are written $\langle n_1, A, n_2 \rangle$ where n_1 and n_2 are nodes and A is either **Id** or a primitive attribute name. In the former case, we sometimes refer to the edge as an "**Id** edge", and in the latter case, we say it is a "non-**Id** edge". Intuitively, nodes of the graph correspond to hypothetical or prototypical objects, while edges correspond to their attribute values.

Each element of the *Refs* set is a pair that binds a name of a reference to some node in N. One of the references will be named *dn* which will be the main "entrance point" into the graph (it corresponds to the *distinguished node* of the graph in [BP94]).

Finally, when a reference *points* to a node (i.e. the reference is bound to the node in the *Refs* set), we use the reference name interchangeably with the node itself. For example, we might denote an edge with $\langle dn, A, n \rangle$ or talk about "node *dn*" as shorthand for the "node referenced by *dn*". We also assume (unless stated otherwise) that the same subscripts and superscripts apply to reference names that are used for the *Refs* component of the graph. For example, by default, $Refs_1'$ contains the distinguished node reference dn_1'.

Chapter 4

Reasoning with Symmetric Uniqueness Constraints

4.1 Problem Definition

In this chapter, we present a solution of the logical implication problem for terminologies when all **fd** constructors inside a terminology are part of symmetric regular uniqueness constraints (defined in Section 3.5) and the posed question is an arbitrary symmetric uniqueness constraint. In other words, the chapter solves what we call the *membership problem*[10] for ORDB functional constraint theory; that is, the problem of determining whether a given functional dependency is logically implied by an ORDB database schema. Note that, using the notation in [Wed89], this is essentially the membership problem for PFDs with a single right-hand-side component:

$$C(\{Pf_1, \ldots, Pf_m\} \rightarrow \{Pf\}).$$

(It is well known that the restriction of having a single right-hand-side component does not reduce the expressiveness of functional dependency constraints.)

By definition of what constitutes a symmetric uniqueness constraint, a view name cannot occur in a posed question. This in turn implies that the view definitions occurring in a terminology will not be relevant to the logical implication problem that we consider. Thus, we assume (without loss of generality) that an argument terminology contains subsumption constraints only. It is then a straightforward consequence that all descriptions that do occur in the terminology or posed question can be generated by the following subset of our description grammar.

[10] The term "membership problem" was used by [BB79] to describe the same implication problem in the relational model where database schemas were assumed to consist of a collection of relational functional dependencies.

D ::= C	(*primitive concept*)
| (**all** Pf D)	(*attribute value restriction*)
| (**fd** C Pfs Pf)	(*functional dependency*)
| (**and** D D Ds)	(*concept intersection*)
Pf ::= A	(*primitive attribute*)
| **Id**	(*identity*)
| (**comp** Pf Pf)	(*attribute composition*)
Pfs ::= ε | Pfs Pf	(*attribute description sequence*)
Ds ::= ε | Ds D	(*concept description sequence*)

Note in particular that the production relating to **same-as** (in addition to views) is now absent since, again by our definitions, occurrences of **same-as** are only possible in view definitions.

4.2 Mapping to Atomic Terminologies

We now present the first of our mappings that were mentioned in Section 1.3. The role of the mapping is to *simplify* an input terminology by breaking up its constraints into a number of what we call *atomic subsumption constraints* that constitute a new *atomic terminology*. Formally, we define an atomic terminology as any terminology T' than can be obtained from an arbitrary terminology T by an exhaustive application of the rewrite rules in Table 4.1. We denote this circumstance by writing *Atomic*(T, T').

One consequence of Proposition 4.1 below is that descriptions in an atomic terminology have the property that they can be generated from the following simplified grammar.

D ::= C	(*primitive concept*)
| (**all** A C)	(*attribute value restriction*)
| (**fd** C Pfs Pf)	(*functional dependency*)
Pf ::= **Id**	(*identity*)
| (**comp** A Pf)	(*attribute composition*)
Pfs ::= ε | Pfs Pf	(*attribute description sequence*)

Another consequence is that this alternative simpler grammar does not reduce the expressiveness of a terminology in any fundamental way; for any terminology T_1 there exists a terminology T_2 for which *Atomic*(T_1, T_2) holds and that *preserves logical consequence*. Thus, although our examples will continue to be based on the more general grammar in Section 4.1, our procedure will assume (without loss of generality) that argument terminologies are atomic.

1. Replace "$C < (\textbf{all Id } D)$" by "$C < D$".
2. Replace "$C_1 < (\textbf{all } (\textbf{comp } Pf_1\ Pf_2)\ D)$" by "$C_1 < (\textbf{all } Pf_1\ C_2)$" and "$C_2 < (\textbf{all } Pf_2\ D)$".*
3. Replace "$C_1 < (\textbf{all } A\ (\textbf{all } Pf\ D))$" by "$C_1 < (\textbf{all } A\ C_2)$" and "$C_2 < (\textbf{all } Pf\ D)$".*
4. Replace "$C_1 < (\textbf{all } A\ (\textbf{fd } C_3\ Pfs\ Pf))$" by "$C_1 < (\textbf{all } A\ C_2)$" and "$C_2 < (\textbf{fd } C_3\ Pfs\ Pf)$".*
5. Replace "$C < (\textbf{all } A\ (\textbf{and } D_1 \ldots D_k))$" by "$C < (\textbf{all } A\ D_1)$", ..., "$C < (\textbf{all } A\ D_k)$".*
6. Replace "$C < (\textbf{and } D_1 \ldots D_k)$" by "$C < D_1$", ..., "$C < D_k$".*
7. Replace "$C_1 < (\textbf{fd } C_3\ Pfs_1\ A\ Pfs_2)$" by "$C_1 < (\textbf{fd } C_3\ Pfs_1\ (\textbf{comp } A\ \textbf{Id})\ Pfs_2)$".

(*where C_2 denotes a new primitive concept not occurring in the given terminology and $k \geq 2$.)

(a) (*rewrites for subsumption constraints*)

8. Replace "$(\textbf{comp Id } Pf)$" by "Pf".
9. Replace "$(\textbf{comp } Pf\ A)$" by "$(\textbf{comp } Pf\ (\textbf{comp } A\ \textbf{Id}))$".
10. Replace "$(\textbf{comp } (\textbf{comp } Pf_1\ Pf_2)\ Pf_3)$" by "$(\textbf{comp } Pf_1\ (\textbf{comp } Pf_2\ Pf_3))$".

(b) (*rewrites for attribute descriptions*)

Table 4.1: Atomic simplification of terminology.

Proposition 4.1 *Let T_1 denote an arbitrary terminology, and $C < D$ a subsumption constraint free of any occurrence of a primitive concept not occurring in T_1. Then an exhaustive application of the rewrite rules in Table 4.1 terminates and obtains a terminology T_2 that satisfies the simplified grammar and such that $T_1 \models C < D$ if and only if $T_2 \models C < D$.*

Proof. (See Appendix A).

For example, the formulation of the UNIV terminology in Figure 4.1 is obtained by the rewrites in Table 4.1 applied to the formulation of the UNIV terminology in Figure 3.1 in Chapter 3. Note, in particular, that all **fd** concept constructors now occur within symmetric uniqueness constraints, and therefore, solution to the membership problem considered in this chapter is applicable to the formulation of the UNIV terminology in Figure 4.1.

4.3 Further Applications

There are many ways in which an efficient means of deducing symmetric uniqueness constraints is very useful in query optimization. For example, to revisit a class of applications relating to the

PROFESSOR < PERSON
PROFESSOR < (**all** Name STRING)
PROFESSOR < (**all** Enum NUMBER)
PROFESSOR < (**all** Boss PROFESSOR)
PROFESSOR < (**all** Dept DEPARTMENT)
PROFESSOR < (**fd** PROFESSOR (**comp** Name **Id**)
(**comp** Enum **Id**))
PROFESSOR < (**fd** PROFESSOR (**comp** Enum **Id**) **Id**)

DEPARTMENT < (**all** Name STRING)
DEPARTMENT < (**fd** DEPARTMENT (**comp** Name **Id**) **Id**)

Figure 4.1: The atomic UNIV terminology.

distinct keyword, consider two formulations in OQL of a request *for all distinct names of professors assigned to a department with a given name*:

select distinct P.Name **as** Name **from** PROFESSOR **as** P
where P.Dept.Name = Param

select P.Name **as** Name
from DEPARTMENT **as** D, PROFESSOR **as** P
where D.Name = Param **and** P.Dept = D

The first formulation expresses the request more clearly, while the second has two performance advantages over the first. First, it allows the possibility of using any DEPARTMENT index to "bind" D prior to any PROFESSOR index to "bind" P; and second, it avoids the use of the **distinct** keyword in the **select** clause. The translation of the former to a version of the latter in which the **distinct** keyword has *not* been removed is straightforward. A query optimizer can then deduce that the **distinct** keyword is unnecessary by first augmenting the UNIV terminology with the following additional subsumption constraints for a new class Q.

Q < (**all** P PROFESSOR)
Q < (**all** D DEPARTMENT)
Q < (**fd** Q (**comp** D Name) Param)
Q < (**fd** Q (**comp** P Dept) D)

The new class serves as an abstraction of the result of the second query; the two **all** constraints correspond to the **from** clause, and the two **fd** constraints correspond to the **where** clause (both

are straightforward consequences of its two conditions). In other words, one can think of Q as the concept containing all the results of the query such that every object in Q defines a binding for a professor P, the name of the professor P.Name, a department D and the query parameter Param. The optimizer then proceeds to check that each of the following is a logical consequence of the modified UNIV terminology.

Q < (**fd** Q (**comp** P Name) P)
Q < (**fd** Q (**comp** P Name) D)
Q < (**fd** Q (**comp** P Name) Param)

If this is true, then the number of objects in Q must be the same as the number of distinct professor names defined by those objects in Q. The optimizer can then safely proceed to remove the **distinct** keyword.

Note that all **fd** concept constructors occur within symmetric uniqueness constraints in this application, and that the three immediately above differ only in their right-hand-side. This latter observation suggests an important way in which an algorithm for reasoning about such constraints should be incremental: subsequent checks for constraint membership that differ only in the right-hand-side should run very efficiently. To enable similar use for another query, it should also be possible to revise a terminology without incurring large "reformatting" costs.

Next, to review how an efficient means of deducing symmetric uniqueness constraints can be useful in schema evaluation, consider the diagnosis of an object normal form proposed by Biskup [Bis89]. His notion of *strong* object normal form requires a relation schema to satisfy two conditions: first, that the schema has a unique minimal key; and second, that the schema is in Boyce-Codd normal form (BCNF). (A relation schema is in *weak* object normal form when it satisfies the latter condition only; that is, if and only if it is in BCNF.)

Now consider the primitive PROFESSOR concept of the UNIV terminology to be an understanding in description logic of a PROFESSOR relation with attributes Name, Enum, Boss and Dept. We can confirm that PROFESSOR is *not* in strong object normal form by asking the following sequence of questions that focus on the particular requirement of having a unique minimal key.[11]

UNIV ⊨ PROFESSOR < (**fd** PROFESSOR Name Enum Boss Dept **Id**)
UNIV ⊨ PROFESSOR < (**fd** PROFESSOR Name Enum Boss **Id**)
UNIV ⊨ PROFESSOR < (**fd** PROFESSOR Name Enum **Id**)
UNIV ⊨ PROFESSOR < (**fd** PROFESSOR Name **Id**)
UNIV ⊨ PROFESSOR < (**fd** PROFESSOR **Id**)

[11] A general procedure that uses our algorithm to diagnose strong object normal form is presented later in Section 4.7.

(Since only the last question is not true, we have that attribute Name is a minimal key.)

$$\text{UNIV} \models \text{PROFESSOR} < (\textbf{fd}\ \text{PROFESSOR Enum Boss Dept } \textbf{Id})$$

(Since this is true, we have that Name is not a *unique* minimal key, and the result follows.)

Again note that all **fd** concept constructors occur within symmetric uniqueness constraints in this application. However, in this case, the only problem parameters that remain fixed are the terminology and the choice of primitive concept. It would therefore be desirable for a membership problem algorithm to be incremental relative to these two parameters as well; that is, subsequent checks for **fd** constraint membership problems should run more efficiently if they only differ in their left and right-hand-sides. This is particularly true if one were to proceed to check if PROFESSOR is in BCNF (and, therefore, in at least weak object normal form) since diagnosing BCNF is likely to require a much larger number of checks for **fd** constraint membership.

4.4 Acceptor Trees and Procedure *Prop*

Our overall approach to having the flexibility needed for these applications is similar to the standard algorithm for the **fd** membership test; we define a data structure, a variation of a description graph, that "grows" according to a given terminology, a primitive concept and a left-hand-side. Adding a further subsumption constraint to the terminology or a path function to the left-hand-side simply causes further growth.

In this chapter, we use description graphs that are trees and are treated as finite state automatons that *accept* precisely the set of right-hand-side path functions. We call such description graphs *acceptor trees*. Thus, an acceptor tree obtained by the "growing process" for a given terminology T, a primitive concept C and a collection of attribute descriptions $\{Pf_1, \ldots, Pf_m\}$ will accept an attribute description Pf precisely when

$$T \models C < (\textbf{fd}\ C\ Pf_1 \ldots Pf_m\ Pf).$$

Formally, an *acceptor tree* G is a tree $\langle N, E, n\rangle$ consisting of a set of nodes N, a set of directed edges E, and distinguished node $n \in N$ (not necessarily the root of the tree). Each node n' in N has two labels: (1) a finite set $Cls(n')$ of concept descriptions, and (2) a finite set $Pfs(n')$ of path functions. The root node is denoted $Root(G)$. Also, each edge is labeled with a primitive attribute A, and as we discussed before, we write $\langle n_1, A, n_2\rangle$ to denote an edge from n_1 to n_2 labeled A.

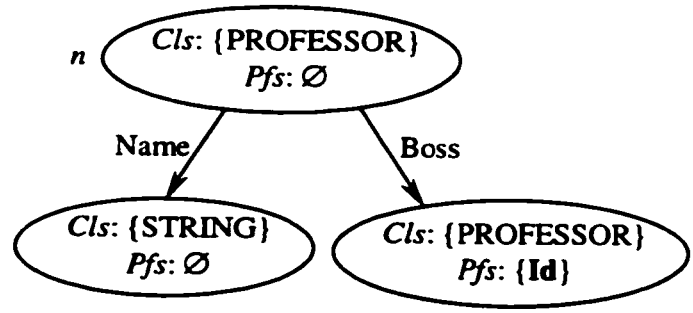
Intuitively, nodes of an acceptor tree represent some objects in an interpretation domain, and every *Cls* label contains concept names to which objects represented by the corresponding node must belong. Edges and their labels represent attributes of the objects. Finally, *Pfs* labels of the nodes are used to identify a set of attribute descriptions on which any two objects represented by the root node must agree. All of the constrains captured by an acceptor tree are derived from a given terminology and a posed question.

Note that in general, acceptor trees are description graphs with a few restrictions. First, acceptor trees are in fact trees with a set of edges rather than arbitrary (description) graphs with a bag of edges. Secondly, acceptor trees do not contain **Id** edges; and finally, while we could use the *Refs* set to store the reference to the distinguished node, it is not necessary since acceptor trees do not use other references. Thus, for the purposes of this chapter, it is sufficient and more convenient to use the distinguished node by itself as the third component of the graph rather than a reference to it.

We say that G ($= \langle N, E, n \rangle$) *accepts* attribute description Pf if and only if either of the following conditions hold:

- $Pfs(n)$ contains a "prefix" of Pf (i.e. there exists $Pf' \in Pfs(n)$ and Pf'' such that $Pf \circ \mathbf{Id}$ is the same as $Pf' \circ Pf''$), or
- $Pf \circ \mathbf{Id}$ has the form "(**comp** A Pf')" and there exists $\langle n, A, n' \rangle \in E$ such that the acceptor tree $\langle N, E, n' \rangle$ accepts Pf'.

For example, consider the following acceptor tree G with distinguished node n.



G accepts attribute descriptions Boss, Boss ∘ Name, and in general any other attribute description that starts with Boss. However, note from our definitions that G does *not* accept Name, despite the existence of both the edge labeled Name and the node to which it points. The reason to allow this relates to the need to remember deductions that will be relevant to subsequent **fd** membership problems that differ only in the left or right-hand-side (as illustrated by the example applications outlined above). In particular, note that G does say (in the sense formally characterized in Lemma 4.3 below) that a professor must have a name that must be a string. It will turn out that,

by first resetting all *Pfs* labels to be empty sets, G can be reused to search for the existence of other functional constraints satisfied by professors.

Let C, $\{Pf_1, \ldots, Pf_m\}$ and T denote a primitive concept, a set of attribute descriptions and a terminology, respectively. Then the *initial acceptor tree* $Init(C, \{Pf_1, \ldots, Pf_m\})$ $(= \langle N, E, n\rangle)$ is the acceptor tree with a single node $\langle\{n\}, \varnothing, n\rangle$ where $Cls(n)$ and $Pfs(n)$ are initialized to $\{C\}$ and $\{Pf_1 \circ \mathbf{Id}, \ldots, Pf_m \circ \mathbf{Id}\}$, respectively. The *propagation of an acceptor tree* G $(= \langle N, E, n\rangle)$ *relative to* T (what we have earlier referred to as a "growing process" or a "propagation procedure"), written $Prop(G, T)$, transforms G by an exhaustive application of the following rewrite rules.

P1. (*composition*) If there exist $n_1 \in N$ and Pf_1 (= "(**comp** A Pf_2)") $\in Pfs(n_1)$, then remove Pf_1 from $Pfs(n_1)$. If $\mathbf{Id} \notin Pfs(n_1)$ then:
   (a) Find $n_2 \in N$ such that $\langle n_1, A, n_2\rangle \in E$. If no such node exists then add a new node n_2 with both its *Cls* and *Pfs* labels initialized to $\varnothing$, and add $\langle n_1, A, n_2\rangle$ to E.
   (b) Add Pf_2 to $Pfs(n_2)$.

P2. (*inheritance*) If there exist $n_1 \in N$ and "$C_1 < C_2$" $\in T$ such that $C_1 \in Cls(n_1)$ and $C_2 \notin Cls(n_1)$, then add C_2 to $Cls(n_1)$.

P3. (*typing*) If there exist $\langle n_1, A, n_2\rangle \in E$ and "$C_1 <$ (**all** A C_2)" $\in T$ such that $C_1 \in Cls(n_1)$ and $C_2 \notin Cls(n_2)$, then add C_2 to $Cls(n_2)$.

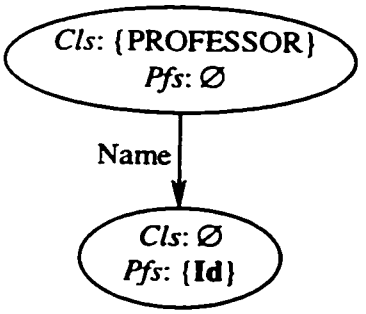
P4. (*uniqueness*) If there exist $n_1 \in N$, $C_1 \in Cls(n_1)$ and "$C_1 <$ (**fd** C_2 Pf_4 ... Pf_m Pf)" $\in T$ such that:
   (a) (*symmetry*) $C_1 = C_2$,
   (b) (*regularity*) there exist $4 \le i \le m$ and path functions Pf_1, Pf_2 and Pf_3 such that $Pf_i = Pf_1 \circ Pf_2$, $Pf = Pf_1 \circ Pf_3$ and $Len(Pf_3) \le 1$,
   (c) $\langle N, E, n_1\rangle$ accepts Pf_j, for all $4 \le j \le m$, and
   (d) $\langle N, E, n_1\rangle$ does *not* accept Pf,
   then add Pf to $Pfs(n_1)$.

(The following rule is optional; it can be added to simplify implementation, and has very little impact on our analytic results.)

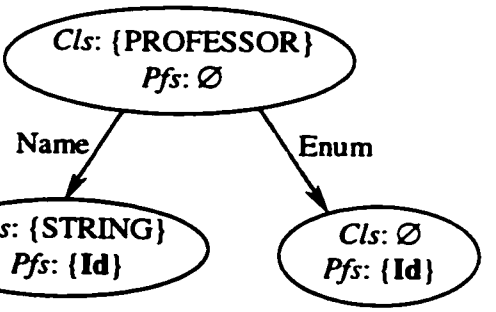
P5. (*simplification*) If there exists $n_1 \in N$ with at least one child and such that $\mathbf{Id} \in Pfs(n_1)$, then remove all other nodes reachable from n_1 along with their incident arcs.

Observe that every transformation obtains a tree when provided with a tree as input. This can be seen in the example of computing *Prop* in Figure 4.2 below. The example presents the sequence of changes made to the acceptor tree G = *Init*(PROFESSOR, {Name}) relative to

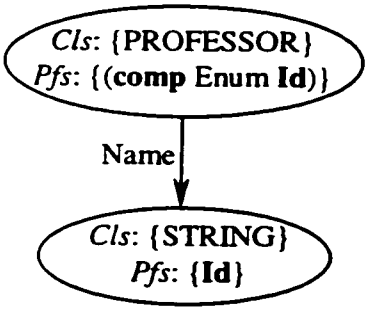
(a) *Init*(PROFESSOR, {Name}).



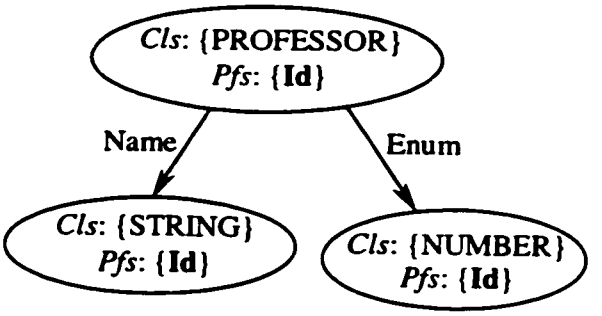
(b) By composition.



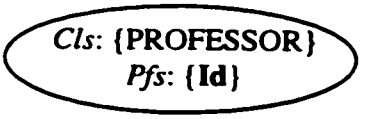
(d) By composition.



(c) By typing and uniqueness.



(e) By typing and uniqueness.



(f) By simplification.

Figure 4.2: Evaluating *Prop*(*Init*(PROFESSOR, {Name}), UNIV).

terminology UNIV, i.e. the set of subsumption constraints appearing in Figure 4.1. Note that the final state of G is an acceptor tree with distinguished node *Root*(G) that accepts any attribute description Pf. And therefore, by the results of the next section, it will follow that

$$\text{UNIV} \models \text{PROFESSOR} < (\textbf{fd}\ \text{PROFESSOR Name}\ Pf).$$

Two observations on the rule of uniqueness are in order. First, condition 4(a) ensures that the propagation of an acceptor tree is unaffected by asymmetric uniqueness constraints (which are beyond the scope of this chapter). And second, condition 4(b) is needed to ensure that *Prop* terminates. For an example of a case in which *Prop* would *not* terminate if condition 4(b) were absent, consider the following terminology T,

$$\{C < (\textbf{all}\ A\ C),\ C < (\textbf{fd}\ C\ B\ (\textbf{comp}\ A\ B))\},$$

and the effect of a call of the form *Prop*(*Init*(C, $\{B\}$), T). Here, the acceptor tree grows indefinitely. Indeed, the condition reflects the result of our best efforts to find the most general class of symmetric uniqueness constraints for which termination of *Prop is* ensured.

4.5 Analysis

Before we present the main results of our analysis of the propagation procedure, we introduce some additional concepts and notation that will be needed in the upcoming proofs. Let G ($= \langle N, E, n\rangle$) denote an acceptor tree and n_1 an arbitrary node in G. The *root path function of* n_1 *in* G, denoted *RootPf*(n_1, G), is recursively defined as follows: **Id** if $n_1 = Root(G)$; and *RootPf*(n_2, G) $\circ A$, where $\langle n_2, A, n_1\rangle \in E$, otherwise. Informally, *RootPf*(n_1, G) is the path function that consists of the primitive attributes labeling the sequence of edges from the root of G to n_1. Now let $\{Pf_3, \ldots, Pf_m\}$ denote a set of path functions, T a terminology and $PA(T)$ the set of all primitive attributes occurring in T. We define *Prefix*($\{Pf_3, \ldots, Pf_m\}$) and *Boundary*($\{Pf_3, \ldots, Pf_m\}$, T) as the respective sets of path functions:

$$\{Pf_1 \circ \textbf{Id} \mid \exists\ Pf_2 \text{ and } 3 \le i \le m\text{: } Pf_1 \circ Pf_2 = Pf_i\}\text{; and}$$
$$\textit{Prefix}(\{Pf_3, \ldots, Pf_m\}) \cup \{Pf_1 \circ A \mid Pf_1 \in \textit{Prefix}(\{Pf_3, \ldots, Pf_m\}) \wedge A \in PA(T)\}.$$

It is a straightforward consequence of our definitions that for any regular symmetric uniqueness constraint that occurs in a terminology T, $Pf \in \textit{Boundary}(\{Pf_3, \ldots, Pf_m\}, T)$. Another consequence of the definitions is that a uniqueness constraint occurring in a terminology T is

symmetric and regular if and only if it satisfies conditions (a) and (b) of the rule of uniqueness of procedure *Prop*.

4.5.1 Termination

We begin the analysis of our procedure by first establishing that the procedure always terminates:

Theorem 4.1 (*termination*) *Let G and T denote an arbitrary finite acceptor tree, i.e. an acceptor tree with finite number of nodes and edges, and an atomic terminology, respectively. Then Prop(G, T) terminates.*

The proof of the theorem will rely on the following additional definitions and observations. Given an acceptor tree G (= $\langle N, E, n \rangle$) and an atomic terminology T, we define acceptor tree prefix and boundary, written $ATPrefix(G)$ and $ATBoundary(G, T)$, as the respective sets:

$$Prefix(\{Pf_1 \circ Pf_2 \mid \exists\, n' \in N\colon Pf_1 = RootPf(n', G) \wedge Pf_2 \in Pfs(n') \cup \{\mathbf{Id}\}\})\text{; and}$$
$$Boundary(\{Pf_1 \circ Pf_2 \mid \exists\, n' \in N\colon Pf_1 = RootPf(n', G) \wedge Pf_2 \in Pfs(n') \cup \{\mathbf{Id}\}\}, T).$$

Observation 4.1 *G will have a finite number of nodes if and only if ATBoundary(G, T) is finite (independent of choice of terminology T).* This follows from the fact that the latter contains $RoofPf(n', G) \circ \mathbf{Id}$ for every node n' in G, that $Pfs(n')$ is finite, and that the number of primitive attributes in T is also finite.

Observation 4.2 *During an invocation of Prop, only rule P4 (uniqueness) can add a path function to ATBoundary(G, T).*

Lemma 4.1 *Let G_0 (= $\langle N, E, n \rangle$) denote an arbitrary finite acceptor tree, T denote an atomic terminology, and $[G_1, G_2, \ldots]$ denote a sequence of acceptor trees obtained by a sequence of applications of rewrite rules defined by $Prop(G_0, T)$. Then, for any $i \geq 0$,*

$$ATBoundary(G_i, T) \subseteq ATBoundary(G_0, T).$$

Proof. We need to show that $RootPf(n', G_i) \circ Pf$ is in $ATBoundary(G_0, T)$, for every $i \geq 0$, n' occurring in G_i, and $Pf \in Pfs(n') \cup \{\mathbf{Id}\}$, which we prove by contradiction.

Let i denote the smallest integer such that G_i (= $\langle N_i, E_i, n\rangle$) contains the first node n_1 with $Pf \in Pfs(n_1) \cup \{\mathbf{Id}\}$ such that $RootPf(n_1, G_i) \circ Pf \notin ATBoundary(G_0, T)$. Clearly $i > 0$, and by Observation 4.2, it follows that the i-th update must have been an application of rule P4 in which Pf was added to $Pfs(n_1)$. Thus, there exists a constraint "$C <$ (**fd** $C\ Pf_4 \ldots Pf_m\ Pf$)" in T with $C \in Cls(n_1)$ such that:

(a) there exist $4 \le j \le m$ and path functions Pf_1, Pf_2 and Pf_3 such that $Pf_j = Pf_1 \circ Pf_2$, $Pf = Pf_1 \circ Pf_3$ and $Len(Pf_3) \le 1$,

(b) $\langle N_{i-1}, E_{i-1}, n_1\rangle$ accepts Pf_k, for $4 \le k \le m$, and

(c) $\langle N_{i-1}, E_{i-1}, n_1\rangle$ does *not* accept Pf.

Conditions (a) and (b) imply that there exist path functions Pf', Pf'' and Pf''' such that:

(d) $Pf' \circ Pf'' \circ Pf''' = Pf_j = Pf_1 \circ Pf_2$, and

(e) there exists a node $n_2 \in N_{i-1}$ reachable from n_1 such that $RootPf(n_2, G_{i-1})$ is the same as $RootPf(n_1, G_{i-1}) \circ Pf'$, and $Pf'' \in Pfs(n_2)$.

By choice of i and condition (e), we have that

$$RootPf(n_2, G_{i-1}) \circ Pf'' \in ATBoundary(G_0, T),$$

and therefore, by condition (e) again, $RootPf(n_1, G_{i-1}) \circ Pf' \circ Pf'' \in ATBoundary(G_0, T)$. But then, conditions (c) and (d) imply that $Pf_1 \in Prefix(\{Pf' \circ Pf''\}) - \{Pf' \circ Pf''\}$, and therefore, that $RootPf(n_1, G_{i-1}) \circ Pf_1 \in ATPrefix(G_0)$. Thus, since $Len(Pf_3) \le 1$ and $Pf = Pf_1 \circ Pf_3$, it follows that $RootPf(n_1, G_{i-1}) \circ Pf \in ATBoundary(G_0, T)$. Finally, since the i-th update does not modify any ancestors of n_1, $RootPf(n_1, G_{i-1}) = RootPf(n_1, G_i)$, and therefore, $RootPf(n_1, G_i) \circ Pf \in ATBoundary(G_0, T)$—contrary to assumptions.

■

Proof of termination. To prove termination, note that Observation 4.1 above and the fact that we have started with a finite acceptor tree imply that $ATBoundary(G_0, T)$ is finite. Therefore, by Lemma 4.1, $ATBoundary(G_i, T)$ is also finite, and Observation 4.1 again implies that G_i is finite. Moreover, by definition of $ATBoundary(G, T)$, the number of nodes in G_i cannot exceed the cardinality of $ATBoundary(G_i, T)$ which in turn is not greater than the cardinality of $ATBoundary(G_0, T)$.

Next consider what happens when we disallow any application of rule P5. Observe that rules P2, P3 and P4 can "fire" at most a finite number of times since (1) the number of nodes is bounded by the cardinality of $ATBoundary(G_0, T)$, (2) no node is ever removed by an application

of rules other than rule P5, and (3) each rule may apply at most once to a given combination of a node and a subsumption constraint in T. Therefore, the total number k of **comp** constructors installed by rule P4 in the *Pfs* labels of nodes is finite, and this in turn ensures that rule P1 can fire at most a finite number of times (since each application decrements k). Thus, the sequence S of applications of rules P1 through P4 is finite.

Now consider the introduction of an application of rule P5 on some node n in S at position j. Clearly, no application of rule P4 occurs after position j in S on node n (condition (d) of rule P4 cannot be satisfied), and no application of rule P1 after position j in S will result in the addition of a new child of n. Since any application of any rule after position j on a descendent of n must be removed from S, since rule P5 itself cannot reapply to node n, and since the application of rule P5 does not make preconditions of other rules true, the resulting sequence must remain finite and the theorem follows.

■

4.5.2 Soundness

The following theorem establishes soundness of the procedure.

Theorem 4.2 (*soundness*) *Let* C, $\{Pf_1, \ldots, Pf_m\}$ *and* T *denote an arbitrary primitive concept, a set of attribute descriptions and an atomic terminology, respectively. Then*

$$T \models C < (\textbf{fd}\ C\ Pf_1 \ldots Pf_m\ Pf)$$

if $Prop(Init(C, \{Pf_1, \ldots, Pf_m\}), T)$ *accepts* Pf.

To prove this theorem, we start by introducing a number of inference axioms in the following lemma.

Lemma 4.2 *For any terminology* T, *primitive concepts* C *and* C_i $(1 \leq i \leq 3)$, *non-negative integers* m *and* k, *and attribute descriptions* Pf, Pf', Pf_j *and* Pf_j' for any $j \geq 1$:

(1) $T \models C < (\textbf{fd}\ C\ Pf_1 \ldots Pf_m\ Pf_j)$, *for* $1 \leq j \leq m$;

(2) $T \models C < (\textbf{all}\ \textbf{Id}\ C)$;

(3) *If* $T \models C_1 < (\textbf{all}\ Pf\ C_2)$ *and* $T \models C_2 < C_3$, *then* $T \models C_1 < (\textbf{all}\ Pf\ C_3)$;

(4) *If* $T \models C_1 < (\textbf{all}\ Pf_1\ C_2)$ *and* $T \models C_2 < (\textbf{all}\ Pf_2\ C_3)$, *then* $T \models C_1 < (\textbf{all}\ Pf_1 \circ Pf_2\ C_3)$;

(5) *If* $T \models C < (\textbf{fd}\ C\ Pf_1 \ldots Pf_m\ Pf)$, *then* $T \models C < (\textbf{fd}\ C\ Pf_1 \ldots Pf_m\ Pf \circ Pf')$;

(6) *If* $T \models C_1 <$ (**all** $Pf\ C_2$) *and* $T \models C_2 <$ (**fd** $C_2\ Pf_1 \ldots Pf_m\ Pf'$), *then* $T \models C_1 <$ (**fd** $C_1\ Pf \circ Pf_1 \ldots Pf \circ Pf_m\ Pf \circ Pf'$);

(7) *If* $T \models C <$ (**fd** $C\ Pf_1 \ldots Pf_k\ Pf_j'$), *for* $1 \leq j \leq m$, *and* $T \models C <$ (**fd** $C\ Pf_1' \ldots Pf_m'\ Pf$), *then* $T \models C <$ (**fd** $C\ Pf_1 \ldots Pf_k\ Pf$).

Proof.

Properties (1) and (2). The properties follow directly from definition of interpretation of the **fd** and **all** constructors.

Property (3). Consider an interpretation **I** that satisfies T and an object x in $C_1^{\mathbf{I}}$. Then, by the first assumption of the property, $Pf^{\mathbf{I}}(x) \in C_2^{\mathbf{I}}$. However, since the second assumption implies that $C_2^{\mathbf{I}} \subseteq C_3^{\mathbf{I}}$, $Pf^{\mathbf{I}}(x) \in C_3^{\mathbf{I}}$. Thus, every object in $C_1^{\mathbf{I}}$ is also in (**all** $Pf\ C_3$)$^{\mathbf{I}}$.

Property (4). Again, consider an interpretation **I** that satisfies T and an object x in $C_1^{\mathbf{I}}$. Then, by the first assumption of the property, $Pf_1^{\mathbf{I}}(x) \in C_2^{\mathbf{I}}$, which in turn implies that $Pf_2^{\mathbf{I}}(Pf_1^{\mathbf{I}}(x)) \in C_3^{\mathbf{I}}$ by the second assumption. Therefore, by property (4) of Claim 3.1, $(Pf_1 \circ Pf_2)^{\mathbf{I}}(x) \in C_3^{\mathbf{I}}$ for any object x in $C_1^{\mathbf{I}}$.

Property (5). The property follows directly from definition of interpretation of the **fd** constructor, from property (4) of Claim 3.1, and from the fact that interpretation of any path function Pf' is a total function.

Property (6). Proof is by contradiction. Let us assume that there is an interpretation **I** that satisfies T, and there exist two objects x_1 and x_2 in $C_1^{\mathbf{I}}$, such that $(Pf \circ Pf_i)^{\mathbf{I}}(x_1) = (Pf \circ Pf_i)^{\mathbf{I}}(x_2)$ for $1 \leq i \leq m$, and $(Pf \circ Pf')^{\mathbf{I}}(x_1) \neq (Pf \circ Pf')^{\mathbf{I}}(x_2)$. Thus, by property (4) of Claim 3.1, $Pf_i^{\mathbf{I}}(Pf^{\mathbf{I}}(x_1)) = Pf_i^{\mathbf{I}}(Pf^{\mathbf{I}}(x_2))$ for $1 \leq i \leq m$, and $Pf'^{\mathbf{I}}(Pf^{\mathbf{I}}(x_1)) \neq Pf'^{\mathbf{I}}(Pf^{\mathbf{I}}(x_2))$. Therefore, if we let y_1 denote $Pf^{\mathbf{I}}(x_1)$ and y_2 $Pf^{\mathbf{I}}(x_2)$, $Pf_i^{\mathbf{I}}(y_1) = Pf_i^{\mathbf{I}}(y_2)$ for $1 \leq i \leq m$, and $Pf'^{\mathbf{I}}(y_1) \neq Pf'^{\mathbf{I}}(y_2)$. By the first assumption of the property, both y_1 and y_2 belong to $C_2^{\mathbf{I}}$. Then, however, restrictions on y_1 and y_2 above contradict the second assumption of the property.

Property (7). Consider an arbitrary interpretation **I** that satisfies T, and any two objects x_1 and x_2 in $C^{\mathbf{I}}$ such that $Pf_i^{\mathbf{I}}(x_1) = Pf_i^{\mathbf{I}}(x_2)$ for $1 \leq i \leq k$. Then, the first assumption of the property implies that $Pf_j'^{\mathbf{I}}(x_1) = Pf_j'^{\mathbf{I}}(x_2)$ for $1 \leq j \leq m$, which in turn implies that $Pf^{\mathbf{I}}(x_1) = Pf^{\mathbf{I}}(x_2)$ by the second assumption of the property.

■

The following lemma introduces an invariant preserved by the *Prop* procedure and is essentially the backbone of the proof of soundness that follows the lemma.

Lemma 4.3 *Let C, $\{Pf_1, \ldots, Pf_m\}$ and T denote an arbitrary primitive concept, a set of attribute descriptions and an atomic terminology, respectively. Also, let G_0 ($= \langle N_0, E_0, n\rangle$) denote the initial acceptor tree $Init(C, \{Pf_1, \ldots, Pf_m\})$ and $G_1, \ldots, G_k$ a (necessarily finite) sequence of acceptor trees ($= \langle N_i, E_i, n\rangle$, $1 \le i \le k$) obtained by a sequence of applications of rewrite rules defined by $Prop(G_0, T)$. Then, for all $0 \le i \le k$:*

(1) *for all $n' \in N_i$, and $Pf \in Pfs(n')$: $T \models C < (\mathbf{fd}\ C\ Pf_1 \ldots Pf_m\ RootPf(n', G_i) \circ Pf)$;*

(2) *for all $n' \in N_i$, and $C' \in Cls(n')$: $T \models C < (\mathbf{all}\ RootPf(n', G_i)\ C')$; and*

(3) *for all $n' \in (N_i \cap N_{i-1})$: $RootPf(n', G_{i-1}) = RootPf(n', G_i)$.*

Proof. Property (3) can be verified by a simple inspection of the rules comprising our *Prop* procedure. We prove properties (1) and (2) by induction on i.

For the base case, G_0 consists of a single node n with labels $Cls(n)$ and $Pfs(n)$ initialized to $\{C\}$ and $\{Pf_1 \circ \mathbf{Id}, \ldots, Pf_m \circ \mathbf{Id}\}$, respectively. (2) is therefore an immediate consequence of Lemma 4.2(2). Lemma 4.2(1) implies

$$T \models C < (\mathbf{fd}\ C\ Pf_1 \ldots Pf_m\ Pf_j)$$

for $1 \le j \le m$, and since $RootPf(n, G_0) \circ Pf_j = Pf_j \circ \mathbf{Id}$, (1) must also hold.

Now assume the lemma holds for all $i < l \le k$, and consider each of the five possible rules that might have obtained G_l. (The rules themselves are reproduced for convenience.)

P1. (*composition*) If there exist $n_1 \in N$ and Pf_1 (= "(**comp** A Pf_2)") $\in Pfs(n_1)$, then remove Pf_1 from $Pfs(n_1)$. If $\mathbf{Id} \notin Pfs(n_1)$ then:

(a) Find $n_2 \in N$ such that $\langle n_1, A, n_2\rangle \in E$. If no such node exists then add a new node n_2 with both its *Cls* and *Pfs* labels initialized to $\emptyset$, and add $\langle n_1, A, n_2\rangle$ to E.

(b) Add Pf_2 to $Pfs(n_2)$.

Observe that no primitive concept is added to any *Cls* label by the rule of composition. Thus, (2) must hold for G_l by the inductive assumption and (3). Also, by the inductive assumption,

$$T \models C < (\mathbf{fd}\ C\ Pf_1 \ldots Pf_m\ RootPf(n_1, G_{l-1}) \circ (\mathbf{comp}\ A\ Pf_2)).$$

And since the rule ensures $RootPf(n_2, G_l) = RootPf(n_1, G_l) \circ A$ and by (3) we have

$$RootPf(n_1, G_l) \circ A = RootPf(n_1, G_{l-1}) \circ A,$$

it follows that

$$T \models C < (\textbf{fd}\ C\ Pf_1 \ldots Pf_m\ RootPf(n_2, G_l) \circ Pf_2).$$

Thus, property (1) must also hold for G_l.

P2. (*inheritance*) If there exist $n_1 \in N$ and "$C_1 < C_2$" $\in T$ such that $C_1 \in Cls(n_1)$ and $C_2 \notin Cls(n_1)$, then add C_2 to $Cls(n_1)$.

Since *Pfs* labels remain unchanged, (1) must hold for G_l by the inductive assumption and property (3). Also, by the inductive assumption,

$$T \models C < (\textbf{all}\ RootPf(n_1, G_{l-1})\ C_1);$$

and by (3), the precondition that "$C_1 < C_2$" $\in T$ and Lemma 4.2(3), it follows that

$$T \models C < (\textbf{all}\ RootPf(n_1, G_l)\ C_2).$$

Thus, property (2) must also hold for G_l.

P3. (*typing*) If there exist $\langle n_1, A, n_2 \rangle \in E$ and "$C_1 < (\textbf{all}\ A\ C_2)$" $\in T$ such that $C_1 \in Cls(n_1)$ and $C_2 \notin Cls(n_2)$, then add C_2 to $Cls(n_2)$.

(A similar line of argument to the previous case applies by substituting a reference to Lemma 4.2(4) in place of Lemma 4.2(3).)

P4. (*uniqueness*) If there exist $n_1 \in N$, $C_1 \in Cls(n_1)$ and "$C_1 < (\textbf{fd}\ C_2\ Pf_4' \ldots Pf_h'\ Pf')$" $\in T$ such that:[12]

(a) (*symmetry*) $C_1 = C_2$,

(b) (*regularity*) there exist $4 \le i \le h$ and path functions Pf_1', Pf_2' and Pf_3' such that $Pf_i' = Pf_1' \circ Pf_2'$, $Pf' = Pf_1' \circ Pf_3'$ and $Len(Pf_3') \le 1$,

[12] Note that notation is adjusted so that path functions inside the rule do not conflict with path functions inside the statement of the lemma.

(c) $\langle N, E, n_1 \rangle$ accepts Pf_j', for all $4 \leq j \leq h$, and

(d) $\langle N, E, n_1 \rangle$ does *not* accept Pf',

then add Pf' to $Pfs(n_1)$.

Since no primitive concept is added to any Cls label by the rule of uniqueness, property (2) must hold for G_l by the inductive assumption and property (3). From condition (b), condition (c), property (3) and definition of acceptance, there must exist nodes n_j and path functions Pf_j^{path}, Pf_j^{label} and Pf_j^{tail}, for each $4 \leq j \leq h$, such that

$$Pf_j' = Pf_j^{path} \circ Pf_j^{label} \circ Pf_j^{tail}, \quad (1)$$

$$RootPf(n_j, G_l) = RootPf(n_1, G_l) \circ Pf_j^{path}, \text{ and} \quad (2)$$

$$Pf_j^{label} \text{ occurs inside label } Pfs(n_j). \quad (3)$$

The inductive assumption, together with (2) and (3), implies that

$$T \models C < (\textbf{fd } C\ Pf_1 \ldots Pf_m\ RootPf(n_1, G_l) \circ Pf_j^{path} \circ Pf_j^{label})$$

must hold for each $4 \leq j \leq h$. And then from (1) and Lemma 4.2(5), we have that

$$T \models C < (\textbf{fd } C\ Pf_1 \ldots Pf_m\ RootPf(n_1, G_l) \circ Pf_j'), \text{ for each } 4 \leq j \leq h. \quad (4)$$

Now consider that $C_1 \in Cls(n_1)$ together with properties (2) and (3) imply that

$$T \models C < (\textbf{all } RootPf(n_1, G_l)\ C_1).$$

Then Lemma 4.2(6) and the fact that $C_1 < (\textbf{fd } C_1\ Pf_4' \ldots Pf_h'\ Pf')$ occurs in T imply

$$T \models C < (\textbf{fd } C\ RootPf(n_1, G_l) \circ Pf_4' \ldots RootPf(n_1, G_l) \circ Pf_h'\ RootPf(n_1, G_l) \circ Pf'), \quad (5)$$

and finally

$$T \models C < (\textbf{fd } C\ Pf_1 \ldots Pf_m\ RootPf(n_1, G_l) \circ Pf')$$

follows from (4), (5) and Lemma 4.2(7). Thus, (1) continues to hold following the addition of the path function Pf' to $Pfs(n_1)$.

P5. If there exists $n_1 \in N$ with at least one child and such that $\mathbf{Id} \in Pfs(n_1)$, then remove all other nodes reachable from n_1 along with their incident arcs.

Since there is no modification to any *Cls* or *Pfs* labels, both (1) and (2) must continue to hold on G_t.

■

Proof of soundness. To prove soundness, let now G denote the acceptor tree computed by

$$Prop(Init(C, \{Pf_1, \ldots, Pf_m\}), T),$$

and let Pf denote an attribute description accepted by G. Then G must contain a node n and there must exist path functions Pf' and Pf'' such that $Pf \circ \mathbf{Id} = RootPf(n, G) \circ Pf' \circ Pf''$ and such that Pf' occurs in $Pfs(n)$. Thus, condition (1) of Lemma 4.3 and Lemma 4.2(5) imply

$$T \models C < (\mathbf{fd}\ C\ Pf_1 \ldots Pf_m\ \ Pf \circ \mathbf{Id}),$$

and since $(\textit{left-hand-side})^{\mathbf{I}} = (\textit{right-hand-side})^{\mathbf{I}}$ for every rewrite rule occurring in Table 4.1(b) and every interpretation $\mathbf{I}$, it then follows that $Pf^{\mathbf{I}} = (Pf \circ \mathbf{Id})^{\mathbf{I}}$ for every interpretation $\mathbf{I}$, and therefore, that

$$T \models C < (\mathbf{fd}\ C\ Pf_1 \ldots Pf_m\ Pf).$$

■

4.5.3 Completeness

Next, completeness of our procedure is established in the following theorem.

Theorem 4.3 (*completeness*) *Let C, $\{Pf_1, \ldots, Pf_m\}$ and T denote an arbitrary primitive concept, a set of attribute descriptions and an atomic terminology, respectively. Then, if all uniqueness constraints occurring in T are symmetric and regular,*

$$Prop(Init(C, \{Pf_1, \ldots, Pf_m\}), T) \textit{ accepts } Pf$$

if $T \models C < (\mathbf{fd}\ C\ Pf_1 \ldots Pf_m\ Pf)$.

Proofs of completeness for the procedures presented in this and the next chapters are analogous. However, since our second procedure is more complicated and can potentially produce cyclic graphs, we present a detailed proof of completeness in the next chapter. On the other hand, while

the proof presented here contains most of the necessary details, it does appeal to intuition on a couple of occasions (for example, by not exactly defining the notion of the *path* in the graph in the proofs of assertions 7 and 8) in order to simplify the presentation and to allow one to follow the general line of argument more easily. Thus, this proof also serves as a guideline for the more difficult proof in Subsection 5.6.3, and at the same time, the details that are left out here can be easily extracted from the next chapter.

Proof of completeness. The overall strategy is to assume that

$$Prop(Init(C, \{Pf_1, \ldots, Pf_m\}), T)$$

does *not* accept Pf, and then to construct an interpretation that satisfies T and does *not* satisfy constraint "$C <$ (**fd** C Pf_1 ... Pf_m Pf)". Our construction starts from the output of procedure $Prop(Init(C, \{Pf_1, \ldots, Pf_m\}), T)$, and considers a "merge" of two identical copies of the resulting tree. The roots of the two trees will represent two objects violating the subsumption constraint, while the rest of the structure will provide the remaining details for an interpretation. After the above-mentioned merge "joins" all isomorphic nodes that have **Id** in their *Pfs* labels, we extend the resulting (description) graph with additional "missing" nodes and edges to represent the attribute values of all the objects in the graph to ensure that the constructed interpretation of every attribute is a total function. Finally, the desired interpretation **I** is obtained by: (1) viewing the set of all nodes as the domain Δ, (2) the interpretation of a concept C as the set of all nodes with C occurring in their *Cls* labels, and (3) the interpretation of an attribute A as the set of pairs of nodes connected by an edge labeled A.

To obtain **I**, we first construct an infinite sequence S_D of (finite) description graphs that starts with acceptor trees

$$G_{-2} = Init(C, \{Pf_1, \ldots, Pf_m\}) = \langle N_{-2}, E_{-2}, dn \rangle \text{ and}$$
$$G_{-1} = Prop(G_{-2}, T) = \langle N_{-1}, E_{-1}, dn \rangle^{13}$$

(we assume that this invocation of *Prop* includes the simplification rule). Note that assuming $m \geq 1$ (the case when $m = 0$ will be discussed at the end of the proof), and due to the composition rule, G_{-1} contains at least one node with **Id** inside its *Pfs* label. Therefore, G_{-1} must contain more than one node, since otherwise, the only node in the tree, i.e. its root, would contain **Id** inside its *Pfs* label, and thus, the tree would accept any attribute description contrary to the assumption that it does not accept Pf. Moreover, due to the simplification rule, only leaves of G_{-1} contain **Id** in their *Pfs* label. Finally, due to the composition and simplification rules, these nodes do not contain any other attribute descriptions in their *Pfs* label, and all other nodes in the graph have empty *Pfs* labels.

[13] Note that neither of the steps of our construction will change the distinguished node.

Next, a copy of tree G_{-1} is constructed, say $G_{-1}' = \langle N_{-1}', E_{-1}', dn' \rangle$, and the following merge of the two trees obtains $G_0 = \langle N_0, E_0, dn \rangle$:

CS_0. Initialize N_0 with $N_{-1} \cup N_{-1}'$, E_0 with $E_{-1} \cup E_{-1}'$, and the distinguished node with dn. Next, for all $n_1 \in N_{-1}$ and $n_1' \in N_{-1}'$ such that $RootPf(n_1, G_{-1}) = RootPf(n_1', G_{-1}')$ and **Id** is in the *Pfs* label of both nodes, and for an edge $\langle n', A, n_1' \rangle \in E_{-1}'$, remove the edge from E_0, add $\langle n', A, n_1 \rangle$ to E_0, and remove node n_1' from N_0.

Note that there must exist a parent for a node n_1' that contains **Id** in its *Pfs* label; or in other words, the edge $\langle n', A, n_1' \rangle$ always exists in the above rule, since otherwise, n_1' would be the root node of G_{-1}'. Then however, G_{-1}', and therefore, G_{-1}, would accept any attribute description, including *Pf*, contrary to the assumption.

It follows that G_0 consists of exactly three non-empty parts. Part 1 contains all nodes (and their outgoing edges if any) that are in N_{-1} and that do not have **Id** in their *Pfs* labels; part 2 contains all nodes (and their outgoing edges) that are in N_{-1}' and that do not have **Id** in their *Pfs* labels; and part 3 contains all nodes that are in N_{-1} and that have **Id** in their *Pfs* labels. Moreover, since **Id** only appears in the *Pfs* labels of the leaves of G_{-1}, part 3 only contains leaf nodes. Finally, due to the structure of G_{-1} that we outlined above, nodes in parts 1 and 2 have empty *Pfs* labels, while nodes in part 3 only contain **Id** in their *Pfs* labels. Graph G_0 is illustrated in Figure 4.3(a) at the end of the proof.

The idea of the construction that follows is to add "missing" attributes to G_0 in a way that eventually ensures that the interpretation of each primitive attribute is a total function. Since nodes in the graphs will be the objects in Δ, and edges will define the interpretation of attributes, each node must have an (outgoing) edge for every attribute in at least one graph. Generally however, we cannot create arbitrary new edges between nodes that already exist in the graph since that might lead to violations of **fd** constraints in T. Thus, for every node n_1 and every primitive attribute A, we create a new node n_2 and an edge $\langle n_1, A, n_2 \rangle$ unless it already exists; deduce the concept(s) that n_2 must "belong" to; and proceed by creating nodes and edges outgoing from n_2. All three parts of G_0 grow with this process.

More formally, let $S_D = [G_{-2}, G_{-1}, G_0, G_1, \ldots, G_i = \langle N_i, E_i, dn \rangle, \ldots]$ denote the infinite sequence of description graphs constructed as follows. G_{-2}, G_{-1}, and G_0 are constructed as defined above. Next, at step i ($i \geq 1$), we add a number of nodes and edges to graph G_{i-1} and invoke procedure *Prop* (without the simplification rule) on the resulting graph to obtain G_i. (Even though we defined *Prop* as a procedure that works on acceptor trees, for the purposes of this proof, we also allow it to work on arbitrary description graphs containing appropriate labels for nodes and edges.) Let N_i' (resp. E_i') denote the set of new nodes (resp. edges) that we add at step i. We will show in Assertion 1 that procedure *Prop* does not add or remove any nodes or edges to or from the resulting graph, and therefore, that $N_i = N_{i-1} \cup N_i'$ and $E_i = E_{i-1} \cup E_i'$.

Let $S_{PA} = [A_1, A_2, \ldots]$ denote a sequence of all primitive attributes[14]. Since we are going to add a countably infinite number of nodes with countably infinite number of outgoing edges, we use a "triangular" construction to keep the graph resulting at each step finite. Thus, we define step i ($i \geq 1$) of the construction, denoted by CS_i, as follows:

> CS_i. For all $n_1 \in N_{i-1}$ and $A_j \in S_{PA}$ ($1 \leq j \leq i$): if there is no n_2 in N_{i-1} such that $\langle n_1, A_j, n_2 \rangle$ is in E_{i-1}, add a new node n_2 to N_i' and edge $\langle n_1, A_j, n_2 \rangle$ to E_i'. G_i is defined as $Prop(\langle N_{i-1} \cup N_i', E_{i-1} \cup E_i', dn \rangle, T)$.

Finally, we define the interpretation **I** as follows:

- $\Delta = N_0 \cup \bigcup_{i \geq 1} N_i'$;
- $C^{\mathbf{I}} = \{n \in \Delta \mid \exists\, G_i\ (i \geq 0)$ such that $n \in N_i$ and $C \in Cls(n)$ inside $G_i\}$ for any primitive concept C; and
- $A^{\mathbf{I}} = \{(n_1, n_2) \mid \exists\, G_i\ (i \geq 0)$ such that $n_1, n_2 \in N_i$ and $\langle n_1, A, n_2 \rangle \in E_i\}$ for any primitive attribute A.

In order to distinguish different invocations of *Prop* procedure during the construction process, we use a subscript to denote the step at which the procedure is executed. Thus, $Prop_i$ ($i \geq 1$) is the procedure invocation at step i. In addition, we denote invocation of *Prop* that constructed graph G_{-1} as $Prop_0$.

Let us now make a number of assertions (and present their proofs) about the nature of this construction and the resulting interpretation **I**.

Assertion 1. *Rules of composition and uniqueness are never applied during an execution of* $Prop_i$ *for any* $i \geq 1$.

Let us assume that the statement is not true, and j is the number of the first step when either uniqueness or composition rule is applied. Also, let $G' = \langle N', E', dn \rangle$ denote the graph right before the rule "fires" for the first time. Since all nodes that are added by CS_i ($i \geq 1$) have empty *Pfs* labels, and the only rules that are applied to G_0 to obtain G' are the rules of inheritance and typing, the only modifications to the nodes and edges of G_0 are the additions of nodes and edges by the CS_i ($1 \leq i \leq j$) steps; and the only nodes that have non-empty *Pfs* labels are the ones that have {**Id**} as their *Pfs* labels (i.e. the leaf nodes "merged" during the construction of G_0). Therefore, the rule that fires next is *not* the rule of composition. In addition, it follows that the

[14] Recall that our assumption that the set of primitive attributes is recursively enumerable ensures that such a sequence exits.

node to which the rule of uniqueness applies, say n_u, belongs to G_0 (since all new nodes that are added by CS_i $(1 \leq i \leq j)$ do not have descendents with **Id** in their *Pfs* labels). However, independently of whether n_u belongs to G_{-1} or G_{-1}', since (a) the structure of those graphs remains unchanged by the construction steps CS_i $(1 \leq i \leq j)$ and by the applications of rules of typing and inheritance; (b) the *Cls* sets of nodes in G_0 are also not modified due to the exhaustive firing of rules of inheritance and typing during the construction of G_{-1}; and (c) no additional incoming edges for the nodes in N_0 are created during the CS_i $(i \geq 1)$ steps, the conditions for firing the rule of uniqueness (on n_u or its isomorphic node if n_u is in G_{-1}') would be satisfied in the graph G_{-1}. However, due to the exhaustive applications of the rules of uniqueness during the construction of G_{-1}, the rule would have been fired. Therefore, condition (d) of the rule would no longer be satisfiable, which implies that it is also not satisfiable in graph G'. Thus, the rule of uniqueness cannot fire. Contradiction.

Note that termination of steps 1 through j above (as well as finiteness of the graphs G_0 through G') is guaranteed by the facts that G_0 is finite, and there is at most a finite number of applications of inheritance and typing rules to a finite number of nodes and edges that are added at each step. At the same time, the assertion itself then guarantees the finiteness of every graph G_i since it establishes that rules of inheritance and typing are the only ones that apply at *every* step CS_i $(i \geq 1)$.

In addition, as we mentioned before, this assertion implies that $N_i = N_{i-1} \cup N_i'$ and $E_i = E_{i-1} \cup E_i'$ $\forall$ $i \geq 1$ since rules of inheritance and typing do not modify nodes and edges in a graph.

Assertion 2. *No step i for* $i \geq 1$ *adds/removes any primitive concepts to/from Cls sets of nodes in* N_{i-1}.

This assertion is a consequence of the facts that every construction step just adds new nodes and edges outgoing from already existing nodes, and by Assertion 1, only rules of inheritance and typing can be applied during the execution of $Prop_i$. First, note that neither of the rules can remove a primitive concept from a *Cls* set of a node. In addition, note that invocation of $Prop_{i-1}$ must have exhaustively explored all applicable rules (if $i = 1$, the rules must have been exhaustively applied during the construction of G_{-1}, and therefore, no rule is applicable in the graph G_0). Thus, the only rules that might be applicable during the invocation of $Prop_i$ are rules of typing that add new concept names to the *Cls* sets of the newly created children (in N_i'). After that, neither of the inheritance or typing rules that apply to nodes in N_i' can possibly affect the *Cls* sets of other nodes. Therefore, $Prop_i$ cannot possibly add new primitive concepts to a *Cls* set of a node in N_{i-1}.

In particular, this assertion implies that if i is the smallest non-negative integer such that a node n is in N_i, $n \in C^{\mathbf{I}}$ for our constructed interpretation **I** if and only if $C \in Cls(n)$ in G_i.

Assertion 3. *For every node $n_1 \in \Delta$, and every $A_i \in S_{PA}$, there is at least one node $n_2 \in \Delta$ such that $(n_1, n_2) \in A^{\mathbf{I}}$.*

Indeed, let j be the smallest non-negative number such that G_j contains n_1 (such j exists since $n_1 \in \Delta$). If there is an edge $\langle n_1, A_i, n_2 \rangle$ in E_j for some node n_2 in N_j, (n_1, n_2) must be in $A_i^{\mathbf{I}}$ by the definition of **I**. Otherwise, a new node n_2 and edge $\langle n_1, A_i, n_2 \rangle$ are created by construction step $max(i, j + 1)$, and thus again, (n_1, n_2) must be in $A^{\mathbf{I}}$.

Assertion 4. *For every node $n_1 \in \Delta$, and every $A_i \in S_{PA}$, there is at most one node $n_2 \in \Delta$ such that $(n_1, n_2) \in A^{\mathbf{I}}$.*

The statement follows from the facts that G_{-1} is a tree; no construction step or an application of a rule adds a new edge if there already exists an edge with the same label and outgoing from the same node; and no edge is ever removed or modified during the construction steps CS_i for all $i \geq 1$.

Assertion 5. **I** *is a valid interpretation.*

This assertion is a straightforward consequence of Assertions 3 and 4 since they ensure that primitive attributes are total functions on Δ.

In order to keep the numbering of the assertions in the proofs of completeness in this chapter and in the next chapter analogous, we skip the assertion number 6 since the analog of the assertion 6 in the next chapter would be a trivial statement in this proof.

Assertion 7. *The distinguished node of G_0 is in $C^{\mathbf{I}}$, and* **I** *satisfies T.*

The first part of the assertion follows directly from Assertion 2, since C is in the *Cls* set of the distinguished node of G_{-2} and remains there throughout the construction process (distinguished node remains unchanged as well).

Next, consider the second part of the assertion. There are three kinds of constraints in T:

(a) $C_1 < C_2$;
(b) $C_1 <$ (**all** A C_2); and
(c) a regular uniqueness constraint $C_1 <$ (**fd** C_1 $Pf_1' \ldots Pf_k'$ Pf').

Let us assume that **I** does not satisfy a constraint of kind (a). Then, there must exist a node n such that n is in $C_1^{\mathbf{I}}$ and not in $C_2^{\mathbf{I}}$. Therefore, by Assertion 2 and definition of **I**, there must exist the smallest non-negative integer i such that graph G_i contains n, C_1 is in $Cls(n)$, and C_2 is not in $Cls(n)$. Then, however, $Prop_i$ would have added C_2 to $Cls(n)$, and therefore, by Assertion 2 (or by construction of G_0 if $i = 0$), n must also belong to $C_2^{\mathbf{I}}$ contrary to the assumption.

Next, let us assume that **I** does not satisfy a constraint of the form (b). Then, there must exist two nodes n and n' such that n is in $C_1^{\mathbf{I}}$, $\langle n, n' \rangle$ is in $A^{\mathbf{I}}$, and n' is not in $C_2^{\mathbf{I}}$. Therefore, by Assertions 2 and 5 and definition of **I**, there must exist the smallest non-negative integer i such that graph G_i contains nodes n and n' and edge $\langle n, A, n' \rangle$, C_1 is in $Cls(n)$, and C_2 is not in $Cls(n')$. Then, however, $Prop_i$ would have added C_2 to $Cls(n')$, and therefore, by Assertion 2 (or by construction of G_0 if $i = 0$), n' must belong to $C_2^{\mathbf{I}}$, contrary to the assumption.

Finally, let us assume that **I** does not satisfy a constraint of the form (c). Then, there must exist nodes n, n', n_1, ..., n_k, n_{k+1}, and n_{k+1}' such that

(1) n and n' are in $C_1^{\mathbf{I}}$;
(2) $Pf_i'^{\mathbf{I}}(n) = Pf_i'^{\mathbf{I}}(n')$ $(= n_i)$ for $1 \leq i \leq k$; and
(3) $Pf'^{\mathbf{I}}(n)$ $(= n_{k+1})$ and $Pf'^{\mathbf{I}}(n')$ $(= n_{k+1}')$ are distinct.

Therefore, by Assertions 2 and 5 and definition of **I**, there must exist the smallest non-negative integer j such that graph G_j contains nodes n, n', n_1, ..., n_k, n_{k+1}, and n_{k+1}', contains a *path* p_i (resp. p_i') from node n (resp. n') to node n_i that passes through the edges labeled with attributes of the path function Pf_i' (for all $1 \leq i \leq k$), and where C_1 is in $Cls(n)$ and in $Cls(n')$. However, since by our construction (and Assertion 1), the only nodes that have more than one parent are the nodes in the part 3 of graph G_0, n and n' must belong to the union of parts 1 and 2 of G_0. Moreover, for every path function Pf_i' $(1 \leq i \leq k)$, there must exist its prefix Pf_i'' such that there is a path that leads from node n (and n') to a node (say n_i') in part 3 of G_0 and that passes through edges with labels corresponding to the attributes of Pf_i'' (see Figure 4.3(b)).

Note that n and n' must be distinct nodes, since otherwise, they would "agree" on the path function Pf', i.e. n_{k+1} and n_{k+1}' would have to be the same. Without loss of generality, let us assume that n is in part 1 of G_0, i.e. $n \in N_{-1}$. Then, our construction of G_0 implies that n' must belong to N_{-1}' and be isomorphic to n. It follows that G_{-1} contains a node n with C in its Cls label. Moreover, $\langle N_{-1}, E_{-1}, n \rangle$ accepts Pf_i' for $1 \leq i \leq k$. Therefore, $Prop_0$ would ensure that $\langle N_{-1}, E_{-1}, n \rangle$ also accepts Pf'. Then, due to the rules of composition and simplification, there would also exist a path in G_{-1} that (a) starts from node n; (b) ends at a node with **Id** in its *Pfs* label; and (c) passes through edges with labels corresponding to the attributes of a prefix of the path function Pf'. Therefore, due to our construction of G_0, nodes n and n' would have to also agree on a prefix of the path function Pf', contrary to condition (3) above.

Assertion 8. *The distinguished node of* G_0 *is* not *in* (**fd** C Pf_1 ... Pf_m Pf)$^{\mathbf{I}}$.

First, note that by the discussion following the definition of the construction step CS_0, *neither dn nor dn'* (the root of G_{-1}') contains **Id** in its *Pfs* label, and therefore, neither of the nodes is removed by the step CS_0. Furthermore, since the *Pfs* label of the distinguished node *dn* contains path functions Pf_i ($1 \le i \le m$) in G_{-2}, by the rule of composition and by construction of G_0, *dn* and *dn'* agree on the paths in G_0 that pass though edges with labels corresponding to the attributes of some prefixes of the path functions Pf_i (for all $1 \le i \le m$). Therefore, by definition of the interpretation **I**, $Pf_i^{\mathbf{I}}(dn) = Pf_i^{\mathbf{I}}(dn')$ ($1 \le i \le m$). In addition, both nodes contain C in their *Cls* labels, and therefore, both belong to $C^{\mathbf{I}}$.

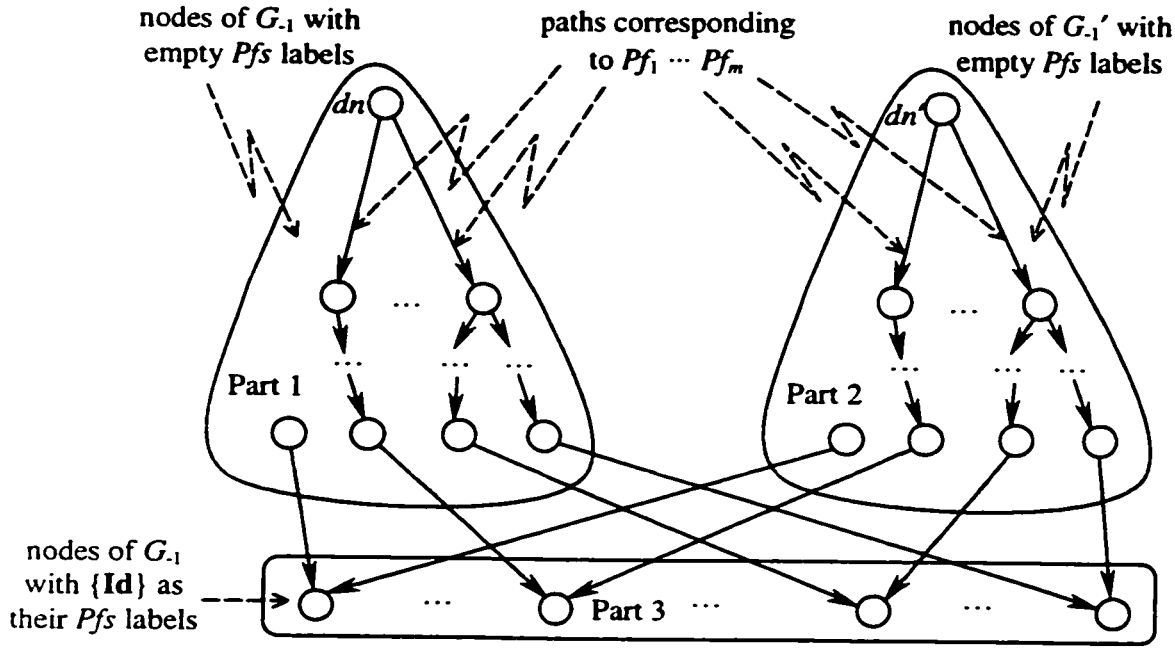
On the other hand, since G_{-1} does not accept the path function Pf, nodes *dn* and *dn'* do *not* agree on a path in G_0 that passes through edges with labels corresponding to the attribute names in any prefix of Pf. Moreover, since our construction steps CS_i for $i \ge 1$ cannot add any nodes with more than one parent, by the definition of **I**, $Pf^{\mathbf{I}}(dn) \neq Pf^{\mathbf{I}}(dn')$. Therefore, the statement of the assertion follows.

Therefore, by Assertions 5, 7 and 8, we constructed an interpretation **I** that satisfies T and contains a node (dn) that is in $C^{\mathbf{I}}$ but *not* in (**fd** C Pf_1 ... Pf_m Pf)$^{\mathbf{I}}$. In other words, **I** satisfies T and does *not* satisfy constraint $C <$ (**fd** C Pf_1 ... Pf_m Pf).

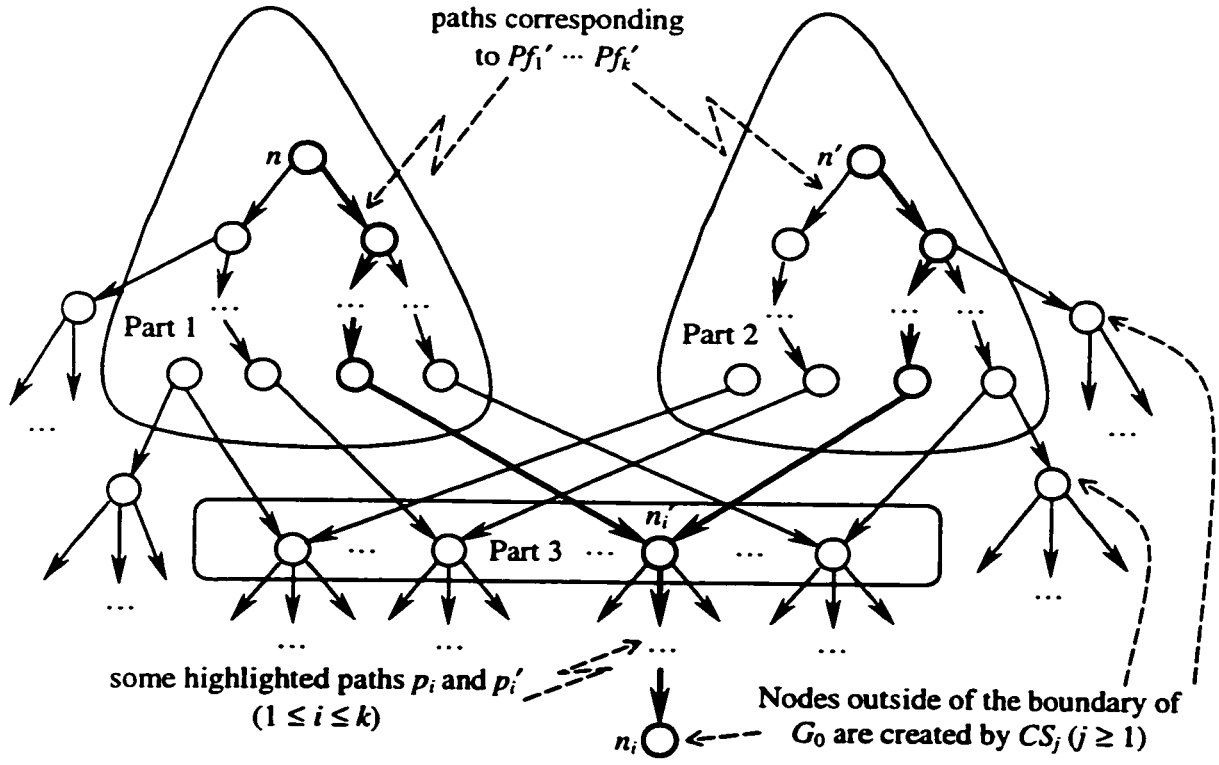
Two final notes are in order. First, the result of the theorem also holds if we consider the invocation of $Prop_0$ (that constructed G_{-1}) *without* the simplification rule. The only change would involve merging descendents of nodes that have **Id** in their *Pfs* labels. (This can be done along the lines of the *Merge* procedure presented in the next chapter.) The descendents would form a forest of trees, and would just slightly complicate the proof. The proof of completeness in the next chapter deals with much more serious complications that includes the one above.

Second, in the case when $m = 0$, our procedure (always) returns false; and indeed, since we require the terminology to contain regular uniqueness constraints, none of the uniqueness constraints in the terminology has the form "$C <$ (**fd** C Pf)", and therefore, none of such constraints can be deduced. To formally show this, one just needs to apply the construction presented above to the case when m is indeed 0. Then, G_0 becomes two unconnected trees with C in the *Cls* labels of the roots. Just like the arguments in the case when $m \ge 1$, it can be shown that the interpretation constructed from the two trees would satisfy terminology T, but at the same time, would clearly violate the constraint "$C <$ (**fd** C Pf)".

■



(a) Graph G_0.



(b) Two nodes that agree on paths corresponding to the path functions Pf_1' through Pf_k'.

Figure 4.3: Construction of the interpretation I in the proof of completeness.

4.5.4 Runtime Complexity

Finally, we consider the runtime complexity of the procedure.

Let G_0 denote $Init(C, \{Pf_1, \ldots, Pf_l\})$, for some primitive concept C and a set of attribute descriptions $\{Pf_1, \ldots, Pf_l\}$, and let G denote $Prop(G_0, T)$, for some terminology T. A straightforward consequence of Lemma 4.1 is that the total number of nodes created during the computation of G will not exceed the size of $ATBoundary(G_0, T)$, which is in turn $O(m \cdot k)$, where m (resp. k) is the encoding length of $\{Pf_1, \ldots, Pf_l\}$ (resp. T). Since each constraint in T will apply at most once to each node, a very loose polynomial bound on the time to compute G is therefore $O(m^2 \cdot k^4)$. This allows $O(m \cdot k^2)$ time for each "iteration" of *Prop* to successfully fire a rule ($O(m \cdot k)$ time to loop through all nodes and $O(k)$ time to check all constraints for each node), together with the possibility that $O(k)$ constraints eventually fire on each node—bounds that would be clearly satisfied by a direct translation of the *Prop* procedure. However, a straightforward indexing of constraints based on their left-hand-side primitive concepts together with a simple optimization that prunes any uniqueness constraints indexed by nodes with **Id** occurring in the *Pfs* label will yield a tighter $O(m^2 \cdot k^2)$ bound on the time to compute G.

Indeed, consider a procedure that performs two major steps. At its first step, the procedure would "expand" the paths corresponding to the path functions Pf_1 through Pf_l and propagate the applicable typing and inheritance constraints. This would take $O(m \cdot k)$ time since $O(m)$ nodes are created and it takes $O(k)$ time to apply all typing and inheritance constraints for each node. Next, since the rule of uniqueness does not fire on the nodes that have **Id** in their *Pfs* label due to condition (d), P4 can only be applied to $O(m)$ nodes; i.e. P4 does not apply to nodes created after the first step. Indeed, since all path functions contain **Id** in the end, the regularity condition of the uniqueness constraints and the rule of composition ensure that the *Pfs* labels of all nodes added during the second step contain **Id**, and that any node added during the second step has a node created during the first step as its parent. It follows that P4 does not apply to the nodes added during the second step. Furthermore, since at most $O(k)$ uniqueness constraints can fire on each node, the second step would consist of $O(m \cdot k)$ iterations that are based on the applications of the rule P4. During each iteration, the procedure would check for applicability of the uniqueness constraints until it finds an applicable one and fires it. This takes $O(m \cdot k)$ time since the procedure goes over $O(m)$ nodes and takes $O(k)$ time to check for applicability of all uniqueness constraints on that node. In addition, it takes $O(k)$ time to propagate all typing and inheritance constraints and to fire them on the new node created due to the application of the uniqueness constraint. Therefore, the total runtime of the procedure is the sum of the $O(m \cdot k)$ component for the first step, and the $O(m \cdot k\ (m \cdot k + k))$ component for the second step which produces the $O(m^2 \cdot k^2)$ bound.

Further, it is even possible to carefully index a terminology and refine our propagation procedure in such a way that the total effort expended on each node is $O(k)$ instead of our original

allowance of $O(m \cdot k^3)$ time. This yields an even better $O(m \cdot k^2)$ bound on the time to compute G. An event driven implementation achieves this by making nodes "wait" for the acceptance of the appropriate path functions in order to apply a uniqueness constraint. For each node and a possibly applicable uniqueness constraint, the procedure keeps track of the progress made at "expanding" the path functions inside the constraint by capturing them as paths inside the acceptor tree. Once all path functions inside the constraint are expanded and accepted by the tree, the uniqueness constraint fires, possibly producing new nodes that allow other path functions to be accepted.

In particular, an implementation would begin by creating the root of the acceptor tree by procedure $Init(C, \{Pf_1, \ldots, Pf_l\})$ and (recursively) adding all primitive concepts that subsume C in $O(k)$ time. The set of concepts in the *Cls* label allows the procedure to deduce the uniqueness constraints that might apply to the root, and for each such uniqueness constraint, the root starts waiting for the necessary edges to be added, and more generally, for the path functions on the "left-hand-side" of the uniqueness constraint to get accepted. Then, the procedure "expands" the path functions Pf_i $(1 \leq i \leq l)$ using the composition rule.

With addition of each new edge $\langle n_1, A, n_2 \rangle$ and after initializing the *Pfs* label of n_2, the implementation updates the progress made on expanding path functions inside uniqueness constraints for the nodes that were waiting for the addition of the edge labeled A and outgoing from the node n_1. The progress is updated either by making the nodes wait for the addition of the next corresponding outgoing edges from n_2 or by accepting the path function if $Pfs(n_2)$ is $\{\mathbf{Id}\}$. If all path functions of a uniqueness constraint turn out to be accepted, the corresponding firing of rule P4 is added to a job queue that contains rules that are still to be fired. Next, all applicable typing constraints for node n_1, i.e. the constraints of the form $C_1 < (\mathbf{all}\ A\ C_2)$ where $C_1 \in Cls(n_1)$, are propagated down the edge and initialize the *Cls* label of node n_2 (with concepts such as C_2). Further, the inheritance constraints (of the form $C_2 < C_3$) are recursively applied to all the concepts inside $Cls(n_2)$. Finally, if $Pfs(n_2)$ does not contain **Id**, n_2 starts waiting for acceptance of the path functions in any uniqueness constraint that can apply to n_2; for example, uniqueness constraints on concepts C_2 and C_3.

After Pf_1 through Pf_l are expanded in the way described above, the jobs from the job queue start adding new edges (if the rules are still applicable). These newly added edges are processed in the same way, and the procedure stops when the job queue becomes empty.

Note that such an implementation that does not attempt to apply inheritance and typing constraints to the ancestors after some work is done on the descendents is possible since there are no cycles in the acceptor trees and applicability of each inheritance and typing constraint does not depend on the descendent nodes. This also means that the set of possibly applicable uniqueness constraints that are chosen based on the concept names inside the *Cls* label of a node does not change after we "process" the *Cls* label for the first time. More elaborate implementation with more extensive use of the job queues will be required and described in the next chapter, where the graph can contain cycles.

To estimate the runtime, note that with appropriate indexing of typing constraints in a terminology and with the concept hierarchy that supports efficient "depth-first" traversal, the time required to propagate all typing constraints from a parent node to its child and the time to add all concepts that subsume an initial set of concepts in the child node is $O(k)$. Therefore, since there are $O(m \cdot k)$ nodes in the acceptor tree, the total time spent on processing of inheritance and typing constraints is $O(m \cdot k^2)$. Next, note that the time spent on processing of a uniqueness constraint for a given node is $O(t)$ where t is the length of the encoding of the constraint. Indeed, for every attribute A in every path function (except for the right-hand-side path function), we spend $O(1)$ time to "follow" the corresponding edge $\langle n_1, A, n_2 \rangle$, determine if **Id** is in $Pfs(n_2)$, and possibly make it wait on addition of the next edge. On the other hand, for every attribute of the right-hand-side path function, we either follow an already existing edge with the corresponding label or create a new edge which also takes $O(1)$ time. Therefore, since the total size of encoding of all uniqueness constraints is $O(k)$, and since there are $O(m)$ nodes to which uniqueness constraints apply, the total time spent on processing of uniqueness constraints is $O(m \cdot k)$. It follows that the total runtime complexity of *Prop* is $O(m \cdot k^2)$.

Observe that our analysis of runtime assumes efficient constant time implementations of such operations as

- adding a concept to a *Cls* label,
- determining whether a given concept is in a *Cls* label,
- adding a node,
- adding an edge between two nodes, and
- checking for existence of an outgoing edge with a given label from a given node.

A technique for such an efficient implementation is described in the appendix of [BW94]. (Note that while the appendix describes a simple array implementation for the last three operations, the first two operations can be implemented in the same way as the last two.)

The following list of $3 \cdot k + 3$ subsumption constraints is a pattern for a terminology that demonstrates that the implementation suggested above obtains the best runtime complexity for our *Prop* procedure.

$C < (\textbf{all}\ A\ C)$;
$C < (\textbf{all}\ B\ C)$;
$C < C_i$, $1 \le i \le k$;
$C < (\textbf{all}\ B_i\ C)$, $1 \le i \le k$;
$C < (\textbf{fd}\ C\ B\ B_i)$, $1 \le i \le k$;
$C < (\textbf{fd}\ C\ (\textbf{comp}\ A\ B_1) \ldots (\textbf{comp}\ A\ B_k)\ B)$.

In particular, consider an acceptor tree G obtained by the evaluation of

$$Prop(Init(C, \{(\mathbf{comp}\ A^m\ B)\}), T),$$

where A^m denotes an attribute description consisting of m copies of the primitive attribute A. It is not difficult to see that the number of nodes occurring in G is $\Theta(m \cdot k)$, and that the inheritance rule will apply to each of these nodes $\Theta(k)$ times.

While $\Theta(m \cdot k^2)$ is a tight bound on the runtime complexity of the procedure, we can still improve the (production) system with two small modifications of rules P2 and P3. An important observation is that we cannot deduce anything about an ancestor based on the typing information in the descendent nodes. Indeed, typing constraints only "propagate" downward, inheritance constraints only operate within a node, and uniqueness constraints only depend on the typing information of the node that they apply to. Therefore, it is not necessary to deduce any typing information about the leaf nodes in order to solve a membership problem. Thus, consider the following modifications of the rules P2 and P3 (the parts that have been added are underlined):

P2'. (*inheritance*) If there exist $n_1 \in N$ and "$C_1 < C_2$" $\in T$ such that $C_1 \in Cls(n_1)$, $C_2 \notin Cls(n_1)$, and $\underline{\mathbf{Id} \notin Pfs(n_1)}$, then add C_2 to $Cls(n_1)$.

P3'. (*typing*) If there exist $\langle n_1, A, n_2 \rangle \in E$ and "$C_1 <$ (**all** A C_2)" $\in T$ such that $C_1 \in Cls(n_1)$, $C_2 \notin Cls(n_2)$, and $\underline{\mathbf{Id} \notin Pfs(n_2)}$, then add C_2 to $Cls(n_2)$.

With these modifications, our implementation above needs to be slightly modified to make sure that no typing information is propagated to the leaf nodes. Then, since the number of non-leaf nodes is only $O(m)$, the total time spent on processing of the inheritance and typing constraints decreases to $O(m \cdot k)$. Therefore, overall, the modified *Prop* procedure, denoted *Prop'*, runs in $O(m \cdot k)$ time!

Observe that if all uniqueness constraints inside terminology and the posted question are restricted to a generalization of relational symmetric uniqueness constraints that allows path functions to be either primitive attributes or **Id**, all nodes in the acceptor tree, except possibly for the root, have to be leaves. Therefore, just as in the case of the relational FD theory, the runtime of the procedure becomes $O(m + k)$ (even with our generalization of the relational uniqueness constraints and inheritance).

Also observe that termination and soundness of *Prop'* directly follow from termination and soundness of *Prop* since we apply the rules in the same manner but decrease the number of applications of the rules. On the other hand, the proof of completeness has to be augmented with additional argument that formally applies the above observation. In particular, in addition to applying *Prop'* at each construction step, we need to propagate typing information to the leaf

nodes during the construction of G_0. Then, we can still prove that all constraints in the terminology are satisfied by the constructed interpretation, while the roots of G_{-1} and G_{-1}' violate the uniqueness constraint in the posed question.

Note that the original procedure represents the *essence* of our production system, while we consider the introduced modifications as more of an *implementation* issue. Analogous refinement of the procedures in the next chapter will emphasize this point. In fact, it is relatively hard to formally modify the rules of the procedures introduced there to capture the improvement. In addition, while the modified procedure is more efficient, we use the original version to simplify our subsequent analysis. For example, we use the original *Prop* to simplify the discussions about multiple applications of the procedure to a series of similar questions in Subsection 5.7.2, or extending the procedure to enable it to answer typing questions in Subsection 6.2.1. While the discussions of such properties also apply to the modified procedure, one would need to keep in mind that there are actually two types of nodes—let us call them *processed* and *unprocessed*. Processed nodes are the ones about which the typing information was deduced from their parents, while the unprocessed nodes are leaves about which no typing information was deduced. Then, whenever one uses an acceptor tree for some deductions that involve an unprocessed node, all appropriate typing information from the parent of such a node has to be "expanded" to make the node processed. In other words, rules P2 and P3 have to be exhaustively applied to the incoming edge of the unprocessed node.

Finally, it should be noted that in practice, the number of constraints applicable to any given primitive concept is usually very small in comparison to the total size of a terminology. Therefore, the $O(k)$ component of $O(m \cdot k)$ is in fact a rather lose bound on the real cost of the work performed on each node.

4.6 Generality

Recall from our comments in Chapters 1 and 2 that existing procedures for deciding the PFD membership problem assume that any PFD in a database schema is a *key* PFD. In terms of our *Prop* procedure, this corresponds to a more constrained condition (b) in our rule of uniqueness in which the upper bound on the length of Pf_3 becomes 0 (which therefore implies that $Pf_3 = \mathbf{Id}$). A natural question to ask is what happens when the condition is slightly relaxed; that is, what happens when one allows the upper bound on the length of Pf_3 to be 2 instead of the current value of 1 studied in this chapter. (And let us call a uniqueness constraint that satisfies this relaxed condition *nearly regular*.) Interestingly, the following shows that this slight generalization is really a complete generalization; or in other words, allowing nearly regular uniqueness constraints inside a terminology is equivalent to allowing arbitrary uniqueness constraints with their last path function independent of the other ones.

Proposition 4.2 *Let T_1 denote an arbitrary atomic terminology and $C < D$ a subsumption constraint free of any occurrence of a primitive attribute not occurring in T_1. Then an application of the following rewrite rule to each constraint in T_1*

Replace "$C <$ (**fd** C *Pfs* $B_1 \circ B_2 \circ \ldots \circ B_k$)", where $k > 2$, by the set of constraints

"$C <$ (**fd** C *Pfs* $B_1 \circ E_2$)",
"$C <$ (**fd** C $B_1 \circ E_2$ $B_1 \circ B_2 \circ E_3$)",
...
"$C <$ (**fd** C $B_1 \circ B_2 \circ \ldots \circ B_{k-2} \circ E_{k-1}$ $B_1 \circ B_2 \circ \ldots \circ B_k$)",

where $E_2, E_3, \ldots, E_{k-1}$ are primitive attributes not occurring in the given terminology.

obtains a T_2 in which all symmetric uniqueness constraints are nearly regular and for which $T_1 \models C < D$ if and only if $T_2 \models C < D$.

Note that while it is still an open problem whether the membership problem with arbitrary uniqueness constraints inside terminology is polynomially decidable, [Wed92] presents some evidence that a chase-like decision procedure for such a problem would take at least exponential time. Thus, overall, our regularity condition (b) of the rule of uniqueness in procedure *Prop* appears to be as general as possible for chase-like decision procedures.

4.7 On Schema Analysis: Diagnosing Object Normal Form

This section presents one possible application of procedure *Prop* in schema analysis. In particular, we present a polynomial time algorithm that determines whether a database schema is in (*strong*) *object normal form* (ONF) as defined in [Bis89]. Although diagnosing BCNF can require exponential time, the addition of a unique minimal key constraint by Biskup makes it possible to check for ONF in polynomial time; and the ability of our *Prop* procedure to reason about regular constraints, and not just key PFDs, allows us to employ it to efficiently diagnose ONF.

Biskup considers a (relational) database schema as a sequence $(\langle R_1, F_1\rangle, \ldots, \langle R_m, F_m\rangle)$ where R_i are distinct relations and F_i are sets of FDs with all their attributes in the corresponding R_i. Without loss of generality, we assume that R_i has attributes $A_1, \ldots, A_n$ and

$$F_i = \{X_1 \rightarrow A_1', \ldots, X_k \rightarrow A_k'\}$$

where X_j are sets of attributes.

A database schema is said to be in ONF if and only if it is in BCNF and every relation schema has a unique minimal key [Bis89]. In our discussion, however, we consider the problem of determining whether a schema is in ONF in terms of our DL. Thus, we will consider a terminology T with a set of constraints of the form

$$
\begin{aligned}
C_i < (&\textbf{and} \\
&(\textbf{all } A_1 \text{ VALUE}) \\
&\quad \ldots \\
&(\textbf{all } A_n \text{ VALUE}) \\
&(\textbf{fd } C_i\ X_1\ A_1')^{15} \\
&\quad \ldots \\
&(\textbf{fd } C_i\ X_k\ A_k') \\
&(\textbf{fd } C_i\ A_1 \ldots A_n\ \textbf{Id})),
\end{aligned}
$$

where VALUE is an arbitrary concept for allowed values of attributes, and relations are considered as concepts with the appropriate number of attributes and the appropriate **fd** constraints directly derived from the F-sets. Also note that we use **Id** as an equivalent of a key in the relational model.

A concept C_i is then said to be in BCNF if for every FD $C_i < (\textbf{fd } C_i\ X\ A)$ implied by T, if all attributes in $X \cup \{A\}$ are attributes of C_i and $A \notin X$, then X is a key of C_i. Also note that a key of a concept C_i is now defined as a set X of attributes of C_i such that $T \models C_i < (\textbf{fd } C_i\ X\ \textbf{Id})$. With this straightforward translation of a database schema into a terminology in our DL, we can now present an algorithm to determine whether such a terminology is in ONF; that is, if every concept in T is in BCNF and has a unique minimal key.

The search space for our algorithm is a tree defined as follows. Given a finite set of primitive attributes $\{A_1, \ldots, A_n\}$, each node in the tree has a label *att* (for "attributes") that contains a permutation of a non-empty subset of these attributes. The root contains sequence $(A_1, \ldots, A_n)$ for an arbitrary permutation of the attributes. Every node with k ($k > 1$) attributes in the label has k children nodes where the label of the i-th node is obtained by removing the i-th attribute from the label of the parent. The leaves have a single attribute in their labels. The general form for such a tree is illustrated in Figure 4.4 below.

The algorithm explores the search space by traversing the nodes of the tree top-down in the following manner: if $T \models C < (\textbf{fd } C\ \textit{att_value}\ \textbf{Id})$, where *att_value* is the sequence of attributes in the *att* label of the current node, then go to the first child of the current node (or output the current node if it is a leaf); otherwise, go to the right sibling of the current node and if

[15] For convenience, given a set of attributes $X = \{A_1, \ldots, A_n\}$, we write $(\textbf{fd } C\ X\ A)$ to denote $(\textbf{fd } C\ A_1 \ldots A_n\ A)$. The same applies when we use more than one set name inside an **fd** constraint.

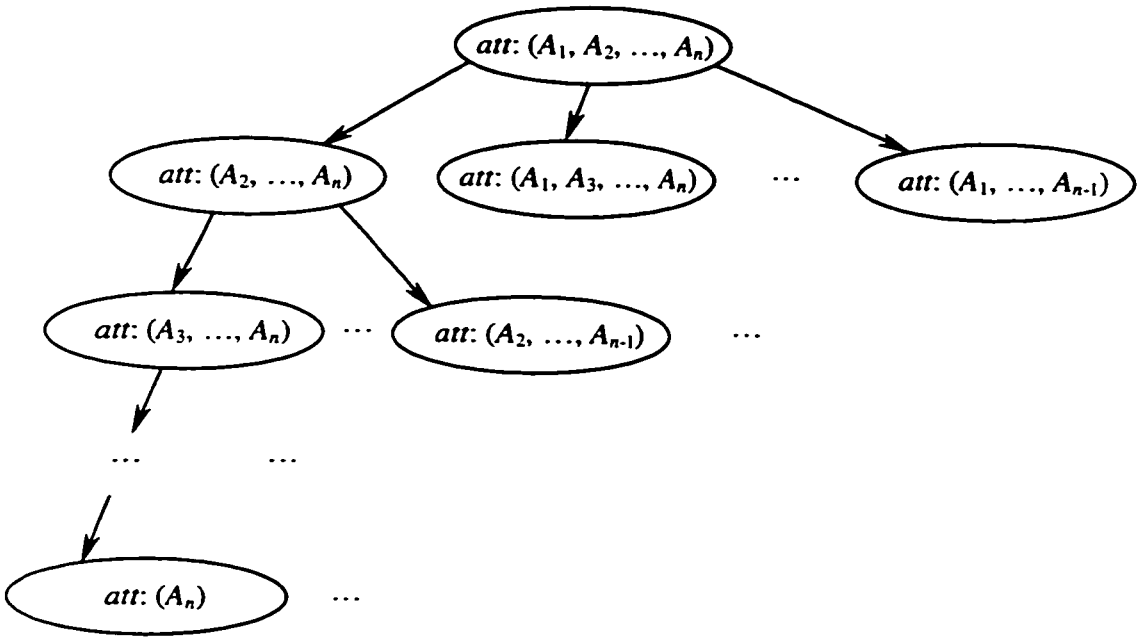


Figure 4.4: The tree used for the algorithm that determines whether a concept C with primitive attributes A_1 through A_n is in ONF.

there are no right siblings, stop and output the parent of the current node as the result of the traversal. Note that since $T \models C <$ (**fd** $C\ A_1\ A_2 \ldots A_n$ **Id**) is always true by our construction of T, there will always be a parent to return when the traversal stops due to non-existence of a right sibling. Also note that the problem of whether or not $T \models C <$ (**fd** C *att_value* **Id**) is efficiently solved by our *Prop* procedure.

Suppose a node with label (A_1', A_2', ..., A_l') is returned, and let us denote the set with these attributes by *att_result*; that is, *att_result* = $\{A_1', A_2', \ldots, A_l'\}$. Then, by the traversal algorithm, *att_result* is a key of C and there is no subset of *att_result* that is a key. Thus, *att_result* is a minimal key. The algorithm now checks (1) whether *att_result* is the *unique* minimal key and, if yes, (2) whether C is in BCNF. If the answer to question (2) is also "yes" then C is in ONF; in any other case it is *not* in ONF.

To answer question (1), the algorithm uses our *Prop* procedure to check whether $T \models C <$ (**fd** $C\ A_1 \ldots A_1' \ldots A_{i-1}' \ldots A_{i+1}' \ldots A_l' \ldots A_n\ A_i'$) for all $1 \leq i \leq l$. If at least one, say j-th, of these FDs is logically implied by T, then *att_result* is *not* the unique minimal key, since there is another key (*possible_key* = $\{A_1, \ldots, A_1', \ldots, A_{j-1}', \ldots, A_{j+1}', \ldots, A_l', \ldots, A_n\}$) which is not a superset or subset of the *att_result*. (Note that *possible_key* cannot be a subset of *att_result*, since otherwise, *att_result* would contain all n attributes, and therefore, the node containing the attributes of *possible_key* in its label would be already checked during the traversal and diagnosed as not being a key.) On the other hand, if none of the FDs are logically implied by T, then *att_result* is

the unique minimal key. Indeed, if there were another minimal key not containing an A_j' as one of its attributes for some $1 \leq j \leq l$, then its superset ($\{A_1, \ldots, A_1', \ldots, A_{j-1}', \ldots, A_{j+1}', \ldots, A_l', \ldots, A_n\}$) would be a key as well, contrary to the assumption that

$$T \not\models C < (\textbf{fd}\ C\ A_1 \ldots A_1' \ldots A_{j-1}' \ldots A_{j+1}' \ldots A_l' \ldots A_n\ A_j').$$

If *att_result* is the unique minimal key, the algorithm proceeds to answer question (2). It follows directly from the definition of BCNF that C is in BCNF if and only if any FD that is logically implied by T that has the following form:

$$C < (\textbf{fd}\ C\ X\ Y\ A_j),$$

where $1 \leq j \leq n$, $X \subseteq$ *att_result*, $Y \cap$ *att_result* $= \emptyset$, and if $A_j \notin X \cup Y$, $T \models C < (\textbf{fd}\ C\ X\ Y\ \textbf{Id})$.

Therefore, to answer question (2), the algorithm checks whether $T \models C < (\textbf{fd}\ C\ Z_{i,j}\ \textbf{Id})$ for all $1 \leq i \leq l$, $1 \leq j \leq n$, where $Z_{i,j}$ is the set of all attributes of C except for A_i' and A_j. (The checks are again conducted by our *Prop* procedure.) If at least one of the FDs is logically implied by T, then C is *not* in BCNF, since the X subset of $Z_{i,j}$ does not contain A_i', and therefore, $Z_{i,j}$ is not a key of C (recall that $\{A_1', A_2', \ldots, A_l'\}$ is the unique minimal key of C) contrary to our assumption that $T \models C < (\textbf{fd}\ C\ Z_{i,j}\ \textbf{Id})$. On the other hand, if none of the FDs are logically implied by T, then C is in BCNF. Indeed, if there were an FD ($\textbf{fd}\ C\ X\ Y\ A_j$) such that $A_j \notin X \cup Y$ and $T \not\models C < (\textbf{fd}\ C\ X\ Y\ \textbf{Id})$, then there would exist an i ($1 \leq i \leq l$) such that $A_i' \notin X$. But then, T would logically imply $C < (\textbf{fd}\ C\ Z_{i,j}\ \textbf{Id})$ contrary to our assumption.

Thus, the above algorithm solves the problem of deciding whether a database schema is in ONF in polynomial time. Note also that since our *Prop* procedure deals with a more general object-relational environment, we can easily extend the allowed database schema to the object-relational case with minor modifications to the algorithm and the proofs above. First, we can replace VALUE in **all** constraints by other concepts in T. In particular, we can allow recursive schemas (unlike the relational model in Biskup). Secondly, since *Prop* allows constraints of the form $C_1 < C_2$ in T, we can have inheritance constraints in the terminology. Finally, we can also allow arbitrary regular **fd** constraints in T in place of just relational functional dependencies of the form $C < (\textbf{fd}\ C\ A_1 \ldots A_n\ A)$.

Chapter 5

On General Logical Implication Problems

5.1 Problem Definition

This section adds further details to the general problem definition stated in Chapter 3. We begin by reviewing the kinds of descriptions that will be relevant in this chapter. We then revisit the notion of a terminology and introduce an example that will be used throughout the chapter to further motivate our results. Finally, we consider some applications of the general logical implication problem in semantic query optimization. In particular, we provide further motivation for extended **fd** descriptions that derives from their ability to pose questions about views that are abstractions of conjunctive queries, and for allowing asymmetric uniqueness constraints in terminologies.

5.1.1 Descriptions

In this chapter, we present procedures that solve the general logical implication problem defined in Chapter 3. However, there are a few additional notes that we need to make about this general formulation.

First note that the problem we address generalizes the membership problem discussed in the previous chapter in a number of ways. We allow a terminology (as well as a posed question) to contain asymmetric **fd** constraints in addition to the previously considered symmetric uniqueness constraints which restrict two objects from the *same* class to agree on the appropriate path functions. More general asymmetric uniqueness constraints can be employed to ensure that a pair of objects from possibly *distinct* classes agree on the path functions. These constraints allow one to capture such facts as "professors do not share offices with any other university employees".

We also allow a terminology to have view definitions, and for any such view definition to contain any number of occurrences of the **same-as** concept constructor. Consequently, it now becomes possible to specify view definitions that fully capture equational restrictions occurring inside conjunctive queries expressed in object-relational query languages.

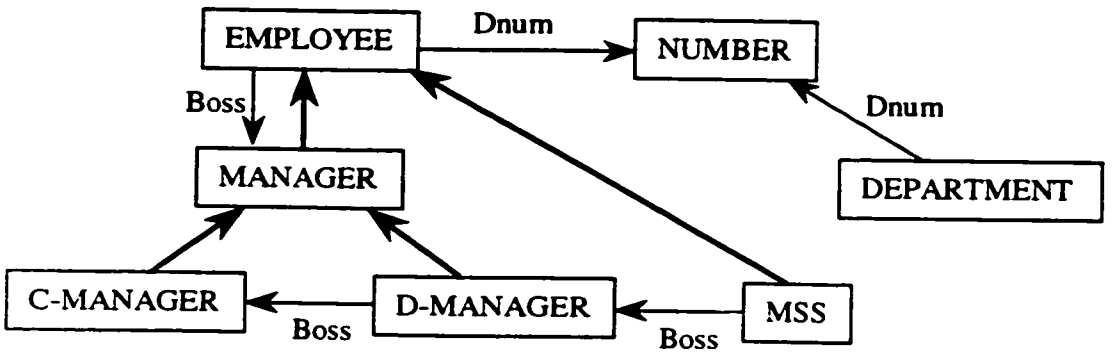
More formally, this chapter considers descriptions that are generated from the grammar presented in Chapter 3 (appropriately adjusted to include a production for extended **fd** descriptions) and repeated here for convenience.

D ::=	C	(*primitive concept*)
	| V	(*view name*)
	| (**all** Pf D)	(*attribute value restriction*)
	| (**fd** D Pfs Pf)	(*functional dependency*)
	| (**and** D D Ds)	(*description intersection*)
	| (**same-as** Pf Pf)	(*equational restriction*)
Pf ::=	A	(*primitive attribute*)
	| **Id**	(*identity*)
	| (**comp** Pf Pf)	(*attribute composition*)
Pfs ::=	ε | Pfs Pf	(*attribute description sequence*)
Ds ::=	ε | Ds D	(*concept description sequence*)

5.1.2 An Example Terminology

Consider a part of an administrative database schema graph shown in Figure 5.1(a). (The example is derived from a real-world case.) Note that the typing and inheritance constraints suggested by the graph can be easily captured by both object-relational and relational models. In the latter model, for example, one would use foreign key constraints to capture the typing restriction on the Boss attribute.

In addition to the restrictions depicted in the figure, the company is structured in such a way that Dnum is a key of the DEPARTMENT concept. Furthermore, "Members of Scientific Staff" (MSS) work for various departments (with such numbers as 7H01, 7H02, etc.) and report to their department ("D-level") managers with each department having exactly one D-level manager and some number of MSSs. D-level managers in turn report to so-called C-level managers with unique department numbers such as 7H00. Thus, each manager is assigned a unique department number (i.e. Dnum is a candidate key of the MANAGER concept), and, in addition, C-level managers have department numbers that are distinct from the department



(a) Diagram of a schema. Thick and thin arrows depict inheritance and typing constraints respectively.

EMPLOYEE < (**and** (**all** Boss MANAGER)
(**all** Dnum NUMBER))

MANAGER < (**and** EMPLOYEE
(**fd** MANAGER Dnum **Id**))

MSS < (**and** EMPLOYEE
(**all** Boss D-MANAGER))

D-MANAGER < (**and** MANAGER
(**all** Boss C-MANAGER)

C-MANAGER < (**and** MANAGER
(**fd** EMPLOYEE Dnum **Id**))

DEPARTMENT < (**and** (**all** Dnum NUMBER)
(**fd** DEPARTMENT Dnum **Id**))

(b) The COMPANY terminology.

Figure 5.1: Example of a terminology representing a schema.

numbers of other employees. Note that, although this latter constraint cannot be captured by standard relational and object-relational models, the constraint *can* be captured in terms of an asymmetric uniqueness constraint. This is illustrated in Figure 5.1(b) that gives a formulation of the administrative database in terms of a COMPANY terminology consisting of six subsumption constraints. Also note that the COMPANY terminology is recursive. For example, the boss of a manager is also a manager.

5.1.3 Applications in Query Optimization

Recall from our comments in Chapter 3 that, for a general logical implication problem of the form $T \models D_1 < D_2$, description D_2 can contain *extended* ***fd*** *descriptions* of the form

$$(\textbf{fd}\ D\ Pf_1 \ldots Pf_m\ Pf).$$

Such descriptions allow us to reason about uniqueness constraints on views and queries. We demonstrate why this is desirable with the following example which further motivates our work as a whole.

Consider the COMPANY terminology presented in Figure 5.1(b) and a request for "all pairs of employees and D-level managers that are in the same existing department". Either of the following two OQL queries expresses this request (note that if an employee is not assigned to a department, their department number will not be a number of an existing department).

```
Q1: select e, dm
    from EMPLOYEE e, D-MANAGER dm
    where e.Dnum = dm.Dnum
    and exists (select *
                from DEPARTMENT d
                where d.Dnum = e.Dnum)

Q2: select distinct e, dm
    from EMPLOYEE e, D-MANAGER dm, DEPARTMENT d
    where e.Dnum = dm.Dnum
    and   d.Dnum = e.Dnum
```

An optimizer that processes Q1 might wish to explore the possibility of absorbing the existential query into the top-level select-from-where clause and rewrite the query into the following form.

```
Q3: select e, dm
    from EMPLOYEE e, D-MANAGER dm, DEPARTMENT d
    where e.Dnum = dm.Dnum
    and   d.Dnum = e.Dnum
```

The optimizer can also rewrite query Q2 into query Q3 by removing the **distinct** keyword. Now, in order to determine if either of these optimizations is possible, the optimizer must check if two

distinct query results can have the same bindings for variables "e" and "dm", but differ in their binding for variable "d".

In order to perform the check, the optimizer can use our results by proceeding as follows. First, query Q3 is abstracted as a subsumption constraint

Q < (**and** (**all** e EMPLOYEE)
(**all** dm D-MANAGER)
(**all** d DEPARTMENT)
(**fd** Q e dm d **Id**))

together with a view definition

QView ≡ (**and** Q
(**same-as** (**comp** e Dnum) (**comp** dm Dnum))
(**same-as** (**comp** d Dnum) (**comp** e Dnum)))

which are then added to the COMPANY terminology to form an expanded COMPANY′ terminology. Thus, we abstract the result of evaluating a query as a set of objects in the denotation of a "query result view" QView, and introduce attributes for QView objects that abstract the query variables. Also observe that capturing the query as a view definition rather than a subsumption constraint allows us to employ the **same-as** concept constructor to capture a given query's join conditions.

The second and final step is to then determine if the COMPANY′ terminology logically implies the following constraint.

QView < (**fd** QView e dm d)

To do this efficiently, the optimizer can use the procedures presented in this chapter.

Thus, the example illustrates that an ability to efficiently reason about uniqueness constraints on views is important and demonstrates the need for extended **fd** descriptions.

Now consider a variation in the example queries. In particular, let Q1′ (resp. Q2′) denote Q1 (resp. Q2) but with D-MANAGER replaced by C-MANAGER. In other words, let Q1′ and Q2′ denote requests for "all pairs of employees and *C-level* managers that are in the same existing department".

```
Q1′: select e, cm
     from EMPLOYEE e, C-MANAGER cm
     where e.Dnum = cm.Dnum
     and exists (select *
                 from DEPARTMENT d
                 where d.Dnum = e.Dnum)

Q2′: select distinct e, cm
     from EMPLOYEE e, C-MANAGER cm, DEPARTMENT d
     where e.Dnum = cm.Dnum
     and   d.Dnum = e.Dnum
```

While the previous optimization of removing the existential quantification (resp. the **distinct** keyword) still applies in this case, an additional optimization is also possible. In particular, the asymmetric uniqueness constraint

C-MANAGER < (**fd** EMPLOYEE Dnum **Id**)

"embedded" in the COMPANY terminology together with the procedures developed in this chapter allow us to efficiently deduce the subsumption constraint

QView′ < (**same-as** e cm)

where QView′ is an abstraction of the query defined as follows.

```
Q′ < (and (all e EMPLOYEE)
          (all cm C-MANAGER)
          (all d DEPARTMENT)
          (fd Q′ e cm Id))

QView′ ≡ (and Q′
              (same-as (comp e Dnum) (comp cm Dnum))
              (same-as (comp d Dnum) (comp e Dnum)))
```

This deduction would allow a query optimizer to select an access plan in which an assignment is used to bind "e" instead of performing an index scan. In effect, the deduction enables the optimizer to rewrite either Q1′ or Q2′ as follows.

select (EMPLOYEE) cm **as** e, cm
from C-MANAGER cm, DEPARTMENT d
where cm.Dnum = d.Dnum

5.2 Mapping to Molecular Terminologies

Analogously to Chapter 4, we simplify terminology before we apply procedures developed in this chapter. In this case however, in addition to simplifying descriptions inside constraints, we *combine* them instead of breaking them up as in Chapter 4. Thus, we call the mapping presented here *Molecular*. Formally, we define a *molecular terminology* as any terminology T' than can be obtained from an arbitrary terminology T by an exhaustive application of the rewrite rule in Table 5.1(a) followed by an exhaustive application of the rewrite rules in Table 5.1(b) and (c) to descriptions inside the resulting constraints. We denote this circumstance by writing *Molecular*(T, T').

One consequence of Proposition 5.1 below is that descriptions in a molecular terminology have the property that they can be generated from the following simplified grammar[16].

D ::= C	(*primitive concept*)
| V	(*view name*)
| (**all** A D)	(*attribute value restriction*)
| (**fd** D Pfs Pf)	(*functional dependency*)
| (**and** D D Ds)	(*description intersection*)
| (**same-as** Pf Pf)	(*equational restriction*)
Pf ::= **Id**	(*identity*)
| (**comp** A Pf)	(*attribute composition*)
Pfs ::= ε | Pfs Pf	(*attribute description sequence*)
Ds ::= ε | Ds D	(*concept description sequence*)

In addition, no two subsumption constraints in a molecular terminology have the same left-hand-sides.

Another consequence is that this alternative simpler grammar does not reduce the expressiveness of a terminology in any fundamental way; for any terminology T_1 there exists a terminology T_2 for which *Molecular*(T_1, T_2) holds and that *preserves logical consequence*. Thus, although our examples will continue to be based on the more general grammar presented in Subsection 5.1.1, our procedures will assume (without loss of generality) that argument terminologies are molecular.

[16] Note that simplified grammar in this chapter is distinct from the simplified grammar in the previous one.

1. Replace "$C < D_1$" and "$C < D_2$" by "$C <$ (**and** D_1 D_2)".

(a) (*combining constraints*)

2. Replace "(**all Id** D)" by "D".
3. Replace "(**all** (**comp** Pf_1 Pf_2) D)" by "(**all** Pf_1 (**all** Pf_2 D))".
4. Replace "(**fd** D Pfs_1 A Pfs_2)" by "(**fd** D Pfs_1 (**comp** A **Id**) Pfs_2)".
5. Replace "(**same-as** A Pf)" by "(**same-as** (**comp** A **Id**) Pf)".
6. Replace "(**same-as** Pf A)" by "(**same-as** Pf (**comp** A **Id**))".

(b) (*rewrites for concept descriptions*)

7. Replace "(**comp Id** Pf)" by "Pf".
8. Replace "(**comp** Pf A)" by "(**comp** Pf (**comp** A **Id**))".
9. Replace "(**comp** (**comp** Pf_1 Pf_2) Pf_3)" by "(**comp** Pf_1 (**comp** Pf_2 Pf_3))".

(c) (*rewrites for attribute descriptions*)

Pf, Pf_1, Pf_2 and Pf_3 are arbitrary path functions; D, D_1 and D_2 are descriptions; and A is a primitive attribute.

Table 5.1: Molecular simplification of terminology.

Proposition 5.1 *Let T_1 denote a terminology and $D_1 < D_2$ a subsumption constraint. Then an exhaustive application of the rewrite rule in Table 5.1(a) to constraints in terminology T_1 followed by exhaustive application of the rewrite rules in Table 5.1(b) and (c) to constraints inside the resulting terminology and to constraint $D_1 < D_2$ terminates and obtains a terminology T_2 and a constraint $D_1' < D_2'$ respectively that can be generated from the simplified grammar and for which $T_1 \models D_1 < D_2$ if and only if $T_2 \models D_1 < D_2$, and $T_1 \models D_1 < D_2$ if and only if $T_1 \models D_1' < D_2'$. In addition, no two constraints in T_2 have the same left-hand-side.*

Proof. (See Appendix B).

5.3 Inference Axioms

The following lemma introduces a number of inference axioms that will be used in the later proofs in this chapter.

Lemma 5.1 *Let Pf, Pf_j and Pf_j' denote attribute descriptions, C denote a primitive concept, and D and D_j denote descriptions for any $j \geq 1$. Then, for any terminology T, the following properties hold:*

(1) *If* $T \models D <$ (**same-as** Pf_1 Pf_2), *then* $T \models D <$ (**same-as** Pf_2 Pf_1).

(2) *If* $T \models D <$ (**same-as** Pf_1 Pf_2) *and* $T \models D <$ (**same-as** Pf_2 Pf_3), *then* $T \models D <$ (**same-as** Pf_1 Pf_3).

(3) *If* $T \models D_1 <$ (**same-as** Pf_1 Pf_2) *and* $T \models D_1 <$ (**all** $Pf_1 \circ Pf_3$ D_2), *then* $T \models D_1 <$ (**all** $Pf_2 \circ Pf_3$ D_2).

(4) *If* $T \models D <$ (**same-as** Pf_1 Pf_2), *then for any attribute description* Pf,

$$T \models D < (\textbf{same-as } Pf_1 \circ Pf \; Pf_2 \circ Pf).$$

(5) *If*
   (a) $T \models D <$ (**all** Pf_2' C),
   (b) $T \models D <$ (**all** Pf_1' (**fd** C Pf_1 Pf_2 ... Pf_m Pf)), *and*
   (c) $T \models D <$ (**same-as** $Pf_1' \circ Pf_i$ $Pf_2' \circ Pf_i$) *for all* $1 \leq i \leq m$,
   then $T \models D <$ (**same-as** $Pf_1' \circ Pf$ $Pf_2' \circ Pf$).

(6) *If* $T \models D <$ (**same-as** $Pf \circ Pf_1$ $Pf \circ Pf_2$), *then* $T \models D <$ (**all** Pf (**same-as** Pf_1 Pf_2)).

(7) $T \models D_1 <$ (**all** $Pf_1 \circ Pf_2$ D_2) *if and only if* $T \models D_1 <$ (**all** Pf_1 (**all** Pf_2 D_2)).

(8) $T \models D_1 <$ (**all Id** D_2) *if and only if* $T \models D_1 < D_2$.

Proof.

Property (1). The property follows directly from definition of interpretation of the **same-as** constructor and from the fact that equality is commutative.

Property (2). This property is also a straightforward consequence of definition of interpretation of the **same-as** constructor and of the fact that equality is transitive.

Property (3). Since $T \models D_1 <$ (**all** $Pf_1 \circ Pf_3$ D_2), for any interpretation **I** that satisfies T and any object $x \in D_1^{\mathbf{I}}$, $(Pf_1 \circ Pf_3)^{\mathbf{I}}(x) \in D_2^{\mathbf{I}}$. Thus, by property (4) of Claim 3.1, $Pf_3^{\mathbf{I}}(Pf_1^{\mathbf{I}}(x)) \in D_2^{\mathbf{I}}$. Next, since $T \models D_1 <$ (**same-as** Pf_1 Pf_2), $Pf_1^{\mathbf{I}}(x) = Pf_2^{\mathbf{I}}(x)$. Therefore, $Pf_3^{\mathbf{I}}(Pf_2^{\mathbf{I}}(x)) \in D_2^{\mathbf{I}}$, or $(Pf_2 \circ Pf_3)^{\mathbf{I}}(x) \in D_2^{\mathbf{I}}$. Thus, $T \models D_1 <$ (**all** $Pf_2 \circ Pf_3$ D_2).

Note that property (1) also allows us to establish that if $T \models D_1 <$ (**same-as** Pf_1 Pf_2) and $T \models D_1 <$ (**all** $Pf_2 \circ Pf_3$ D_2), then $T \models D_1 <$ (**all** $Pf_1 \circ Pf_3$ D_2).

Property (4). Consider an interpretation **I** that satisfies T and an object x in $D^{\mathbf{I}}$. Then, by the assumption of the property, $Pf_1^{\mathbf{I}}(x) = Pf_2^{\mathbf{I}}(x)$, and thus, $Pf^{\mathbf{I}}(Pf_1^{\mathbf{I}}(x)) = Pf^{\mathbf{I}}(Pf_2^{\mathbf{I}}(x))$. Therefore, by property (4) of Claim 3.1, $(Pf_1 \circ Pf)^{\mathbf{I}}(x) = (Pf_2 \circ Pf)^{\mathbf{I}}(x)$. Thus, $T \models D <$ (**same-as** $Pf_1 \circ Pf$ $Pf_2 \circ Pf$).

Property (5). Proof is by contradiction. Let us assume that there is an interpretation **I** that satisfies T such that (a), (b), and (c) are true while T does *not* logically imply constraint $D <$ (**same-as** $Pf_1' \circ Pf$ $Pf_2' \circ Pf$). Then, there must exist an object

$$x \in D^{\mathbf{I}} \tag{1}$$

such that, by (a), (b), (c) and (1),

$$y = Pf_2'^{\mathbf{I}}(x) \in C^{\mathbf{I}}, \tag{2}$$
$$w = Pf_1'^{\mathbf{I}}(x) \in (\mathbf{fd}\ C\ Pf_1\ Pf_2 \ldots Pf_m\ Pf)^{\mathbf{I}}, \tag{3}$$
$$Pf_i^{\mathbf{I}}(Pf_1'^{\mathbf{I}}(x)) = Pf_i^{\mathbf{I}}(Pf_2'^{\mathbf{I}}(x)), \text{ for } 1 \le i \le m \tag{4}$$

and $(Pf_1' \circ Pf)^{\mathbf{I}}(x) \neq (Pf_2' \circ Pf)^{\mathbf{I}}(x)$. However, (2), (3), and (4) imply that

$$Pf_i^{\mathbf{I}}(w) = Pf_i^{\mathbf{I}}(y) \text{ for } 1 \le i \le m. \tag{5}$$

Then, by (2), (3), (5) and definition of $(\mathbf{fd}\ C\ Pf_1\ Pf_2 \ldots Pf_m\ Pf)^{\mathbf{I}}$, $Pf^{\mathbf{I}}(w) = Pf^{\mathbf{I}}(y)$. Therefore, $Pf^{\mathbf{I}}(Pf_1'^{\mathbf{I}}(x)) = Pf^{\mathbf{I}}(Pf_2'^{\mathbf{I}}(x))$. Thus, by property (4) of Claim 3.1, $(Pf_1' \circ Pf)^{\mathbf{I}}(x) = (Pf_2' \circ Pf)^{\mathbf{I}}(x)$ contrary to the assumption.

Property (6). Since for any interpretation **I** that satisfies T and any object $x \in D^{\mathbf{I}}$,

$$x \in (\mathbf{same\text{-}as}\ Pf \circ Pf_1\ Pf \circ Pf_2)^{\mathbf{I}},$$

by property (4) of Claim 3.1, $Pf_1^{\mathbf{I}}(Pf^{\mathbf{I}}(x)) = (Pf \circ Pf_1)^{\mathbf{I}}(x) = (Pf \circ Pf_2)^{\mathbf{I}}(x) = Pf_2^{\mathbf{I}}(Pf^{\mathbf{I}}(x))$. Therefore, $Pf^{\mathbf{I}}(x) \in$ (**same-as** Pf_1 Pf_2)$^{\mathbf{I}}$ which in turn implies that $x \in$ (**all** Pf (**same-as** Pf_1 Pf_2))$^{\mathbf{I}}$.

Property (7). Consider an arbitrary interpretation **I** that satisfies T and an object x in $D^{\mathbf{I}}$. Then, by property (4) of Claim 3.1, $(Pf_1 \circ Pf_2)^{\mathbf{I}}(x) \in D_2$ if and only if $Pf_2^{\mathbf{I}}(Pf_1^{\mathbf{I}}(x)) \in D_2$. In other words, $x \in$ (**all** $Pf_1 \circ Pf_2$ D_2)$^{\mathbf{I}}$ if and only if $x \in$ (**all** Pf_1 (**all** Pf_2 D_2))$^{\mathbf{I}}$.

Property (8). For an arbitrary interpretation **I** that satisfies T,

$$(\textbf{all Id } D_2)^{\mathbf{I}} = \{\, x \in \Delta \mid \mathbf{Id}^{\mathbf{I}}(x) = D_2^{\mathbf{I}} \,\} = D_2^{\mathbf{I}}.$$

Therefore, an object x in $D_1^{\mathbf{I}}$ is in $(\textbf{all Id } D_2)^{\mathbf{I}}$ if and only if it is in $D_2^{\mathbf{I}}$.

■

Additional useful properties that can be derived from the inference axioms are introduced in the following lemma.

Lemma 5.2 *For any attribute descriptions Pf_1, Pf_2, Pf_3, and Pf_4, a description D, and a terminology T the following properties hold:*

(1) *If* $T \models D <$ (**same-as** Pf_1 Pf_2) *and* $T \models D <$ (**same-as** $Pf_1 \circ Pf_3$ Pf_4), *then*

$$T \models D < (\textbf{same-as } Pf_2 \circ Pf_3 \; Pf_4).$$

(2) *If* $T \models D <$ (**same-as** Pf_1 Pf_2) *and* $T \models D <$ (**same-as** Pf_3 $Pf_1 \circ Pf_4$), *then*

$$T \models D < (\textbf{same-as } Pf_3 \; Pf_2 \circ Pf_4).$$

(3) *If* $T \models D <$ (**same-as** Pf_1 Pf_2) *and* $T \models D <$ (**same-as** $Pf_2 \circ Pf_3$ Pf_4), *then*

$$T \models D < (\textbf{same-as } Pf_1 \circ Pf_3 \; Pf_4).$$

(4) *If* $T \models D <$ (**same-as** Pf_1 Pf_2) *and* $T \models D <$ (**same-as** Pf_3 $Pf_2 \circ Pf_4$), *then*

$$T \models D < (\textbf{same-as } Pf_3 \; Pf_1 \circ Pf_4).$$

Proof. We will only consider the proof of property (1) since properties (2) through (4) can be proven analogously. Since $T \models D <$ (**same-as** Pf_1 Pf_2), by property (4) of Lemma 5.1,

$$T \models D < (\textbf{same-as } Pf_1 \circ Pf_3 \; Pf_2 \circ Pf_3).$$

Thus, by property (1) of Lemma 5.1, $T \models D <$ (**same-as** $Pf_2 \circ Pf_3$ $Pf_1 \circ Pf_3$). Combining this with the assumption that $T \models D <$ (**same-as** $Pf_1 \circ Pf_3$ Pf_4) and applying property (2) of Lemma 5.1 lets us conclude that $T \models D <$ (**same-as** $Pf_2 \circ Pf_3$ Pf_4).

■

5.4 Description Graphs and Paths

In this chapter, we consider description graphs with the following node labels: a finite set $\alpha(n, G)$ of descriptions and a finite set *fired*(n, G)[17] of primitive concept names and view names (we write $\alpha(n)$ and *fired*(n) when G is understood). Also, in addition to the distinguished node reference, *dn*, in the *Refs* set, procedures of this chapter use another reference, named *fdcn* for "functional dependency check node", that will be used to check if a functional dependency is satisfied. Other references are defined as they become necessary.

Given a description graph $G = \langle N, E, Refs \rangle$, let us define a *path* from node n_1 to node n_k as a sequence of nodes and edge labels $\langle n_1, A_1, n_2, A_2, \ldots, A_{k-1}, n_k \rangle$ where $k \geq 1$, $n_1, n_2, \ldots, n_k$ are nodes in N and $A_1, A_2, \ldots, A_{k-1}$ are edge labels such that edges $\langle n_1, A_1, n_2 \rangle$, $\langle n_2, A_2, n_3 \rangle$, $\ldots$, $\langle n_{k-1}, A_{k-1}, n_k \rangle$ are in E. Then, we define the *length* of the path as $k - 1$. Thus, paths of length 0 consist of a single node (e.g. $\langle n \rangle$), and paths of length 1 are edges. We say a path $\langle n_1, A_1, n_2, A_2, \ldots, A_{k-1}, n_k \rangle$ *goes* or *passes through* nodes n_i for $1 \leq i \leq k$ and edges $\langle n_{i-1}, A_{i-1}, n_i \rangle$ for $2 \leq i \leq k$. We also say that a path is a *loop* if it has the same first and last nodes, and a *simple loop* if it is a loop and no other nodes in the path equal the ones at its ends. (Note that any loop l that starts and ends with a node n can be broken into a number of simple loops $l_1, \ldots, l_m$ that start and end with node n by just going through the nodes in l successively and completing a new loop l_i every time one comes across n.) In addition, we sometimes omit nodes and/or edges in a path when they are not important or known. Thus, for example, $\langle n_1, A_1, A_2, \ldots, A_{k-1} \rangle$ denotes a path that starts at n_1 and goes through edges labeled $A_1, A_2, \ldots, A_{k-1}$.

Next, we define composition of paths and sub-paths. Given two paths $p_1 = \langle n_1, A_1, n_2, A_2, \ldots, A_{k-1}, n_k \rangle$ and $p_2 = \langle n_k, A_k, n_{k+1}, A_{k+1}, \ldots, A_{m-1}, n_m \rangle$, the *concatenation* or *composition of paths* p_1 and p_2, denoted $p_1 \circ p_2$, is the path[18]

- $\langle n_1, A_1, n_2, A_2, \ldots, A_{m-1}, n_m \rangle$ if $1 < k < m$;
- $\langle n_1, A_1, n_2, A_2, \ldots, A_{k-1}, n_k \rangle$ if $1 < k = m$;
- $\langle n_1, A_1, n_2, A_2, \ldots, A_{m-1}, n_m \rangle$ if $1 = k < m$; and
- $\langle n_1 \rangle$ if $1 = k = m$.

Note that composition is only defined for paths that are *attachable*, i.e. the last node of the first path has to be the same as the first node of the second path. Thus, when we talk about path concatenation, it is implicit that the paths must be appropriately attachable. It is easy to see that, as with attribute descriptions, the concatenation operator $\circ$ for paths is associative, and therefore, we do not need parentheses to indicate the order of applications of path composition. Finally, we

[17] Set *fired* is used as a technical device to avoid "firing" the same constraint in T more than once on a given node.

[18] Note overloading of the $\circ$ operator which we find intuitive. We will back up this intuition by Claim 5.1.

define a path p_3 to be a *sub-path* of a path p_1 if and only if $p_1 = p_2 \circ p_3 \circ p_4$ for some paths p_2 and p_4. If p_2 is of length 0, we call sub-path p_3 a *head* of p_1, whereas if p_4 is of length 0, we say that p_3 is a *tail* of p_1.

Let us now define a mapping of paths to path functions: given a path $p = \langle n_1, A_1, n_2, A_2, \ldots, A_{k-1}, n_k \rangle$, function $\gamma(p)$ maps p to path function

- $A_1 \circ A_2 \circ \ldots \circ A_{k-1}$ if $k > 1$, or
- **Id** otherwise (i.e. if $p = \langle n_1 \rangle$).

Note that by property (3) of Claim 3.1, $\gamma(p) = A_1 \circ A_2 \circ \ldots \circ A_{k-1} \circ \mathbf{Id}$ for any $k \geq 1$.

Claim 5.1 $\gamma(p_1 \circ p_2) = \gamma(p_1) \circ \gamma(p_2)$ *for any attachable paths* p_1 *and* p_2 *in any description graph* G.

Proof. Without loss of generality, let $p_1 = \langle n_1, A_1, n_2, A_2, \ldots, A_{k-1}, n_k \rangle$ and $p_2 = \langle n_k, A_k, n_{k+1}, A_{k+1}, \ldots, A_{m-1}, n_m \rangle$ $(1 \leq k \leq m)$. Then, by definitions of γ function and $\circ$ operators for paths and attribute descriptions, by property (3) of Claim 3.1, and associativity of $\circ$ operator for attribute descriptions,

- If $1 < k < m$,

$$\gamma(p_1 \circ p_2) = \gamma(\langle n_1, A_1, n_2, A_2, \ldots, A_{m-1}, n_m \rangle) = A_1 \circ A_2 \circ \ldots \circ A_{m-1} =$$
$$(A_1 \circ A_2 \circ \ldots \circ A_{k-1}) \circ (A_k \circ A_{k+1} \circ \ldots \circ A_{m-1}) = \gamma(p_1) \circ \gamma(p_2).$$

- If $1 < k = m$,

$$\gamma(p_1 \circ p_2) = \gamma(\langle n_1, A_1, n_2, A_2, \ldots, A_{k-1}, n_k \rangle) = A_1 \circ A_2 \circ \ldots \circ A_{k-1} =$$
$$(A_1 \circ A_2 \circ \ldots \circ A_{k-1}) \circ (\mathbf{Id}) = \gamma(p_1) \circ \gamma(p_2).$$

- If $1 = k < m$,

$$\gamma(p_1 \circ p_2) = \gamma(\langle n_1, A_1, n_2, A_2, \ldots, A_{m-1}, n_m \rangle) = A_1 \circ A_2 \circ \ldots \circ A_{m-1} =$$
$$(\mathbf{Id}) \circ (A_1 \circ A_2 \circ \ldots \circ A_{k-1}) = \gamma(p_1) \circ \gamma(p_2).$$

- If $1 = k = m$,

$$\gamma(p_1 \circ p_2) = \gamma(\langle n_1 \rangle) = \mathbf{Id} = \mathbf{Id} \circ \mathbf{Id} = \gamma(p_1) \circ \gamma(p_2).$$

■

Note that Claim 5.1 implies that given an interpretation **I** and a description graph G, $\gamma(p_1 \circ p_2)^{\mathbf{I}} = (\gamma(p_1) \circ \gamma(p_2))^{\mathbf{I}} = \gamma(p_2)^{\mathbf{I}} \circ \gamma(p_1)^{\mathbf{I}}$ for any attachable paths p_1 and p_2 in G.

Next, we say a node n_2 is *directly reachable* from a node n_1, n_1 is a *parent* of n_2, or n_2 is a *child* of n_1 in a description graph $G = \langle N, E, Refs \rangle$ if there exists an edge $\langle n_1, A, n_2 \rangle \in E$ for some A that is either a primitive attribute or **Id**. On the other hand, we say that n_2 is *reachable* from n_1 if there exists any path p from n_1 to n_2 in G. Finally, we say that n_2 is a *descendant* of n_1, or n_1 is an *ancestor* of n_2 if n_2 is reachable and distinct from n_1.

In this chapter, we sometimes need to refer to reachability only through non-**Id** edges. We therefore define additional *pa-* notions as follows ("pa" stands for "primitive attribute"). A node n_2 is *pa-directly reachable* from a node n_1, n_1 is a *pa-parent* of n_2, or n_2 is a *pa-child* of n_1 in $G = \langle N, E, Refs \rangle$ if there exists an edge $\langle n_1, A, n_2 \rangle \in E$ for some primitive attribute A. Additionally, we say that n_2 is *pa-reachable* from n_1 if there exists a path p from n_1 to n_2 in G that only passes through edges labeled by primitive attributes. Finally, we say that n_2 is a *pa-descendant* of n_1, or n_1 is a *pa-ancestor* of n_2 if n_2 is pa-reachable and distinct from n_1.

Let us now define two auxiliary procedures that work on description graphs.

By $Reachable_G(n_1, Pf, n_2)$ we denote a function that tells us whether a node n_2 is pa-reachable from node n_1 in G via a path with corresponding path function Pf. Formally, if $Pf = A_1 \circ A_2 \circ \ldots \circ A_k \circ \mathbf{Id}$ for some $k \geq 0$ and primitive attributes $A_1, \ldots, A_k$, $Reachable_G(n_1, Pf, n_2)$ is true if and only if

- $k = 0$ (i.e. $Pf = \mathbf{Id}$) and $n_1 = n_2$; or
- $k = 1$ and there is an edge $\langle n_1, A_1, n_2 \rangle$ in E; or
- $k \geq 2$ and there exist $k - 1$ nodes $n_1', \ldots, n_{k-1}'$ in N, and k edges $\langle n_1, A_1, n_1' \rangle$, $\langle n_1', A_2, n_2' \rangle$, ..., $\langle n_{k-2}', A_{k-1}, n_{k-1}' \rangle$, $\langle n_{k-1}', A_k, n_2 \rangle$ in E.

As a straightforward consequence of definitions of paths, function γ, and procedure *Reachable*, it follows that if $Reachable_G(n_1, Pf, n_2)$ is true, then there exists a path p from n_1 to n_2 in G such that $\gamma(p) = Pf$.

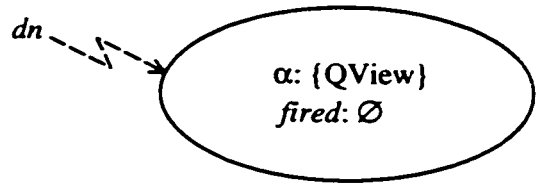
Similarly, by $Create_G(n_1, Pf, n_2)$ we denote a procedure that creates a path with corresponding path function Pf from a node n_1 to a node n_2 in G. Formally, if $Pf = A_1 \circ A_2 \circ \ldots \circ A_k \circ \mathbf{Id}$ for some $k \geq 0$ and primitive attributes $A_1, \ldots, A_k$, then

- if $k = 0$ (i.e. $Pf = \mathbf{Id}$), $Create_G(n_1, Pf, n_2)$ adds $\langle n_1, \mathbf{Id}, n_2 \rangle$ to E;
- if $k = 1$, $Create_G(n_1, Pf, n_2)$ adds $\langle n_1, A_1, n_2 \rangle$ to E; and
- if $k \geq 2$, $Create_G(n_1, Pf, n_2)$ adds $k - 1$ new nodes $n_1', \ldots, n_{k-1}'$ to N, and k new edges $\langle n_1, A_1, n_1' \rangle$, $\langle n_1', A_2, n_2' \rangle$, ..., $\langle n_{k-2}', A_{k-1}, n_{k-1}' \rangle$, $\langle n_{k-1}', A_k, n_2 \rangle$ to E.

Again, as a straightforward consequence of our definitions, it follows that after $Create_G(n_1, Pf, n_2)$ is done, there exists a path p from n_1 to n_2 in G such that $\gamma(p) = Pf$.

For convenience, we omit the subscript when we refer to procedures *Reachable* and *Create* if graph G is understood from the context.

Given a description D, the *initial description graph*, $Init(D)$, is defined as $\langle \{n\}, \emptyset, \{\langle dn, n \rangle\} \rangle$ with $\alpha(n) = \{D\}$ and $fired(n) = \emptyset$. For example, a graphical representation of $Init$(QView) is depicted as follows:



5.5 Procedures

This section presents our algorithms in the form of procedures that manipulate description graphs and that can be used to solve logical implication problems. We also discuss a number of their properties.

5.5.1 Procedure *Merge*

Procedure *Merge*(G) for a description graph $G = \langle N, E, Refs \rangle$ transforms G by an exhaustive application of the following rewrite rules.

M1. If there exists an edge $\langle n_1, \mathbf{Id}, n_2 \rangle$ in E where $n_1 \neq n_2$, then merge nodes n_1 and n_2 as follows:
   (a) change $\alpha(n_1)$ to $\alpha(n_1) \cup \alpha(n_2)$, and *fired*($n_1$) to *fired*($n_1$) $\cup$ *fired*(n_2);
   (b) modify all edges touching n_2 to touch n_1 instead;
   (c) $\forall \langle r, n_2 \rangle \in Refs$, replace $\langle r, n_2 \rangle$ by $\langle r, n_1 \rangle$ in *Refs*; and
   (d) remove n_2 from N.

M2. If there exists an edge $\langle n, \mathbf{Id}, n \rangle$ in E then remove it.

M3. If there exist edges $\langle n_1, A, n_2 \rangle$ and $\langle n_1, A, n_3 \rangle$ in E for a primitive attribute A, then: if $n_2 = n_3$, then remove the second edge from E; otherwise, add edge $\langle n_2, \mathbf{Id}, n_3 \rangle$ to E if such an edge does not exist already.

Claim 5.2 *Let $G_0 = \langle N_0, E_0, Refs_0 \rangle$ denote a finite description graph, i.e. a graph with finite set N_0 and bag E_0, and let $[G_1, G_2, \ldots]$ (where $G_i = \langle N_i, E_i, Refs_i \rangle$) denote a sequence of description graphs obtained by a sequence of applications of rewrite rules defined by Merge(G_0). Then, Merge(G_0) terminates, and the resulting graph, $G_k = \langle N_k, E_k, Refs_k \rangle$, satisfies the following properties:*

(a) *there are no* **Id** *edges in G_k;*

(b) *no node in N_k has two outgoing edges with the same label; and*

(c) *if* $\langle ref_1, A, ref_2 \rangle \in E_0$ *for any references* ref_1 *and* ref_2 *in* $Refs_0$ *and a primitive attribute* A, *then* $\langle ref_1, A, ref_2 \rangle \in E_k$.

Proof. Let $|N_i|$ denote number of nodes, and $|E_i|$ number of edges in the graph G_i. Consider partitioning of edges in E_i into three groups: E_i^1, edges of the form $\langle n, \mathbf{Id}, n \rangle$; E_i^2, edges of the form $\langle n_1, \mathbf{Id}, n_2 \rangle$, where $n_1 \neq n_2$; and E_i^3, edges of the form $\langle n_1, A, n_2 \rangle$ for some nodes n, n_1, n_2 and a primitive attribute A. Let $|E_i^1|$, $|E_i^2|$, and $|E_i^3|$ denote sizes of E_i^1, E_i^2, and E_i^3 respectively.

Now, to prove termination, consider the following *weight* function of the graph G_i:

$$(|N_0|^2 + |E_0|) + 3 \cdot (|N_0|^2 + |E_0|) \cdot (|N_i| + 1) + |E_i^1| + |E_i^3| - |E_i^2|.$$

First, note that no rule in *Merge* adds nodes. Therefore, since (1) only rule M3 adds new edges, (2) M3 adds an edge $\langle n_1, \mathbf{Id}, n_2 \rangle$ only if such an edge does not already exist, and (3) no rule in *Merge* removes an edge $\langle n_1, \mathbf{Id}, n_2 \rangle$ for distinct nodes n_1 and n_2, the maximum number of edges that can ever be added does not exceed $|N_0|^2$. It follows that $|E_i| \leq |N_0|^2 + |E_0|$ which implies that $(|N_0|^2 + |E_0|) - |E_i^2| \geq 0$. Therefore, the weight function is non-negative. Next, observe that every rule in *Merge* decreases the weight function as follows:

- Rule M1 decreases $|N_i|$ by one, and since it does not modify the number of edges, even if all edges move from group E_i^2 to groups E_i^1 and E_i^3, the constant $3 \cdot (|N_0|^2 + |E_0|)$ would still be larger than the change in $|E_i^1| + |E_i^3| - |E_i^2|$ part that cannot exceed $2 \cdot (|N_0|^2 + |E_0|)$.
- Rule M2 decreases $|E_i^1|$ by one.
- If $n_2 = n_3$ in rule M3, $|E_i^3|$ is decreased by one. Otherwise, $|E_i^2|$ is increased by one.

Therefore, since the weight function is finite, decreases after each step of *Merge* and must be non-negative, the procedure must terminate.

Now we can assume that $Merge(G_0)$ terminates after k steps for some $k \geq 0$. Part (a) of the claim follows directly from the rules M1 and M2, since if there was an edge $\langle n_1, \mathbf{Id}, n_2 \rangle \in E_k$, then either M1 or M2 would still be applicable contrary to assumption that G_k is the last graph produced by *Merge*. Similarly, part (b) of the claim follows directly from the rule M3 and part (a): since there are no **Id** edges, if there was a node in N_k with two outgoing edges with the same label, M3 would still be applicable after step k.

To prove part (c), note that A is a primitive attribute, and therefore, $\langle ref_1, A, ref_2 \rangle$ cannot be removed by M2. If the edge is removed by M3 at some step i, exactly the same edge must already exist in E_i. Finally, if the edge is modified by M1, node(s) n_2 is(are) replaced by node(s) n_1 in the edge. However, step (c) of M1 ensures that all references pointing to n_2 are modified to point to n_1, and therefore, edge $\langle ref_1, A, ref_2 \rangle$ must still exist after M1 is applied.

■

If a finite description graph satisfies properties (a) and (b) of the Claim 5.2, we call it a *well-formed description graph*. Note that property (b) implies that the second component is actually a set rather than a bag of edges in a well-formed description graph. One of the important uses of procedure *Merge* inside other procedures presented in the chapter is to transform a modified description graph into a well-formed one. Note that no rule of procedure *Merge* is applicable to a well-formed graph.

5.5.2 Procedure *Exp*

As in Chapter 4, we restrict **fd** descriptions inside any terminology to be regular. In addition, if we are considering a problem of whether a terminology logically implies a constraint $D_1 < D_2$, we assume that all **fd** descriptions inside D_1 are also regular. (On the other hand, **fd** descriptions inside D_2 can be extended and not necessarily regular.) The regularity condition is essential for the termination of the following key procedure.

Expansion of a description graph $G = \langle N, E, Refs \rangle$ over a molecular terminology T, written $Exp(G, T)$, transforms G by an exhaustive application of the following rewrite rules.

E1. If there exists a node n with $\alpha(n) = \{D_1, \ldots, D_k, (\textbf{and } D_{k+1} \ldots D_m)\}$ in N, then change $\alpha(n)$ to $\{D_1, \ldots, D_k, D_{k+1}, \ldots, D_m\}$.

E2. If there exist nodes n_1 with $\alpha(n_1) = \{D_1, \ldots, D_k, (\textbf{all } A\ D)\}$ and n_2 in N, and an edge $\langle n_1, A, n_2 \rangle$ in E then:
   (a) remove (**all** A D) from $\alpha(n_1)$; and
   (b) add D to $\alpha(n_2)$.

E3. If there exists a node n_1 with $\alpha(n_1) = \{D_1, \ldots, D_k, (\textbf{all } A\ D)\}$, there is *no* edge $\langle n_1, A, n_2 \rangle$ in E for any node n_2 in N, and $D_{[u]}$ contains a **same-as** constructor anywhere inside it then:
   (a) add a new node n_2 to N and a new edge $\langle n_1, A, n_2 \rangle$ to E;
   (b) remove (**all** A D) from $\alpha(n_1)$; and
   (c) add D to $\alpha(n_2)$.

E4. If there exists a node n with $\alpha(n) = \{D_1, \ldots, D_k, (\textbf{same-as } Pf_1\ Pf_2)\}$ in N then:
   (a) remove (**same-as** Pf_1 Pf_2) from $\alpha(n)$;
   (b) add a new node n' to N;
   (c) execute $Create(n, Pf_1, n')$ and $Create(n, Pf_2, n')$; and
   (d) invoke $Merge(G)$.

E5. If there exist nodes n_1 with $\alpha(n_1) = \{D_1, \ldots, D_k, (\mathbf{fd}\ C\ Pf_1 \ldots Pf_m\ Pf)\}$ and n_2 (distinct from n_1) in N, such that
(1) $(\mathbf{fd}\ C\ Pf_1 \ldots Pf_m\ Pf)$ is regular;
(2) $C \in \alpha(n_2)$;
(3) for every Pf_i, $1 \le i \le m$, there is a prefix Pf_i' of Pf_i and a node n_i' in N such that $Reachable(n_1, Pf_i', n_i')$ and $Reachable(n_2, Pf_i', n_i')$ are true; and
(4) there is no prefix Pf' of Pf and a node n in N such that $Reachable(n_1, Pf', n)$ and $Reachable(n_2, Pf', n)$ are true;
then
(a) add a new node n to N;
(b) execute $Create(n_1, Pf, n)$ and $Create(n_2, Pf, n)$; and
(c) invoke $Merge(G)$.

E6. If there exists a node n in N, and a constraint $C < D$ in T such that $C \in \alpha(n)$ and $C \notin fired(n)$, then add D to $\alpha(n)$ and C to $fired(n)$.

E7. If there exists a node n in N, and a view definition $V \equiv D$ in T such that $V \in \alpha(n)$ and $V \notin fired(n)$, then add D to $\alpha(n)$ and V to $fired(n)$.

For simplicity, we omit the second parameter of *Exp* when the terminology is understood from the context.

Claim 5.3 *Let $G_0 = \langle N_0, E_0, Refs_0\rangle$ denote a well-formed description graph, and let $[G_1, G_2, \ldots]$ (where $G_i = \langle N_i, E_i, Refs_i\rangle$) denote a sequence of description graphs obtained by a sequence of applications of rewrite rules E1 through E7 defined by $Exp(G_0)$. Then, every G_i is also well-formed for all $i \ge 0$.*

Proof. Since procedure *Merge* is invoked as the last step of rules E4 and E5, Claim 5.2 ensures that the graph is well-formed after E4 and E5 are applied. The graph is also well-formed after rules E1, E2, E6, and E7 are applied since they do not modify nodes and edges in the graph. Finally, the graph is well-formed after E3 is applied since it only creates one new non-**Id** edge such that no other edge outgoing from the same node has the same label.

■

Our next claim establishes some useful properties about a graph resulting after a finite sequence of applications of rewrite rules defined by procedures *Merge* and *Exp* as well as of the following additional rules:

A1. Given a description graph $G = \langle N, E, Refs \rangle$, a node $n \in N$, and an attribute description A which is either a primitive attribute or **Id**,
(a) add a new node n' to N; and
(b) add a new edge $\langle n, A, n' \rangle$ to E.

A2. Given a description graph $G = \langle N, E, Refs \rangle$, add a new node n to N.

A3. Given a description graph $G = \langle N, E, Refs \rangle$, nodes n_1 and n_2 in N, and an attribute description A which is either a primitive attribute or **Id**, add a new edge $\langle n_1, A, n_2 \rangle$ to E.

Claim 5.4 *Let $G_0 = \langle N_0, E_0, Refs_0 \rangle$ denote a finite description graph. Then, the following properties hold for any graph $G = \langle N, E, Refs \rangle$ that results from applying any number of rewrite rules from the set R = {M1, M2, M3, E1, E2, E3, E4, E5, E6, E7, A1} to G_0:*

(a) *Change of a reference in set Refs neither affects applicability of rules from set R, nor their effect on nodes, edges, α sets or other references in set Refs;*
(b) *if $G_0 = Init(D_1)$ for some description D_1, there is a path from dn to every node in N; and*
(c) *if $\langle ref_1, A, ref_2 \rangle \in E_0$ for any references ref_1 and ref_2 in $Refs_0$ and a primitive attribute A, then $\langle ref_1, A, ref_2 \rangle \in E$ even if R is extended to include rules A2 and A3.*

Proof. Part (a) follows from the fact that neither conditions of applicability of rules from set R, nor the modifications to the nodes, edges, α sets or other references in set *Refs* by the rules in R depend on a value of any of the bindings in *Refs*.

Next, consider part (b). Procedure *Reachable* does not change the graph. On the other hand, procedure *Create* and rule A1 only create new nodes that are reachable from already reachable nodes. Thus, in order to prove part (b), it suffices to show that no rules in *Merge* or *Exp* create an "unreachable" node. However, rules E1, E2, E6, and E7 only modify α sets; rule E3 creates only nodes that are reachable from other nodes; and rules E4 and E5 also create only nodes that are reachable from already reachable nodes before invoking *Merge* procedure. Thus, it is sufficient just to consider procedure *Merge*.

By definition of M1, for any path p from dn to some node n before M1 is applied, there is a path p' from dn to n (assuming n is not removed by M1) after M1 applies. Path p' is exactly the same as p except for every occurrence of node n_2 is replaced by n_1. Thus, M1 preserves property (b). M2 also preserves the property since one can remove edge $\langle n,$ **Id**, $n \rangle$ from any path without "invalidating" the path. Finally, M3 preserves property (b) as well, since it either removes an edge that has a duplicate in E, or adds an edge that connects two already existing nodes. Therefore, since the only node in G_0 is reachable from itself (by path $\langle dn_0 \rangle$), G must satisfy property (b).

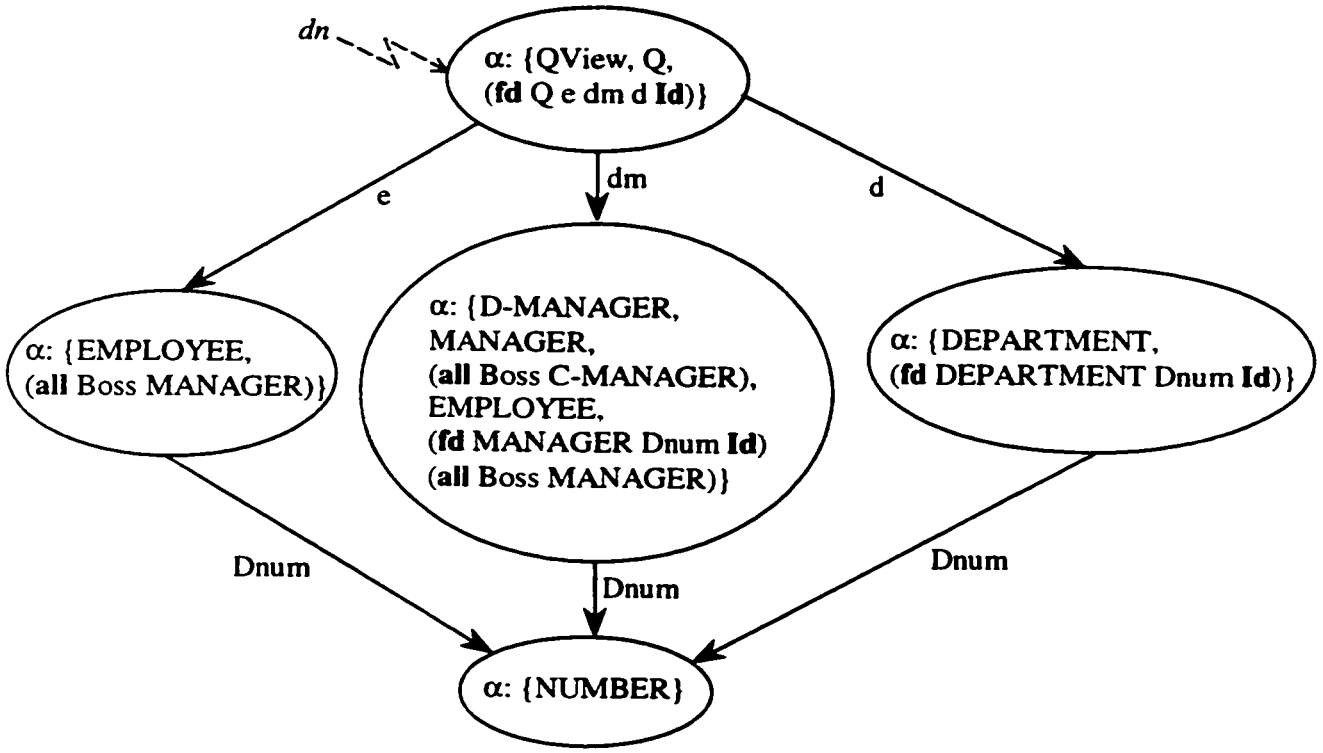


Figure 5.2: The graph resulting after invocation of *Exp*(*Init*(QView), COMPANY').

Part (c) of the claim follows directly from part (c) of Claim 5.2 and from the fact that none of the rules E1 through E7 or A1 through A3 changes references in the *Refs* set or removes any edges, except possibly inside invocations of procedure *Merge*.

■

Note that part (a) of Claim 5.4 allows us to use extra references to nodes in our proofs knowing that they will not affect applications of the rules.

As an example of an application of procedure *Exp*, consider the call to *Exp*(*Init*(QView), COMPANY'). The graph resulting after the call is presented in Figure 5.2. Note that the graph does not show the *fired* sets for brevity and uses dashed arrows to show references to nodes (in this case, the *dn* reference). Each node contains exactly the same concepts and view names in its *fired* set as the ones that are in its α set since otherwise rule E6 or E7 would still be applicable.

5.5.3 Procedure *Subsumes*

Given a description D, a description graph $G = \langle N, E, Refs\rangle$ and a terminology T, $Subsumes(D, \langle N, E, Refs\rangle, T)$ returns true if and only if any of the following cases returns true.

S1. If D is a primitive concept name, then return true if and only if D occurs in $\alpha(dn)$.

S2. If D has the form (**and** D_1 ... D_m), return true if and only if $Subsumes(D_i, \langle N, E, Refs\rangle, T)$ is true for every $1 \le i \le m$.

S3. If D has the form (**all** A D'), modify G as follows:
   (a) add a new node n to N;
   (b) add a new edge $\langle dn, A, n\rangle$ to E;
   (c) change reference dn to point to the node n but remember the current distinguished node: construct a set $Refs'$ to be the same as $Refs$ except for the old dn binding $\langle dn, n'\rangle$ for some node n' replaced by $\langle dn, n\rangle$ and for the additional binding $\langle dn', n'\rangle$ for some new reference dn' not occurring in set $Refs$;
   (d) invoke $Merge(G)$; and
   (e) invoke $Exp(\langle N, E, Refs'\rangle, T)$.
   After a call to $Subsumes(D', \langle N, E, Refs'\rangle, T)$ and then restoring the distinguished node binding dn to $\langle dn, dn'\rangle$, return the result of the recursive call.

S4. If D has the form (**same-as** Pf_1 Pf_2), modify G as follows:
   (a) add two new nodes n_1 and n_2 to N;
   (b) execute $Create(dn, Pf_1, n_1)$, and $Create(dn, Pf_2, n_2)$;
   (c) invoke $Merge(G)$; and
   (d) invoke $Exp(\langle N, E, Refs\rangle, T)$.
   Return true if and only if there exists a node n in N, such that $Reachable(dn, Pf_1, n)$ and $Reachable(dn, Pf_2, n)$ are true.

S5. If D has the form (**fd** D' Pf_1 ... Pf_m Pf) for some description D', modify a copy $G_c = \langle N_c, E_c, Refs_c\rangle$ of G as follows:
   (a) add a new node n_0 with $\alpha(n_0) = \{D'\}$ to N_c;
   (b) add $\langle fdcn_c, n_0\rangle$ to $Refs_c$;
   (c) add m new nodes $n_1, \ldots, n_m$ (with empty α sets) to N_c;
   (d) execute $Create(n_0, Pf_i, n_i)$, and $Create(dn_c, Pf_i, n_i)$ for every $1 \le i \le m$;
   (e) invoke $Merge(G_c)$; and
   (f) invoke $Exp(\langle N_c, E_c, Refs_c\rangle, T)$.
   Return true if and only if there exists a prefix Pf' of Pf and a node n in N_c such that $Reachable(dn_c, Pf', n)$ and $Reachable(fdcn_c, Pf', n)$ are true.

S6. If D is a view name V with view definition $V \equiv D'$ in T, then if V occurs in $\alpha(dn)$ return true; otherwise, return true if and only if *Subsumes*(D', $\langle N, E, Refs \rangle$, T) is true.

Note that as discussed before, we allow extended **fd** descriptions inside the parameter description D. Also note that the **fd** descriptions in D do *not* have to be regular.

As an example, consider the call to

Subsumes((**fd** QView e dm d), *Exp*(*Init*(QView)), COMPANY').

As a result of the call, a copy of the graph *Exp*(*Init*(QView)) is constructed and an additional node referenced by $fdcn_c$ with "QView" in its α set is added along with two pairs of edges labeled "e" and "dm". Next, after the invocation of *Merge* in part (e) of step S5, procedure *Exp* "expands" the "QView" description and, at some point, might obtain a result depicted in Figure 5.3. Then, after the merges shown in the figure by the two dashed lines are complete, the graph becomes the same as the one presented in Figure 5.2 (except for the additional $fdcn_c$ reference pointing at the same node as dn_c). Therefore, the appropriate check at the end of step S5 returns true. As our further analysis shows, this result confirms that QView < (**fd** QView e dm d) is indeed logically implied by the COMPANY' terminology.

Note that while our examples have been using path functions of only lengths one and zero for simplicity, the procedures are general enough to work with path functions of arbitrary length (unless the path functions violate the regularity condition of the **fd** descriptions in a terminology). For example, the procedures would work with such constraints as

MSS < (**fd** MSS (**comp** Boss Dnum) Boss)

and

MSS < (**fd** MSS (**comp** Boss Dnum) Dnum)

that require any pair of MSS with the same department number of their bosses to have the same boss and the same department number respectively.

5.6 Analytic results

In this section, we will establish the main results about our procedures in this chapter. We first show that procedures *Exp* and *Subsumes* terminate. Then, we consider a problem of whether a

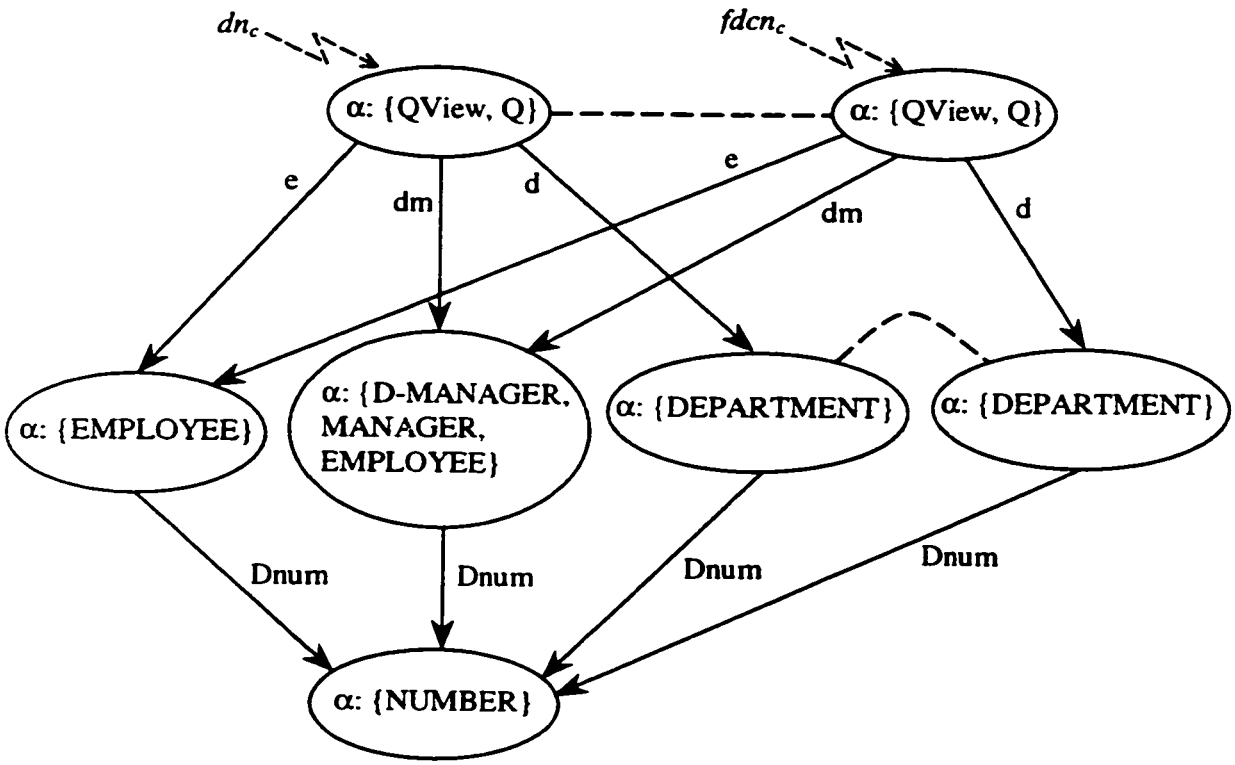


Figure 5.3: Execution of *Subsumes*((**fd** QView e dm d), *Exp*(*Init*(QView)), COMPANY'). Nodes that are connected by the dashed lines are merged. Only concepts and view names are shown inside α sets.

terminology T logically implies a constraint $D_1 < D_2$ for any descriptions D_1 and D_2 that satisfy the conditions we outlined above. We claim that the problem can be solved by invoking procedure

$$Subsumes(D_2, Exp(Init(D_1), T), T)$$

and prove that our solution is sound and complete. Finally, we establish a bound on the running time of our procedures.

5.6.1 Termination

The most difficult part in proving that $Subsumes(D_2, Exp(Init(D_1), T), T)$ terminates is establishing termination of the *Exp* procedure. In order to do that, we first prove an auxiliary claim about procedure *Merge*.

Claim 5.5 *Let $G_0 = \langle N_0, E_0, Refs_0 \rangle$ denote a finite description graph, i.e. a graph with finite set N_0 and finite bag E_0, and let $[G_1, G_2, \ldots, G_k]$ (where $G_i = \langle N_i, E_i, Refs_i \rangle$) denote a sequence of description graphs obtained by a sequence of applications of rewrite rules defined by Merge(G_0). Then:*

(a) *for every $0 \le j \le i \le k$, if ref_1 and ref_2 point to the same node in $Refs_j$, ref_1 and ref_2 point to the same node in $Refs_i$ as well;*

(b) *for every $0 \le j \le i \le k$, if $\langle ref_1, A, ref_2 \rangle \in E_j$ for any references ref_1 and ref_2 in $Refs_j$ and a primitive attribute A, then $\langle ref_1, A, ref_2 \rangle \in E_i$;*

(c) *for every $0 \le i \le k$, if $\langle ref_1, \mathbf{Id}, ref_2 \rangle \in E_i$ for some references ref_1 and ref_2 in $Refs_i$, ref_1 and ref_2 point to the same node in $Refs_k$; and*

(d) *for every $0 \le i \le k$, if $\langle ref_1, A, ref_2 \rangle \in E_i$ and $\langle ref_1, A, ref_3 \rangle \in E_i$ for some references ref_1, ref_2, and ref_3 in $Refs_i$ and a primitive attribute A, then ref_2 and ref_3 point to the same node in $Refs_k$.*

Proof. Part (a) follows from the facts that rules M2 and M3 of *Merge* do not touch any references, while M1 moves *all* references that point to a node that is being removed to another node.

Since A is not "**Id**" in part (b), M2 does not affect edge $\langle ref_1, A, ref_2 \rangle$ in any way. If the edge is modified by step (b) of M1, step (c) of M1 ensures that the references ref_1 and ref_2 are modified accordingly. Finally, the edge is removed by M3 only if a duplicate edge exists in the graph. Therefore, part (b) of the claim must hold.

Analogously to the proof of part (b), we can show that the edge $\langle ref_1, \mathbf{Id}, ref_2 \rangle$ in part (c) must exist in all graphs G_j ($i \le j \le k$) until the references start pointing to the same node, since if they point to distinct nodes, M2 is not applicable; M1 changes edges according to changes to the references to corresponding nodes; and M3 does not modify **Id** edges. However, *Merge* cannot finish until it removes all **Id** edges from the graph (by part (a) of Claim 5.2). Therefore, there must be a point when rule M1 merges nodes referenced by ref_1 and ref_2. After M1 is applied ref_1 and ref_2 continue to reference the same node by part (a) of Claim 5.5.

Finally, part (d) follows from part (b) of Claim 5.2, the fact that by part (b) of Claim 5.5, both edges $\langle ref_1, A, ref_2 \rangle$ and $\langle ref_1, A, ref_3 \rangle$ must exist in G_j for all $i \le j \le k$, and the fact that edges would be different if ref_2 and ref_3 pointed to different nodes.

■

Now we establish the termination of procedure *Exp*.

Lemma 5.3 *Let $G_0 = \langle N_0, E_0, Refs_0 \rangle$ denote a description graph that satisfies the following properties:*

(a) *G_0 is finite; i.e. it has finite set N_0 and finite bag E_0;*

(b) *there is no* **Id** *edges in E_0; and*

(c) *no node in N_0 has two outgoing edges with the same label.*

Then, for any terminology T, $Exp(G_0, T)$ terminates.

Proof. Consider any sequence $seq_0 = [G_1, G_2, \ldots]$ (where $G_i = \langle N_i, E_i, Refs_i \rangle$) of description graphs obtained by a sequence of applications of rewrite rules defined by $Exp(G_0)$. To prove the lemma, we are going to show that the sequence must be finite.

Note that G_i must be a finite graph since G_1 is finite, G_i was obtained only by a finite number of applications of rules E1 through E7, and application of each rule adds at most a finite number of nodes, edges and/or descriptions to the α and *fired* sets.

Our general plan is as follows. We first show that the number of applications of rules E7 and E4 must be finite. We then deduce that the number of applications of rule E3 must be also finite. That leaves only rule E5 as a possible source of new nodes in the graph. Then, we show that a certain *boundary property* is preserved by rule E5, and that E5 can only "fire" on a finite number of pairs of nodes in a graph that satisfies the boundary property. After proving that rule E5 can also fire only a finite number of times, we finish the proof by verifying that rules E1, E2, and E6 can also be applied only a finite number of times.

First, let $Descs(G)$ denote union of α sets of all nodes in a description graph G. Clearly, rewrite rules E1 to E5 do not introduce new occurrences of view names or **same-as** constructor to $Descs(G)$. Neither does rule E6 since, by definition of terminology, constraints of the form $C < D$ in T cannot contain view names or **same-as** constructor inside D. Therefore, only rule E7 can introduce new occurrences of view names and **same-as** constructor. Further note that when rule E7 is applied to an occurrence of a view name, say V, it cannot be ever applied to that occurrence of V again due to the addition of V to the *fired* set (note that even when a node with α set containing V gets merged, both α and *fired* sets still include V after the merge). Moreover, the non-recursive property of views guarantees that a new occurrence of V will never be introduced by an application of rule E7 to V or any occurrence of a view name that was produced as a result of "expanding" V (after one or more expansions by rule E7). Therefore, after E7 applies to an occurrence of a view name V, it never applies to it again, and it never produces V in another description again. Since the number of view definitions in terminology is finite, and the number of occurrences of view names in $Descs(G_0)$ is finite, it follows that rule E7 can "fire" at most a finite number of times.

Moreover, since each application of E7 adds at most a finite number of occurrences of the same-as constructor to a finite number of occurrences of the **same-as** constructor in $Descs(G_0)$,

the total number of occurrences of the **same-as** constructor that are ever added and exist in the graph (without subtracting the ones that get removed) is finite.

On the other hand, each application of rule E4 decreases the number of occurrences of the **same-as** constructor in *Descs*(G). Therefore, E4 can also be applied only a finite number of times. Thus, there must exist some graph $G_1' = \langle N_1', E_1', Refs_1' \rangle = G_i$ in seq_0 such that rules E7 and E4 are *not* applied to any graph G_j in seq_0 for all $j \geq i$. Therefore, we can now concentrate on the part of the seq_0 that starts with G_1' and does not use rules E7 and E4 to obtain further graphs in the sequence. Let us denote this "tail" of the sequence by seq_1, i.e. $seq_1 = [G_1' = \langle N_1', E_1', Refs_1' \rangle, \ldots]$.

Since each application of rule E3 in seq_1 decreases a finite number of concept constructors of a description containing an occurrence of a **same-as** constructor or a view name in G_1', and rules E1, E2, E3, E5 and E6 do not increase the number of concept constructors containing the description, E3 can only be applied a finite number of times to a description with a given occurrence of a **same-as** constructor or a view name. However, since rules E4 and E7 are not applied in seq_1, and since no new occurrences of the **same-as** constructor or view names can be introduced to *Descs*(G) by other rules, there must exist some graph $G_2' = \langle N_2', E_2', Refs_2' \rangle = G_i$ in seq_1 such that rules E3, E4 and E7 are *not* applied to any graph G_j in seq_1 for all $j \geq i$. Therefore, we can now concentrate on the part of the seq_1 that starts with G_2' and does not use rules E3, E4 and E7 to obtain further graphs in the sequence. Let us denote this "tail" of the sequence by seq_2, i.e. $seq_2 = [G_2' = \langle N_2', E_2', Refs_2' \rangle, \ldots]$.

Since rules E3 and E4 are not applied in seq_2, only rule E5 can produce new nodes in the graph or, in general, modify nodes and edges in the graph. However, since nodes can be "merged" by procedure *Merge* in step (c) of E5, we use references to keep track of nodes in the graph. Thus, let us assume that every node in G_2' has a reference pointing to it in $Refs_2'$ set, and every time we create new nodes (by E5), we create new references in the *Refs* set pointing to those nodes.

We now partition *Refs* set of a graph G in seq_2 into two subsets R_{old} and R_{new}: we say a reference *ref* is in R_{old} if there is a reference *ref'* in $Refs_2'$ that points to the same node as *ref*; otherwise, *ref* is in R_{new}. Essentially, R_{old} is a set of references that point to "old" nodes in N_2' possibly merged with other nodes, whereas R_{new} is a set of references to "new" nodes added by applications of rule E5 and that did not get merged with old nodes. Note that by definition, every node n is either referenced only by references from R_{old} or only by references from R_{new}. In the first case, we say that *n relates to* R_{old}; and in the second case, we say that *n relates to* R_{new}. Since we create a reference with every new node and the reference moves from a node only if the node is removed by M1, every node must be pointed to by at least one reference, and therefore, every node either relates to R_{old} or R_{new}.

Let us now show that any graph G in seq_2 satisfies so-called *boundary property*, which states that no node n that relates to R_{new} has any pa-children (i.e. outgoing non-**Id** edges). A structure enforced by the boundary property is depicted in Figure 5.4(a) at the end of the proof.

Since only rule E5 modifies nodes and edges in the graph, in order to prove that the boundary property is preserved, it is sufficient to consider only applications of rule E5 in seq_2.

The boundary property trivially holds in G_2' since every reference in $Refs_2'$ is in R_{old}. Assume now that the property holds for some graph $G = \langle N, E, Refs \rangle$ in seq_2 and, without loss of generality, consider an application of rule E5 to some nodes n_1 and n_2 in N for description $D = (\mathbf{fd}\ C\ Pf_1\ \ldots\ Pf_m\ Pf)$ in α set of one of those nodes. Additionally, let us assume that $Pf = (\mathbf{comp}\ A_1\ (\mathbf{comp}\ A_2\ \ldots\ (\mathbf{comp}\ A_k\ \mathbf{Id})\ \ldots\ ))$ for some $k \geq 0$ and primitive attributes $A_1, \ldots, A_k$. Since description D must be regular by property (1) of E5, there must exist a Pf_i $(1 \leq i \leq m)$ that has the form $(\mathbf{comp}\ A_1\ (\mathbf{comp}\ A_2\ \ldots\ (\mathbf{comp}\ A_{k-1}\ Pf'')\ \ldots\ ))$ for some path function Pf''. In addition, by condition (3) of E5, there must exist a node n in N and a prefix Pf_{prefix} of Pf_i such that $Reachable(n_1, Pf_{prefix}, n)$ and $Reachable(n_2, Pf_{prefix}, n)$ are true. In other words, there must exist paths p_1 and p_2 from n_1 and n_2, respectively, to n such that $\gamma(p_1) = \gamma(p_2) = Pf_{prefix}$.

Therefore, there are three possible cases relating Pf and Pf_{prefix}:

(i) $Pf = \mathbf{Id}$;
(ii) length of Pf is greater than 0 and $Pf_{prefix} = (\mathbf{comp}\ A_1\ \ldots\ (\mathbf{comp}\ A_l\ \mathbf{Id})\ \ldots\ ))$ for some $1 \leq l \leq k-1$; or
(iii) length of Pf is greater than 0 and $Pf_{prefix} = (\mathbf{comp}\ A_1\ \ldots\ (\mathbf{comp}\ A_{k-1}\ Pf')\ \ldots\ ))$ for some prefix Pf' of Pf''.

Note that the length of Pf_{prefix} cannot be 0, since n_1 and n_2 must be distinct nodes. Moreover, n_1 and n_2 must relate to R_{old}. Indeed, since they are distinct, and paths p_1 and p_2 consist of non-**Id** edges that end at the same node, both n_1 and n_2 must have pa-children. However, since G satisfies the boundary property, neither n_1 nor n_2 can relate to R_{new}.

Before we proceed with proving that the boundary property is preserved by cases (i) to (iii), let us also note that **Id** edges that can be removed by M2 do not affect applications of rules M1 and M3. Therefore, such **Id** edges can be removed at any time during an execution of *Merge* without changing nodes, other edges or references in the graph, and thus, without affecting the modifications produced by a sequence of applications of rules M1 and M3. Thus, while considering a sequence of steps during an invocation of *Merge* procedure we can disregard such edges and applications of rule M2.

Now, consider case (i). E5 adds a new node n' to N (with corresponding new reference ref'), and edges $\langle n_1, \mathbf{Id}, n' \rangle$ and $\langle n_2, \mathbf{Id}, n' \rangle$ to E. Since the only added node does not have any pa-children, the boundary property is preserved at this point. Next, consider the invocation of procedure *Merge*. Since removal of a duplicate edge or addition of an **Id** edge by M3 and removal

of an **Id** edge by M2 cannot violate the boundary property that only considers non-**Id** edges, it is sufficient to consider merges by M1. There are three possible cases: both nodes that are merged relate to R_{old}; one of the nodes relates to R_{old} and the other relates to R_{new}; and both nodes relate to R_{new}. In the first two cases, the node resulting after application of M1 relates to R_{old} by our definitions of R_{old} and R_{new} sets. Moreover, all pa-ancestors of both nodes remain in R_{old}. Similarly, in the third case, all pa-ancestors of the nodes remain in R_{old}. However, the node resulting after application of M1 remains in R_{new} and does not have any pa-children since neither of the nodes have pa-children before M1 is applied. Therefore, applications of the rules of *Merge* preserve the boundary property.

It follows that, the boundary property is preserved by case (i). It is also preserved by case (ii) since condition (4) of E5 ensures that the rule does not "fire" in this case, and therefore, the graph is not modified.

Finally, consider case (iii). Without loss of generality, let us assume that $p_1 = \langle n_1, A_1, ref_2', A_2, \ldots, ref_{k-1}', A_{k-1}, ref_k' \rangle \circ p_3$ and $p_2 = \langle n_2, A_1, ref_2'', A_2, \ldots, ref_{k-1}'', A_{k-1}, ref_k'' \rangle \circ p_4$ for some paths p_3 and p_4 such that $\gamma(p_3) = \gamma(p_4) = Pf'$. Also, assume that part (a) of E5 added new node n' (with reference r') and that $Create(n_1, Pf, r')$ created path $\langle n_1, A_1, r_2', A_2, \ldots, r_k', A_k, n' \rangle$, whereas $Create(n_2, Pf, r')$ created path $\langle n_2, A_1, r_2'', A_2, \ldots, r_k'', A_k, n' \rangle$ for some references r_i' and r_i'' $(2 \leq i \leq k)$. An example structure of such a graph is depicted in Figure 5.4(b). Note that if ref_k' and ref_k'' pointed to the same node, i.e. if paths p_1 and p_2 "merged" before those nodes, E5 would not "fire" due to condition (4).

Next, consider what happens during the invocation of *Merge* procedure. It follows from Claim 5.3 that G is a well-formed graph, and therefore, rules of *Merge* are *not* applicable to any part of G itself, and can only either merge nodes referenced by ref_2' and r_2' or ref_2'' and r_2''. After ref_2' is merged with r_2' (resp. ref_2'' with r_2''), ref_3' has to be merged with r_3' (resp. ref_3'' with r_3''). This process continues until every node ref_i' is merged with node r_i' (resp. ref_i'' with r_i'') for $2 \leq i \leq k$. By constructions of nodes referenced by r_i' and r_i'' $(2 \leq i \leq k)$ and by definition of the *Merge* procedure, the net result of the merges so far are just additional references r_i' and r_i'' pointing to nodes that already existed in G, additional **Id** edges (to be removed by M2), and edges $\langle ref_k', A_k, n' \rangle$ and $\langle ref_k'', A_k, n' \rangle$. As noted above, ref_k' and ref_k'' point to distinct nodes since E5 would not fire otherwise. Therefore, both ref_k' and ref_k'' as well as their pa-ancestors including ref_i' and ref_i'' $(2 \leq i \leq k-1)$ must be in R_{old} since they have a common pa-descendent. Thus, when these nodes are merged with r_i' and r_i'' $(2 \leq i \leq k)$, the nodes that result from the merges relate to R_{old}. It follows that the only new node at this point that relates to R_{new} is n'. Moreover, since other nodes, edges and references remain the same, both pa-parents of n' relate to R_{old}, and therefore, the graph satisfies the boundary property. Finally, using the argument presented in case (i) that establishes that every rule of *Merge* preserves the boundary property, we can conclude that the graph resulting after the application of *Merge* in step (c) of E5 in case (iii) satisfies the boundary property.

It follows that in general, application of rule E5 preserves the boundary property of the graph; and therefore, every graph in seq_2 must satisfy it. This implies, however, that E5 can only be applied to nodes that are referenced by references in $Refs_2'$. Indeed, since a node that relates to R_{new} does not have any pa-children, it cannot "agree" on a path with a distinct node (as required by part (3) of E5).

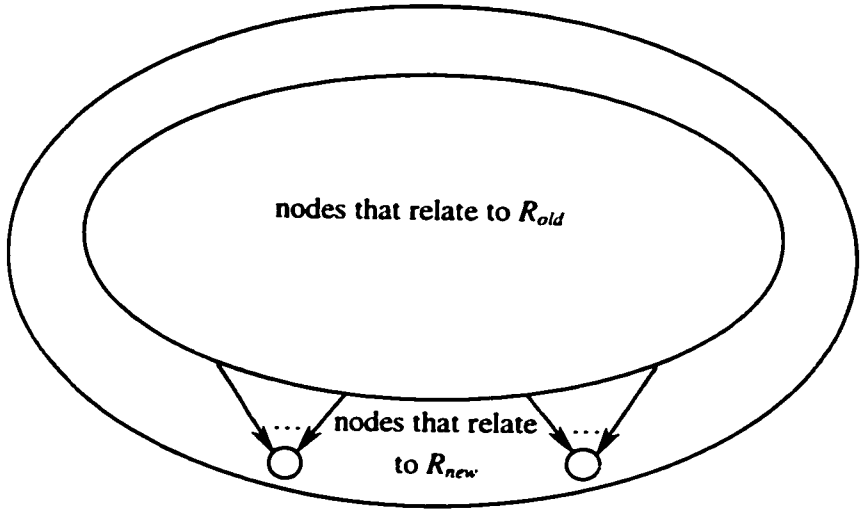
Let us now show that given any references ref_1 and ref_2, and a description D = (**fd** C Pf_1 ... Pf_m Pf), E5 can be applied to the pair (ref_1, ref_2) at most once for the description D; in other words, E5 cannot "fire" more than once given nodes referenced by ref_1 and ref_2 and a particular **fd** description D in $\alpha(ref_1)$.

There are two cases: Pf = **Id**, or Pf = (**comp** A_1 (**comp** A_2 ... (**comp** A_k **Id**) ...)) for some $k > 0$ and primitive attributes A_1, ..., A_k. In the first case, application of the rule E5 produces a new node n and two edges $\langle ref_1, \textbf{Id}, n\rangle$ and $\langle ref_2, \textbf{Id}, n\rangle$. By part (c) of Claim 5.5, after *Merge* is done, ref_1 and ref_2 point to the same node, and therefore, they would never again point to distinct nodes. Thus, E5 cannot be applied to those references again.

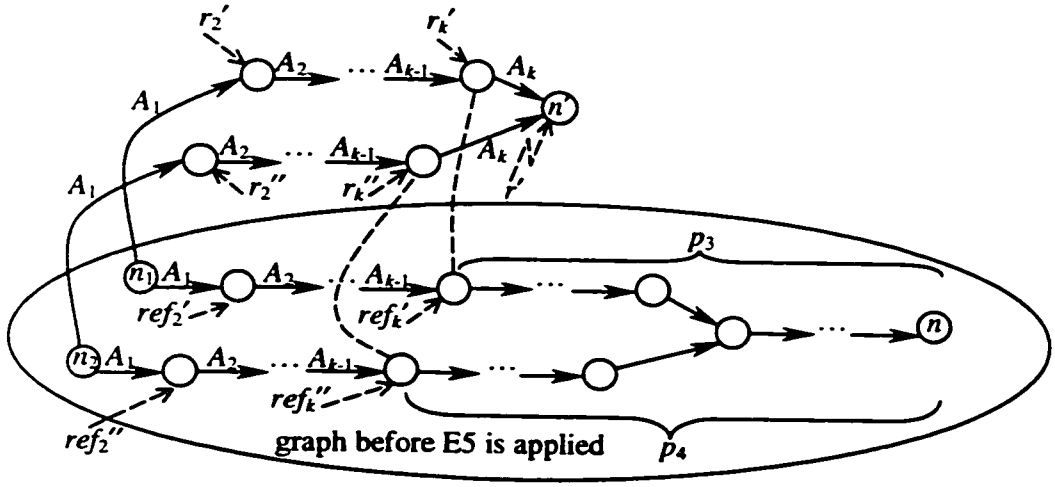
In the second case, when Pf = (**comp** A_1 (**comp** A_2 ... (**comp** A_k **Id**) ...)) for some $k > 0$, part (c) of Claim 5.2, part (c) of Claim 5.4, and the fact that parts (a) and (b) of E5 do *not* change nodes, edges and references that already exist in the graph imply that after E5 is applied, there will always exist paths that start at ref_1 and ref_2, have edges labeled A_1, ..., A_k, and end at the same node. Therefore, condition (4) of E5 prevents it from "firing" again for the description D.

We can conclude now that since $Refs_2'$ has a finite number of references; rule E5 is applicable only to nodes that are referenced by references in $Refs_2'$; T has finite number of **fd** descriptions inside its constraints; and E5 can "fire" at most once for a given (ordered) pair of references and an **fd** description, the rewrite rule E5 can be applied only a finite number of times to a graph. Therefore, there must exist a graph $G_3' = \langle N_3', E_3', Refs_3'\rangle = G_i$ in seq_2 such that rule E5 is not applied to any graph G_j in seq_2 for all $j \geq i$. Thus, we can now concentrate on the part of the seq_2 that starts with G_3' and does not use rules E3, E4, E5, and E7 to obtain further graphs in the sequence. Let us denote this "tail" of the sequence by seq_3, i.e. $seq_3 = [G_3', \ldots]$.

Since rules E1, E2 and E6 do not modify nodes and edges in the graph, the structure of the graph stays the same in seq_3. However, since rule E6 can be applied at most once to a node for a particular constraint $C < D$ in T (addition of C to the *fired* set of a node n the first time the constraint is applied to n prevents it from "firing" again), and since T has a finite number of constraints, rule E6 can be applicable only a finite number of times in seq_3. Thus, there must exist a graph $G_4' = \langle N_4', E_4', Refs_4'\rangle = G_i$ in seq_3 such that rule E6 is not applied to any graph G_j in seq_3 for all $j \geq i$. Therefore, we can now concentrate on the part of seq_3 that starts with G_4' and does not use rules E3, E4, E5, E6, and E7 to obtain further graphs in the sequence. Let us denote this "tail" of the sequence by seq_4, i.e. $seq_4 = [G_4', \ldots]$.



(a) General structure of a graph that satisfies the boundary property.



(b) A graph structure for case (iii). Dashed lines show some nodes that are to be merged and dashed arrows show some references.

Figure 5.4: Graph structures in the proof of termination.

Since $Descs(G_4')$ contains a finite number of **and** and **all** constructors, and since every time rule E1 or E2 is applied, the number of **and** or **all** constructors is decreased by 1; E1 and E2 can fire at most a finite number of times. Therefore, seq_4, and thus seq_0, must be finite.

■

Next, termination of *Subsumes* follows and is stated in the following theorem.

Theorem 5.1 (*termination*) *Let $G = \langle N, E, Refs \rangle$ denote a finite description graph; i.e. a graph with finite set N and finite bag E. Then, Subsumes(D, G, T) terminates for any description D and terminology T, and the possibly modified description graph remains finite.*

Proof. First, note that applications of rules M1 to M3, E1 to E7 and all other steps during invocation of *Subsumes* add/remove at most a finite number of nodes and edges to/from the graph and descriptions to/from the node labels and take a finite amount of time for all necessary checks to be performed. Therefore, in order to show termination of *Merge*, *Exp*, or any part of *Subsumes*, it is sufficient to show that they execute at most a finite number of steps.

We first consider applications of steps of *Subsumes* that do not make any recursive calls: S1, S4 and S5. S1 trivially takes a finite amount of time and does not modify the graph. Further, S4 (resp. S5) trivially takes a finite amount of time to execute parts (a) and (b) (resp. (a) through (d)). Moreover, the graph remains finite. Thus, by Claim 5.2, invoking *Merge* in part (c) (resp. (e)) terminates and obtains a graph that satisfies conditions (b) and (c) of Lemma 5.3. In addition, the obtained graph is finite by the reasoning above. Therefore, by Lemma 5.3, invoking procedure *Exp* at part (d) (resp. (f)) terminates, and thus, obtains a finite graph as well (in case of S5, both the original graph and the modified copy of the graph are finite).

Analogously to S4 and S5, following a trivially terminating sequence of parts (a) through (c), invocations of *Merge* and *Exp* in step S3 terminate and obtain a finite graph. Additionally, steps S2 and S6 trivially take at most a finite amount of time before making recursive calls (if any in case of S6). Therefore, to prove termination, it is sufficient to show that S2, S3, and S6 make at most a finite number of recursive calls since each recursive call passes a finite description and steps S1, S4 and S5 terminate for any description (that applies to them).

Steps S2 and S3 "break up" a description in a finite amount of time and recursively call *Subsumes* passing descriptions with at least one concept constructor less as a parameter. Step S6, on the other hand, either trivially executes in a finite amount of time if the view name is found in the α set, or recursively calls *Subsumes* passing description that corresponds to the view in its view definition. However, by the non-recursive property of views, and since only step S6 can consider new view names, each occurrence of a view name can be "unfolded" by only a finite number of applications of S6. Indeed, since terminology is finite and substituting view names by

the corresponding descriptions can never produce an original view name, each occurrence of a view name can produce at most a finite number of such substitutions. In other words, for an occurrence of a view V, applications of S6 can produce at most the number of constructors in $V_{[u]}$. Therefore, S6 can be applied at most a finite number of times and produce at most a finite number of concept constructors. As we noted above, however, each application of S2 and S3 decreases the number of concept constructors "considered" by *Subsumes*, and therefore, they can be applied at most a finite number of times as well.

■

5.6.2 Soundness

Next we prove soundness of our procedures. We show that whenever

$$Subsumes(D_2, Exp(Init(D_1), T), T)$$

returns true for any terminology T, and descriptions D_1 and D_2,

$$T \models D_1 < D_2.$$

We start by proving the following lemma that establishes preservation of an important invariant by rewrite rules defined in procedures *Merge* and *Exp* and by rule A1.

Lemma 5.4 *Let $G_0 = \langle N_0, E_0, Refs_0 \rangle = Init(D_1)$ for some description D_1. Then, given a terminology T, the following properties hold for any graph G that results from applying any number of rewrite rules from the set R = {M1, M2, M3, E1, E2, E3, E4, E5, E6, E7, A1} to G_0:*

(1) *If there exists a path $p = \langle dn, A_1, n_1, A_2, n_2, \ldots, A_m, n_m \rangle$ in the description graph G such that $D \in \alpha(n_m)$, then $T \models D_1 <$ (**all** $\gamma(p)$ D). (If $m = 0$, $T \models D_1 <$ (**all Id** D), or $T \models D_1 < D$.)*

(2) *If there exist paths $p_1 = \langle dn, A_1, n_1', A_2, n_2', \ldots, A_k, n_k' \rangle$ ($k \geq 0$) and $p_2 = \langle dn, B_1, n_1'', B_2, n_2'', \ldots, B_m, n_m'' \rangle$ ($m \geq 0$) in G such that $n_k' = n_m''$, then $T \models D_1 <$ (**same-as** $\gamma(p_1)$ $\gamma(p_2)$).*

Proof. First, we prove that the properties are satisfied by the graph $Init(D_1)$. Then, we show that the properties are preserved by every rewrite rule in R.

By definition of $Init(D_1)$, the only way D could be in the α set of n_0, the only node created by *Init*, is if $D_1 = D$; but then, $T \models D_1 < D =$ (**all Id** D) ($m = 0$). Since no other nodes are

created by *Init* and no other descriptions are placed into $\alpha(n_0)$, property (1) holds for the graph $Init(D_1)$. Property (2) is trivially satisfied as the only path that exists in $Init(D_1)$ is $\langle dn \rangle$.

For each rewrite rule, let $G_1 = \langle N_1, E_1, Refs_1 \rangle$ denote the graph before the rule is applied, and let $G_2 = \langle N_2, E_2, Refs_2 \rangle$ denote the graph after the rule is applied. We assume that properties (1) and (2) of the lemma are satisfied by G_1 and prove that they must also be then satisfied by G_2. (The rules themselves are reproduced for convenience.)

First, consider rules of the *Merge* procedure. In the proof, we will call two nodes n_1 and n_2 in a graph $G = \langle N, E, Refs \rangle$ *equivalent* if either $n_1 = n_2$ or $\langle n_1, \mathbf{Id}, n_2 \rangle \in E$. Note that if there exist paths p_1 and p_2 from dn_1 to equivalent nodes in G_1, property (2) of the lemma, properties (3) and (5) of Claim 3.1, and Claim 5.1 imply that $T \models D_1 <$ (**same-as** $\gamma(p_1)$ $\gamma(p_2)$) since we can add an **Id** edge to one of the paths to "meet" the other if they do not already agree on the last node. This feature of the equivalent nodes will be referred to as the *equivalent nodes property*.

M1. If there exists an edge $\langle n_1, \mathbf{Id}, n_2 \rangle$ in E where $n_1 \neq n_2$, then merge nodes n_1 and n_2 as follows:

(a) change $\alpha(n_1)$ to $\alpha(n_1) \cup \alpha(n_2)$, and $fired(n_1)$ to $fired(n_1) \cup fired(n_2)$;

(b) modify all edges touching n_2 to touch n_1 instead;

(c) $\forall \langle r, n_2 \rangle \in Refs$, replace $\langle r, n_2 \rangle$ by $\langle r, n_1 \rangle$ in $Refs$; and

(d) remove n_2 from N.

To prove that M1 preserves property (1), consider an arbitrary path p' in G_2 that starts at dn_2 and a description D inside α set of the node at the end of p'. To show that $T \models D_1 <$ (**all** $\gamma(p')$ D), let us first find an "equivalent" path p'' in G_1 such that $T \models D_1 <$ (**same-as** $\gamma(p')$ $\gamma(p'')$), p'' starts at dn_1, and p' and p'' end at the same node. If p' does *not* go through n_1, p'' that equals to p' satisfies the conditions above as p' must also exist in G_1. Any path from dn_2 that does go through n_1 in G_2 can in general pass through n_1 more than once, and therefore, has the form

$$p' = p_0' \circ p_1' \circ p_2' \circ \ldots \circ p_m' \circ p_{m+1}'$$

where $m \geq 0$, p_i' for $1 \leq i \leq m$ are simple loops of length greater than 0 that start and end at node n_1; p_0' starts at dn_2, ends at n_1 and does not have n_1 among its other nodes; and p_{m+1}' starts at n_1 and does not have n_1 among its other nodes.

First, let us consider the case when p_0' has length greater than 0. Since p_0' does not pass through n_1 until the last node, all of the edges in p_0' must exist in G_1 except possibly for the last one. However, the edge that formed the last edge of p_0' would either point to n_1 or n_2 in G_1, and can be attached to the other edges in p_0' to form a path p in G_1 that would go through the edges with the same sequence of labels as inside p_0'. Therefore, by definition of γ,

$$\gamma(p_0') = \gamma(p). \qquad (1)$$

Since both n_1 and n_2 are equivalent to n_1, by property (2) of the assumption and by the equivalent nodes property,

$$T \models D_1 < (\textbf{same-as } \gamma(p_0'') \ \gamma(p)) \tag{2}$$

for some path p_0'' from dn_1 to n_1 in G_1 (such path p_0'' must exist by Claim 5.4). Therefore, by (1), (2), and property (1) of Lemma 5.1,

$$T \models D_1 < (\textbf{same-as } \gamma(p_0') \ \gamma(p_0'')). \tag{3}$$

In case when length of p_0' is 0, $p_0' = \langle n_1 \rangle$ and dn_1 is either n_1 or n_2. If dn_1 is n_1, then we can set p_0'' to $\langle n_1 \rangle$ and (3) remains satisfied. If $dn_1 = n_2$, then set p_0'' to some path from n_2 to n_1 (which must exist by Claim 5.4). Then, since n_1 and n_2 are equivalent, $T \models D_1 < (\textbf{same-as Id } \gamma(p_0''))$ by property (2) of the assumption and by the equivalent nodes property. Therefore, (3) still holds in this case as $\gamma(p_0') = \textbf{Id}$. Thus, in both cases of dn_1 being n_1 or n_2, we constructed p_0'' from dn_1 to n_1 in G_1 that satisfies (3).

Next, consider a simple loop $p_{loop}' = \langle n_1, A_1, n_2', \ldots, n_k', A_k, n_1 \rangle$ in G_2 with length greater than 0. Since every edge $\langle n_j', A_j, n_{j+1}' \rangle$ for $2 \le j \le k-1$ does not touch n_1, the same edge also exists in E_1. Now let $\langle n', A_1, n_2' \rangle$ be the edge in E_1 that formed the edge $\langle n_1, A_1, n_2' \rangle$ in E_2, and let $\langle n_k', A_k, n'' \rangle$ be the edge in E_1 that formed the edge $\langle n_k', A_k, n_1 \rangle$ in E_2. Note that even though n' and n'' could be different, each must be either n_1 or n_2. Thus, by constructions in rule M1, the following path p_{loop}'' must exist in G_1:

- $p_{loop}'' = \langle n_1, A_1, n_2', \ldots, n_k', A_k, n'' \rangle$ if $n' = n_1$; and
- $p_{loop}'' = \langle n_1, \textbf{Id}, n_2, A_1, n_2', \ldots, n_k', A_k, n'' \rangle$ if $n' = n_2$.

Since p_{loop}'' and p_{loop}' pass through the sequence of equivalently labeled edges (modulo an additional **Id** edge), by definition of γ, Claim 5.1 and property (3) of Claim 3.1, $\gamma(p_{loop}'') = \gamma(p_{loop}')$. Also, since p_{loop}'' starts with node n_1, it is attachable to path p_0''; and thus, by property (2) of the assumption and by the equivalent nodes property, $T \models D_1 < (\textbf{same-as } \gamma(p_0'' \circ p_{loop}'') \ \gamma(p_0''))$. Therefore, for every $1 \le i \le m$, there exists a path p_i'' in G_1 such that

$$\gamma(p_i') = \gamma(p_i''), \text{ and} \tag{4}$$

$$T \models D_1 < (\textbf{same-as } \gamma(p_0'' \circ p_i'') \ \gamma(p_0'')). \tag{5}$$

(4), (5) and Claim 5.1 imply that

$$T \models D_1 < (\textbf{same-as } \gamma(p_0'') \circ \gamma(p_i') \ \gamma(p_0'')) \text{ for } 1 \le i \le m. \tag{6}$$

Therefore, (3), (6), and Lemma 5.2 imply that

$$T \models D_1 < (\textbf{same-as } \gamma(p_0'')\circ\gamma(p_i') \ \gamma(p_0')) \text{ for } 1 \le i \le m.$$

Further, the following implications can be deduced:

$$\begin{aligned}
&T \models D_1 < (\textbf{same-as } \gamma(p_0'')\circ\gamma(p_m') \ \gamma(p_0')),\\
&T \models D_1 < (\textbf{same-as } \gamma(p_0')\circ\gamma(p_m') \ \gamma(p_0')),\\
&T \models D_1 < (\textbf{same-as } \gamma(p_0'')\circ\gamma(p_{m-1}')\circ\gamma(p_m') \ \gamma(p_0')),\\
&T \models D_1 < (\textbf{same-as } \gamma(p_0')\circ\gamma(p_{m-1}')\circ\gamma(p_m') \ \gamma(p_0')),\\
&\ldots\\
&T \models D_1 < (\textbf{same-as } \gamma(p_0')\circ\gamma(p_1')\circ\ldots\circ\gamma(p_m') \ \gamma(p_0')),\\
&T \models D_1 < (\textbf{same-as } \gamma(p_0'\circ p_1'\circ p_2'\circ\ldots\circ p_m') \ \ \gamma(p_0')), \text{ and finally,}
\end{aligned}$$

$$T \models D_1 < (\textbf{same-as } \gamma(p_0'\circ p_1'\circ p_2'\circ\ldots\circ p_m') \ \gamma(p_0'')). \qquad (7)$$

Thus, by property (4) of Lemma 5.1 and by Claim 5.1,

$$T \models D_1 < (\textbf{same-as } \gamma(p_0'\circ p_1'\circ p_2'\circ\ldots\circ p_m'\circ p_{m+1}') \ \gamma(p_0'')\circ\gamma(p_{m+1}')). \qquad (8)$$

Now consider p_{m+1}'. First, assume that the length of p_{m+1}' is greater than 0. Then, since $p_{m+1}' = \langle n_1, A, n_2'\rangle \circ p_{tail}'$ (for some node n_2') does not go through node n_1 in G_2 after the first node, p_{tail}' must exist in G_1. Let $\langle n', A, n_2'\rangle$ be the edge in E_1 that originated the edge $\langle n_1, A, n_2'\rangle$ in E_2. Note that n' must either be n_1 or n_2 by constructions in M1. Therefore, the following path exists in G_1:

- $p_{m+1}'' = \langle n_1, A, n_2'\rangle \circ p_{tail}'$ if $n' = n_1$; and
- $p_{m+1}'' = \langle n_1, \textbf{Id}, n_2, A, n_2'\rangle \circ p_{tail}'$ if $n' = n_2$.

Since p_{m+1}'' and p_{m+1}' pass through the sequence of equivalently labeled edges (modulo an additional **Id** edge), by definition of γ, Claim 5.1 and property (3) of Claim 3.1,

$$\gamma(p_{m+1}'') = \gamma(p_{m+1}'). \qquad (9)$$

Also, note that p_{m+1}'' and p_{m+1}' end at the same node. In case when length of p_{m+1}' is 0, let $p_{m+1}'' = p_{m+1}' = \langle n_1\rangle$. Clearly, (9) holds in this case as well and p_{m+1}' and p_{m+1}'' end at the same node again.

Observe now that by "construction" of p_{m+1}'' from p_{m+1}', they both start at n_1 and end at the same node. Therefore, by property (1) of the assumption,

$$T \models D_1 < (\textbf{all}\ \gamma(p_0'' \circ p_{m+1}'')\ D) \qquad (10)$$

if p_{m+1}' and p_{m+1}'' end at any node other than n_1, since α sets of nodes do not change except for node n_1. However, in the case when p_{m+1}' and p_{m+1}'' end at n_1, (10) is still satisfied, since either $D \in \alpha(n_1, G_1)$ in which case (10) holds by assumption, or $D \in \alpha(n_2, G_1)$ in which case (10) holds due to property (3) of Claim 3.1, Claim 5.1, and the fact that by assumption,

$$T \models D_1 < (\textbf{all}\ \gamma(p_0'' \circ p_{m+1}'' \circ \langle n_1, \textbf{Id}, n_2 \rangle)\ D).$$

Let p'' now denote the path $p_0'' \circ p_{m+1}''$. Then, (8), (9) and Claim 5.1 imply that

$$T \models D_1 < (\textbf{same-as}\ \gamma(p')\ \gamma(p'')). \qquad (11)$$

Finally, (10), (11) and properties (1) and (3) of Lemma 5.1 imply that $T \models D_1 < (\textbf{all}\ \gamma(p')\ D)$. Therefore, M1 preserves property (1) of the lemma.

Let us now consider property (2). If a path in G_2 does not go through n_1, exactly the same path exists in G_1. On the other hand, if we have a path in G_2 from dn_2 to n_m that passes through node n_1, we can find an "equivalent" path from dn_1 to n_m in G_1 in the way presented in the proof of property (1) above. Thus, for both paths p_1 and p_2 (from the statement of property (2)) in G_2, there must exist paths p_1' and p_2' in G_1 that start at dn_1 and such that

$$T \models D_1 < (\textbf{same-as}\ \gamma(p_1)\ \gamma(p_1')) \text{ and } T \models D_1 < (\textbf{same-as}\ \gamma(p_2)\ \gamma(p_2')). \qquad (12)$$

Since p_1 and p_1' end at the same node, p_2 and p_2' end at the same node, and p_1 and p_2 end at the same node, p_1' and p_2' must also end at the same node. Therefore, by property (2) of the assumption,

$$T \models D_1 < (\textbf{same-as}\ \gamma(p_1')\ \gamma(p_2')). \qquad (13)$$

Then, however, by (12), (13) and Lemma 5.2, $T \models D_1 < (\textbf{same-as}\ \gamma(p_1)\ \gamma(p_2))$. Therefore, M1 preserves property (2) of the lemma as well.

M2. If there exists an edge $\langle n, \mathbf{Id}, n\rangle$ in E then remove it.

This case trivially satisfies properties (1) and (2) since except for removal of some paths in the graph, nothing else changes.

M3. If there exist edges $\langle n_1, A, n_2\rangle$ and $\langle n_1, A, n_3\rangle$ in E for a primitive attribute A, then: if $n_2 = n_3$, then remove the second edge from E; otherwise, add edge $\langle n_2, \mathbf{Id}, n_3\rangle$ to E if such an edge does not exist already.

In this case, either some paths from dn to a node are removed if $n_2 = n_3$, or an **Id** edge is added otherwise. In the first case, properties (1) and (2) are trivially preserved since (just as in M2) except for removal of some paths, nothing else changes. Therefore, it is sufficient to consider the case when $n_2 \neq n_3$ and edge $\langle n_2, \mathbf{Id}, n_3\rangle$ is added.

To prove that M3 preserves property (1), consider an arbitrary path p' in G_2 that starts at dn_2 and a description D inside α set of the node at the end of p'. Our goal is then to prove that $T \models D_1 < (\mathbf{all}\ \gamma(p')\ D)$. If p' does *not* pass though $\langle n_2, \mathbf{Id}, n_3\rangle$ edge in G_2, exactly the same path exists in G_1, and therefore property (1) remains satisfied. On the other hand, if p' does go through edge $\langle n_2, \mathbf{Id}, n_3\rangle$ in G_2, it can in general pass through the edge more than once, and therefore, has the form

$$p' = p_0' \circ p_1' \circ p_2' \circ \ldots \circ p_m' \circ p_{m+1}'$$

where $m \geq 0$, p_i' for $1 \leq i \leq m$ are loops of length greater than 0 that start with edge $\langle n_2, \mathbf{Id}, n_3\rangle$ and do not pass though $\langle n_2, \mathbf{Id}, n_3\rangle$ afterwards, p_0' starts at dn_2, ends at n_2 and does not pass through $\langle n_2, \mathbf{Id}, n_3\rangle$, and p_{m+1}' starts with edge $\langle n_2, \mathbf{Id}, n_3\rangle$ and does not pass though $\langle n_2, \mathbf{Id}, n_3\rangle$ afterwards.

Since p_0' does not pass through $\langle n_2, \mathbf{Id}, n_3\rangle$, p_0' must also exist in G_1. Let p_0'' be some path from dn_1 to n_1 in G_1 (such path must exist by Claim 5.4). Then, by property (2) of the assumption,

$$T \models D_1 < (\mathbf{same\text{-}as}\ \gamma(p_0')\ \ \gamma(p_0'' \circ \langle n_1, A, n_2\rangle)). \tag{14}$$

By definition of γ and Claim 5.1,

$$T \models D_1 < (\mathbf{same\text{-}as}\ \gamma(p_0'' \circ \langle n_1, A, n_2\rangle)\ \ \gamma(p_0'' \circ \langle n_1, A, n_3\rangle)). \tag{15}$$

Next, consider a loop $p_{loop}' = \langle n_2, \mathbf{Id}, n_3, A_1, n_1', \ldots, n_{k-1}', A_k, n_2\rangle$ in G_2. Since every edge after the first one is not $\langle n_2, \mathbf{Id}, n_3\rangle$, the same edge exists in E_1. Therefore, path $p_{loop}'' = \langle n_3, A_1, n_1', \ldots, n_{k-1}', A_k, n_2\rangle$ exists in G_1. By definition of γ, Claim 5.1 and property (3) of Claim 3.1, $\gamma(p_{loop}') = \gamma(p_{loop}'')$.

Also, since p_{loop}'' starts at node n_3, it is attachable to path $p_0''\circ\langle n_1, A, n_3\rangle$; and thus, by property (2) of the assumption, $T \models D_1 <$ (**same-as** $\gamma(p_0''\circ\langle n_1, A, n_3\rangle\circ p_{loop}'')$ $\gamma(p_0''\circ\langle n_1, A, n_2\rangle)$). Therefore, for every $1 \le i \le m$, there exists a path p_i'' in G_1 such that

$$\gamma(p_i') = \gamma(p_i''), \text{ and} \tag{16}$$

$$T \models D_1 < (\textbf{same-as } \gamma(p_0''\circ\langle n_1, A, n_3\rangle\circ p_i'')\ \ \gamma(p_0''\circ\langle n_1, A, n_2\rangle)). \tag{17}$$

Then, (15), (16), (17), Lemma 5.2, and Claim 5.1 imply that

$$T \models D_1 < (\textbf{same-as } \gamma(p_0''\circ\langle n_1, A, n_2\rangle)\circ\gamma(p_i')\ \gamma(p_0''\circ\langle n_1, A, n_2\rangle)) \text{ for } 1 \le i \le m. \tag{18}$$

Therefore, the following implications can be deduced from (14), (18), Lemma 5.2, and Claim 5.1:

$$T \models D_1 < (\textbf{same-as } \gamma(p_0''\circ\langle n_1, A, n_2\rangle)\circ\gamma(p_m')\ \gamma(p_0''\circ\langle n_1, A, n_2\rangle)),$$
$$T \models D_1 < (\textbf{same-as } \gamma(p_0''\circ\langle n_1, A, n_2\rangle)\circ\gamma(p_{m-1}')\circ\gamma(p_m')\ \gamma(p_0''\circ\langle n_1, A, n_2\rangle)),$$
$$\ldots$$
$$T \models D_1 < (\textbf{same-as } \gamma(p_0''\circ\langle n_1, A, n_2\rangle)\circ\gamma(p_1')\circ\ldots\circ\gamma(p_m')\ \gamma(p_0''\circ\langle n_1, A, n_2\rangle)),$$
$$T \models D_1 < (\textbf{same-as } \gamma(p_0')\circ\gamma(p_1')\circ\ldots\circ\gamma(p_m')\ \gamma(p_0''\circ\langle n_1, A, n_2\rangle)), \text{ and finally,}$$

$$T \models D_1 < (\textbf{same-as } \gamma(p_0'\circ p_1'\circ p_2'\circ\ldots\circ p_m')\ \gamma(p_0''\circ\langle n_1, A, n_2\rangle)). \tag{19}$$

Thus, by property (4) of Lemma 5.1 and by Claim 5.1,

$$T \models D_1 < (\textbf{same-as } \gamma(p_0'\circ p_1'\circ p_2'\circ\ldots\circ p_m'\circ p_{m+1}')\ \ \gamma(p_0''\circ\langle n_1, A, n_2\rangle)\circ\gamma(p_{m+1}')). \tag{20}$$

Now consider p_{m+1}'. Recall that it has to have the form $\langle n_2, \mathbf{Id}, n_3\rangle \circ p_{tail}$, where no edges in p_{tail} are $\langle n_2, \mathbf{Id}, n_3\rangle$. Therefore, path $p_{m+1}'' = p_{tail}$ must exist in G_1. Note that by construction of p_{m+1}'', by definition of γ, by property (3) of Claim 3.1 and by Claim 5.1,

$$\gamma(p_{m+1}'') = \gamma(p_{m+1}'). \tag{21}$$

Also, note that p_{m+1}'' and p_{m+1}' end at the same node. Observe now that by construction of p_{m+1}'', it starts at n_3. Therefore, by property (1) of the assumption,

$$T \models D_1 < (\textbf{all}\ \gamma(p_0'' \circ \langle n_1, A, n_3\rangle \circ p_{m+1}'')\ D), \tag{22}$$

since α sets of nodes do not change. Let p'' now denote the path $p_0'' \circ \langle n_1, A, n_3\rangle \circ p_{m+1}''$. By (21), definition of γ and Claim 5.1, $\gamma(p'') = \gamma(p_0'' \circ \langle n_1, A, n_2\rangle) \circ \gamma(p_{m+1}')$. Then, considering (20), it follows that

$$T \models D_1 < (\textbf{same-as}\ \gamma(p')\ \gamma(p'')). \tag{23}$$

Finally, (22), (23) and properties (1) and (3) of Lemma 5.1 imply that $T \models D_1 < (\textbf{all}\ \gamma(p')\ D)$. Therefore, M3 preserves property (1) of the lemma.

Let us now consider property (2). If a path in G_2 does not go through edge $\langle n_2, \textbf{Id}, n_3\rangle$, exactly the same path exists in G_1. On the other hand, if we have a path in G_2 from dn_2 to n_m that passes through edge $\langle n_2, \textbf{Id}, n_3\rangle$, we can find an "equivalent" path from dn_1 to n_m in G_1 in the way presented in the proof of property (1) above. Thus, for both paths p_1 and p_2 (from the statement of property (2)) in G_2, there must exist paths p_1' and p_2' in G_1 that start at dn_1 and such that

$$T \models D_1 < (\textbf{same-as}\ \gamma(p_1)\ \gamma(p_1')) \text{ and } T \models D_1 < (\textbf{same-as}\ \gamma(p_2)\ \gamma(p_2')). \tag{24}$$

Since p_1 and p_1' end at the same node; p_2 and p_2' end at the same node; and p_1 and p_2 end at the same node; p_1' and p_2' must also end at the same node. Therefore, by property (2) of the assumption,

$$T \models D_1 < (\textbf{same-as}\ \gamma(p_1')\ \gamma(p_2')). \tag{25}$$

Then, however, by (24), (25) and Lemma 5.2, $T \models D_1 < (\textbf{same-as}\ \gamma(p_1)\ \gamma(p_2))$. Therefore, M3 preserves property (2) of the lemma as well.

Note that in the rest of the rules that we consider, distinguished node is the same in G_1 and G_2; and therefore, we denote dn_1 and dn_2 just by dn.

E1. If there exists a node n with $\alpha(n) = \{D_1, \ldots, D_k, (\textbf{and}\ D_{k+1} \ldots D_m)\}$ in N, then change $\alpha(n)$ to $\{D_1, \ldots, D_k, D_{k+1}, \ldots, D_m\}$.

This rule preserves property (2) since no paths are changed by E1. Property (1) is satisfied for all nodes except possibly n, since neither paths to those nodes nor their α sets are changed. By assumption, for any path p from dn to n (p exists in G_1 if and only if it exists in G_2), $T \models D_1 < (\textbf{all}\ \gamma(p)\ (\textbf{and}\ D_{k+1} \ldots D_m))$. Therefore, for any interpretation **I** that satisfies T and for

any object $x \in D_1^{\mathbf{I}}$, $\gamma(p)^{\mathbf{I}}(x) \in D_i^{\mathbf{I}}$ for $k+1 \leq i \leq m$. Also, by assumption, $T \models D_1 < (\textbf{all}\ \gamma(p)\ D_j)$ for $1 \leq j \leq k$. Thus, $T \models D_1 < (\textbf{all}\ \gamma(p)\ D_i)$ for $1 \leq i \leq m$.

E2. If there exist nodes n_1 with $\alpha(n_1) = \{D_1, \ldots, D_k, (\textbf{all}\ A\ D)\}$ and n_2 in N, and an edge $\langle n_1, A, n_2\rangle$ in E then:

(a) remove (**all** A D) from $\alpha(n_1)$; and

(b) add D to $\alpha(n_2)$.

This rule also preserves property (2) since no paths are changed. Property (1) is satisfied for all nodes except possibly n_2, since all paths remain the same, and α set of any of those nodes does *not* obtain any new description. Node n_2, on the other hand, has a new description D in its α set. Consider a path p from dn to n_1 in G_2 (which must exist by Claim 5.4). Since G_1 has the same paths, p must also exist in G_1. Then, by assumption,

$$T \models D_1 < (\textbf{all}\ \gamma(p)\ (\textbf{all}\ A\ D)).$$

Therefore, for any interpretation **I** that satisfies T and for any object $x \in D_1^{\mathbf{I}}$, $A^{\mathbf{I}}(\gamma(p)^{\mathbf{I}}(x)) \in D^{\mathbf{I}}$. However, since p ends at n_1, p is attachable to $\langle n_1, A, n_2\rangle$ (in both G_1 and G_2). Thus, by property (4) of Claim 3.1 and by Claim 5.1, $\gamma(p \circ \langle n_1, A, n_2\rangle)^{\mathbf{I}}(x) \in D^{\mathbf{I}}$.

Finally, since any path p' from dn to n_2 in G_2 agrees with path $p \circ \langle n_1, A, n_2\rangle$ on the last node (n_2), by property (2) of the assumption, $\gamma(p')^{\mathbf{I}}(x) \in D^{\mathbf{I}}$. Therefore, $T \models D_1 < (\textbf{all}\ \gamma(p')\ D)$. Property (1) remains satisfied for all other descriptions in n_2 since they are also present in $\alpha(n_2, G_1)$ and no paths are changed by E2.

E3. If there exists a node n_1 with $\alpha(n_1) = \{D_1, \ldots, D_k, (\textbf{all}\ A\ D)\}$, there is *no* edge $\langle n_1, A, n_2\rangle$ in E for any node n_2 in N, and $D_{[u]}$ contains a **same-as** constructor anywhere inside it then:

(a) add a new node n_2 to N and a new edge $\langle n_1, A, n_2\rangle$ to E;

(b) remove (**all** A D) from $\alpha(n_1)$; and

(c) add D to $\alpha(n_2)$.

First, consider property (2). Since $N_2 = N_1 \cup \{n_2\}$ and $E_2 = E_1 \cup \{\langle n_1, A, n_2\rangle\}$, no new paths are added that would end at a node in N_1, and thus, property (2) remains satisfied for any pair of paths that end at any node in N_1. Consider now any paths p_1 and p_2 from dn to n_2 in G_2. Since n_2 is only directly reachable by the added edge $\langle n_1, A, n_2\rangle$, there must exist paths p_1' and p_2' (in both G_1 and G_2) such that $p_1 = p_1' \circ \langle n_1, A, n_2\rangle$ and $p_2 = p_2' \circ \langle n_1, A, n_2\rangle$. Since paths p_1' and p_2' are from dn to node n_1, by property (2) of the assumption, $T \models D_1 < (\textbf{same-as}\ \gamma(p_1')\ \gamma(p_2'))$. Therefore, by definition of γ, Claim 5.1 and property (4) of Lemma 5.1, $T \models D_1 < (\textbf{same-as}\ \gamma(p_1)\ \gamma(p_2))$.

Now consider property (1). It is satisfied for all nodes except possibly n_2, since all paths to those nodes remain the same, and α set of any of those nodes does not obtain any new description. Node n_2, on the other hand, is new and only has description D in its α set. Consider a path p from dn to n_1 in G_1 and G_2. Then, by assumption,

$$T \models D_1 < (\textbf{all } \gamma(p) \ (\textbf{all } A \ D)).$$

Therefore, for any interpretation $\mathbf{I}$ that satisfies T and for any object $x \in D_1^{\mathbf{I}}$, $A^{\mathbf{I}}(\gamma(p)^{\mathbf{I}}(x)) \in D^{\mathbf{I}}$. However, since p ends at n_1, p is attachable to $\langle n_1, A, n_2 \rangle$ in G_2. Thus, by property (4) of Claim 3.1 and Claim 5.1, $\gamma(p \circ \langle n_1, A, n_2 \rangle)^{\mathbf{I}}(x) \in D^{\mathbf{I}}$.

Finally, since any path p' from dn to n_2 in G_2 agrees with path $p \circ \langle n_1, A, n_2 \rangle$ on the last node (n_2), by the fact that property (2) is preserved, $T \models D_1 < (\textbf{same-as } \gamma(p \circ \langle n_1, A, n_2 \rangle) \ \gamma(p'))$, and thus, $\gamma(p')^{\mathbf{I}}(x) \in D^{\mathbf{I}}$. Therefore, $T \models D_1 < (\textbf{all } \gamma(p') \ D)$.

E4. If there exists a node n with $\alpha(n) = \{D_1, \ldots, D_k, (\textbf{same-as } Pf_1 \ Pf_2)\}$ in N then:

(a) remove $(\textbf{same-as } Pf_1 \ Pf_2)$ from $\alpha(n)$;

(b) add a new node n' to N;

(c) execute $Create(n, Pf_1, n')$ and $Create(n, Pf_2, n')$; and

(d) invoke $Merge(G)$.

Note that *Merge*, and thus step (d) of rule E4, preserves properties (1) and (2) of the lemma as was proven above. Therefore, it is sufficient to show that properties (1) and (2) are preserved just for steps (a), (b) and (c), i.e. G_2 in this case is the graph obtained after step (c) is executed and before step (d) invokes *Merge*.

Property (1) remains satisfied since for any node n_1 in N_1 all paths that end at n_1 remain the same, $\alpha(n_1, G_2) \subseteq \alpha(n_1, G_1)$, and every added node has empty α set.

Since no edge that would end at a node in N_1 is added, all paths that end at a node in N_1 continue to satisfy property (2). Further, without loss of generality, let us assume that

- Pf_1 has the form $(\textbf{comp } A_1 \ (\textbf{comp } A_2 \ \ldots \ (\textbf{comp } A_k \ \textbf{Id}) \ \ldots))$, $k \geq 0$;
- Pf_2 has the form $(\textbf{comp } B_1 \ (\textbf{comp } B_2 \ \ldots \ (\textbf{comp } B_m \ \textbf{Id}) \ \ldots))$, $m \geq 0$;
- if $k \geq 2$, $Create(n, Pf_1, n')$ adds $k - 1$ new nodes $n_1', \ldots, n_{k-1}'$ and k new edges $\langle n, A_1, n_1' \rangle$, $\langle n_1', A_2, n_2' \rangle, \ldots, \langle n_{k-2}', A_{k-1}, n_{k-1}' \rangle, \langle n_{k-1}', A_k, n' \rangle$.
- if $m \geq 2$, $Create(n, Pf_2, n')$ adds $m - 1$ new nodes $n_1'', \ldots, n_{m-1}''$ and m new edges $\langle n, B_1, n_1'' \rangle$, $\langle n_1'', B_2, n_2'' \rangle, \ldots, \langle n_{m-2}'', B_{m-1}, n_{m-1}'' \rangle, \langle n_{m-1}'', B_m, n' \rangle$.

Now, consider two paths p_1 and p_2 in G_2 that go from dn to n_i' for some $1 \le i \le k-1$. By construction in procedure *Create* and step E4, n_j' is only directly reachable from n_{j-1}' (by edge $\langle n_{j-1}', A_j, n_j' \rangle$) for every $2 \le j \le k-1$; n_1' is only directly reachable from n by $\langle n, A_1, n_1' \rangle$; and for any node n_l' ($1 \le l \le k-1$), only nodes n_j' ($l \le j \le k-1$) and n' are reachable from n_l' (recall that n' is a new node that does not have any outgoing arcs). Thus, both paths p_1 and p_2 must go through edges $\langle n, A_1, n_1' \rangle$, $\langle n_1', A_2, n_2' \rangle$, ..., $\langle n_{i-1}', A_i, n_i' \rangle$, and the paths pass through this sequence of edges only once (in the end), since if the edge labeled A_{i+1} is followed, the path cannot come back to node n_i'. In addition, the paths cannot use any edges added by *Create*(n, Pf_2, n') since if any of those edges is in a path, the path cannot reach n_i' afterwards. Thus, only nodes in N_1 can be used in the paths p_1 and p_2 before they reach n_1'. Therefore, there must exist paths p_1' and p_2' in G_1 (and G_2) such that $p_1 = p_1' \circ p_{tail}$ and $p_2 = p_2' \circ p_{tail}$, where $p_{tail} = \langle n, A_1, n_1', \ldots, A_i, n_i' \rangle$. Note that p_1' and p_2' must end at the same node (n) and thus, by property (2) of the assumption, $T \models D_1 <$ (**same-as** $\gamma(p_1')$ $\gamma(p_2')$). Therefore, by property (4) of Lemma 5.1 and by Claim 5.1, $T \models D_1 <$ (**same-as** $\gamma(p_1)$ $\gamma(p_2)$).

The proof that property (2) is also satisfied for any two paths from dn to n_i'' for some $1 \le i \le m-1$ is completely analogous.

Finally, consider two paths p_1 and p_2 that go from dn to n' in G_2. If they both use the edge $\langle n, A_1, n_1' \rangle$ (or $\langle n, B_1, n_1'' \rangle$), the proof that the property (2) is satisfied for the two paths is completely analogous to the proof presented above for the paths ending at a node n_i'. (In this case, the fact that n' cannot be somewhere in the middle of a path follows right away from the fact that there is no outgoing edges from node n'.) Thus, it is sufficient to consider the case when $p_1 = p_1' \circ p_{tail}'$ and $p_2 = p_2' \circ p_{tail}''$, where $p_{tail}' = \langle n, A_1, n_1', \ldots, A_k, n' \rangle$, and $p_{tail}'' = \langle n, B_1, n_1'', \ldots, B_m, n' \rangle$. The proof that such paths p_1' and p_2' exist in both G_1 and G_2 is again analogous to the proof of their existence for paths ending at n_i'. Note that by construction, and from definition of γ,

$$\gamma(p_{tail}') = Pf_1 \text{ and } \gamma(p_{tail}'') = Pf_2. \tag{26}$$

By properties (1) and (2) of the assumption,

$$T \models D_1 < (\textbf{same-as } \gamma(p_1') \ \gamma(p_2')), \text{ and} \tag{27}$$

$$T \models D_1 < (\textbf{all } \gamma(p_1') \ (\textbf{same-as } Pf_1 \ Pf_2)). \tag{28}$$

(27) and (28) imply that for any interpretation **I** that satisfies T and for any object $x \in D_1^{\mathbf{I}}$, $\gamma(p_1')^{\mathbf{I}}(x) \in (\textbf{same-as } Pf_1 \ Pf_2)^{\mathbf{I}}$, or $Pf_1^{\mathbf{I}}(\gamma(p_1')^{\mathbf{I}}(x)) = Pf_2^{\mathbf{I}}(\gamma(p_1')^{\mathbf{I}}(x))$. Therefore, by (26), Claim 5.1 and property (4) of Claim 3.1, $\gamma(p_1' \circ p_{tail}')^{\mathbf{I}}(x) = \gamma(p_2' \circ p_{tail}'')^{\mathbf{I}}(x)$, or $\gamma(p_1)^{\mathbf{I}}(x) = \gamma(p_2)^{\mathbf{I}}(x)$. Therefore, $T \models D_1 <$ (**same-as** $\gamma(p_1)$ $\gamma(p_2)$).

Finally, note that the proof for cases when k and/or m are either 0 or 1 is essentially the same, except that the part that discusses nodes n_i' and/or n_i'' should be skipped since *Create* just adds an edge and no new nodes.

E5. If there exist nodes n_1 with $\alpha(n_1) = \{D_1, \ldots, D_k, (\textbf{fd } C\ Pf_1 \ldots Pf_m\ Pf)\}$ and n_2 (distinct from n_1) in N, such that

(1) (**fd** $C\ Pf_1 \ldots Pf_m\ Pf$) is regular;

(2) $C \in \alpha(n_2)$;

(3) for every Pf_i, $1 \leq i \leq m$, there is a prefix Pf_i' of Pf_i and a node n_i' in N such that *Reachable*(n_1, Pf_i', n_i') and *Reachable*(n_2, Pf_i', n_i') are true; and

(4) there is no prefix Pf' of Pf and a node n in N such that *Reachable*(n_1, Pf', n) and *Reachable*(n_2, Pf', n) are true;

then

(a) add a new node n to N;

(b) execute *Create*(n_1, Pf, n) and *Create*(n_2, Pf, n); and

(c) invoke *Merge*(G).

Note that *Merge*, and thus step (c) of rule E5, preserves properties (1) and (2) of the lemma as was proven above. Therefore, it is sufficient to show that properties (1) and (2) are preserved just for steps (a) and (b), i.e. G_2 in this case is the graph obtained after step (b) is executed and before step (c) invokes *Merge*.

Property (1) remains satisfied since for any node n' in N_1 all paths that end at n' remain the same, $\alpha(n', G_1) = \alpha(n', G_2)$, and every added node has empty α set.

Since no edge that would end at a node in N_1 is added, all paths that end at a node in N_1 continue to satisfy property (2). Further, without loss of generality, let us assume that

- Pf has the form (**comp** A_1 (**comp** $A_2 \ldots$ (**comp** A_k **Id**) ...)), $k \geq 0$;
- if $k \geq 2$, *Create*(n_1, Pf, n) adds $k - 1$ new nodes $n_1', \ldots, n_{k-1}'$ and k new edges $\langle n_1, A_1, n_1' \rangle$, $\langle n_1', A_2, n_2' \rangle, \ldots, \langle n_{k-2}', A_{k-1}, n_{k-1}' \rangle, \langle n_{k-1}', A_k, n \rangle$.
- if $k \geq 2$, *Create*(n_2, Pf, n) adds $k - 1$ new nodes $n_1'', \ldots, n_{k-1}''$ and k new edges $\langle n_2, A_1, n_1'' \rangle$, $\langle n_1'', A_2, n_2'' \rangle, \ldots, \langle n_{k-2}'', A_{k-1}, n_{k-1}'' \rangle, \langle n_{k-1}'', A_k, n \rangle$.

The proof that property (2) is satisfied for any two paths from dn to n_i' (resp. from dn to n_i'') for some $1 \leq i \leq k - 1$ is completely analogous to the part of the proof for rule E4 that discusses paths from dn to n_i' (resp. from dn to n_i'').

Finally, consider two paths p_1 and p_2 that go from dn to n in G_2. If they both use edge $\langle n_1, A_1, n_1' \rangle$ (resp. $\langle n_2, A_1, n_1'' \rangle$), the proof that property (2) is satisfied for the two paths is completely analogous to the part of the proof for rule E4 that discusses paths from dn to n' that go

through the edge $\langle n, A_1, n_1' \rangle$ (resp. $\langle n, B_1, n_1'' \rangle$). Thus, it is sufficient to consider the case when path p_1 goes though edge $\langle n_1, A_1, n_1' \rangle$ and path p_2 goes through edge $\langle n_2, A_1, n_1'' \rangle$. By Claim 5.4 and by the construction, there must exist paths p_1' (from dn to n_1) and p_2' (from dn to n_2) in G_1 and G_2 such that $p_1 = p_1' \circ p_{tail}'$ and $p_2 = p_2' \circ p_{tail}''$, where $p_{tail}' = \langle n_1, A_1, n_1', \ldots, n_{k-1}', A_k, n \rangle$ and $p_{tail}'' = \langle n_2, A_1, n_1'', \ldots, n_{k-1}'', A_k, n \rangle$. (The proof that such paths p_1' and p_2' exist in both G_1 and G_2 is again completely analogous to the proof of existence of paths p_1' and p_2' in the proof for rule E4.) Note that by construction and from definition of γ,

$$\gamma(p_{tail}') = \gamma(p_{tail}'') = Pf. \tag{29}$$

By properties (1) and (2) of the assumption,

$$T \models D_1 < (\textbf{all}\ \gamma(p_1')\ (\textbf{fd}\ C\ Pf_1 \ldots Pf_m\ Pf)), \text{ and} \tag{30}$$
$$T \models D_1 < (\textbf{all}\ \gamma(p_2')\ C). \tag{31}$$

By condition (3) of E5, for every Pf_i, $1 \le i \le m$, there are path functions Pf_i' and Pf_i'' and a node n_i in N_1 such that

$$Pf_i' \circ Pf_i'' = Pf_i, \text{ and} \tag{32}$$

Reachable(n_1, Pf_i', n_i) and *Reachable*(n_2, Pf_i', n_i) are true. Thus, by definition of *Reachable*, for every $1 \le i \le m$, there must exist paths p' from n_1 to n_i and p'' from n_2 to n_i in G_1 such that their edge labels "correspond" to Pf_i'. Therefore, by definition of γ, $\gamma(p') = \gamma(p'') = Pf_i'$. However, since paths $p_1' \circ p'$ and $p_2' \circ p''$ go from dn to n_i in G_1, by property (2) of the assumption,

$$T \models D_1 < (\textbf{same-as}\ \gamma(p_1' \circ p')\ \gamma(p_2' \circ p'')),$$

which in turn implies $T \models D_1 < (\textbf{same-as}\ \gamma(p_1') \circ \gamma(p') \circ Pf_i''\ \gamma(p_2') \circ \gamma(p'') \circ Pf_i'')$ by Claim 5.1 and property (4) of Lemma 5.1. Therefore, considering (32),

$$T \models D_1 < (\textbf{same-as}\ \gamma(p_1') \circ Pf_i\ \gamma(p_2') \circ Pf_i), \text{ for every } 1 \le i \le m. \tag{33}$$

Therefore, from (30), (31), (33) and property (5) of Lemma 5.1,

$$T \models D_1 < (\textbf{same-as}\ \gamma(p_1') \circ Pf\ \gamma(p_2') \circ Pf). \tag{34}$$

Thus, by (34), (29), and Claim 5.1, $T \models D_1 < (\textbf{same-as}\ \gamma(p_1)\ \gamma(p_2))$.

Finally, note that the proof for cases when k is either 0 or 1 is the same, except that the part that discusses nodes n_i' and/or n_i'' should be skipped since *Create* just adds an edge and no new nodes.

E6. If there exists a node n in N, and a constraint $C < D$ in T such that $C \in \alpha(n)$ and $C \notin \mathit{fired}(n)$, then add D to $\alpha(n)$ and C to $\mathit{fired}(n)$.

By the same reasoning as presented in the proof of the case of rule E1, property (2) remains satisfied for all paths, and property (1) is satisfied by all paths, nodes and their descriptions, except possibly for description D in $\alpha(n, G_2)$. Given a path p from dn to n in G_1 and G_2, by property (1) of the assumption, $T \models D_1 <$ (**all** $\gamma(p)$ C). Therefore, for any interpretation $\mathbf{I}$ that satisfies T and for any object $x \in D_1^{\mathbf{I}}$, $\gamma(p)^{\mathbf{I}}(x) \in C^{\mathbf{I}}$, but since $C < D \in T$, $\gamma(p)^{\mathbf{I}}(x) \in D^{\mathbf{I}}$. Thus, $T \models D_1 <$ (**all** $\gamma(p)$ D).

E7. If there exists a node n in N, and a view definition $V \equiv D$ in T such that $V \in \alpha(n)$ and $V \notin \mathit{fired}(n)$, then add D to $\alpha(n)$ and V to $\mathit{fired}(n)$.

The proof is analogous to the proof of the case of rule E6.

A1. Given a description graph $G = \langle N, E, \mathit{Refs} \rangle$, a node $n \in N$, and an attribute description A which is either a primitive attribute or **Id**,

(a) add a new node n' to N; and

(b) add a new edge $\langle n, A, n' \rangle$ to E.

First, observe that $N_2 = N_1 \cup \{n'\}$ and $E_2 = E_1 \cup \{\langle n, A, n' \rangle\}$. Property (1) is satisfied in this case for any node n'' in N_1 since no new edge is added to E_1 that ends at n'', and since $\alpha(n'', G_2) = \alpha(n'', G_1)$. On the other hand, property (1) is also satisfied for any path that ends at n' since $\alpha(n', G_2) = \varnothing$.

Again, because no new paths are added that would end at a node in N_1, property (2) remains satisfied for any pair of paths that end at any node in N_1. Consider now any two paths p_1 and p_2 from dn to n' in G_2. Since a path can get to n' only by the added edge $\langle n, A, n' \rangle$, there must exist paths p_1' and p_2' (in both G_1 and G_2) such that $p_1 = p_1' \circ \langle n, A, n' \rangle$ and $p_2 = p_2' \circ \langle n, A, n' \rangle$. Since paths p_1' and p_2' are from dn to node n, by property (2) of the assumption, $T \models D_1 <$ (**same-as** $\gamma(p_1')$ $\gamma(p_2')$). Therefore, by Claim 5.1, and property (4) of Lemma 5.1, $T \models D_1 <$ (**same-as** $\gamma(p_1)$ $\gamma(p_2)$).

■

Next, we verify that our solution is sound when D_2 does not contain any **and** constructors or view names; that is, when D_2 has one of the forms

(a) (**all** A_1 (**all** A_2 ... (**all** A_k C) ...)),
(b) (**all** A_1 (**all** A_2 ... (**all** A_k (**same-as** Pf_1 Pf_2)) ...)), or
(c) (**all** A_1 (**all** A_2 ... (**all** A_k (**fd** C Pf_1 ... Pf_m Pf)) ...)).

First, we use Lemma 5.4 to establish soundness for the first two cases above in Lemma 5.5.

Lemma 5.5 *Let $G = \langle N, E, Refs \rangle$ denote a graph obtained by a number of applications of rewrite rules from the set R = {M1, M2, M3, E1, E2, E3, E4, E5, E6, E7, A1} to graph $G_0 = Init(D_1)$ for some description D_1 and a terminology T, and let D_2 be a description*

(**all** A_1 (**all** A_2 ... (**all** A_k C) ...)) *or* (**all** A_1 (**all** A_2 ... (**all** A_k (**same-as** Pf_1 Pf_2)) ...))

for some $k \geq 0$, where C is a primitive concept, A_i is a primitive attribute for all $1 \leq i \leq k$, and Pf_1 and Pf_2 are path functions. Then, if Subsumes(D_2, G, T) returns true, $D_1 < D_2$ is a logical consequence of T.

Proof. Intuitively, we note that *Subsumes* constructs a chain of new nodes $n_1, n_2, \ldots, n_k$ that form a path $\langle dn, A_1, n_1, A_2, n_2, \ldots, A_k, n_k \rangle$, where dn is the initial distinguished node. This construction as well as constructions in rule S4 are specific cases of applications of rewrite rule A1. The only construction that does not fit into the set of rules R = {M1, M2, M3, E1, E2, E3, E4, E5, E6, E7, A1} is the modification of set *Refs* in S3. However, it does not influence applicability of rules or the results of rule applications by part (a) of Claim 5.4. Therefore, the proof follows from Lemma 5.4. Since *Merge* might remove some of the new nodes, we use references instead of nodes in the formal proof. Part (c) of Claim 5.4 allows us to establish that paths between references (except for reference dn that gets modified by S3) do not change even though nodes that are referenced could be merged. A more formal proof is as follows.

By definitions of procedure *Subsumes* and rule A1, given a description D of the form (**all** A D'), a call to procedure *Subsumes*(D, G, T) essentially applies rule A1, creates a new reference dn', modifies reference dn in the *Refs* set, invokes procedures *Merge* and *Exp*, and makes the recursive call. Therefore, since procedure *Subsumes*(D, G, T) does *not* make recursive calls when D is a primitive concept C or description (**same-as** Pf_1 Pf_2), *Subsumes*(D_2, G, T) executes step S3 exactly k times and then either executes S1 or S4 depending on the form of D_2.

First, consider the case of $k > 0$. For the purposes of the proof, consider creating additional references $ref_0, ref_1, ref_2, \ldots, ref_k$ to the node referenced by original dn and to the newly created nodes. Thus, the first application of S3 creates a reference ref_0 to the node referenced by dn in G and a reference ref_1 that references the new node (n_1). Therefore, edge $\langle ref_0, A_1, ref_1 \rangle$ is in

E. After that, any i-th ($2 \le i \le k$) call to *Subsumes* adds a new node n_i to N and a new binding $\langle ref_i, n_i \rangle$ to *Refs*. Note that *dn* is modified to reference node n_i during this step. Therefore, by part (c) of Claim 5.4, and since modification of *dn* or creation of new references by part (c) of S3 does not change other references, after k steps, we have a graph with path $\langle ref_0, A_1, ref_1, A_2, ref_2, \ldots, A_k, ref_k \rangle$ in it.

If $k = 0$, no new nodes are created and *Subsumes* just executes step S1 or S4.

Since S3 of *Subsumes* returns true only if the recursive call does, the top level call returns true only if the last call does. Note that besides the change of the set *Refs* by S3, all other steps executed during an invocation of *Subsumes* are just applications of the rules from set R. However, since by part (a) of Claim 5.4, modifications of the *Refs* set do *not* influence any other graph modifications, Lemma 5.4 would apply to the resulting graph if *dn* did not change and would still be pointing to the node referenced by ref_0.

Consider the case when D_2 = (**all** A_1 (**all** A_2 ... (**all** A_k C) ...)). In the last call to *Subsumes*, $dn = ref_k$ and procedure returns true only if $C \in \alpha(dn)$. Therefore, if *dn* was not modified by S3 and remained the same as ref_0, we would have a sequence of applications of rewrite rules from set R to graph $Init(D_1)$ that results in a graph with path $p = \langle ref_0, A_1, ref_1, A_2, ref_2, \ldots, A_k, ref_k \rangle$ and $C \in \alpha(ref_k)$ for some $k \ge 0$. Thus, by property (1) of Lemma 5.4, $T \models D_1 <$ (**all** $\gamma(p)$ C). However, by definition of function γ, $\gamma(p) = A_1 \circ A_2 \circ \ldots \circ A_k \circ \mathbf{Id}$. Therefore, by properties (7) and (8) of Lemma 5.1,

$$T \models D_1 < (\mathbf{all}\ A_1\ (\mathbf{all}\ A_2 \ldots (\mathbf{all}\ A_k\ C) \ldots)).$$

Next consider the case when D_2 = (**all** A_1 (**all** A_2 ... (**all** A_k (**same-as** Pf_1 Pf_2)) ...)). If the last call to *Subsumes* returns true, it follows that there exists a node n in N, such that $Reachable(dn, Pf_1, n)$ and $Reachable(dn, Pf_2, n)$ are true where $dn = ref_k$. Thus, there exist paths p_1 and p_2 from ref_k to n such that $\gamma(p_1) = Pf_1$ and $\gamma(p_2) = Pf_2$.

Note that constructions in S4 are just applications of rule A1. Therefore, Lemma 5.4 applies to this case as well: if *dn* was not modified by S3 and remained the same as ref_0, we would have a sequence of applications of rewrite rules from set R to graph $Init(D_1)$ that results in a graph with paths $\langle dn, A_1, ref_1, A_2, ref_2, \ldots, A_k, ref_k \rangle \circ p_1$ and $\langle dn, A_1, ref_1, A_2, ref_2, \ldots, A_k, ref_k \rangle \circ p_2$. Therefore, by Lemma 5.4,

$$T \models D_1 < (\mathbf{same\text{-}as}\ \gamma(\langle dn, A_1, ref_1, A_2, ref_2, \ldots, A_k, ref_k \rangle \circ p_1)\ \gamma(\langle dn, A_1, ref_1, A_2, ref_2, \ldots, A_k, ref_k \rangle \circ p_2)),$$

or by Claim 5.1,

$$T \models D_1 < (\mathbf{same\text{-}as}\ Pf \circ \gamma(p_1)\ \ Pf \circ \gamma(p_2)),$$

where $Pf = \gamma(\langle dn, A_1, ref_1, A_2, ref_2, \ldots, A_k, ref_k\rangle) = A_1 \circ A_2 \circ \ldots \circ A_k \circ \mathbf{Id}$. Therefore, by property (6) of Lemma 5.1, $T \models D_1 < (\mathbf{all}\ Pf\ (\mathbf{same\text{-}as}\ \gamma(p_1)\ \gamma(p_2)))$, or

$$T \models D_1 < (\mathbf{all}\ A_1 \circ A_2 \circ \ldots \circ A_k \circ \mathbf{Id}\ (\mathbf{same\text{-}as}\ Pf_1\ Pf_2)),$$

which in turn implies $T \models D_1 < (\mathbf{all}\ A_1\ (\mathbf{all}\ A_2\ \ldots\ (\mathbf{all}\ A_k\ (\mathbf{same\text{-}as}\ Pf_1\ Pf_2))\ \ldots\ ))$ by properties (7) and (8) of Lemma 5.1.

■

Next, in order to establish soundness for the case when D_2 has the form

$$(\mathbf{all}\ A_1\ (\mathbf{all}\ A_2\ \ldots\ (\mathbf{all}\ A_k\ (\mathbf{fd}\ C\ Pf_1\ \ldots\ Pf_m\ Pf))\ \ldots\ )),$$

we first prove an auxiliary Lemma 5.6 that allows us to use Lemma 5.5 in the proof of this case.

Lemma 5.6 *For any terminology T, descriptions D_1 and D_2, and path functions $Pf_1, \ldots, Pf_m, Pf$, and Pf' (where $m \geq 1$),*

$$T \models D_1 < (\mathbf{all}\ Pf'\ (\mathbf{fd}\ D_2\ Pf_1\ \ldots\ Pf_m\ Pf))$$

if and only if, for new attributes P and Q (that do not occur in T, D_1, D_2, or paths $Pf_1, \ldots, Pf_m, Pf$, and Pf'),

$$\begin{aligned} T \models\ & (\mathbf{and}\ (\mathbf{all}\ P\ D_2)\ (\mathbf{all}\ Q\ D_1) \\ & \quad (\mathbf{same\text{-}as}\ P \circ Pf_1\ \ Q \circ Pf' \circ Pf_1) \\ & \quad \ldots \\ & \quad (\mathbf{same\text{-}as}\ P \circ Pf_m\ \ Q \circ Pf' \circ Pf_m)) \\ & < (\mathbf{same\text{-}as}\ P \circ Pf\ \ Q \circ Pf' \circ Pf). \end{aligned}$$

Proof of if-part. We first prove the if-part of the lemma by contradiction. Let us assume that

(1) for any interpretation $\mathbf{J} = \langle \Delta', \cdot^{\mathbf{J}} \rangle$ that satisfies T and any object o in Δ', if

$$\begin{aligned} o \in\ & (\mathbf{and}\ (\mathbf{all}\ P\ D_2)\ (\mathbf{all}\ Q\ D_1) \\ & \quad (\mathbf{same\text{-}as}\ P \circ Pf_1\ \ Q \circ Pf' \circ Pf_1) \\ & \quad \ldots \\ & \quad (\mathbf{same\text{-}as}\ P \circ Pf_m\ \ Q \circ Pf' \circ Pf_m))^{\mathbf{J}}, \end{aligned}$$

then $o \in$ (**same-as** $P \circ Pf\ Q \circ Pf' \circ Pf$)$^{\mathbf{J}}$; and

(2) there exists an interpretation $\mathbf{I} = \langle \Delta, \cdot^{\mathbf{I}} \rangle$ that satisfies T for which there is an object $x \in \Delta$ such that $x \in D_1{}^{\mathbf{I}}$ and $x \notin$ (**all** Pf' (**fd** $D_2\ Pf_1 \ldots Pf_m\ Pf$))$^{\mathbf{I}}$.

By definition of the interpretation function, (2) implies that $Pf'^{\mathbf{I}}(x) \notin$ (**fd** $D_2\ Pf_1 \ldots Pf_m\ Pf$)$^{\mathbf{I}}$, which in turn implies that there is an object $y \in \Delta$ such that $y \in D_2{}^{\mathbf{I}}$, $Pf_1{}^{\mathbf{I}}(Pf'^{\mathbf{I}}(x)) = Pf_1{}^{\mathbf{I}}(y)$, ..., $Pf_m{}^{\mathbf{I}}(Pf'^{\mathbf{I}}(x)) = Pf_m{}^{\mathbf{I}}(y)$, and $Pf^{\mathbf{I}}(Pf'^{\mathbf{I}}(x)) \neq Pf^{\mathbf{I}}(y)$.

Consider an interpretation $\mathbf{J}$ that is exactly the same as $\mathbf{I}$ except possibly for interpretations of attributes P and Q. The only restriction on the interpretations of P and Q is that there is an object o in Δ (not necessarily distinct from x or y) such that $P^{\mathbf{J}}(o) = y$ and $Q^{\mathbf{J}}(o) = x$. Clearly, interpretation $\mathbf{J}$ can be easily constructed from $\mathbf{I}$ and it would continue to satisfy T as well as any implications derived from T, since P and Q are attributes that do not occur in T.

By our assumptions and constructions, $o \in$ (**all** $P\ D_2$)$^{\mathbf{J}}$ and $o \in$ (**all** $Q\ D_1$)$^{\mathbf{J}}$. In addition, since $Pf_i{}^{\mathbf{I}}(Pf'^{\mathbf{I}}(x)) = Pf_i{}^{\mathbf{I}}(y)$ for $1 \leq i \leq m$, $Pf_i{}^{\mathbf{J}}(Pf'^{\mathbf{J}}(x)) = Pf_i{}^{\mathbf{J}}(y)$, and thus, $Pf_i{}^{\mathbf{J}}(Pf'^{\mathbf{J}}(Q^{\mathbf{J}}(o))) = Pf_i{}^{\mathbf{J}}(P^{\mathbf{J}}(o))$. Therefore, by property (4) of Claim 3.1, $o \in$ (**same-as** $P \circ Pf_i\ \ Q \circ Pf' \circ Pf_i$)$^{\mathbf{J}}$ for $1 \leq i \leq m$.

It follows that

$$
\begin{aligned}
o \in (\mathbf{and}\ &(\mathbf{all}\ P\ D_2)\ (\mathbf{all}\ Q\ D_1) \\
&(\mathbf{same\text{-}as}\ P \circ Pf_1\ \ Q \circ Pf' \circ Pf_1) \\
&\ldots \\
&(\mathbf{same\text{-}as}\ P \circ Pf_m\ \ Q \circ Pf' \circ Pf_m))^{\mathbf{J}},
\end{aligned}
$$

and thus, by part (1) of the assumption above, $o \in$ (**same-as** $P \circ Pf\ Q \circ Pf' \circ Pf$)$^{\mathbf{J}}$. Therefore, $Pf^{\mathbf{J}}(Pf'^{\mathbf{J}}(Q^{\mathbf{J}}(o))) = Pf^{\mathbf{J}}(P^{\mathbf{J}}(o))$, which in turn implies that $Pf^{\mathbf{J}}(Pf'^{\mathbf{J}}(x)) = Pf^{\mathbf{J}}(y)$. However, this contradicts our statement above that $Pf^{\mathbf{I}}(Pf'^{\mathbf{I}}(x)) \neq Pf^{\mathbf{I}}(y)$ (since Pf and Pf' do not contain attributes P and Q, their $\mathbf{I}$ and $\mathbf{J}$ interpretations must the same for all objects in Δ).

Proof of only-if-part. We prove the only-if-part also by contradiction. Let us assume that

(1) for any interpretation $\mathbf{I} = \langle \Delta', \cdot^{\mathbf{I}} \rangle$ that satisfies T and an object $x \in \Delta'$,

if $x \in D_1{}^{\mathbf{I}}$ then $x \in$ (**all** Pf' (**fd** $D_2\ Pf_1 \ldots Pf_m\ Pf$))$^{\mathbf{I}}$; and

(2) there exists a terminology $\mathbf{J} = \langle \Delta, \cdot^{\mathbf{J}} \rangle$ that satisfies T and an object o in Δ, such that

$$o \in (\textbf{and } (\textbf{all } P\ D_2)\ (\textbf{all } Q\ D_1)$$
$$(\textbf{same-as } P \circ Pf_1\ \ Q \circ Pf' \circ Pf_1)$$
$$\ldots$$
$$(\textbf{same-as } P \circ Pf_m\ \ Q \circ Pf' \circ Pf_m))^{\mathbf{J}},$$

and $o \notin (\textbf{same-as } P \circ Pf\ \ Q \circ Pf' \circ Pf)^{\mathbf{J}}$.

Let $x = Q^{\mathbf{J}}(o)$ and $y = P^{\mathbf{J}}(o)$. Then, by part (2) of the assumption, $x \in D_1^{\mathbf{J}}$, $y \in D_2^{\mathbf{J}}$, and $(P \circ Pf_i)^{\mathbf{J}}(o) = (Q \circ Pf' \circ Pf_i)^{\mathbf{J}}(o)$ for $1 \le i \le m$. Thus, by property (4) of Claim 3.1, $Pf_i^{\mathbf{J}}(Pf'^{\mathbf{J}}(Q^{\mathbf{J}}(o))) = Pf_i^{\mathbf{J}}(P^{\mathbf{J}}(o))$, which in turn implies that $Pf_i^{\mathbf{J}}(Pf'^{\mathbf{J}}(x)) = Pf_i^{\mathbf{J}}(y)$ for $1 \le i \le m$. Additionally, $(P \circ Pf)^{\mathbf{J}}(o)$ must be different from $(Q \circ Pf' \circ Pf)^{\mathbf{J}}(o)$, and thus, $Pf^{\mathbf{J}}(Pf'^{\mathbf{J}}(x)) \neq Pf^{\mathbf{J}}(y)$.

Since $\mathbf{J}$ satisfies T, it is a valid instance of interpretation that must also satisfy part (1) of the assumption. Therefore, since $x \in D_1^{\mathbf{J}}$, $x \in (\textbf{all } Pf'\ (\textbf{fd } D_2\ Pf_1\ \ldots\ Pf_m\ Pf))^{\mathbf{J}}$, and thus, by definition of the interpretation function, $Pf'^{\mathbf{J}}(x) \in (\textbf{fd } D_2\ Pf_1\ \ldots\ Pf_m\ Pf)^{\mathbf{J}}$. Since $y \in D_2^{\mathbf{J}}$, and $Pf_i^{\mathbf{J}}(Pf'^{\mathbf{J}}(x)) = Pf_i^{\mathbf{J}}(y)$ for $1 \le i \le m$, by definition of interpretation function, $Pf^{\mathbf{J}}(Pf'^{\mathbf{J}}(x))$ must be the same object as $Pf^{\mathbf{J}}(y)$ which contradicts our statement above that $Pf^{\mathbf{J}}(Pf'^{\mathbf{J}}(x)) \neq Pf^{\mathbf{J}}(y)$.

■

Lemma 5.7 *Let $G = \langle N, E, Refs \rangle$ denote a graph obtained by a number of applications of rewrite rules from the set R = {M1, M2, M3, E1, E2, E3, E4, E5, E6, E7, A1} to graph $G_0 = Init(D_1)$ for some description D_1 and a terminology T, and let D_2 be a description*

$$(\textbf{all } A_1\ (\textbf{all } A_2\ \ldots\ (\textbf{all } A_k\ (\textbf{fd } D\ Pf_1\ \ldots\ Pf_m\ Pf))\ \ldots\ ))$$

for some $k \ge 0$ and $m \ge 1$, where D is a description, A_i is a primitive attribute for all $1 \le i \le k$, and Pf and Pf_j ($1 \le j \le m$) are path functions. Then, if Subsumes(D_2, G, T) returns true, $D_1 < D_2$ is a logical consequence of T.

Proof. Intuitively, the lemma is a consequence of Lemmas 5.4 and 5.6. We first consider sequences of applications of rewrite rules in construction of a graph (G_4) that results after the application of step S5 of *Subsumes*. Then, we use these sequences to construct an almost the same graph (G_4') starting with an initial graph for an equivalent problem, $T \models D_3 < D_4$, that is defined by Lemma 5.6. Since *Subsumes* returns true, there must exist paths from nodes dn_c and $fdcn_c$ that "agree" on a node in G_4 and that have edge labels corresponding to a prefix of Pf. We use these paths in G_4' to prove that the equivalent problem must follow by Lemma 5.4, which in turn implies that T must logically imply $D_1 < D_2$ if *Subsumes*(D_2, G, T) returns true. Let us now proceed with a more formal proof.

Let rewrite rules E4 and E5 without their last parts that invoke procedure *Merge* be denoted by $E4_{short}$ and $E5_{short}$ respectively. Then, any application of rule E4 [E5] to a description graph consists of an application of rule $E4_{short}$ [$E5_{short}$] followed by a sequence of applications of rules M1, M2 and M3. Since our proofs of Claim 5.4 and Lemma 5.4 for rules E4 and E5 essentially consider $E4_{short}$ and $E5_{short}$ separately from the rules of *Merge*, Claim 5.4 and Lemma 5.4 still hold if set R is generalized to have $E4_{short}$ and $E5_{short}$ instead of E4 and E5. Therefore, if we apply a sequence of rewrite rules from the set R' = {M1, M2, M3, E1, E2, E3, $E4_{short}$, $E5_{short}$, E6, E7, A1} to a graph $Init(D_1)$, properties (1) and (2) of Lemma 5.4 hold on the resulting graph.

Consider how the initial graph $Init(D_1)$, denoted by G_1, is modified. First, we apply a sequence, denoted by seq_1, of rewrite rules from R' to get the new graph G. Then, *Subsumes* recursively executes step S3 (if $k \geq 1$) and constructs a chain of new nodes $n_1, n_2, \ldots, n_k$ that form a path $\langle dn, A_1, n_1, A_2, n_2, \ldots, A_k, n_k \rangle$, where dn is the initial distinguished node. Just as in the proof of Lemma 5.5, we consider references $ref_0, ref_1, \ldots, ref_k$ to nodes $dn, n_1, n_2, \ldots, n_k$ respectively. Thus, path $\langle ref_0, A_1, ref_1, A_2, ref_2, \ldots, A_k, ref_k \rangle$ is constructed by the applications of rule S3. The graph obtained so far is denoted by G_2 and depicted in Figure 5.5(a). (*Remark*: Some nodes ref_i can actually reside "outside" of the boundary of G. This depends on which nodes are actually "merged in" by the *Merge* procedure. Our proof, however, does not depend on the distinction between the nodes that are "outside" and those that are "inside".)

Next, step S5 of *Subsumes* is applied. Let seq_2 denote the sequence of applications of rules from the set {M1, M2, M3, E1, E2, E3, $E4_{short}$, $E5_{short}$, E6, E7} that are executed during the applications of rule S3 and by parts (a) through (d) of S5 of *Subsumes*, and let G_3 denote the graph obtained after part (d) of S5 is applied. G_3 is graphically depicted in Figure 5.5(b). Note that seq_2 neither includes construction of new nodes and edges nor the modifications to the *Refs* set by the steps of procedure *Subsumes*. Such modifications of the *Refs* set do not affect applicability of other rules or their effect by part (a) of Claim 5.4. However, seq_2 does include all other modifications to the graph by procedures *Merge* and *Exp* invoked during executions of step S3. Finally, let seq_3 denote the sequence of rule applications by parts (e) and (f) of S5, and let G_4 denote the resulting graph.

Note that *Subsumes* returns true in step S3 if and only if the recursive call to *Subsumes* returns true. Therefore, the top level *Subsumes* returns true only if step S5 returns true. Since lemma assumes that *Subsumes*(D_2, G, T) returns true, there must exist a prefix Pf' of Pf and a node n in G_4 such that *Reachable*(dn_c, Pf', n) and *Reachable*($fdcn_c$, Pf', n) are true. Thus, there exists a path p_1 from dn_c to n and a path p_2 from $fdcn_c$ to n in G_4 such that $\gamma(p_1) = \gamma(p_2) = Pf'$. However, since dn_c is the same as ref_k when S5 is executed, by Claim 5.1, there must exist a path p_3 from ref_0 to n such that $\gamma(p_3) = A_1 \circ A_2 \circ \ldots \circ A_k \circ Pf'$.

Let us denote description

(and (all P D**) (all** Q D_1**)**
(same-as $P \circ Pf_1$ $Q \circ Pf_{prefix} \circ Pf_1$**)**
...
(same-as $P \circ Pf_m$ $Q \circ Pf_{prefix} \circ Pf_m$**))**

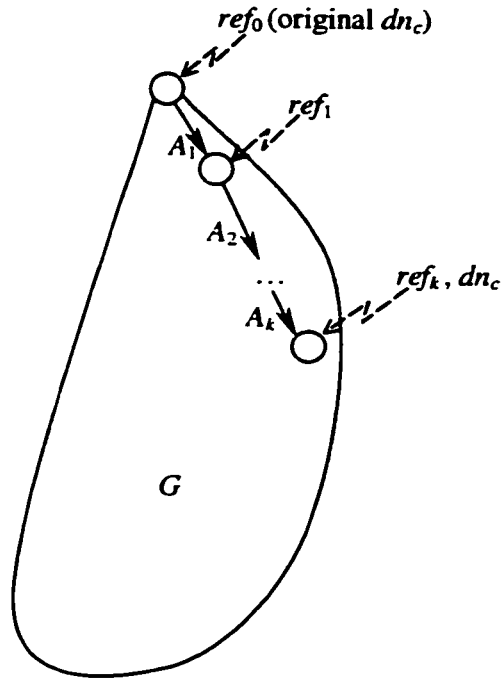
by D_3, and description **(same-as** $P \circ Pf$ $Q \circ Pf_{prefix} \circ Pf$ **)** by D_4 where P and Q are some new attributes and $Pf_{prefix} = A_1 \circ A_2 \circ \ldots \circ A_k \circ \mathbf{Id}$. Then, by Lemma 5.6 and properties (7) and (8) of Lemma 5.1, $T \models D_3 < D_4$ if and only if $T \models D_1 < D_2$.

We now present a sequence of applications of rewrite rules from R' that starts with $G_1' = Init(D_3)$ graph and ends with a graph which has G_4 as its sub-graph. Just as in the proof of Lemma 5.5, since some nodes could be removed when merged with other nodes, we use references to denote the nodes of interest. By part (a) of Claim 5.4, these references do not affect modifications to nodes, edges, α sets or other references in the graph.

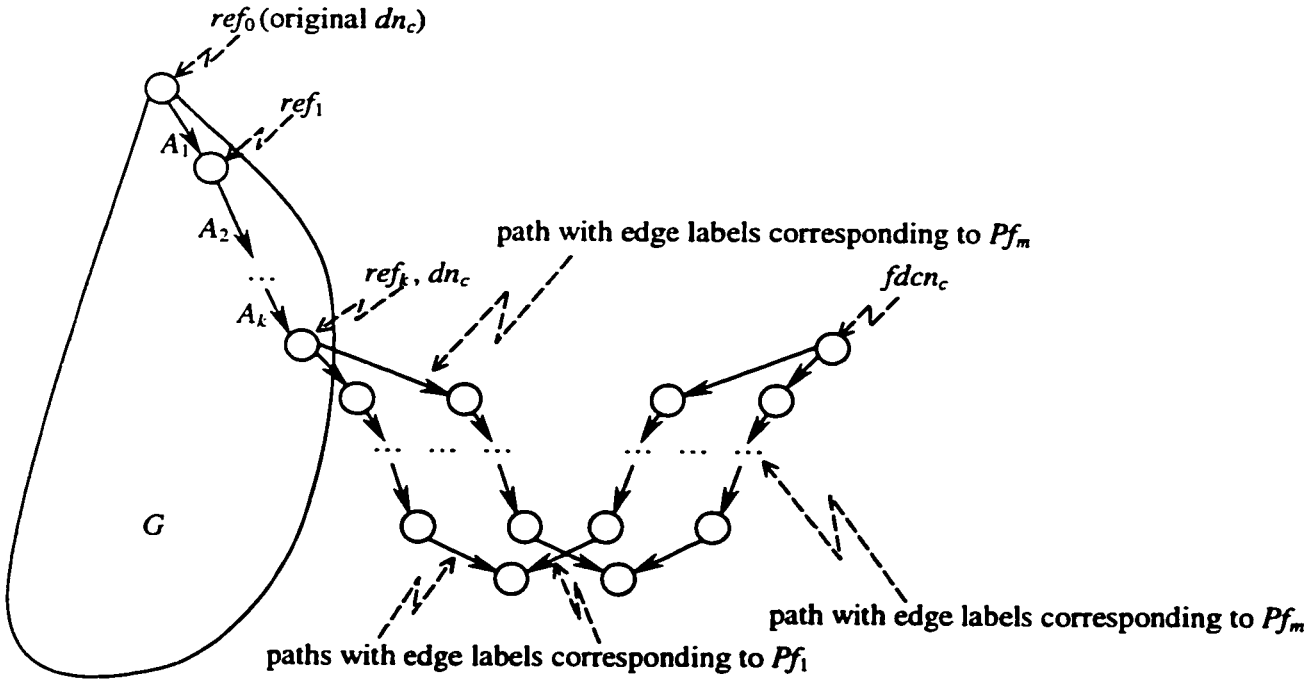
By definition of procedure *Init*, G_1' consists of a single node, for which we create a new reference r, with $\alpha(r) = \{D_3\}$ (i.e. *dn* of G_1' references the same node as r at this point). First, we apply rule E1 to node r. The rule "expands" the **and** description into the set consisting of the descriptions inside D_3. Then, we apply rules $E4_{short}$ to all **same-as** descriptions in $\alpha(r)$ (recall that $m \geq 1$). In particular, this constructs m edges that start at r and are labeled with Q and m edges that start at r and are labeled with P. We can merge these into two edges $\langle r, Q, u \rangle$ and $\langle r, P, v \rangle$ by rules M1 and M3, where u and v are two new references to the corresponding nodes. In the same manner, we also merge all edges along the path that starts at node u and has labels from Pf_{prefix} (we first merge all edges labeled A_1 that start at u, then we merge edges labeled A_2, etc.). Let ref_1', ..., ref_k' denote references to new nodes produced along the path. In other words, the graph now contains path $\langle u, A_1, ref_1', A_2, ref_2', \ldots, A_k, ref_k' \rangle$.

Next, we apply rules E2 to descriptions **(all** P D**)** and **(all** Q D_1**)** in r. As a result, description D moves into $\alpha(v)$ and description D_1 moves into $\alpha(u)$. The resulting graph, G_2', is presented in Figure 5.6(a). Note that node u is now the same as the graph $Init(D_1)$. Thus, we can now apply sequence seq_1 to node u to produce G as a sub-graph.

Then, we merge nodes along the path $\langle u, A_1, ref_1', A_2, ref_2', \ldots, A_k, ref_k' \rangle$ with the rest of the graph if possible, and apply the rules from seq_2 to the resulting graph (note that we do not need to perform any constructions that were completed during the applications of steps S3 and S5 in the construction of G_3 since all necessary nodes and edges already exist). Clearly, the resulting graph, G_3' (depicted in Figure 5.6(b)), will be isomorphically the same as graph G_3 except for a "hat" consisting of node r and edges $\langle r, Q, u \rangle$ and $\langle r, P, v \rangle$. Moreover, the isomorphism maps node ref_0 to node u and node $fdcn_c$ to v. Therefore, we can now apply the rules in the sequence seq_3 to obtain graph G_4' that would be isomorphically the same as graph G_4 except for the "hat".

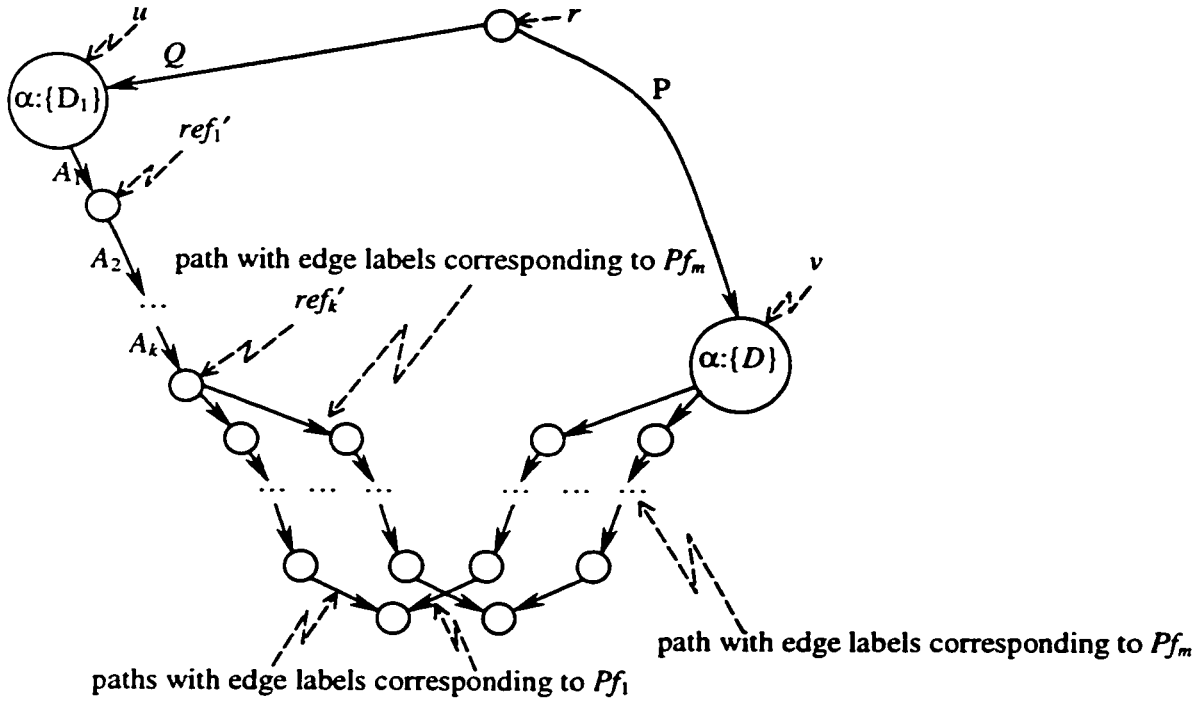


(a) Graph G_2.

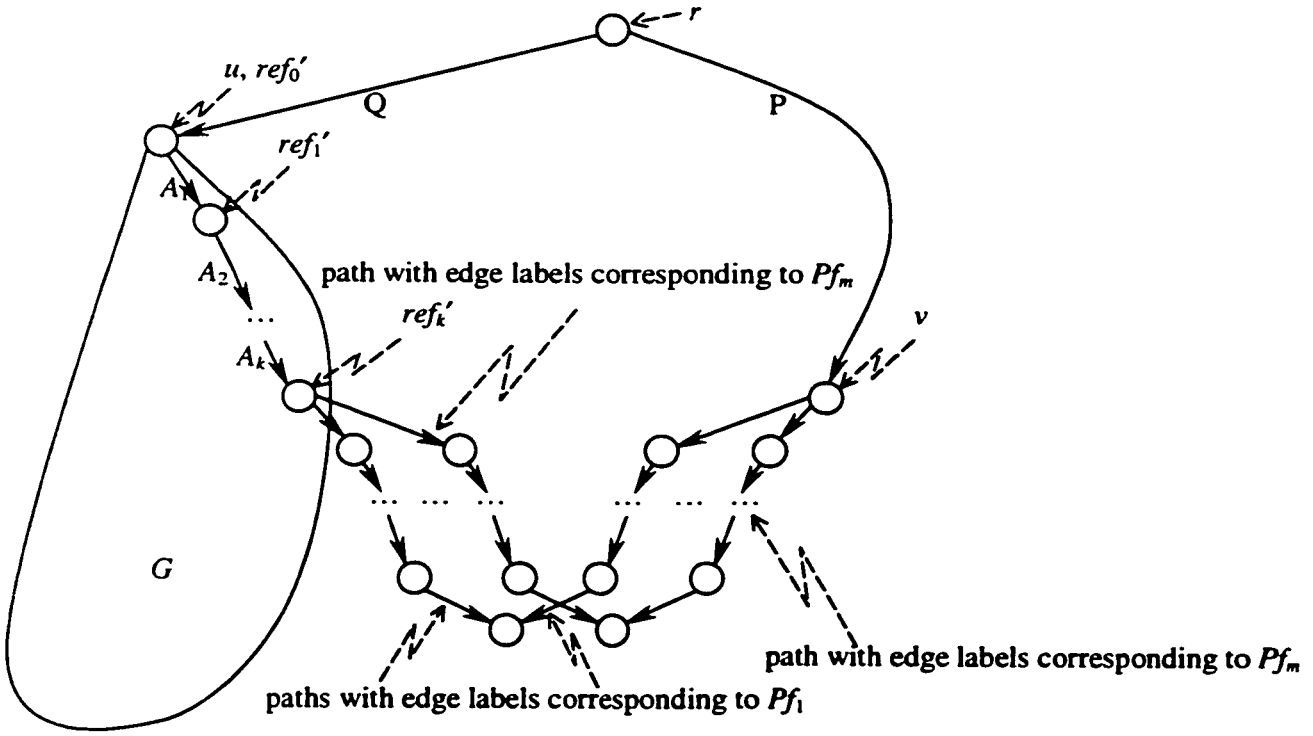


(b) Graph G_3.

Figure 5.5: Some steps in the first construction of the proof of Lemma 5.7.



(a) Graph G_2'.



(b) Graph G_3'.

Figure 5.6: Some steps in the second construction of the proof of Lemma 5.7.

Since, as noted above, there exists a path p_3 from ref_0 to n and a path p_2 from $fdcn_c$ to n in G_4 such that $\gamma(p_3) = Pf_{prefix} \circ Pf'$ and $\gamma(p_2) = Pf'$, there must exist a path p_3' from u to n' and a path p_2' from v to n' for some node n' in G_4' such that $\gamma(p_3') = Pf_{prefix} \circ Pf'$ and $\gamma(p_2') = Pf'$. Therefore, by Claim 5.1, there exist paths p_4' and p_5' from r to n' in G_4' such that $\gamma(p_4') = Q \circ Pf_{prefix} \circ Pf'$ and $\gamma(p_5') = P \circ Pf'$. Thus, by Lemma 5.4, $T \models D_3 <$ (**same-as** $P \circ Pf'$ $Q \circ Pf_{prefix} \circ Pf'$). Then, by definition of a prefix of a path function and property (4) of Lemma 5.1, $T \models D_3 < D_4$, and therefore, $T \models D_1 < D_2$.

■

At this point, we can prove soundness for the general case by induction on the number of view substitutions necessary to obtain $D_{2[u]}$ from D_2 and the number of occurrences of the **and** constructor inside $D_{2[u]}$. Lemmas 5.5 and 5.7 serve as the base case for such an induction.

Theorem 5.2 (*soundness*) *Let* $G = \langle N, E, Refs \rangle = Exp(Init(D_1), T)$ *for some description* D_1 *and a terminology* T. *Then, if* $Subsumes(D_2, G, T)$ *returns true for some description* D_2, *then* $D_1 < D_2$ *is a logical consequence of* T.

Proof. We present a proof of the more general statement of the theorem where G is allowed to be an arbitrary graph obtained by a number of applications of rewrite rules from the set R = {M1, M2, M3, E1, E2, E3, E4, E5, E6, E7, A1} to graph $G_0 = Init(D_1)$ for some description D_1 and a terminology T.

The proof that follows is an induction on the *and-size* of D_2, denoted $\|D_2\|$, which we define here as the sum of the number of view substitutions necessary to obtain $D_{2[u]}$ from D_2 and the number of **and** concept constructors occurring in $D_{2[u]}$. First, note that Lemmas 5.5 and 5.7 prove the generalized statement of the theorem for the base case of the induction, i.e. when $\|D_2\|$ is 0, or when D_2 has one of the forms

(**all** A_1 (**all** A_2 ... (**all** A_k C) ...)),
(**all** A_1 (**all** A_2 ... (**all** A_k (**same-as** Pf_1 Pf_2)) ...)), or
(**all** A_1 (**all** A_2 ... (**all** A_k (**fd** C Pf_1 ... Pf_m Pf)) ...)).

Next, assume that the statement holds for an arbitrary parameter description of *Subsumes* with and-size no greater than m for some $m \geq 0$, and then, consider a particular description D_2 with $\|D_2\| = m + 1$. Since $\|D_2\| \geq 1$, D_2 has one of the forms

(a) (**all** A_1 (**all** A_2 ... (**all** A_k (**and** D_1' D_2' ... D_t')) ...)), or
(b) (**all** A_1 (**all** A_2 ... (**all** A_k V) ...))

for some $k \geq 0$, $t \geq 2$, primitive attributes A_i, descriptions D_i', and a view name V.

First, consider case (a). If $k > 0$, the call to *Subsumes*(D_2, G, T) creates a path $p = \langle dn_0, A_1, n_1, A_2, n_2 \ldots, A_k, n_k \rangle$ from the original distinguished node dn_0 to some (possibly new) node n in G by k applications of step S3. Without loss of generality, let us assume *Subsumes* recursively "explores" descriptions D_1', D_2', ..., D_t' in that order. Then, by step S2, the procedure returns true only if *Subsumes*(D_i', G', T) returns true for a modified graph G' for $1 \leq i \leq t$.

First, consider the recursive call with parameter description D_1'. Note that a call to *Subsumes*((**all** A_1 (**all** A_2 ... (**all** A_k D_1') ...)), G, T) would modify the graph exactly the same way, and therefore, would also return true. Since ||(**all** A_1 (**all** A_2 ... (**all** A_k D_1') ...))|| is no more than m however, it follows from the inductive assumption that

$$T \models D_1 < (\textbf{all}\ A_1\ (\textbf{all}\ A_2 \ldots (\textbf{all}\ A_k\ D_1') \ldots)).$$

Next, assume that recursive calls to *Subsumes* with parameter descriptions D_1' through D_{i-1}' returned true and consider the recursive call *Subsumes*(D_i', G', T) for some $2 \leq i \leq t$. Observe that steps S1, S2, S5 and S6 do not modify the graph (S5 only modifies a copy of the graph), and all modifications by steps S3 and S4 are essentially applications of rules from set R, except for the modifications of the set *Refs* by S3. These modifications (i) do not affect applicability and the results of firing of rules by part (a) of Claim 5.4, and (ii) make dn point to n_k right before *Subsumes*(D_i', G', T) is invoked. In addition, since both S3 and S4 invoke *Merge* and *Exp* in the end, and since we are considering the case when $k > 0$, G' must be a result of an application of *Exp* procedure and must be well-formed.

Now consider what would happen if we invoked

$$\textit{Subsumes}((\textbf{all}\ A_1\ (\textbf{all}\ A_2 \ldots (\textbf{all}\ A_k\ D_i') \ldots)), G'', T)$$

where G'' is a graph obtained from *Init*(D_1) by the same sequence of applications of rules from set R that obtained graph G'. By part (a) of Claim 5.4, the only difference between G' and G'' is the new references in *Refs* created by S3 and the fact that reference dn in G'' still points to node dn_0 instead of node n_k. The first thing *Subsumes* would do is execute step S3 and create an edge $\langle dn_0, A_1, n_1' \rangle$ for some new node n_1'. Therefore, since G' and G'' are well-formed, the only thing that *Merge* can do in part (d) of S3, is to merge node n_1' with n_1, and therefore, again obtain the same graph as G'' (except for the new dn' reference created by S3 and reference dn pointing to the node resulting after merge of n_1 and n_1'). Moreover, since G' is a result of an application of *Exp* procedure and by part (a) of Claim 5.4, invocation of *Exp* in step (e) of S3 cannot execute any steps on the resulting graph.

The same argument applies to the rest of $k-1$ executions of step S3 that follow. Every step j $(2 \leq j \leq k)$ just creates a new edge labeled A_j and ending at a new node n_j' that is merged with already existing edge $\langle n_{j-1}, A_j, n_j \rangle$ of path p. Thus, the total effect of the k executions of step S3 is the same graph as G' including reference dn pointing to the result of merge of nodes n_k and n_k' (with the exception of some new references). Therefore, the results of the steps of *Subsumes* that follow are the same as the results of the recursive call

$$Subsumes(D_i', G', T),$$

which in turn implies that since $Subsumes(D_i', G', T)$ returns true, so does

$$Subsumes((\textbf{all}\ A_1\ (\textbf{all}\ A_2\ \ldots\ (\textbf{all}\ A_k\ D_i')\ \ldots\ )), G'', T).$$

However, since the parameter description of the above call has the and-size of at most m, and since G'' was obtained as a number of applications of rules from set R to $Init(D_1)$, it follows by the inductive assumption that

$$T \models D_1 < (\textbf{all}\ A_1\ (\textbf{all}\ A_2\ \ldots\ (\textbf{all}\ A_k\ D_i')\ \ldots\ )).$$

Therefore, $T \models D_1 < (\textbf{all}\ A_1\ (\textbf{all}\ A_2\ \ldots\ (\textbf{all}\ A_k\ D_i')\ \ldots\ ))$ for $1 \leq i \leq t$. It follows that for any interpretation $\mathbf{I}$ that satisfies T and any object $x \in D_1^{\mathbf{I}}$,

$$x \in (\textbf{all}\ A_1\ (\textbf{all}\ A_2\ \ldots\ (\textbf{all}\ A_k\ D_i')\ \ldots\ ))^{\mathbf{I}} \text{ for } 1 \leq i \leq t, \text{ or}$$

by definition of the interpretation function,

$$A_k^{\mathbf{I}}(\ \ldots\ (A_2^{\mathbf{I}}(A_1^{\mathbf{I}}(x)))\ \ldots\ ) \in D_i'^{\mathbf{I}} \text{ for } 1 \leq i \leq t.$$

Therefore, $A_k^{\mathbf{I}}(\ \ldots\ (A_2^{\mathbf{I}}(A_1^{\mathbf{I}}(x)))\ \ldots\ ) \in (\textbf{and}\ D_1'\ D_2'\ \ldots\ D_t')^{\mathbf{I}}$, or

$$x \in (\textbf{all}\ A_1\ (\textbf{all}\ A_2\ \ldots\ (\textbf{all}\ A_k\ (\textbf{and}\ D_1'\ D_2'\ \ldots\ D_t'))\ \ldots\ ))^{\mathbf{I}}.$$

Thus, $T \models D_1 < (\textbf{all}\ A_1\ (\textbf{all}\ A_2\ \ldots\ (\textbf{all}\ A_k\ (\textbf{and}\ D_1'\ D_2'\ \ldots\ D_t'))\ \ldots\ ))^{\mathbf{I}}$.

If $k = 0$, the proof is analogous except for the fact that no new path is constructed by step S3, and therefore, G'' is exactly the same as G' since set *Refs* is only modified inside recursive calls to S3. In other words, even if there are any invocations of S3 during explorations of D_1' through D_{i-1}', restoring of reference dn by S3 make sure that recursive call $Subsumes(D_i', G', T)$ still has reference dn pointing to dn_0.

Next, consider case (b), or when D_2 has the form (**all** A_1 (**all** A_2 ... (**all** A_k V) ...)) for some view name V with corresponding view definition $V \equiv D$ in T. Just as in case (a), if $k > 0$, *Subsumes*(D_2, G, T) creates a path $p = \langle dn_0, A_1, n_1, A_2, n_2 \ldots, A_k, n_k \rangle$ from the original distinguished node dn_0 to some (possibly new) node n_k in G by step S3. Then, there are two possible cases: (i) *Subsumes* returns true because $V \in \alpha(n_k, G)$, or (ii) the recursive call to *Subsumes*(D, G, T) returns true.

Case (i) is completely analogous to the case when D_2 has the form

$$(\textbf{all}\ A_1\ (\textbf{all}\ A_2 \ldots (\textbf{all}\ A_k\ C) \ldots))$$

for some primitive concept C: if set *Refs* was not modified by S3 and dn remained the same as dn_0, we would have a sequence of applications of the rewrite rules from set R to the graph *Init*(D_1) that results in a graph with path p and where $V \in \alpha(n_k)$. Thus, by property (1) of Lemma 5.4, $T \models D_1 < (\textbf{all}\ \gamma(p)\ V)$. However, by definition of function γ, $\gamma(p) = A_1 \circ A_2 \circ \ldots \circ A_k \circ \textbf{Id}$. Therefore, by properties (7) and (8) of Lemma 5.1,

$$T \models D_1 < (\textbf{all}\ A_1\ (\textbf{all}\ A_2 \ldots (\textbf{all}\ A_k\ V) \ldots)).$$

Next, consider case (ii). Note that by definition of *Subsumes*, and step S6 in particular, invocation of

$$\textit{Subsumes}((\textbf{all}\ A_1\ (\textbf{all}\ A_2 \ldots (\textbf{all}\ A_k\ D) \ldots)), G, T)$$

would perform exactly the same actions on G as

$$\textit{Subsumes}((\textbf{all}\ A_1\ (\textbf{all}\ A_2 \ldots (\textbf{all}\ A_k\ V) \ldots)), G, T),$$

except for the execution of step S6 that does not modify G or affect the result of the procedure. It follows that the former invocation would also return true. However, the and-size of the description in that invocation is no greater than m, and therefore, by the inductive assumption,

$$T \models D_1 < (\textbf{all}\ A_1\ (\textbf{all}\ A_2 \ldots (\textbf{all}\ A_k\ D) \ldots)).$$

Therefore, for any interpretation **I** that satisfies T and an object x in D_1,

$$A_k^{\mathbf{I}}(\ldots (A_2^{\mathbf{I}}(A_1^{\mathbf{I}}(x))) \ldots) \in D^{\mathbf{I}}.$$

Since **I** satisfies T, however, it follows that $V^{\mathbf{I}} = D^{\mathbf{I}}$, and thus, $A_k^{\mathbf{I}}(\ldots (A_2^{\mathbf{I}}(A_1^{\mathbf{I}}(x))) \ldots) \in V^{\mathbf{I}}$, or

$$x \in (\textbf{all}\ A_1\ (\textbf{all}\ A_2\ \ldots\ (\textbf{all}\ A_k\ V)\ \ldots\ ))^{\mathbf{I}}.$$

Therefore, $T \models D_1 < (\textbf{all}\ A_1\ (\textbf{all}\ A_2\ \ldots\ (\textbf{all}\ A_k\ V)\ \ldots\ ))^{\mathbf{I}}$.

If $k = 0$, the proof is again analogous except for the fact that no new path is constructed by step S3.

■

5.6.3 Completeness

Our completeness result is stated in the following theorem.

Theorem 5.3 (*completeness*) *Let* $G = \langle N, E, Refs \rangle = Exp(Init(D_1), T)$ *for some description* D_1 *and a terminology* T. *Then, if* $D_1 < D_2$ *is a logical consequence of* T *for some description* D_2, *Subsumes*(D_2, G, T) *returns true.*

Proof. The proof is by contrapositive. We assume that *Subsumes*(D_2, G, T) returns false and then show that there exists an interpretation **I** that satisfies T and does not satisfy constraint $D_1 < D_2$. To obtain **I**, we first construct an infinite sequence S_D of (finite) description graphs that starts with graphs $G_{-2} = Init(D_1)$, $G_{-1} = Exp(Init(D_1), T)$, and the graph G_0 "accountable" for the false result of *Subsumes*(D_2, G, T). Then, we use nodes of the graphs in S_D to define Δ, and their α sets to define the interpretations of primitive concepts over Δ. In addition, we use edges in the graphs to define interpretations of primitive attributes. We show that this sequence of graphs can be used to produce the desired interpretation **I**.

As in the proof of completeness in Chapter 4, the idea of the construction is to add "missing" attributes to the graph originally responsible for the false result of *Subsumes*(D_2, G, T) in a way that eventually ensures that the interpretation of each primitive attribute is a total function. Nodes in the graphs will be the objects in Δ, and edges will determine the interpretation of attributes. Thus, each node must have an edge for every attribute in at least one graph. Recall, however, that creation of arbitrary new edges between nodes that already exist in the graph might violate uniqueness constraints in T. Thus, for every pair of a node n_1 and a primitive attribute A, we create a new node n_2 and an edge $\langle n_1, A, n_2 \rangle$ unless such an edge already exists; determine the class(es) that n_2 must belong to; and then proceed by creating new edges outgoing from n_2.

More formally, let $S_D = [G_{-2}, G_{-1}, G_0, G_1, \ldots, G_i = \langle N_i, E_i, Refs_i \rangle, \ldots]$ denote the infinite sequence of finite description graphs constructed as follows. Let $G_{-2} = Init(D_1)$ and $G_{-1} = Exp(Init(D_1), T)$. Now consider what happens during a call to *Subsumes*(D_2, G_{-1}, T) that returns false. Note that steps S3 and S4 of *Subsumes* modify the graph, whereas the modifications of step S5 do not affect the graph since they are applied to a copy of the graph. Also, note that

Subsumes recursively calls itself only when it deals with either **and** constructors, **all** constructors, or view names. In all three cases, it returns false if and only if at least one recursive call returns false. Therefore, there must be a recursive call with parameter description being either a primitive concept, a **same-as** or an **fd** description that returns false. In other words, some recursive call to *Subsumes* must return false either at step S1, S4, or S5. Consider the first time when *Subsumes* returns false. If this happens during step S1 or S4, we define G_0 to be the (modified) graph that is obtained right after *Subsumes* returns false. In the negative result is returned by step S5, we define G_0 to be the (modified) copy of the graph that is obtained right after *Subsumes* returns false.

Next, we define the rest of the sequence S_D. We obtain G_i ($i \geq 1$) by adding a number of nodes and edges to graph G_{i-1} and then invoking procedure *Exp* on the resulting graph at step i. As in the proof of completeness in Chapter 4, let N_i' (resp. E_i') denote the set of new nodes (resp. edges) that we add at step i. We will show in Assertion 1 that procedure *Exp* does not add or remove any nodes or edges to or from the resulting graph, and therefore that $N_i = N_{i-1} \cup N_i'$ and $E_i = E_{i-1} \cup E_i'$.

Again, analogously to the proof of completeness in Chapter 4, given a sequence $S_{PA} = [A_1, A_2, \ldots]$ of all primitive attributes[19], we are going to add a countably infinite number of nodes with countably infinite number of outgoing edges. We use a "triangular" construction to keep the graph resulting at each step finite. Thus, we define step i ($i \geq 1$) of the construction, denoted by CS_i, as follows:

> CS_i. For all $n_1 \in N_{i-1}$ and $A_j \in S_{PA}$ ($1 \leq j \leq i$): if there is no n_2 in N_{i-1} such that $\langle n_1, A_j, n_2 \rangle$ is in E_{i-1}, add a new node n_2 to N_i' and edge $\langle n_1, A_j, n_2 \rangle$ to E_i'. G_i is defined as $Exp(\langle N_{i-1} \cup N_i', E_{i-1} \cup E_i', Refs_{i-1} \rangle, T)$.

Finally, we define the interpretation **I** as follows:

- $\Delta = N_0 \cup \bigcup_{i \geq 1} N_i'$;
- $C^{\mathbf{I}} = \{n \in \Delta \mid \exists\, G_i\ (i \geq 0)$ such that $n \in N_i$ and $C \in \alpha(n, G_i)\}$ for any primitive concept C; and
- $A^{\mathbf{I}} = \{(n_1, n_2) \mid \exists\, G_i\ (i \geq 0)$ such that $n_1, n_2 \in N_i$ and $\langle n_1, A, n_2 \rangle \in E_i\}$ for any primitive attribute A.

In addition, for every view definition $V_j \equiv D_j$ ($1 \leq j \leq m$) in T, we define $V_j^{\mathbf{I}}$ (in the order of increasing j) as all those elements of Δ that are in $D_j^{\mathbf{I}}$. Such definition is possible due to the non-recursive property of views that ensures that D_j does not contain any V_i for $i \geq j$.

---

[19] Recall that our assumption that the set of primitive attributes is recursively enumerable ensures that such a sequence exits.

Note that since rules of *Exp* (resp. steps of *Subsumes*) either do not violate properties of a well-formed graph or invoke *Merge* (resp. *Merge* and *Exp*) in the end, they produce a well-formed graph given one as input. (The graph remains finite since each rule makes at most a finite number of modifications to nodes, edges, and descriptions inside α sets.)

In addition, since graph $G_{-2} = Init(D_1)$ just consists of one node, it is in fact well-formed. Therefore, by the discussion above, graphs G_{-1} and G_0 must be also well-formed. It follows that we do not have to call *Merge* during the steps of construction of G_i ($i \geq 1$) to ensure that G_i is well-formed. Indeed, our construction step only adds a finite number of non-**Id** edges outgoing from nodes that do not have other outgoing edges with the same label. Therefore, this addition of new edges as well as the invocation of *Exp* that follows preserve the properties of a well-formed graph. (Considering the facts that (a) G_0 is finite by Theorem 5.1, (b) there is a finite number of attributes in sequence $[A_1, \ldots, A_i]$, and (c) each of the rules E1 through E7 and M1 through M3 modifies at most a finite number of nodes/edges/descriptions in the graph, every G_i is finite.)

Next, in order to distinguish different invocations of *Exp* procedure during the construction of G_i ($i \geq 1$), we use a subscript to denote the step at which the procedure is executed. Thus, Exp_i ($i \geq 1$) denotes the invocation of *Exp* procedure at step i. Since G_{-1} is a result of an application of *Exp*, steps S1, S2, and S6 of *Subsumes* do not modify the graph, while steps S3, S4 and S5 invoke *Exp* in the end, G_0 is also a result of an invocation of the procedure *Exp*. We denote that invocation by Exp_0.

Let us now make a number of assertions (and present their proofs) about the process of construction and the resulting interpretation **I**.

Assertion 1. *E3, E4, and E5 are never applied during the execution of* Exp_i *for any* $i \geq 1$.

Let us first show that any description D that contains **same-as** constructor(s) inside its unfolded form $D_{[u]}$ must be eliminated by Exp_0 (if it still exists by the time Exp_0 is called) and no such descriptions can be added afterwards. Observe that for every view name V in an α set of a node in G_0, V must be also in *fired* set of that node since otherwise rule E7 would still be applicable after Exp_0 terminates. It follows that if a **same-as** constructor is inside $V_{[u]}$ for some view name V inside an α set, it must also be inside $D_{[u]}$ for some description D added by corresponding application(s) of rule E7. Therefore, if a **same-as** constructor is inside an unfolded description $D_{[u]}$ for some description D in an α set of a node in G_0, there are three possible cases: D is a **same-as** description, D is an **and** description, or D is an **all** description. However, in the first case, rule E4 would still be applicable to G_0; in the second case, E1 would still be applicable; and in the third case, either E2 or E3 would be applicable. Moreover, no step during our construction process can possibly add a description D with the **same-as** constructor inside $D_{[u]}$ (recall that E6 cannot add such descriptions due to the restrictions on the constraints inside T, and

E7 cannot "extract" such descriptions since all of them would be eliminated by Exp_0 by reasoning above).

It follows that a description D with the **same-as** constructor inside $D_{[u]}$ never occurs inside an α set of a node in G_i for $i \geq 0$, which in turn implies that rules E3 and E4 are not applicable during the execution of $Exp_i \; \forall \; i \geq 1$.

Let us now show that E5 is also not applied during any invocation of Exp_i ($i \geq 1$). Indeed, assume the opposite, and let i ($i \geq 1$) be the first step at which E5 "fires" for the first time and let $G' = \langle N', E', Refs' \rangle$ be the graph right before E5 "fires" for the first time. Note that since rules E3 and E4 are not applied as we showed above, besides the nodes and edges added by construction step(s) before invocation(s) of *Exp*, E5 is the only rule that can add new nodes and edges. Thus, G' can differ from G_{i-1} only by nodes and edges added before the invocation of *Exp* during step i and by α sets that could be changed by applications of rules E1, E2, E6 and E7 during the invocation of Exp_i prior to firing rule E5 for the first time. Moreover, G' must have the same nodes and edges as G_0 in addition to nodes and edges added by a number of construction steps that form non-overlapping "trees" in following sense. Let N_{added} denote the set of nodes added to G_0 by the construction step(s) 1 through i, i.e.

$$N_{added} = \bigcup_{i \geq j \geq 1} N_j'.$$

Then, by our construction and the fact that nodes and edges added during the construction steps 1 through $i - 1$ are not modified, $N' = N_0 \cup N_{added}$, every node in N_{added} has exactly one parent and all its descendents, if any, are also in N_{added}. A graph satisfying this *tree property* is illustrated in Figure 5.7.

Without loss of generality, let us assume that rule E5 is applied to distinct nodes n_1 and n_2 in N' for a description (**fd** $C \; Pf_1 \; \ldots \; Pf_m \; Pf$) in n_1. There are three possible cases: (i) n_1 and n_2 are both in N_0; (ii) one of n_1 and n_2 is in N_0 and the other is in N_{added}; and (iii) both n_1 and n_2 are in N_{added}. Let us prove that all three cases lead to contradiction.

First, consider case (i). We first show that the same rule E5 must have fired during the invocation of Exp_0. Indeed, condition (1) of E5 is satisfied independently of the construction process. If condition (2) and the fact that (**fd** $C \; Pf_1 \; \ldots \; Pf_m \; Pf$) is in $\alpha(n_1)$ are true in G', they must be also true in G_0 since no step j ($j \geq 1$) of the construction process can add a description to an α set of a node in N_{j-1} only by using rules E1, E2, E6, and E7 in Exp_j (tree property ensures that there is no edge from a node in N_j' to a node in N_{j-1}). In addition, since only rules E1, E2, E6 and E7 can be applied during the steps $1 \leq j \leq i$, Exp_j do not change nodes and edges of the graphs. Therefore, condition (4) is satisfied by G' only if it is true in G_0. Finally, condition (3) is also satisfied by G' only if it is satisfied by G_0 since the tree property ensures that no two paths from

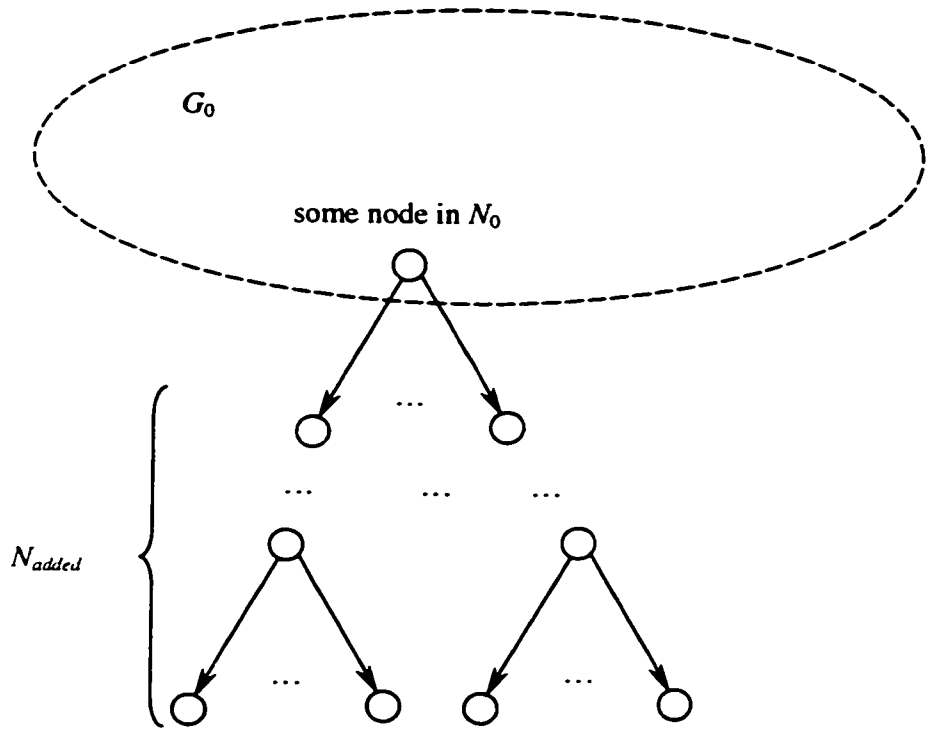


Figure 5.7: An i-th step in construction of the graph in the proof of completeness.

nodes in N_0 can go to the same node in N_{added} without having a prefix that goes to the same node in N_0 (or intuitively, if two nodes in N_0 "agree" on a path in G', they must also agree on a prefix of that path in G_0). Indeed, tails of such two paths that go from nodes in N_0 to a node in N_{added} would have to be the same, since every node in N_{added} has at most one parent. Therefore, the paths must have heads of the same length that go to the same node in N_0 which, in turn, does not have any ascendants in N_{added}, and thus, both heads must exist in G_0 as well. Therefore, Exp_0 would not terminate without applying E5 rewrite rule first. This application, however, would create paths p_1 and p_2 from nodes n_1 and n_2 to the same node such that $\gamma(p_1) = \gamma(p_2) = Pf$. Existence of such paths would then violate the fact that condition (4) is satisfied by G_0.

Cases (ii) and (iii) cannot occur either, since by the tree property, a path p starting from a node in N_{added} would only pass through nodes in N_{added}. However, since each node in N_{added} has only one parent, there can only be one node from which a path of the same length as p can originate. Thus, nodes n_1 and n_2 cannot be distinct.

Therefore, rule E5 cannot be applied during any step i ($i \geq 1$) in the construction of S_D.

As we mentioned before, this Assertion implies, in particular, that $N_i = N_{i-1} \cup N_i'$ and $E_i = E_{i-1} \cup E_i'$ $\forall$ $i \geq 1$ since rules E1, E2, E6 and E7 do not modify nodes and edges in a graph.

Assertion 2. *No step i for $i \geq 1$ adds/removes any primitive concepts or* **fd** *descriptions to/from* α *sets of nodes in* N_{i-1}.

This Assertion is a consequence of the facts that every construction step just adds new nodes and edges outgoing from already existing nodes, and by Assertion 1, only rules E1, E2, E6 and E7 can be applied during the execution of Exp_i. First, note that neither of the four rules E1, E2, E6 and E7 can remove a primitive concept or an **fd** description from an α set of a node. In addition, note that invocation of *Exp* procedure that created G_{i-1} must have exhaustively explored all applicable rules. Therefore, no α set of a node n in G_{i-1} can contain an **and** description, an (**all** A D) description if there is an edge outgoing from n and labeled A, a primitive concept C such that there is a constraint $C < D$ in T which has not "fired" and added C into the *fired*(n) set, or a view name V such that there is a view definition $V \equiv D$ in T which has not "fired" and added V into the *fired*(n) set. Thus, the only rules that might be applicable during the invocation of Exp_i are E2 rules that remove (**all** A D) descriptions from α sets of nodes in N_{i-1} and add descriptions (D) to α sets of their newly created children in N_i'. After that, neither of the rules E1, E2, E6 or E7 that apply to the nodes in N_i' can possibly affect α sets of other nodes. Therefore, Exp_i cannot possibly add new primitive concepts or **fd** descriptions (as well as any other descriptions) to an α set of a node in N_{i-1}.

In particular, this Assertion implies that if i is the smallest non-negative integer such that a node n is in N_i, steps $i + 1$, $i + 2$, etc. do *not* change the set of primitive concepts in the α set of n; or in other words, $n \in C^{\mathbf{I}}$ for our constructed interpretation **I** if and only if $C \in \alpha(n, G_i)$.

Assertion 3. *For every node $n_1 \in \Delta$, and every $A_i \in S_{PA}$, there is at least one node $n_2 \in \Delta$ such that $(n_1, n_2) \in A^{\mathbf{I}}$.*

Indeed, let j be the smallest non-negative number such that G_j contains n_1 (such j exists since $n_1 \in \Delta$). If there is an edge $\langle n_1, A_i, n_2 \rangle$ in E_j for some node n_2 in N_j, (n_1, n_2) must be in $A_i^{\mathbf{I}}$ by the definition of **I**. Otherwise, a new node n_2 and edge $\langle n_1, A_i, n_2 \rangle$ are created by construction step $max(i, j + 1)$, and thus, (n_1, n_2) must be in $A^{\mathbf{I}}$.

Assertion 4. *For every node $n_1 \in \Delta$, and every $A_i \in S_{PA}$, there is at most one node $n_2 \in \Delta$ such that $(n_1, n_2) \in A^{\mathbf{I}}$.*

By Assertion 3, there must be a smallest non-negative integer j such that G_j contains node n_1 and an edge $\langle n_1, A_i, n_2 \rangle$ for some node n_2 in N_j. Therefore, it is sufficient to show that (i) G_j contains at most one edge labeled A_i and outgoing from n_1, and (ii) no G_{j+k} ($k \geq 1$) contains any

other edges labeled A_i and outgoing from n_1. However, (i) follows from the facts that G_{-1} is a result of an application of *Exp* procedure to a well-formed graph, and the graph remains well-formed after each step of the construction process. On the other hand, (ii) follows from Assertion 1 and definition of the construction process since they ensure that edge $\langle n_1, A_i, n_2 \rangle$ is never modified by any step $j + k$ and no new edge with label A_i is ever added to node n_1.

Assertion 5. **I** *is a valid interpretation.*

This Assertion is a straightforward consequence of Assertions 3 and 4 since they ensure that primitive attributes are total functions on Δ.

Note that proofs of Assertions 1 through 5 would apply to an arbitrary construction process that started with any well-formed graph G_{-2} and not necessarily $Init(D_1)$. This fact will be used in the proofs of assertions that follow.

Consider an arbitrary well-formed graph G_{-2} and the process of construction of graph G_0 (as described above) which is based on the invocation of $Subsumes(D_2, Exp(G_{-2}, T), T)$ that returns false. When the rules of *Merge*, *Exp* and steps of *Subsumes* are applied to G_{-2}, new "intermediate" graphs are obtained such that the last intermediate graph is G_0. The following Assertion establishes that if a description D is ever added to an α set of a node of either an intermediate graph or graph G_i ($i \geq 0$), the "corresponding" node n in Δ must be in $D^{\mathbf{I}}$. In other words, if a description D is ever added to an α set of a node during the construction of G_0 or by the construction step CS_i ($i \geq 1$), the object of Δ that corresponds to the node (i.e. either the node itself or a result of its merges with other nodes) must be in $D^{\mathbf{I}}$.

Assertion 6. *Consider the process of construction of graph G_0 starting from an arbitrary well-formed description graph G_{-2} as described above. Also, consider an arbitrary intermediate graph $G = \langle N, E, Refs \rangle$, a reference $\langle ref_1, n_1 \rangle$ in the Refs set and a node n in $G_0 = \langle N_0, E_0, Refs_0 \rangle$ such that $\langle ref_1, n \rangle \in Refs_0$ and ref_1 is* not *dn. Then, if $D \in \alpha(n_1, G)$ for some description D, $n \in D^{\mathbf{I}}$. In addition, if $D \in \alpha(n, G_i)$ for some $i \geq 0$, $n \in D^{\mathbf{I}}$.*

Note that we use references instead of nodes in the statement of the Assertion when we deal with the intermediate graph since its nodes are *not* necessarily in Δ; however, the property still holds for the corresponding nodes that might be a result of merge(s) of the original nodes with some other nodes. In addition, we prove the property only for references that are *not dn* since *dn* is modified by S3, and thus we "lose" the node for which the property holds. Finally, without loss of generality, by part (a) of Claim 5.4, we assume in the rest of the proof that every node in G_{-2} is pointed to by some reference (distinct from *dn*) in the set *Refs*, and whenever we add a new node to the graph, we also a new reference to that node.

Our plan is to first prove the Assertion for the cases when D has one of the forms

(a) C (for a primitive concept C),
(b) (**same-as** Pf_1 Pf_2), or
(c) (**fd** C Pf_1 ... Pf_m Pf).

Then, using the fact that for any description D, there is a finite number of view substitutions to obtain $D_{[u]}$ from D, we show that the Assertion holds for an arbitrary description D by induction on the number of substitutions necessary to obtain $D_{[u]}$ plus the number of concept constructors in $D_{[u]}$.

Let us first consider case (a) for the part of the Assertion that deals with an intermediate graph G, i.e. when $\alpha(ref_1, G)$, or $\alpha(n_1, G)$, contains a primitive concept C. The only way node n_1 can disappear is in a step M1 of *Merge* procedure. However, if this happens, concept C would "move" to the new node along with the reference ref_1. Additionally, no rule or step applied during our process removes primitive concepts from the α set of any node. Therefore, if $\langle ref_1, n\rangle \in Refs_0$, $C \in \alpha(n, G_0)$, and thus, by Assertions 1 and 2 and definition of **I**, $n \in C^{\mathbf{I}}$. In addition, Assertions 1 and 2 and the definition of **I** imply that if $C \in \alpha(n, G_i)$ *for some* $i \geq 0$, $n \in C^{\mathbf{I}}$.

In case (b), it follows from the proof of Assertion 1 that no α set of any node in G_i (for all $i \geq 0$) can contain a description with the **same-as** constructor inside its unfolded form. Therefore, it is sufficient to consider case (b) for the part of the Assertion that deals with an intermediate graph G, i.e., when $\alpha(ref_1, G)$, or $\alpha(n_1, G)$, contains D = (**same-as** Pf_1 Pf_2). As in case (a), D moves with the reference to the node whenever the node is merged by M1. Therefore, the only other rule that deals with description D is E4. Moreover, rule E4 has to be applied to the description during an invocation of *Exp* procedure. (Note that unless D is removed by an application of E4 before Exp_0 is invoked, Exp_0 has to apply E4.) When E4 is applied, it creates two paths, say p_1 and p_2, that start at ref_1, end at the same node (say referenced by ref), and such that $\gamma(p_1) = Pf_1$ and $\gamma(p_2) = Pf_2$. It follows from part (c) of Claim 5.4, however, that paths p_1 and p_2 starting at ref_1 and ending at ref are "preserved" by all other rules and constructions that are applied to the graph, and thus, both paths would exist in G_0. Therefore, by our construction of the interpretation **I**, since the edges are "translated" into interpretations of the attributes, node n referenced by ref_1 in $Refs_0$ must belong to $D^{\mathbf{I}}$ = (**same-as** Pf_1 Pf_2)$^{\mathbf{I}}$.

Next, consider case (c) for the part of the Assertion that deals with an intermediate graph G, i.e. when $\alpha(ref_1, G)$, or $\alpha(n_1, G)$, contains D = (**fd** C Pf_1 ... Pf_m Pf). Just as in case (a), we can show that description D must be in the α set of the node (n) referenced by ref_1 in G_0. By Assertion 2, the **fd** description would stay in $\alpha(n)$ during the rest of the construction process.

Let us now assume that $n \notin D^{\mathbf{I}}$, or in other words, that there is a node $n_2 \in C^{\mathbf{I}}$ such that $Pf_j^{\mathbf{I}}(n) = Pf_j^{\mathbf{I}}(n_2)$ for $1 \leq j \leq m$, whereas $Pf^{\mathbf{I}}(n) \neq Pf^{\mathbf{I}}(n_2)$. Then, considering Assertions 1 and 2, it follows that there must exist a graph G_l ($l \geq r$), and nodes $n_2, n_1', \ldots, n_m', n_{m+1}', n_{m+2}'$, such that

(i) $n, n_2, n_1', \ldots, n_m', n_{m+1}', n_{m+2}'$ are in N_l;
(ii) $C \in \alpha(n_2, G_l)$;
(iii) for every $1 \leq j \leq m$, $Reachable(n, Pf_j, n_j')$ and $Reachable(n_2, Pf_j, n_j')$ are true;
(iv) $Reachable(n, Pf, n_{m+1}')$ and $Reachable(n_2, Pf, n_{m+2}')$ are true; and
(v) n is distinct from n_2, and n_{m+1}' is distinct from n_{m+2}'.

These conditions, however, are contradictory since Exp_l would make sure that rule E5 would be applied if (i) through (v) were satisfied and that application of E5 would violate (iv) and (v) afterwards.

If $n \in G_i$ for some $i \geq 0$, and (**fd** C Pf_1 ... Pf_m Pf) $\in \alpha(n, G_i)$, we can use the same argument to deduce that E5 would be applied whenever there was a node that could possibly violate the **fd** constraint. (Note that Assertion 1 implies that new nodes are added carefully by CS_i, $i \geq 1$, and do not violate any **fd** constraints.) Thus, we would again conclude that n must be in $D^{\mathbf{I}}$.

To formulate the inductive hypothesis, let us define *the size of D*, denoted $|D|$, as the number of necessary view substitutions to obtain $D_{[u]}$ from D plus the number of concept constructors in $D_{[u]}$. Next, we make the inductive assumption that the Assertion holds for any description D of size no greater than m for some $m \geq 1$. (Note that cases (a) through (c) are the base case for the induction.)

For the inductive step, consider a description D with size $m + 1$. Since $m + 1 \geq 2$, D has to have one of the following forms:

(d) (**all** B D'),
(e) (**and** D_1' D_2' ... D_t') for some $t \geq 2$, or
(f) V for some view definition $V \equiv D_V$ in T.

First, consider case (d) for the part of the Assertion that deals with an intermediate graph G, i.e. when $\alpha(ref_1, G)$, or $\alpha(n_1, G)$, contains D = (**all** B D'). Just as in case (a), we can show that the description D must be in the α set of the node referenced by ref_1 until either rule E2 or E3 apply to the description. Note that since no nodes are merged at step i for all $i \geq 1$ by Assertion 1, ref_1 will always point to the same node (n) after G_0 is constructed. It follows then from Assertion 3 that an edge labeled B and outgoing from ref_1, say $\langle ref_1, B, ref_2 \rangle$, is going to be added at some point, and procedure *Exp* will apply after that. Therefore, either E2 or E3 must fire on the description D. Both rules remove D from the node referenced by ref_1 and add D' to the α set of the node referenced by ref_2. Since $|D'| \leq m$, however, by our inductive assumption, node referenced by ref_2 is in $D'^{\mathbf{I}}$. Then, by definition of **I** and the fact that $\langle ref_1, B, ref_2 \rangle$ gets preserved by part (c) of Claim 5.4, it follows that n must be in (**all** B D')$^{\mathbf{I}}$.

Proof for the part of the Assertion that deals with graphs G_i (for $i \geq 0$) is analogous and in fact easier since by Assertion 1, one does not have to worry about keeping track of nodes by references and can reason directly about nodes.

Next, consider case (e) for the part of the Assertion that deals with an intermediate graph G, i.e. when $\alpha(ref_1, G)$, or $\alpha(n_1, G)$, contains $D = (\textbf{and } D_1' \; D_2' \; \ldots \; D_t')$ for some $t \geq 2$. As before, we note that D must be in the α set of the node referenced by ref_1 until rule E1 applies to the description. (E1 does apply to D since it is the only rule applicable to an **and** description and there is always a chance for E1 to "fire" inside Exp_0 if it does not apply before.) When E1 fires however, it inserts descriptions D_i' $(1 \leq i \leq t)$ into the node referenced by ref_1, and therefore, since the size of each D_i' is no greater than m, by our inductive assumption, $n \in D_i'^{\mathbf{I}}$ $(1 \leq i \leq t)$, which in turn implies that $n \in (\textbf{and } D_1' \; D_2' \; \ldots \; D_t')^{\mathbf{I}}$.

As with case (d), proof for the part of the Assertion that deals with graphs G_i (for $i \geq 0$) is analogous. In this case, however, the fact that E1 fires is guaranteed by invocation of Exp_i at the end of each CS_i step.

Finally, consider case (f) for the part of the Assertion that deals with an intermediate graph G, i.e. when $\alpha(ref_1, G)$, or $\alpha(n_1, G)$, contains $D = V$ for some view definition $V \equiv D_V$ in T. Similarly to the reasoning above, we note that V is never removed from the α set of the node referenced by ref_1, and that rule E7 must apply to the description. (E7 does apply to V since it is the only rule applicable to the description and there is always a chance for E7 to "fire" inside an invocation of *Exp*.) When E7 fires, it adds D_V to the α set of the node referenced by ref_1, and therefore, since the size of D_V is no greater than m, by our inductive assumption, $n \in D_V^{\mathbf{I}}$, and thus, $n \in V^{\mathbf{I}}$.

As before, proof for the part of the Assertion that deals with graphs G_i (for $i \geq 0$) is analogous.

Now, to complete the proof of completeness we are going to prove that **I** satisfies T, whereas there is a node in G_0 that is in $D_1^{\mathbf{I}}$ but is *not* in $D_2^{\mathbf{I}}$. The node we are interested in is the distinguished node of G_{-2}. *Subsumes*, however, may change the *dn* reference to that node in step S3. Therefore, in order to keep a reference to the "original" distinguished node, we assume without loss of generality (by part (a) of Claim 5.4 and the fact that applicability as well as results of steps of *Subsumes* do not depend on a reference that is not *dn*), that there is an additional reference dn_0 to the distinguished node of G_{-2}.

Assertion 7. *The node referenced by dn_0 in G_0 is in $D_1^{\mathbf{I}}$, and* **I** *satisfies T.*

The first part of the Assertion follows directly from Assertion 6, since $D_1 \in \alpha(dn_0, G_{-2})$, and thus, the node referenced by dn_0 in $Refs_0$ must be in $D_1^{\mathbf{I}}$. (Note that $Init(D_1)$ is a well-formed graph since it only contains one node.)

The second part also follows from Assertion 6. Indeed, let us assume to the contrary that there is a constraint $C < D \in T$ and a node $n \in \Delta$ such that $n \in C^{\mathbf{I}}$ and $n \notin D^{\mathbf{I}}$. By our definition of **I**, it follows that there must exist a graph G_i, where i is the smallest non-negative integer such that $n \in G_i$. Assertion 2 implies that C is in $\alpha(n, G_i)$. Then, however, Exp_i (or possibly an earlier invocation of Exp if $i = 0$) would apply rule E6 to node n and constraint $C < D$ since this is the only way for C to be added to $fired(n)$, and until C is in $fired(n)$, rule E6 is applicable. Thus, D must have been added to $\alpha(n)$. By Assertion 6, however, it follows that $n \in D^{\mathbf{I}}$, contrary to the assumption. Therefore, **I** satisfies every subsumption constraint $C < D$ in T. The fact that **I** satisfies every view definition $V \equiv D$ in T directly follows from the definition of $V^{\mathbf{I}}$ that consists of all nodes that are in $D^{\mathbf{I}}$.

Assertion 8. *The node referenced by dn_0 in G_0 is* not *in $D_2^{\mathbf{I}}$.*

To prove the Assertion, let us make a small generalization of the construction process as in Assertion 6 and consider an arbitrary well-formed description graph G_{-2} instead of $Init(D_1)$. Also, let us assume that dn_0 is a reference in $Refs_{-2}$ that points to the same node as dn, $Subsumes(D_2, G_{-1}, T)$ returns false where $G_{-1} = Exp(G_{-2}, T)$, and graphs G_0, G_1, etc. and the interpretation **I** are constructed in the same way as described by our construction process. Clearly, to prove Assertion 8, it is sufficient to show that the node referenced by dn_0 in G_0 is *not* in $D_2^{\mathbf{I}}$ for this more general construction process.

We prove this more general assertion by induction on the number of steps executed by $Subsumes(D_2, G_{-1}, T)$. For the base case, we consider invocations of *Subsumes* that (return false and) execute only one step, and therefore, do not make any recursive calls. Therefore, D_2 must have one of the forms (a) C, (b) (**same-as** Pf_1 Pf_2), or (c) (**fd** D Pf_1 ... Pf_m Pf). (Note that D_2 cannot be a view name since S6 only returns true if it does not make a recursive call.)

In case (a), since *Subsumes* returns false by step S1, C is *not* in $\alpha(dn_0)$. Therefore, by Assertion 2, the node referenced by dn_0 in G_0 is *not* in $C^{\mathbf{I}}$.

In case (b), *Subsumes* executes S4. Thus, G_0 contains paths p_1 and p_2 that start at dn_0 and where $\gamma(p_1) = Pf_1$ and $\gamma(p_2) = Pf_2$. Since *Subsumes* returns false and considering part (c) of Claim 5.4, the two paths do *not* end at the same node in G_0, and by Assertion 1, the paths stay the same. Thus, by our construction of **I**, dn_0 is *not* in (**same-as** Pf_1 Pf_2)$^{\mathbf{I}}$.

Next, consider case (c). *Subsumes* adds a new node referenced by *fdcn*, and adds D to $\alpha(fdcn)$. For every $1 \leq j \leq m$, S5 creates some paths p_j' and p_j'' from dn_0 and *fdcn* respectively to some new node, say, referenced by ref_j. Then, it invokes *Merge* and *Exp*. Since *Subsumes* returns false, there are no two paths p' and p'' in G_0 that start at dn_0 and *fdcn*, respectively, end at the same node, and such that $\gamma(p') = \gamma(p'')$ $(= Pf')$ and Pf' is some prefix of Pf. Since our construction of S_D only adds trees of nodes under already existing nodes, and since by Assertion 1, no nodes are merged after they are added, it follows that there will never be such paths p' and p''. On the other hand, nodes referenced by dn_0 and *fdcn* will always "agree" on paths $Pf_1, \ldots, Pf_m$ by part (c) of Claim 5.4. Therefore, since node referenced by *fdcn* is in $D^{\mathbf{I}}$ by Assertion 6, node referenced by dn_0 is *not* in $(\textbf{fd}\ C\ Pf_1 \ldots Pf_m\ Pf)^{\mathbf{I}}$.

Next, for inductive hypothesis, let us assume that if $Subsumes(D_2, G_{-1}, T)$ executes p steps for some description D_2 and returns false, dn_0 is not in $D_2^{\mathbf{I}}$ for any $p \leq m$ for some $m \geq 1$. Consider what happens when an invocation of $Subsumes(D_2, G_{-1}, T)$ returns false after executing $m + 1$ steps. Since *Subsumes* executes at least 2 steps ($m + 1 \geq 2$), it makes at least one recursive call. It follows that D_2 has one of the forms (d) (**all** B D) for some primitive attribute B and a description D, (e) (**and** D_1' D_2' $\ldots$ D_t') for some $t \geq 2$ and descriptions D_i' $(1 \leq i \leq t)$, or (f) V for some view name V with corresponding view definition $V \equiv D$ in T.

In case (d), *Subsumes* executes S3 which adds a new edge labeled B and calls procedures *Merge* and *Exp*. We obtain a graph with path $\langle dn_0, B, n\rangle$ for some node n. Note that as was discussed above, after invocation of *Merge* by part (d) of S3, the graph remains well-formed. Therefore, we can consider it as G_{-2}, and the graph resulting after invocation of *Exp* procedure (by part (e) of S3) as G_{-1}. In addition, considering that the recursive call to *Subsumes* passes n as the new distinguished node and returns false, and since the recursive call to *Subsumes* executes at most m steps, if we consider a reference *ref* to n as dn_0 *inside* the recursive call, we can use the inductive assumption to conclude that the node referenced by *ref* in G_0 is *not* in $D^{\mathbf{I}}$. It follows however, that dn_0 is not in $(\textbf{all}\ B\ D)^{\mathbf{I}}$ since $\langle dn_0, B, ref\rangle$ exists in G_0 by part (c) of Claim 5.4.

In case (e), *Subsumes* applies S2. Since one of the recursive calls to S1, S4 or S5 returns false, we can assume without loss of generality that D_i' is the description such that S1, S4 or S5 returns false inside the recursive call to *Subsumes* that passes D_i' as the parameter description. In other words, G_0 is obtained somewhere inside the call to $Subsumes(D_i', G, T)$ where G is the graph obtained from G_{-1} by previous steps of *Subsumes*. Since steps S1, S2, and S6 do not modify the graph, steps S2, S3, and S4 call *Merge* and then *Exp* in the end, and by Claim 5.2, it follows that G is a result of an application of *Exp* to some well-formed graph G_w. In addition, note that although reference *dn* might be changed by step S3 in the recursive calls to *Subsumes* preceding the invocation of $Subsumes(D_i', G, T)$, S3 makes sure that *dn* is restored, and thus points to the same node as dn_0 when the recursive call is made. Therefore, considering G_w as G_{-2}, graph G as G_{-1}, and taking into account the fact that $Subsumes(D_i', G, T)$ executes at most m steps, we can use the inductive assumption to conclude that dn_0 is *not* in $D_i'^{\mathbf{I}}$. Therefore, dn_0 cannot be in $(\textbf{and}\ D_1' \ldots D_i' \ldots D_t')^{\mathbf{I}}$ either.

Finally, consider case (f). *Subsumes* applies step S6, and since *Subsumes* returns false, it must make a recursive call to *Subsumes*(D, G_{-1}, T) that executes m steps. Therefore, using our inductive assumption, we conclude that dn_0 is not in $D^{\mathbf{I}}$, and thus not in $V^{\mathbf{I}}$.

Therefore, by Assertions 5, 7 and 8, we constructed an interpretation **I** that satisfies T and contains a node that is in $D_1^{\mathbf{I}}$ but *not* in $D_2^{\mathbf{I}}$. In other words, **I** satisfies T and does *not* satisfy constraint $D_1 < D_2$.

■

5.6.4 Runtime complexity

Consider a problem of determining whether a constraint "$D_1 < D_2$" is logically implied by a terminology T. Let m denote the encoding length of the problem; that is, the sum of the encoding lengths of descriptions $D_{1[u]}$ and $D_{2[u]}$, and k denote the encoding length of T. Our procedures solve the problem by first constructing graph $G = Exp(Init(D_1), T)$ and then invoking procedure *Subsumes*(D_2, G, T). We start the analysis by estimating the time complexity of procedure *Exp* without counting the time spent on the invocations of procedure *Merge*. Next, we estimate the total "cost" of all invocations of *Merge*, and conclude the analysis by evaluating the time spent by *Subsumes*.

First, consider an implementation of *Exp* that consists of two phases: the first removes all **same-as** constructors from the unfolded descriptions of all α sets by applying rules E1, E2, E3, E4 and E7; and the second starts with the first application of either rule E5 or E6. This is well defined since no rule of *Exp* introduces new **same-as** constructors inside the (unfolded versions of) descriptions in the α sets, and one can remove all **same-as** constructors by applying rules E1 through E4 and E7 first. Indeed, if a **same-as** constructor is nested inside an (unfolded version of a) description in an α set, either rule E1, E2, E3 or E7 will be applicable and will reduce the level of nesting, whereas rule E4 will remove an occurrence of **same-as** when it is unnested. Moreover, implementation of this way of eliminating occurrences of **same-as** is straightforward by means of a recursive procedure that starts at the distinguished node of the graph and recursively applies rules E1 through E3 and E7 to all descriptions containing an occurrence of same-as inside their unfolded forms, and then applying E4 at the base case of the recursion.

Let G_0 denote $Init(D_1)$, and G_1 denote a graph obtained by the first phase. It is not hard to see that the number of nodes, edges, and combined sizes of α sets of G_1 are $O(m)$. Indeed, graph transformations only consist of (a) removing some **and** descriptions by E1, (b) turning some **all** descriptions into edges with corresponding labels by E2 and E3, and (c) removing all same-as descriptions (by E4) while creating paths with edge labels corresponding to the primitive attributes inside the eliminated **same-as** descriptions. Thus, the number of created edges does *not* exceed the number of occurrences of primitive attributes in $D_{1[u]}$, and α sets of nodes only contain

"components" of $D_{1[u]}$ without repetition. In addition, since the graph remains connected, the number of nodes does not exceed the number of edges by more than one.

Note that the above analysis does not take invocations of *Merge* by E4 into account. Therefore, even without merging nodes and edges inside *Merge*, there would be O(m) edges and nodes in the graph. In addition, our recursive procedure for obtaining G_1 takes O(1) time to find an applicable rule. Moreover, the total cost of all applications of rules E1 through E4 and E7 (without counting the time spent on invocations of *Merge* by E4) is O(m) since $D_{1[u]}$ can be broken up into at most O(m) "pieces". Therefore, the total time to obtain G_1 is O(m) if we do not count the time spent on *Merge*.

Now consider what happens during the second phase. *Exp* proceeds to exhaustively apply rules E1, E2, E5, E6, and E7 to obtain graph $G_2 = G = Exp(Init(D_1), T)$. (Rules E3 and E4 are no longer applicable since no descriptions that contain **same-as** constructors inside their unfolded versions are in or can be introduced to G_1.) As was shown in the proof of termination, G_1, G_2 and all intermediate graphs that are produced during the second phase satisfy the boundary property (with the structure depicted in Figure 5.4(a)) where nodes that relate to R_{old} are essentially nodes of G_1 (possibly merged with other nodes), whereas nodes that relate to R_{new} are all other nodes. In addition, the proof establishes that new nodes and edges can only be produced by applications of rule E5.

Next, to obtain a bound on the complexity of the second phase, consider an event driven implementation of *Exp*. In the following description, procedure keeps track of possible future applications of rule E5 *before* all conditions for such applications are satisfied. Using an efficient way of obtaining all nodes that contain a given concept name in their α set, procedure *Exp* would start "waiting" for E5 to fire whenever an **fd** description

$$D = (\mathbf{fd}\ C\ Pf_1\ \ldots\ Pf_l\ Pf)$$

is added to a node (n_1). Similarly, using an efficient way of obtaining all nodes that contain **fd** descriptions with a given concept name in their α sets, the procedure would start "waiting" for E5 to fire whenever a concept C is added to an α set of a node (n_2) different from a node n_1 with D in its α set. Thus, whenever a situation arises with some nodes n_1 with $D \in \alpha(n_1)$ and n_2 with $C \in \alpha(n_2)$, the procedure would create a record $\langle n_1, n_2, Pf_1, \ldots, Pf_l, Pf\rangle$ and check for paths with edge labels corresponding to every Pf_i starting at nodes n_1 and n_2. For every node n_1' (resp. n_2') in a path that starts at node n_1 (resp. n_2) and has edge labels corresponding to a prefix of Pf_i for which there exists a corresponding path starting at node n_2 (resp. n_1), the procedure would make the node n_1' (resp. n_2') "wait" for a match (i.e. a merge) with the corresponding node n_2' (resp. n_1'). In addition, if no "corresponding" node exists for a node along one of the paths, the last node at the end of the other path is set to wait for an addition of the next appropriate edge along the path. When the edge is added, the two nodes start waiting for a match as well as for the next edge along one (or both) path(s) if there are more edges in Pf_i. Next, when at least one of the

matches occurs, i.e. two same-length prefixes of the paths matched, Pf_i is set to be "satisfied" in the record. Finally, when all Pf_i are satisfied, E5 can fire by adding paths with labels corresponding to Pf, starting at n_1 and n_2 and ending at the same node.

With this implementation, a check for a possible application of an **fd** constraint on any particular pair of nodes as well as "firing" the constraint will take at most $O(t)$ time, where t is the size of its encoding. Now observe that there are two kinds of **fd** constraints: those introduced by the terminology and those occurring in description $D_{1[u]}$. Since (a) by the proof of termination, rule E5 can only be applied to nodes that relate to R_{old}; (b) there are at most $O(m)$ such nodes; and (c) the sum of the encoding lengths of the **fd** descriptions inside terminology that can cause rule E5 to fire on a pair of nodes is $O(k)$, it takes $O(m^2 \cdot k)$ time to process applications of rule E5 that use **fd** descriptions introduced by the terminology. Conversely, each **fd** description inside $D_{1[u]}$ is added to at most one α set in the graph. Therefore, it can cause at most $O(m)$ firings of rule E5. However, the sum of lengths of **fd** descriptions inside $D_{1[u]}$ does not exceed $O(m)$. It follows that the total cost to process **fd** descriptions that come from $D_{1[u]}$ is $O(m^2)$, which in turn implies that the time to check for an applicability of *all* **fd** constraints as well as to fire them is $O(m^2 \cdot k)$.

We can also use the event driven approach to efficiently implement firing of rules E1, E2, E6 and E7. Whenever an **all** description is added to a node, the node starts "waiting" for the appropriate edge to be added. On the other hand, when a concept name, a view name, or an **and** description are added to an α set, rule E6, E7, or E1 respectively, is ready to fire and can be directly added to a job queue.

Thus, *Exp* can employ a job queue to contain all rules that are ready to fire. As a job gets processed, new rules become available for firing and other rules make progress in satisfying their conditions. The procedure ends when no more rules can fire.

With such an implementation, the checks for possible applications of rules E1, E2, E6 and E7 as well as firing them will require $O(t)$ time for any given node, where t is the size of the encoding of the appropriate part of the description.

Note that by the boundary property introduced in the proof of termination, the number of edges and nodes in the graph after each application of a rule of *Exp* does not exceed $O(m \cdot (m + k))$, since there are $O(m)$ edges and nodes in R_{old} and each of the nodes can have at most $O(m + k)$ outgoing edges labeled with an attribute name—there are $O(k)$ attributes in terminology and $O(m)$ attributes in D_1. Moreover, none of the rules of *Exp* can add more than $O(m + k)$ new edges and nodes to the graph. Therefore, the total number of edges and nodes in the graph never exceeds $O(m \cdot (m + k))$. Thus, since each node can introduce a number of descriptions by rules E6 and E7 with total encoding size of $O(k)$, the total cost of firing rules E1, E2, E6 and E7 is $O(m \cdot k \cdot (m + k))$. (Note that $O(m \cdot k \cdot (m + k))$ also covers relatively negligible $O(m)$ time taken by applications of rules E1 and E2 that are "caused" by description D_1.) It

follows that the total time to compute $Exp(Init(D_1), T)$ is $O(m^2 \cdot k + m \cdot k^2)$ without considering the overhead of the *Merge* procedure.

Now, consider the total cost of this overhead. To begin, note that the rules of *Merge* are applicable if and only if (1) there are **Id** edges in the graph, and/or (2) there are two edges outgoing from the same node and labeled with the same label. Also, note that procedure *Exp* invokes *Merge* in every rule that could possibly make conditions (1) and/or (2) satisfied. Moreover, observe that the only way an application of a rule of *Merge* can "cause" further applications is by merging two distinct nodes that obtain a node with a number of outgoing edges with the same labels. These facts allow an efficient implementation of *Merge* by employing a job queue specifically for procedure *Merge* with each element of the queue containing either an **Id** edge or a node and its two outgoing edges with the same label. An element is added to the queue whenever a new **Id** edge or a non-**Id** edge satisfying condition (2) is added to the graph by rules of *Exp*. *Merge* then removes a job from the queue, merges the appropriate nodes, and adds new jobs to the queue that are "induced" by the performed merge.

In the following analysis of *Merge*, we assume an efficient constant time implementation of such operations on the graph as

- addition of a node,
- addition of an outgoing edge from a node,
- checking for existence of an outgoing edge with a given label from a given node, and
- finding a list of all outgoing edges given a node.

An example implementation of such efficient operations is presented in the appendix of [BW94]. (The difference in our case is that in the encoding of a node that relates to R_{old}, the third array C in that implementation now contains a pointer to a *list* of nodes instead of just one node since we allow multiple outgoing edges with the same label inside procedure *Merge*. Conversely, the structure of the node that relates to R_{new} does not have to have any space for outgoing edges.)

Observe that each merge of two nodes involves combining their incoming edges as well as searching for the duplicates among the labels of the outgoing edges (to find the "induced" merges) and also merging the outgoing edges. First, observe that merging (the sets of) incoming edges can be implemented by a fast disjoint-set union algorithm (for example, Algorithm 4.3 presented in [AHU74]). Such an algorithm takes $O(t \cdot invA(t))$ time for t merges (limited by the number of nodes in the graph) and accesses to the edges, where $invA(t)$ is the inverse Ackerman function (it can be considered as constant in practice). Since we account for the costs of accessing the edges elsewhere and can add the extra constant component there, the total cost of merging the incoming edges by the invocations of the *Merge* procedure can be evaluated as the maximum number of nodes, i.e. $O(m \cdot (m + k))$.

Next, consider searching for duplicates among the labels of the outgoing edges from some nodes n_1 and n_2 that are being merged (let us denote the node resulting from this merge by n). In order to efficiently implement the search (as well as the merge of the outgoing edges), we keep track of the number of outgoing edges for every node and then look through every outgoing edge of the node that has the smaller number of the outgoing edges (say n_1) and check (in constant time) if there is an edge with the same label, say A, outgoing from the other node (n_2). If such an edge exists, we combine the lists of nodes that are reachable from n_1 and n_2 through edges labeled A and add a job to the queue that will merge an (arbitrary) node in the list of nodes that are reachable from n_1 through an edge labeled A with an arbitrary node in the list of nodes that are reachable from n_2 through an edge labeled A. Otherwise, we just "transfer" all outgoing edges labeled A from node n_1 to node n.

Now, consider the time complexity of this algorithm. The structure of the graph is such that only $O(m)$ nodes have any outgoing edges (i.e. the nodes that relate to R_{old}). For every merge with both nodes containing outgoing edges and for every edge label out of $O(m + k)$ edge labels, at most one new job queue is added, since every merge decreases the number of distinct nodes in R_{old} by 1. Therefore, there will be at most $O(m \cdot (m + k))$ jobs added to the queue. Thus, the total cost of all invocations of procedure *Merge*, except for the "search for duplicates" part, is $O(m \cdot (m + k))$. Indeed, all the work that is done for every job takes $O(1)$ time plus the time it takes to merge the incoming edges and the time to search for more jobs to add to the queue (as well as the time to merge outgoing edges which is already accounted for inside the search time, since we combine outgoing edges during the searches). The first two components add up to $O(m \cdot (m + k))$ as shown above.

Next, since only nodes that relate to R_{old} have outgoing edges, only merges that involve nodes from R_{old} would incur any cost for the "search for duplicates" part. Moreover, since each such merge decreases the number of nodes in R_{old}, there would be at most $O(m)$ merges that have to search for duplicates. Thus, the total cost of searching for duplicate edge labels requires $O(m \cdot (m + k))$ time since we spend constant time for at most $O(m + k)$ labels for each merge. It follows that the combined cost of *all* invocations of the procedure *Merge* is $O(m \cdot (m + k))$. Therefore, the total cost of a call to $Exp(Init(D_1), T)$ remains $O(m^2 \cdot k + m \cdot k^2)$.

The following example shows that our procedure *Exp* can in fact reach the bounds of $O(m^2 \cdot k)$ for firing rule E5 and $O(m \cdot k^2)$ for firing rules E1, E2, E6, and E7 in the worst case. Consider a terminology T with constraint

$C <$ (**and** C_i (for all $1 \leq i \leq k$)
(**all** A C)
(**all** B C)
(**all** B_i C) (for all $1 \leq i \leq k$)
(**fd** C B B_i) (for all $1 \leq i \leq k$)
(**fd** C (**comp** A B_1) ... (**comp** A B_k) B))

and a problem $D_1 < D_2$ for an arbitrary description D_2, where D_1 is

(**and** C (**same-as** $B_1.A^m.B$ $B_2.A^m.B$))

with $B_j.A^m.B$ denoting the attribute description (**comp** B_j (**comp** A (**comp** A ... (**comp** A B) ...))) of length $m + 2$ (i.e. with m occurrences of attribute A). It is not difficult to see that the number of nodes occurring in $Exp(Init(D_1), T)$ is $\Theta(m \cdot k)$ since for each of the m pairs of nodes of the same "depth" from the distinguished node and with incoming edges labeled A, the procedure creates k new nodes with incoming edges labeled B_1 through B_k. Moreover, the procedure must insert $\Theta(k)$ concept names into each of the $\Theta(m \cdot k)$ nodes.

In addition, since there are $\Theta(m)$ nodes in R_{old} with C in their α sets, our implementation would create a record for each pair of these nodes and each of the $\Omega(k)$ **fd** descriptions in the terminology. Therefore, the implementation would take $\Omega(m^2 \cdot k)$ time for processing the **fd** constraints. Thus, the total time *Exp* would take is $\Omega(m^2 \cdot k + m \cdot k^2)$.

Let us now consider the runtime complexity of procedure *Subsumes*. First, note that invocations of *Merge* inside S3, S4, and S5 can efficiently integrate added paths with already existing paths of the graph. Therefore, they only take $O(t)$ time, where t is the length of the paths added by the parts that precede the invocations of *Merge*. Since there are at most $O(m)$ new nodes added by those parts of the steps S3, S4, and S5, all invocations of *Merge* take $O(m)$ time.

The running time of S2 consists of the sum of the runtimes of the recursive calls plus O(1) time. Since there are at most $O(m)$ **and** constructors in $D_{2[u]}$, the total time spent on S2 is $O(m)$ not counting the time spent on the recursive calls.

The runtime complexity of an execution of S1 is at most $O(m + k)$ since (1) S1 simply looks for a specific concept name inside an α set and (2) there are at most $O(m + k)$ concept names that can be introduced by T and $D_{1[u]}$. An execution of step S6 takes $O(k)$ time for the same reasons as S1 (and considering that no new view names can be introduced by $D_{1[u]}$) plus the time for the recursive call if necessary. It follows that without the recursive calls, steps S1 and S6 take $O(m \cdot (m + k))$ time total since there are at most $O(m)$ concepts and view names inside $D_{2[u]}$.

Next, consider the time spent on the execution of steps S3 and S4 not counting the invocations of *Merge* in parts (d) and (c) respectively. First, note that by similar analysis to the

proof of assertion 1 in the proof on completeness, rules E3, E4 and E5 are never applied during the invocation of *Exp* inside the execution of rules S3 and S4. It follows that steps S3 and S4 add at most $O(m)$ new edges and nodes. Therefore, the total number of edges and nodes in the graph is still limited by $O(m \cdot (m + k))$, and we can use the discussion of the runtime complexity of applications of rules E1, E2, E6 and E7 by procedure *Exp* to conclude that the total time spent on executions of rules S3 and S4 is $O(m^2 \cdot k + m \cdot k^2)$.

Finally, consider the time spent during an execution of step S5 without the time spent by the invocation of *Merge* in part (e). First, note that making a copy of the graph can be efficiently implemented, for example, by using additional lists of nodes, edges, and descriptions inside node labels that should be removed after S5 is done. Next, note that parts (a) through (e) take $O(t)$ time where t is the size of the encoding of the **fd** constraint. Additionally, analysis of the runtime complexity of the invocation of procedure *Exp* in part (f) is analogous to the analysis of the runtime to compute $Exp(Init(D_1), T)$. The only difference is that nodes that relate to R_{old} are now introduced in three different ways: one group of size $O(m)$ are nodes that relate to R_{old} in $Exp(Init(D_1), T)$; the second group is $O(m)$ nodes added by steps S3 and S4, and nodes added in parts (a) through (d) of the step S5 (possibly merged with other nodes in the graph by part (e)); and finally, the third group is $O(m)$ nodes created by the expansion of the description D_2. (Recall that D_2 is placed inside α set of the node referenced by $fdcn_c$ by the first phase of procedure *Exp* invoked in part (f) of S5.) Therefore, the boundary property is satisfied for the resulting graph, there are $O(m)$ nodes that relate to R_{old}, and the argument analogous to the one presented for the analysis of the runtime complexity of procedure *Exp* implies that an execution of step S5 takes $O(m^2 \cdot k + m \cdot k^2)$ time. (This bound also includes a check for dn_c and $fdcn_c$ to agree on a prefix of Pf which takes $O(m)$ time.)

Ultimately, therefore, we have the following:

1. $O(m)$ time *total* for step S2.
2. $O(m \cdot (m + k))$ time *total* for steps S1 and S6.
3. $O(m^2 \cdot k + m \cdot k^2)$ time *total* for steps S3 and S4 including the $O(m)$ time spent by procedure *Merge*.
4. $O(m^2 \cdot k + m \cdot k^2)$ time for *each* execution of step S5 including the $O(m)$ time spent by procedure *Merge* in part (e).

Thus, letting f denote the number of occurrences of the **fd** constructor in $D_{2[u]}$, the total runtime complexity of $Subsumes(D_2, Exp(Init(D_1), T), T)$ is $O(f \cdot (m^2 \cdot k + m \cdot k^2))$.

Similarly to Chapter 4, we can further improve the runtime complexity by observing that we cannot deduce anything about an ancestor based on the typing information in the descendent nodes that do not have outgoing edges; that is, based on descriptions inside α sets of such descendents. Therefore, it is not necessary to deduce any typing information about nodes that

relate to R_{new} in order to deduce anything about nodes that relate to R_{old}. One of the ways to capture these modifications inside the *Merge*, *Exp*, and *Subsumes* procedures is to add an extra label to nodes which would denote whether the node relates to R_{old} or R_{new}, and then make the rules of the procedures to take the new label into account. For example, rule E4 would only create nodes that relate to R_{old}, whereas rule E5 would create nodes that relate to R_{new} but could later turn into nodes that relate to R_{old} by the rules of *Merge*. However, since these modifications are more of an implementation issue rather than an essential part of the algorithm, we will not attempt to further introduce them into the production system. Instead, we expect implementation *not* to apply rule E2 during the second phase of *Exp* to any edge $\langle n_1, A, n_2 \rangle$ if n_2 is in R_{new}. Then, if steps S3, S4, parts (a) through (d) of step S5, or the first phase of an invocation of *Exp* in part (f) of step S5 create any paths that pass through nodes that relate to R_{new} (thus, essentially making them relate to R_{old} for the purposes of the following *Exp* procedure), the implementation would propagate typing information from the parents of such nodes and expand it according to the applicable rules of *Exp*.

It follows however, that since the number of nodes that relate to R_{old} is only $O(m)$, the total time spent on processing the rules E1, E2, E6 and E7 inside *Exp* would decrease to $O(m \cdot k)$. Therefore, the total runtime complexity of *Exp* becomes $O(m^2 \cdot k)$. Moreover, by the same reasoning, the total runtime complexity of *Subsumes* becomes only $O(f \cdot m^2 \cdot k)$.

Note that as in Chapter 4, termination and soundness of the modified procedures directly follow from termination and soundness of the original procedures since we apply the rules in the same manner but only decrease the number of applications of the rules. On the other hand, the proof of completeness would have to be augmented with additional argument that formally applies the above observation. In particular, in addition to applying the modified *Exp* and *Subsumes* procedures during the construction steps of the proof, we need to propagate typing information to *all* nodes during the construction of G_0. Then, we can still prove that all constraints in the terminology are satisfied by the constructed interpretation, while node dn_0 violates the subsumption constraint in the posed question.

Finally, note that in practice, the number of constraints applicable to any given primitive concept is usually very small in comparison to the total size of a terminology. Therefore, the $O(k)$ component of $O(f \cdot m^2 \cdot k)$ is in fact a rather lose bound on the real cost of the work performed.

5.7 Integration

5.7.1 Inter Procedure Reuse

So far we presented an efficient procedure to solve membership problems in Chapter 4 and less efficient procedures to solve more general logical implication problems in this chapter. One of the features of these two solutions is that we can integrate them to solve a series of problems using the more efficient *Prop* procedure for the membership problems and using the *Exp* and *Subsumes* procedures for the remainder. In particular, work performed by *Prop* to solve a membership problem can be reused by the procedures in this chapter to solve similar logical implication problems.

Note that both *Prop* and *Exp* procedures perform essentially the same work at "expanding" typing information in the description graphs. In other words, all deductions that *Prop* makes about concept names inside labels of nodes as well as its "propagation" of other constructors would also be performed by the *Exp* procedure. In particular, consider an application of the *Prop* procedure to an initial acceptor tree with concept C in the *Cls* label of the root. Procedures *Exp/Subsumes* can then reuse the information embedded inside the resulting acceptor tree by copying the tree under any node n of a description graph if $\alpha(n)$ contains C. Clearly, such reuse is possible if the terminology for the membership problem is essentially a subset of the terminology for the more general logical implication problem that reuses the acceptor tree.

Thus, to combine the procedures, one would maintain both an acceptor tree for membership problems and a description graph for more general logical implication problems. Then, each question either modifies the acceptor tree or the description graph depending on whether it can be classified as a membership problem. In addition, the acceptor tree is reused inside the description graph if the corresponding membership and logical implication problems are related in the right way (for example, in the way described above).

This kind of inter procedure reuse is facilitated by the incremental nature of our procedures discussed below.

5.7.2 Intra Procedure Reuse

In Chapter 4, we motivated and started to discuss the incremental requirement for procedures that solve logical implication problems. This subsection shows how our procedures can be used incrementally; that is, in a way that will reuse descriptions graphs resulting after our procedures solve one problem in order to solve the next.

First, consider two consecutive problems in a series of logical implication problems of the form:

$$T \models D_1 < (\textbf{fd}\ D_2\ Pf_1\ \ldots\ Pf_m\ Pf) \text{ and } T' \models D_1' < (\textbf{fd}\ D_2'\ Pf_1'\ \ldots\ Pf_k'\ Pf').$$

If the two problems above satisfy all appropriate requirements to be membership problems, we can reuse the acceptor tree obtained by the *Prop* procedure that solves the first problem to solve the second problem in a number of cases. First, if only Pf and Pf' are different, exactly the same acceptor tree can be used to solve both problems. Next, if T is the same as T', D_1 is the same as D_1', and for every Pf_i in the first problem there is a Pf_j' in the second problem which is a prefix of Pf_i, we can reuse the tree by (a) adding the rest of the path functions to the root of the tree, (b) inserting "**Id**" into the *Pfs* labels of nodes that correspond to the appropriate Pf_j' prefixes, and (c) invoking *Prop* on the resulting graph. If in addition, D_1' is subsumed by D_1, we could still reuse the acceptor tree by adding "new" constraints in D_1' to the root of the tree and then propagating them. Finally, if in addition to the possible differences above, T' contains all of the constraints in T augmented with some new ones, we can apply new constraints on the appropriate nodes of the tree without redoing the work for the "common" terminology constraints that were already explored by the *Prop* procedure when it was solving the first problem.

Prop can be also reused when it is *not* the case that for every Pf_i in the first problem there is a Pf_j' in the second problem which is a prefix of Pf_i. If terminologies and concept names in the membership problems are related in the ways described above, we could reuse the acceptor tree by (a) resetting all *Pfs* labels of nodes to empty sets, (b) reinitializing the *Pfs* label of the root with path functions Pf_j', and (c) invoking *Prop* on the resulting tree. Differences in terminologies and concept names would be accounted for as before.

Exp/Subsumes procedures are also incremental in cases analogous to those discussed above. If the two logical implication problems only differ in the last path function, exactly the same description graph can be used to solve both problems. Reuse of the graph is also possible if T is the same as T', D_1 is the same as D_1', D_2 is the same as D_2', and for every Pf_i in the first problem there is a Pf_j' in the second problem which is a prefix of Pf_i. In this case, we can reuse the description graph obtained by solution to the first problem by (a) making *dn* and *fdcn* agree on paths that correspond to the additional path functions, (b) merging nodes that correspond to the appropriate Pf_j' prefixes, and (c) invoking *Exp* and *Subsumes* procedures to obtain the new graph. We would then check for *dn* and *fdcn* to agree on paths that correspond to Pf'. Moreover, reuse is also possible if D_1 "contains" description D_1' and D_2 contains description D_2', or more generally, if $D_1' < D_1$ and $D_2' < D_2$. We would then only have to add "new" constraints in D_1' and D_2' to *dn* and *fdcn* respectively (followed by expansion of those constraints by the rules of *Exp/Subsumes*). Finally, if in addition to the possible differences above, T' contains all of the constraints in T augmented with some new ones, we again start with description graph resulting from solving the first problem and make the additional constraints in T' fire on the appropriate nodes.

The need to be able to efficiently reason about a series of problems that differ in the ways that we discussed above is demonstrated, for instance, by examples in Sections 4.3.

Next, consider logical implication problems that do not contain **fd** constructors on the right-hand-side of posed questions; let us call them *non-fd problems*. In other words, non-**fd** problems have the form $T \models D_1 < D_2$ where D_2 only contains **and**, **all**, and **same-as** concept constructors. Such problems do not "invalidate" any part of the description graph and can reuse the description graph resulting after solving one problem to solve the next one. On the other hand, processing of **fd** constructors by step S5 of *Subsumes* may cause some nodes to get merged that would not get merged otherwise, and may also cause some descriptions to be added to a node that would not be there in general. This is exactly the reason why we create a copy of the graph at step S5 and only work with that copy. Thus, non-**fd** problems can reuse the state of the graph before it is used to solve a problem that contains an **fd** constructor on the right-hand-side of the posed question (let us call such problems *fd problems*).

In practice, instead of creating a copy of the graph, one might use two *versions* of nodes and edges in the graph. For example, version 1 could indicate that a particular edge or a node has been processed by step S5. Therefore, it should *not* be considered as part of the graph for next non-**fd** problem. On the other hand, if next **fd** problem relates to the previous **fd** problem in one of the ways that we discussed above, such an edge or a node could be reused.

Chapter 6

Conclusion

The first section of this chapter summarizes the main results of the thesis. We then explore a number of ways in which the expressiveness of our languages can be enhanced, in many cases without compromising efficiency. Finally, we discuss some directions for future work.

6.1 Summary

This thesis addresses problems that arise in many areas of information technology. In particular, our work considers effective representations of semantic constraints commonly used in object-relational database systems and examines efficient algorithms to reason about such constraints. We expand data dependencies that are commonly used by relational models and combine them with constraints arising in object-relational models to form a theory that allows one to reason about both kinds of constraints. We also present procedures that can efficiently reason in such a theory. The procedures can be used to help solve problems relating to both object-relational and relational databases. A fundamental reason that we are able to derive such procedures relates to the variety of *uniqueness constraints* incorporated into the theory, which strictly generalize the notions of keys and functional dependencies that are inherent parts of relational and object-relational models.

We investigate the interaction between such constraints and other data dependencies, including inheritance, typing and equational constraints. In addition, the problem is explored in the context of *description logics*, which are a family of knowledge representation schemas that have found myriad applications in information systems technology. From this perspective, we capture uniqueness constraints inside the DL framework by introducing an **fd** concept constructor that is more expressive than other related constructors in earlier works [GL95, CGL95, BW97]. We then show how various DL dialects that include this constructor can be used to address

problems in information technology and present efficient decision procedures for subsumption checking in these dialects.

Among other contributions (listed in Chapter 1), our procedures are further analyzed with respect to their generality and incremental abilities. In addition, their possible extensions are considered in the next section.

We now present a detailed overview of the main problems solved by our procedures in preceding chapters. The logical implication problem of Chapter 4 has the form

$$T \models C < (\textbf{fd}\ C\ Pf_1\ \ldots\ Pf_m\ Pf),$$

where $m \geq 0$, C is a primitive concept name, Pf_1 through Pf_m and Pf are arbitrary attribute descriptions, and T is a set of constraints of the form $C < D$, where D satisfies the following grammar:

D ::= C	*(primitive concept)*
| **(all** Pf D)	*(attribute value restriction)*
| **(fd** C Pfs Pf)	*(functional dependency)*
| **(and** D D Ds)	*(concept intersection)*
Pf ::= A	*(primitive attribute)*
| **Id**	*(identity)*
| **(comp** Pf Pf)	*(attribute composition)*
Pfs ::= ε | Pfs Pf	*(attribute description sequence)*
Ds ::= ε | Ds D	*(concept description sequence)*

Moreover, all uniqueness constraints resulting after terminology simplification must be symmetric and regular. The chapter presents a very efficient (sound and complete) algorithm which runs in time corresponding to the product of the encoding length of the terminology and the posed question.

An additional factor of roughly the encoding length of posed question allows one to use a more general algorithm presented in Chapter 5. The problem considered there has the form

$$T \models D_1 < D_2.$$

Terminology T now consists of two parts—a set of subsumption constraints and a set of non-recursive view definitions. Each subsumption constraint has the form $C < D$ where D satisfies the same grammar as above with the exception that uniqueness constraints can be asymmetric. Every view definition has the form $V \equiv D_V$ where D_V satisfies the grammar above in addition to allowing view names and equational restrictions:

D_V ::=	C	(*primitive concept*)
	| (**all** Pf D_V)	(*attribute value restriction*)
	| (**fd** C Pfs Pf)	(*functional dependency*)
	| (**and** D_V D_V Ds)	(*concept intersection*)
	| V	(*view name*)
	| (**same-as** Pf Pf)	(*equational restriction*)
Pf ::=	A	(*primitive attribute*)
	| **Id**	(*identity*)
	| (**comp** Pf Pf)	(*attribute composition*)
Pfs ::=	ε | Pfs Pf	(*attribute description sequence*)
Ds ::=	ε | Ds D_V	(*concept description sequence*)

Next, D_1 satisfies the same grammar as the one for the view definition descriptions D_V. Finally, description D_2 satisfies the grammar of D_V in addition to extended **fd** descriptions:

D_2 ::= (**fd** D_{fd} Pfs Pf)	(*extended fd description*)

(+ grammar of D_V where every D_V is replaced by D_2)

This **fd** description does not have to be regular and D_{fd} satisfies the grammar of D_V.

6.2 Extensions

The following subsections describe some possible extensions of our work. We consider potential problems arising with some extensions, thus illustrating the need for some of the restrictions that we have imposed on the logical implication problems that were considered. A guideline on how to (efficiently) incorporate most of the extensions with our procedures is included.

6.2.1 Procedure *Accepts*

Our procedure in Chapter 4 can be extended to solve arbitrary problems of the form $T \models C < D$, where the right-hand-side of the posed question satisfies the grammar of terminology constraints. The extension is analogous to the way such problems are solved in Chapter 5. More specifically, this can be achieved by an additional procedure, say *Accepts*, which deals with generalized problems analogously to the way procedure *Subsumes* does this in Chapter 5.

Since Chapter 4 only deals with symmetric uniqueness constraints, any **fd** constructor inside D must have C as its concept name and cannot be "buried" inside an **all** description (but

only inside **and** descriptions). In other words, if the constraint is decomposed by the rules in Table 4.1, all resulting uniqueness constraints must be symmetric. Then, each of such constraints forms a membership problem that can be solved as described in Chapter 4. Therefore, analogously to *Subsumes*, procedure *Accepts* would recursively call itself to deal with **and** and **all** constructors, and in the "simple" base case, when the description passed to *Accepts* is just a primitive concept C', the procedure would check for the presence of C' in the *Cls* label of the passed distinguished node. In the "involved" base case, when D is an **fd** description, *Accepts* would empty all *Pfs* labels, initialize the *Pfs* label of the root with the appropriate path functions, fire procedure *Prop*, and finally, check for acceptance of the last path function in D. Note the importance of the incremental nature of the procedure here: we can reuse typing information that is already deduced to solve further problems about uniqueness constraints that differ in their path functions. (In some cases, such as those described in Subsection 5.7.2, we can reuse other work as well.)

Formally, the addition of procedure *Accepts* would require an extension to the proofs presented in Chapter 4 in the direction of the proofs in Chapter 5. Note that the new proof of soundness would now also use property (2) of Lemma 4.3, construction would remain the same in the proof of completeness, and the essential notion of *ATBoundary* would still be the central part in the proof of termination.

6.2.2 THING

Concept constructor THING denotes all objects in the domain, i.e. $\text{THING}^{\mathbf{I}} = \Delta$ for any interpretation $\mathbf{I}$ ($= \langle \Delta, \cdot^{\mathbf{I}} \rangle$). Therefore, THING naturally represents the top of a class hierarchy, which sometimes is also referred to as class OBJECT. In addition, we allow THING to appear anywhere inside terminology or posed question where primitive concept can appear.

There are two ways in which our procedures can be modified in order to reason about THING, and both ways only slightly distinguish THING from a primitive concept name.

The first way is just to add THING to every *Cls* (resp. α) label of every node created in a description graph. This in turn fires all terminology constraints with THING on their left-hand-side on every node.

The second way, on the other hand, preserves node creation, and uses mechanisms already built into the procedures to "spread" THING to all nodes. Therefore, it allows our procedures to consider THING just as a primitive concept name at a cost of some "preprocessing". First, for every concept C and every attribute A occurring in a terminology T, we add constraints

C < THING, and
C < (**all** A THING)

to T. These constraints, along with propagation of the typing information by our procedures, ensure that THING is inside the *Cls* (resp. α) label of every node in the description graph that is reachable from a node with non-empty *Cls* (resp. α) label.[20] Such changes are sufficient in case of *Prop/Accept* procedures since they consider only description graphs that are trees, and since posed questions processed by these procedures have a primitive concept on the left-hand-side. On the other hand, questions solved by *Exp/Subsumes* procedures have to be slightly modified to ensure that every node has THING inside its α set. In particular, every posed question of the form $D_1 < D_2$ should be modified into (**and** THING D_1) < D_2', where every description D' of every **fd** description inside D_2' is changed to (**and** THING D').

Note that the equivalence of the original and modified questions as well as the fact that every constraint added to the terminology is satisfied by any interpretation follows directly from the definition of the interpretation function. In addition, proofs of soundness do not change significantly. In fact, the same invariants should be used and only slight extensions are needed to account for THING constructor. Proofs of termination would also remain almost unchanged since procedures would consider THING as any primitive concept name. Finally, the proofs of completeness also do not change much. One just has to formally show that every node in the constructed description graphs is reachable from a node with a concept name inside it (*dn* and *dn'* nodes in case of *Prop/Accepts*; and *dn* and *fdcn* nodes in case of *Prop/Subsumes*). Then, it would follow that every object in the constructed interpretation **I** is in fact in THING$^{\mathbf{I}}$.

6.2.3 Concept Conjunction

Another way to extend our procedures is to allow concept conjunction, i.e. descriptions of the form (**and** C_1 ... C_k) for $k \geq 1$, on the *left-hand-sides* of terminology subsumption constraints. For example, a set of constraints

{TA < STUDENT, TA < TEACHER, (**and** STUDENT TEACHER) < TA}

allows us to capture the fact that "teaching assistants are the only students that are also teachers."

Our procedures can be relatively easily extended to deal with such constraints by looking for concepts C_1 through C_k inside a *Cls* (or α) label of a node rather than just one concept name before applying the appropriate rule. For example, rule P3 of *Prop*,

[20] Note though that for efficiency reasons, we do not have to propagate THING or any concept name to leaf nodes as discussed in the descriptions of the suggested efficient implementations.

(*typing*) If there exist $\langle n_1, A, n_2 \rangle \in E$ and "$C_1 <$ (**all** A C_2)" $\in T$ such that $C_1 \in Cls(n_1)$ and $C_2 \notin Cls(n_2)$, then add C_2 to $Cls(n_2)$,

would be replaced by

(*typing*) If there exist $\langle n_1, A, n_2 \rangle \in E$ and "(**and** C_1 ... C_k) < (**all** A C')" $\in T$ such that $C_i \in Cls(n_1)$ for all $1 \leq i \leq k$ and $C' \notin Cls(n_2)$, then add C' to $Cls(n_2)$.[21]

Similarly, we can extend our procedures to deal with **fd** descriptions that contain concept conjunction instead of a single concept inside. (We allow such descriptions on the right-hand-sides of terminology constraints, including view definitions in Chapter 5, on the right-hand-side of the posed question and, in case of Chapter 5, on the left-hand-side of the posed question.) Then, for example, rule E5 of *Exp*,

If there exist nodes n_1 with $\alpha(n_1) = \{D_1, ..., D_k,$ (**fd** C Pf_1 ... Pf_m Pf)} and n_2 (distinct from n_1) in N, such that

(1) (**fd** C Pf_1 ... Pf_m Pf) is regular;

(2) $C \in \alpha(n_2)$;

... ,

would be replaced by

If there exist nodes n_1 with $\alpha(n_1) = \{D_1, ..., D_k,$ (**fd** (**and** C_1 ... C_l) Pf_1 ... Pf_m Pf)} and n_2 (distinct from n_1) in N, such that

(1) (**fd** (**and** C_1 ... C_l) Pf_1 ... Pf_m Pf) is regular;

(2) $C_i \in \alpha(n_2)$ for all $1 \leq i \leq l$;

...

Note that a constraint of the form (**and** C_1 ... C_k) < (**fd** (**and** C_1' ... C_l') Pf_1 ... Pf_m Pf)[22] would be considered symmetric (for the purposes of procedures in Chapter 4) if and only if $k = l$, both **and** descriptions contain only distinct concept names[23], and for every $1 \leq i \leq k$, there exists $1 \leq j \leq l$ such that C_i and C_j' denote the same primitive concept name.

---

[21] Note that for simplicity, we presume that every constraint $C < D$ is converted to (**and** C) < D before the procedures are applied. Also note that set *fired* now contains concept conjunctions in addition to primitive concept names and view names.

[22] Analogously to introduction of concept conjunction on the left-hand-side of constraints, we presume that every description (**fd** C Pf_1 ... Pf_m Pf) is converted to (**fd** (**and** C) Pf_1 ... Pf_m Pf) before the procedures are applied.

[23] This condition can be ensured by a step of primitive concept duplicate elimination from these (and other) **and** descriptions during the terminology simplification phase.

It should be also mentioned that proofs of termination, soundness and completeness would have to be only slightly modified by noting at appropriate places that for any positive integer k and any interpretation **I**, an object belongs to (**and** $C_1 \ldots C_k$)$^{\mathbf{I}}$ if and only if it belongs to C_i for all $1 \leq i \leq k$.

Finally, as discussed in the previous subsection, we can also allow THING inside the conjunctions without significant modifications.

6.2.4 Equational Restrictions in Terminology Constraints

For the procedures in Chapter 5, the left-hand-sides of terminology subsumption constraints can be further extended to allow equational restrictions, i.e. descriptions of the form (**same-as** Pf_1 Pf_2) for some path functions Pf_1 and Pf_2. For example, such a constraint

(**and** PROFESSOR (**same-as** Salary (**comp** Dept HighestSalary))) < CHAIRMAN

allows us to capture the fact that "all professors with highest salaries in their departments are chairmen."

To deal with terminology constraints that have left-hand-side of the form

(**and** $C_1 \ldots C_k$ (**same-as** Pf_1 Pf_2))[24],

in addition to checking whether concept names C_i are in the α label of a node n before applying the constraint to n, our procedures have to ensure that there exist two paths outgoing from n with edge labels corresponding to (some prefixes of) Pf_1 and Pf_2 that agree on the last node.

More formally, let us define a notion of a *suffix* of a path function Pf as any path function Pf_{suf} such that there exists a path function Pf_{pref} for which $Pf = Pf_{pref} \circ Pf_{suf}$. Pf_{suf} is called a *common suffix* of two path functions if it is a suffix of both. Finally, Pf_{suf} is the *greatest common suffix* of two path functions Pf_1 and Pf_2, denoted $GCF(Pf_1, Pf_2)$, if no other common suffix exceeds it in length. (It is not hard to establish that the greatest common suffix of any two path functions always exists and there is only one such suffix.)

Consider a common suffix Pf of path functions Pf_1 and Pf_2, and let $Pf_1 = Pf_1' \circ Pf$ and $Pf_2 = Pf_2' \circ Pf$. Clearly, due to the functionality of the attributes, (**same-as** Pf_1' Pf_2') implies (**same-as** Pf_1 Pf_2). This observation suggests that to process description (**same-as** Pf_1 Pf_2) on the left-hand-side of a constraint, the procedure should check if there exist two paths with edge labels

[24] Note that k can be 0 in this case. Then, the left-hand-side would just have the form (**same-as** Pf_1 Pf_2). Also note that while we omit the discussion of this, we could substitute THING for any of the C_i's.

corresponding to the prefixes of Pf_1 and Pf_2 such that the "remainder" of those path functions form a common suffix (starting with $GCS(Pf_1, Pf_2)$).

For more details, let Pf_1'' and Pf_2'' be such path functions that $Pf_1 = Pf_1'' \circ GCS(Pf_1, Pf_2)$ and $Pf_2 = Pf_2'' \circ GCS(Pf_1, Pf_2)$. Then, the procedure would start at a node to which the constraint is possibly to be applied (after checking for the presence of necessary concept names inside the α set), and would follow the paths with edge labels corresponding to Pf_1'' and Pf_2''. If at least one of these paths stops before reaching the last node, i.e. one of the edges with a needed label was not created, the constraint is not applicable (yet). If both paths end at the same node, the constraint can be applied. Otherwise, when both paths reach their destinations that are distinct nodes, we move one edge further along the paths (and along the greatest common suffix) and check if we end up at the same node. The process of moving along the paths continues until we either do not find a necessary edge, two paths agree on an appropriate node, or when we reach the ends of the paths. (Note that our event driven approach would also work here if we make the nodes wait either for the corresponding "next" nodes along the path to be added or for an agreement with the appropriate nodes along the other path. This would only require a total of O(1) work for every edge in the path functions.)

Thus, rule E6 of *Exp*,

> If there exists a node n in N, and a constraint $C < D$ in T such that $C \in \alpha(n)$ and $C \notin fired(n)$, then add D to $\alpha(n)$ and C to $fired(n)$,

would now be replaced by the following two rules:

> If there exists a node n in N, and a constraint (**and** $C_1 \ldots C_k$) $< D$ in T ($k \geq 1$) such that $C_i \in \alpha(n)$ for all $1 \leq i \leq k$ and (**and** $C_1 \ldots C_k$) $\notin fired(n)$, then add D to $\alpha(n)$ and (**and** $C_1 \ldots C_k$) to $fired(n)$,[25]

and

> If there exist nodes n and n' in N, a constraint (**and** $C_1 \ldots C_k$ (**same-as** Pf_1 Pf_2)) $< D$ in T ($k \geq 0$), prefixes Pf_1' and Pf_2' of Pf_1 and Pf_2 respectively, and a common suffix Pf_{suf} of Pf_1 and Pf_2 such that
>
> (1) $C_i \in \alpha(n)$ for all $1 \leq i \leq k$;
>
> (2) $Pf_1 = Pf_1' \circ Pf_{suf}$ and $Pf_2 = Pf_2' \circ Pf_{suf}$;

[25] As before, note that for simplicity, we presume that every constraint $C < D$ is converted to (**and** C) $< D$ before the procedures are applied. Also note that set *fired* should be extended again to allow descriptions of the form (**and** $C_1 \ldots C_k$ (**same-as** Pf_1 Pf_2)).

(3) *Reachable*(n, Pf_1', n') and *Reachable*(n, Pf_2', n') are true; and

(4) (**and** C_1 ... C_k (**same-as** Pf_1 Pf_2)) $\notin$ *fired*(n);

then add D to $\alpha(n)$ and (**and** C_1 ... C_k (**same-as** Pf_1 Pf_2)) to *fired*(n).

Proceeding analogously to the previous subsection, we can further extend our procedures in Chapter 5 to deal with extended **fd** descriptions that contain conjunctions of concepts and equational restrictions[26]. (We allow such descriptions on the right-hand-sides of terminology constraints, including view definitions, and on both sides of the posed question.) For example, to express the fact that "each department chairman teaches at most one course," one could use the constraint

(**and** COURSE
    (**same-as** TaughtBy (**comp** TaughtBy (**comp** Dept Chairman)))) <
(**fd** (**and** COURSE
    (**same-as** TaughtBy (**comp** TaughtBy (**comp** Dept Chairman))))
                                                TaughtBy **Id**).

(The constraint states that if any two courses taught by a department chairman have the same instructor, they must be the same course.)

Finally, going one step further, it is easy to see that the left-hand-sides of the constraints and the extended **fd** descriptions can in fact deal with any description satisfying the following grammar:

D ::= C		(*primitive concept*)
	| THING	(*all objects in the domain*)
	| (**and** D Ds)	(*description intersection*)
	| (**same-as** Pf Pf)	(*equational restriction*)

where Ds and Pf are a sequence of descriptions and an attribute description respectively with grammars provided in Section 6.1. In other words, we can allow an arbitrary number of the equational restrictions and an arbitrary nesting of the **and** constructors (which can be easily "unnested").

[26] Note that rule E3 of *Exp* has to be modified *not* to account for such occurrences of the **same-as** constructor inside an extended **fd** description.

6.2.5 Roles

Our procedures can be further extended to deal with roles and corresponding typing restrictions; i.e. to deal with descriptions of the form (**all** R D) for some role R and description D. (Recall that roles are set-valued properties of objects; i.e. interpretation of a role is a subset of $\Delta \times \Delta$ for a domain Δ.) As in the case of **fd** descriptions extended with concept conjunction, we allow role typing restrictions on the right-hand-sides of terminology constraints, including view definitions in Chapter 5, on the right-hand-side of the posed question and, in case of Chapter 5, on the left-hand-side of the posed question.

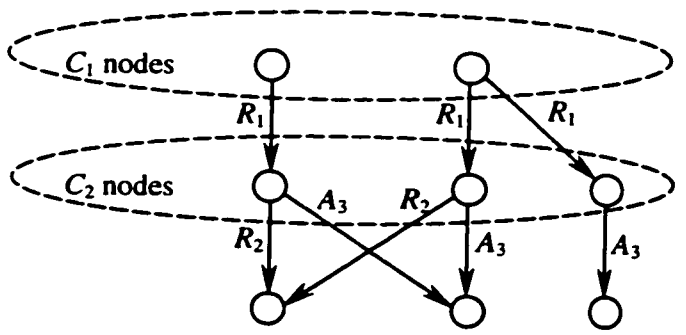
Note that due to complicated interaction of roles with **same-as** constructor (as was mentioned in Chapter 2) and with **fd** constructor, we have to disallow any occurrence of a role inside **same-as** and **fd** descriptions. To illustrate such "non-standard" interaction, note for example that constraints

$$C_1 < (\textbf{all}\ R_1\ C_2) \text{ and}$$
$$C_2 < (\textbf{fd}\ C_2\ R_2\ A_3)$$

do *not* imply the constraint

$$C_1 < (\textbf{fd}\ C_1\ (\textbf{comp}\ R_1\ R_2)\ (\textbf{comp}\ R_1\ A_3))$$

where R_1 and R_2 are roles (contrast this with inference axiom 6 in Lemma 4.2). An example interpretation **I** proving the above claim is represented in the following figure:



Analogously to the proofs of completeness, the nodes in the graph are the domain of **I** and labeled edges correspond to role/attribute interpretations[27]. With this interpretation, we have two distinct nodes in $C_1^{\mathbf{I}}$ whose interpretations of (**comp** R_1 R_2) are the same (singleton) set, while the

[27] Note that only "important" nodes and edges are shown. Totality of A_3, if we want it to be an attribute, would require additional edges and nodes that could be added, for example, along the same lines as in the proofs of completeness.

interpretations of (**comp** R_1 A_3) are distinct sets (one is actually a proper subset of the other). At the same time, both of the originally given constraints are satisfied by **I**.

With above restrictions, our procedures can "process" (**all** R D) descriptions as if R were an attribute. It is not hard to see (in particular, from the arguments similar to the ones made in the complexity proofs) that since no **same-as** or **fd** description includes a role name, the description graph essentially becomes a high-level *tree* where nodes are connected only by edges labeled with role names, and each node is a description graph that contains only edges labeled with attribute names, i.e. exactly the description graphs studied in earlier chapters. (This kind of description graph is in fact discussed in [BP94].)

Note that addition of *either* (**at-most** R m), $m \geq 1$, *or* (**at-least** R k), $k \geq 0$, would involve a little extra processing that would adjust the upper or lower number restriction based on newly discovered constraints that apply to an edge labeled R. Moreover, allowing number restrictions of the form (**at-most** R 0) makes the reasoning just a bit harder, since the procedures have to make sure that for any question asked about something "below" role R, the affirmative answer is returned.

On the other hand, addition of both kinds of number restrictions would make the procedures exponential. Indeed, allowing both kinds of the constructors creates the possibility of *incoherent descriptions*, i.e. descriptions that necessarily do not contain any objects. (Note that such descriptions are subsumed by NOTHING which in turn is subsumed by *any* description.) Incoherent descriptions arise from constraints that force the lower limit on the number of role fillers for some occurrence of a role to exceed the corresponding upper limit. Since an incoherent description is subsumed by any other description, the procedures have to make sure that they discover all *incoherent nodes* [BP94] that essentially "correspond" to incoherent descriptions. However, this requires exploration of *all* chains of attributes and roles (with at least 1 role filler) that might lead to a role which "causes" an incoherence, since such incoherence would be propagated back to the original node. While we can make this process terminate when we reach nodes that only contain descriptions that were already explored for incoherence or are in the process of being explored, it certainly does not look like the procedure can be made to run in less than exponential time in the worst case.

Finally, it should be noted that the addition of constructor NOTHING and/or negation of primitive concepts runs into the same difficulties.

6.2.6 Extending Uniqueness Constraints

As was mentioned in Section 4.6, nearly regular uniqueness constraints turn out to be as expressive as the general uniqueness constraints that are likely to have only exponential decision procedures. Here, we consider another possible extension of the regular uniqueness constraints.

As with other relational functional dependencies, consider the natural generalization of *coupled functional dependencies* [CK85] and *union functional dependencies* [CV83] to use path functions instead of attributes. Even though some forms of this generalization can be captured by our asymmetric uniqueness constraints, extending our regular uniqueness constraints to fully capture such generalization would lead to an undecidable problem. Indeed, if we just consider the extension of symmetric regular constraints, and allow two objects to agree on *different* path functions, the extended constraints would have the form

$$C < (\textbf{fd}\ C\ \langle Pf_1\ Pf_1'\rangle \ldots \langle Pf_k\ Pf_k'\rangle\ \langle Pf\ Pf'\rangle),$$

where $\langle Pf_i\ Pf_i'\rangle$ are pairs of path functions on which a pair of objects in C have to agree. In other words, the constraint is satisfied by an interpretation $\mathbf{I}$ if and only if for any two objects o_1 and o_2 in $C^{\mathbf{I}}$, if $Pf_i^{\mathbf{I}}(o_1) = Pf_i'^{\mathbf{I}}(o_2)$ for $1 \le i \le k$, then $Pf^{\mathbf{I}}(o_1) = Pf'^{\mathbf{I}}(o_2)$. An extension of our regularity condition would then require an existence of $1 \le i \le k$ such that Pf (resp. Pf'), possibly without its last attribute, is a prefix of Pf_i (resp. Pf_i').

Then however, constraint $C <$ (**fd** $C\ \langle B_1 \circ B_2\ B_1 \circ B_2\rangle\ \langle B_1 \circ B_2\ A\rangle$) is equivalent to the constraint $C <$ (**same-as** $B_1 \circ B_2\ A$) for any concept C and attributes A, B_1 and B_2. It is not hard to show (see [BW97]) that constraints of the form $C <$ (**same-as** $B_1 \circ B_2\ A$) are as expressive as arbitrary constraints of the form $C <$ (**same-as** $Pf_1\ Pf_2$) for any path functions Pf_1 and Pf_2. However, it is well known that the problem of logical implication with arbitrary equational constraints is undecidable (e.g. see [BW94]).

6.2.7 Summary of Extensions

Here, we summarize how the problems in Chapters 4 and 5 would be modified in the presence of the "efficient" extensions discussed in the previous subsections.

Procedures in Chapter 4 would now deal with logical implication problem of the form $T \models D' < D''$ where T is a set of constraints of the form $D_{left} < D_{right}$. D_{left} satisfies the following grammar:

D_{left} ::=	C	(*primitive concept*)
	| THING	(*all objects in the domain*)
	| (**and** D_{left} Ds_{left})	(*description intersection*)
Ds_{left} ::=	ε | Ds_{left} D_{left}	(*concept description sequence*)

D_{right} satisfies the following grammar:

D_{right} ::=	C	*(primitive concept)*
	| THING	*(all objects in the domain)*
	| (**all** Pd D_{right})	*(attribute value restriction)*
	| (**fd** $D_{fd\text{-}right}$ Pfs Pf)	*(functional dependency)*
	| (**and** D_{right} D_{right} Ds)	*(concept intersection)*
Pf ::=	A	*(primitive attribute)*
	| **Id**	*(identity)*
	| (**comp** Pf Pf)	*(attribute composition)*
Pd ::=	A	*(primitive attribute)*
	| R	*(primitive role)*
	| **Id**	*(identity)*
	| (**comp** Pd Pd)	*(role composition)*
$D_{fd\text{-}right}$ ::=	C	*(primitive concept)*
	| THING	*(all objects in the domain)*
	| (**and** $D_{fd\text{-}right}$ $Ds_{fd\text{-}right}$)	*(description intersection)*
Pfs	::= ε | Pfs Pf	*(attribute description sequence)*
Ds	::= ε | Ds D_{right}	*(concept description sequence)*
$Ds_{fd\text{-}right}$	::= ε | $Ds_{fd\text{-}right}$ $D_{fd\text{-}right}$	*(concept description sequence)*

All uniqueness constraints still have to be symmetric and regular. Finally, D' satisfies the grammar of D_{left}, and D'' satisfies the grammar of D_{right} with exception that the **fd** descriptions do not have to be regular.

Next, consider the logical implication problem of Chapter 5. It now has the form $T \models D_1' < D_2'$, where subsumption constraints in T have the form $D_{left}' < D_{right}'$ and non-recursive view definitions have the form $V \equiv D_V'$. The grammar of D_{left}' extends the grammar of D_{left} with the following description:

D_{left}' ::= (**same-as** Pf Pf) *(equational restriction)*

(+ grammar of D_{left} where every D_{left} is replaced by D_{left}')

The grammar of D_{right}' extends the grammar of D_{right} with addition of equational restrictions to descriptions inside the extended **fd** descriptions:

$D_{fd\text{-}right}$ ::= (**same-as** Pf Pf) *(equational restriction)*

(+ grammar of $D_{fd\text{-}right}$ above)

Note that uniqueness constrains in T are no longer required to be symmetric.

Next, D_V' must satisfy the grammar of D_{right}' in addition to equational restriction and other view names:

$D_V' ::=$ V *(view name)*
| **(same-as** Pf Pf**)** *(equational restriction)*

(+ grammar of D_{right}' where every D_{right} is replaced by D_V')

Finally, D_1' satisfies the grammar of D_V', and description D_2' satisfies the grammar of D_V' in addition to extended **fd** descriptions:

$D_2' ::=$ **(fd** D_{fd}' Pfs Pf**)** *(extended fd description)*

(+ grammar of D_V' where every D_V' is replaced by D_2')

This **fd** description is no longer required to be regular, and D_{fd}' satisfies the grammar of D_V'.

6.3 Research Directions

Many of the underlying ideas have been incorporated into parts of the DEMO[28] system that perform query optimization [Cha97]. In that case, a variation of description graphs, called *existential graphs*, is used, and all inferences are based on a resource bounded graphical analogue of unit resolution. In this context, one of the contributions of our work is to the understanding of the underlying computational properties of the system—determining, for example, the limits of resource bounding that ensure complete reasoning for patterns of "graphical" constraints.

A study of how our procedures can be further incorporated into a practical database system and further exploration of the interaction of the underlying algorithms with other constructors (possibly with constraints that cannot be captured in a resulting theory) merit additional work. In general, although there is already a great deal of evidence for this, we believe further work is still needed to improve our understanding of the benefits of semantic query optimization in "industrial strength" IT applications. Conversely, work on exploring the possible efficacy of new (not currently existing) DL concept constructors is also needed.

Another important direction for future work is to consider how our procedures can be adapted to incorporate other kinds of DL concept and attribute constructors in ways that preserve tractability. Indeed, the previous section begins to explore this for a number of such extensions.

---

[28] "DEMO" stands for *Design Environment for Main Memory Object-Oriented databases*. The DEMO system is under development at the University of Waterloo in collaboration with Nortel Networks Ltd. It explores how to adopt database technology to enable it to be used to manage control data for embedded control programs.

The addition of two cases not yet considered, however, seem desirable: some capacity for reasoning about (necessarily restricted forms of) disjunction, and some ability to include equational restrictions on the right-hand-sides of subsumption constraints in terminologies.

Appendix A

Simplifying Terminology for the Membership Problem

We prove Proposition 4.1 that relates to a mapping presented in Chapter 4 for constructing so-called *atomic terminologies*. We begin by reviewing the proposition, starting with the relevant grammars and rewrite rules.

General grammar:

(1) **D ::= C**
(2) **| (all Pf D)**
(3) **| (fd C Pfs Pf)**
(4) **| (and D D Ds)**

(5) **Pf ::= A**
(6) **| Id**
(7) **| (comp Pf Pf)**

(8) **Pfs ::= ε**
(9) **| Pfs Pf**

(10) **Ds ::= ε**
(11) **| Ds D**

Simplified grammar:

(1') **D ::= C**
(2') **| (all A C)**
(3') **| (fd C Pfs Pf)**

(6') **Pf ::= Id**
(7') **| (comp A Pf)**

(8') **Pfs ::= ε**
(9') **| Pfs Pf**

Rule number	Replace	By
1	$C <$ (**all Id** D)	$C < D$
2*	$C_1 <$ (**all** (**comp** $Pf_1\ Pf_2$) D)	$C_1 <$ (**all** $Pf_1\ C_2$) and $C_2 <$ (**all** $Pf_2\ D$)
3*	$C_1 <$ (**all** A (**all** $Pf\ D$))	$C_1 <$ (**all** $A\ C_2$) and $C_2 <$ (**all** $Pf\ D$)
4*	$C_1 <$ (**all** A (**fd** $C_3\ Pfs\ Pf$))	$C_1 <$ (**all** $A\ C_2$) and $C_2 <$ (**fd** $C_3\ Pfs\ Pf$)
5*	$C <$ (**all** A (**and** $D_1 \ldots D_k$))	$C <$ (**all** $A\ D_1$), ..., $C <$ (**all** $A\ D_k$)
6*	$C <$ (**and** $D_1 \ldots D_k$)	$C < D_1$, ..., $C < D_k$
7	$C_1 <$ (**fd** $C_3\ Pfs_1\ A\ Pfs_2$)	$C_1 <$ (**fd** $C_3\ Pfs_1$ (**comp** A **Id**) Pfs_2)
8	(**comp Id** Pf)	Pf
9	(**comp** $Pf\ A$)	(**comp** Pf (**comp** A **Id**))
10	(**comp** (**comp** $Pf_1\ Pf_2$) Pf_3)	(**comp** Pf_1 (**comp** $Pf_2\ Pf_3$))

(* where C_2 denotes a new primitive concept not occurring in the given terminology and $k \geq 2$.)

Table A.1: Atomic simplification of terminology in Chapter 4 (revisited).

Let us call the general grammar $\mathbf{G_1}$ and the simplified grammar $\mathbf{G_2}$. Also, we say a *subsumption constraint* $C < D$ is *generated* by a *grammar* ($\mathbf{G_1}$ or $\mathbf{G_2}$) if, after replacing every primitive concept name in D by **C** and every primitive attribute name in D by **A**, D can be generated by the grammar. We say a *terminology is generated by a grammar* if every subsumption constraint in the terminology is generated by the grammar. Finally, for any rewrite rule in Table A.1, we call the string to be replaced the *left-hand-side* of the rule, and the string(s) that the left-hand-side is replaced with the *right-hand-side(s)* of the rule.

Proposition 4.1 *(revisited) Let T_1 denote an arbitrary terminology generated by grammar G_1, and $C < D$ a subsumption constraint free of any occurrence of a primitive concept not occurring in T_1. Then an exhaustive application of the rewrite rules in Table A.1 terminates and obtains a terminology T_2 that can be generated from the grammar G_2 for which $T_1 \models C < D$ if and only if $T_2 \models C < D$.*

Proof. The proof of the proposition consists of three parts: termination, translation, and equivalence. In other words, we have to prove that the process of exhaustive rule application terminates; that a terminology T_1 generated by $\mathbf{G_1}$ is transformed into a terminology T_2 that can be generated by $\mathbf{G_2}$; and that $T_1 \models C < D$ if and only if $T_2 \models C < D$.

Part 1: Proof of termination. To prove that an exhaustive application of the rewrite rules in Table A.1 terminates, we will assign *costs* to path functions, descriptions, subsumption

constraints and terminologies. Let us consider a function, *Cost*[], that "charges" the following amounts for the expressions generated by $\mathbf{G_1}$[29]:

$$Cost[C] = 0;$$
$$Cost[(\textbf{all } Pf\ D)] = 1 + Cost[Pf] + 2 \cdot Cost[D];$$
$$Cost[(\textbf{fd } C\ Pfs\ Pf)] = 3 + Cost[Pfs] + Cost[Pf] + Cost_{sp}[Pf];$$
$$Cost[(\textbf{and } D_1\ D_2\ Ds)] = 1 + Cost[D_1] + 1 + Cost[D_2] + Cost[Ds];$$
$$Cost[A] = 0;$$
$$Cost[\textbf{Id}] = 1;$$
$$Cost[(\textbf{comp } Pf_1\ Pf_2)] = 2 + 4 \cdot Cost[Pf_1] + Cost[Pf_2] + Cost_{sp}[Pf_2];$$
$$Cost[Pfs\ Pf] = Cost[Pfs] + Cost[Pf] + Cost_{sp}[Pf];$$
$$Cost[\varepsilon] = 0;$$
$$Cost[Ds\ D] = Cost[Ds] + 1 + Cost[D];$$

where $Cost_{sp}[Pf]$ is 10 if Pf is a primitive attribute A, and 0 otherwise. For any subsumption constraint $C < D$, we define its cost as $Cost[D]$, and for any terminology, we define its cost as sum of all costs of the constraints in the terminology. It is not hard to see that for any constraint generated by $\mathbf{G_1}$, its cost is non-negative (since it is computed as a number of additions and multiplications of non-negative numbers). Therefore, for any terminology, its cost is also non-negative (as well as finite since we add and multiply a finite number of non-negative numbers).

Also note that after applying any rule in Table A.1 to a constraint generated by grammar $\mathbf{G_1}$, the resulting constraint(s) can also be generated by $\mathbf{G_1}$. This fact follows from the observation that the right-hand-sides of the rules can be generated from the grammar assuming their components (such as D, D_1, D_2, A, Pf, Pfs, Pf_1, Pf_2, C, etc.) can. However, since right-hand-sides only use components of the left-hand-sides (with the exception of primitive concept C_2 that can be generated from $\mathbf{G_1}$ by the first production), and since left-hand-sides can be generated from $\mathbf{G_1}$ by assumption, so can the resulting subsumption constraint(s). Therefore, to prove that the process of exhaustive rule application to a terminology (generated by $\mathbf{G_1}$) terminates, it is sufficient to show that an application of any rewrite rule to a constraint decreases the (finite, integer and non-negative) cost of the terminology.

Further note that the first seven rewrite rules deal with rewriting concept descriptions while the last three rules rewrite attribute descriptions. Since attribute descriptions do not contain concept descriptions, and since the cost of every inner concept description is always added to the total cost of the outer (concept) description by our cost function definition, it is sufficient to show

[29] Note that by C (A), we mean any symbol in D that is replaced by **C** (**A**) before its derivation from $\mathbf{G_1}$. We use *italic* symbols here in place of **bold** non-terminals to indicate strings generated from the corresponding non-terminals, and we use subscripts to distinguish between different strings generated from the same non-terminal.

that the cost of the right-hand-side of each of the seven rules is less than the cost of the corresponding left-hand-side.

On the other hand, note that the cost of an inner attribute description is either added (and possibly modified by a constant positive factor) to the cost of the outer description by itself or with the addition of the special cost. However, both left- and right-hand-sides of rewrite rules 9 and 10 as well as any attribute description that contains them have special costs of 0. It follows that in order to prove that an application of any of these rules decreases the total cost of the terminology, it is also sufficient to show that the cost of the right-hand-side of the rule is less than the cost of the corresponding left-hand-side.

Now consider rule 8. An attribute description Pf' that might contain the left-hand-side of the rule, (**comp Id** Pf), computes its cost using a product of a positive constant factor and either the cost of the left-hand-side of the rule by itself or in addition to its special cost. However, since Pf' and any "higher-level" descriptions containing Pf' have special costs of 0, it is sufficient to show that the cost of the right-hand-side, $Cost[Pf]$, of rule 8 is less than the cost of the left-hand-side, $Cost$[(**comp Id** Pf)], as well as that the cost of the right-hand-side plus its special cost, $Cost_{sp}[Pf]$, is less than the cost of the left-hand-side plus its special cost $Cost_{sp}$[(**comp Id** Pf)].

The following shows the sufficient differences of the costs of the right- and left-hand-sides of the rewrite rules. All of the equalities follow directly from the definitions of costs.

(1) $Cost[C < (\textbf{all Id } D)] - Cost[C < D] = Cost[(\textbf{all Id } D)] - Cost[D] =$
$2 + 2 \cdot Cost[D] - Cost[D] > 0;$

(2) $Cost[C_1 < (\textbf{all } (\textbf{comp } Pf_1\ Pf_2)\ D)] - Cost[C_1 < (\textbf{all } Pf_1\ C_2)] - Cost[C_2 < (\textbf{all } Pf_2\ D)] =$
$1 + (2 + 4 \cdot Cost[Pf_1] + Cost[Pf_2] + Cost_{sp}[Pf_2]) + 2 \cdot Cost[D] -$
$(1 + Cost[Pf_1]) - (1 + Cost[Pf_2] + 2 \cdot Cost[D]) =$
$1 + 3 \cdot Cost[Pf_1] + Cost_{sp}[Pf_2] > 0;$

(3) $Cost[C_1 < (\textbf{all } A\ (\textbf{all } Pf\ D))] - Cost[C_1 < (\textbf{all } A\ C_2)] - Cost[C_2 < (\textbf{all } Pf\ D)] =$
$1 + Cost[A] + 2 \cdot (1 + Cost[Pf] + 2 \cdot Cost[D]) -$
$(1 + Cost[A]) - (1 + Cost[Pf] + 2 \cdot Cost[D]) =$
$1 + Cost[Pf] + 2 \cdot Cost[D] > 0;$

(4) $Cost[C_1 < (\textbf{all } A\ (\textbf{fd } C_3\ Pfs\ Pf))] - Cost[C_1 < (\textbf{all } A\ C_2)] - Cost[C_2 < (\textbf{fd } C_3\ Pfs\ Pf)] =$
$1 + Cost[A] + 2 \cdot (3 + Cost[Pfs] + Cost[Pf] + Cost_{sp}[Pf]) -$
$(1 + Cost[A]) - (3 + Cost[Pfs] + Cost[Pf] + Cost_{sp}[Pf]) =$
$3 + Cost[Pfs] + Cost[Pf] + Cost_{sp}[Pf] > 0;$

(5) $Cost[C < (\textbf{all}\ A\ (\textbf{and}\ D_1 \ldots D_k))] - Cost[C < (\textbf{all}\ A\ D_1)] - \ldots - Cost[C < (\textbf{all}\ A\ D_k)] =$
$1 + 2 \cdot (k + Cost[D_1] + \ldots + Cost[D_k]) - (1 + 2 \cdot Cost[D_1]) - \ldots -$
$(1 + 2 \cdot Cost[D_k]) = 1 + k > 0;$

(6) $Cost[C < (\textbf{and}\ D_1 \ldots D_k)] - Cost[C < D_1] - \ldots - Cost[C < D_k] =$
$k + Cost[D_1] + \ldots + Cost[D_k] - Cost[D_1] - \ldots - Cost[D_k] = k > 0;$

(7) $Cost[C_1 < (\textbf{fd}\ C_3\ Pfs_1\ A\ Pfs_2)] - Cost[C_1 < (\textbf{fd}\ C_3\ Pfs_1\ (\textbf{comp}\ A\ \textbf{Id})\ Pfs_2)] =$
$3 + Cost[Pfs_1] + 10 + Cost[Pfs_2] - (3 + Cost[Pfs_1] + (2 + 1) + Cost[Pfs_2]) =$
$7 > 0;$

(8) $Cost[(\textbf{comp}\ \textbf{Id}\ Pf)] - Cost[Pf] = 2 + 4 \cdot 1 + Cost[Pf] + Cost_{sp}[Pf] - \text{Cost}[Pf] =$
$6 + Cost_{sp}[Pf] > 0;$

$Cost[(\textbf{comp}\ \textbf{Id}\ Pf)] + Cost_{sp}[(\textbf{comp}\ \textbf{Id}\ Pf)] - Cost[Pf] - Cost_{sp}[Pf] =$
$2 + 4 \cdot 1 + Cost[Pf] + Cost_{sp}[Pf] + 0 - Cost[Pf] - Cost_{sp}[Pf] = 6 > 0;$

(9) $Cost[(\textbf{comp}\ Pf\ A)] - Cost[(\textbf{comp}\ Pf\ (\textbf{comp}\ A\ \textbf{Id}))] =$
$2 + 4 \cdot Cost[Pf] + 10 - (2 + 4 \cdot Cost[Pf] + (2 + 1)) = 7 > 0;$

(10) $Cost[(\textbf{comp}\ (\textbf{comp}\ Pf_1\ Pf_2)\ Pf_3)] - Cost[(\textbf{comp}\ Pf_1\ (\textbf{comp}\ Pf_2\ Pf_3))] =$
$2 + 4 \cdot (2 + 4 \cdot Cost[Pf_1] + Cost[Pf_2] + Cost_{sp}[Pf_2]) + Cost[Pf_3] + Cost_{sp}[Pf_3] -$
$(2 + 4 \cdot Cost[Pf_1] + (2 + 4 \cdot Cost[Pf_2] + Cost[Pf_3] + Cost_{sp}[Pf_3])) =$
$10 + 16 \cdot Cost[Pf_1] + 4 \cdot Cost[Pf_2] + 4 \cdot Cost_{sp}[Pf_2] + Cost[Pf_3] + Cost_{sp}[Pf_3] -$
$(4 + 4 \cdot Cost[Pf_1] + 4 \cdot Cost[Pf_2] + Cost[Pf_3] + Cost_{sp}[Pf_3]) =$
$6 + 12 \cdot Cost[Pf_1] + 4 \cdot Cost_{sp}[Pf_2] > 0.$

Part 2: Proof of translation. Next, we prove that after an exhaustive application of the rules from Table A.1 to a terminology T_1 generated by $\mathbf{G_1}$, one obtains a terminology T_2 that can be generated by $\mathbf{G_2}$.

As pointed out in the proof of termination, if the left-hand-side of a rule can be generated by $\mathbf{G_1}$, so can the right-hand-side(s); therefore, T_2 can be generated from $\mathbf{G_1}$. Let us assume T_2 cannot be generated from $\mathbf{G_2}$. Then, by our definitions, there must be a constraint $C < D$ in T_2 such that, after replacing every concept name in D by **C** and every primitive attribute name in D by **A**, D cannot be generated from $\mathbf{G_2}$ (and can be generated from $\mathbf{G_1}$).

Let us consider the process of generating D from $\mathbf{G_1}$. The first production used for this generation must be one of the four productions for **D**. However, for all four cases, we would obtain a contradiction as follows.

Production 1. **D** ::= **C** production also exists in $\mathbf{G_2}$ (production 1'), and therefore, we would be able to generate *D* from $\mathbf{G_2}$. Contradiction.

Production 2. (**all Pf D**). In this case, we should be able to generate *D* from (**all A C**). Indeed, if **Pf** did *not* generate **A**, it would have to generate either (**comp Pf Pf**), in which case rewrite rule 2 in Table A.1 would still be applicable contrary to our assumption that all rules were already exhaustively applied; or **Id**, in which case rule 1 would still be applicable.

Now that we know that **Pf** generates **A**, let us show that only **C** can be generated from **D**. We cannot generate (**all Pf D**) from **D**, since rule 3 in Table A.1 would be applicable. Analogously, we cannot generate (**fd C Pfs Pf**) because of rule 4, or (**and D D Ds**) because of rule 5. Therefore, *D* must have the form (**all** *A C*). However, we can generate such *D* from $\mathbf{G_2}$ by production 2'. Contradiction.

Production 3. (**fd C Pfs Pf**). Note that $\mathbf{G_2}$ has the same production (3'), and $\mathbf{G_1}$ and $\mathbf{G_2}$ have the same productions for **Pfs**. Therefore, we must have a **Pf** that cannot be generated from $\mathbf{G_2}$ (since all other non-terminal symbols can be generated by $\mathbf{G_2}$). Let us consider a string *Pf* generated from such **Pf**.

Pf cannot be **Id** as $\mathbf{G_2}$ would be able to generate it with production 6'. *Pf* cannot be an attribute **A** either, since a rewrite rule 7 in Table A.1 would be applicable. The only other choice we have for *Pf* is production 7 of $\mathbf{G_1}$. Thus, *Pf* has the form (**comp** $\mathbf{Pf_1}$ $\mathbf{Pf_2}$)[30].

Note that for *any* (**comp** $\mathbf{Pf_1}$ $\mathbf{Pf_2}$) expression, $\mathbf{Pf_1}$ cannot generate **Id** (by production 6 of $\mathbf{G_1}$) since rewrite rule 8 would be applicable. Analogously, $\mathbf{Pf_1}$ cannot generate (**comp** $\mathbf{Pf_3}$ $\mathbf{Pf_4}$) (by production 7 of $\mathbf{G_1}$) since the rule 10 would be applicable. Therefore, $\mathbf{Pf_1}$ must generate **A** (by production 5 of $\mathbf{G_1}$).

Since, as we just showed, any (**comp** $\mathbf{Pf_1}$ $\mathbf{Pf_2}$) can only generate (**comp A** $\mathbf{Pf_2}$), *Pf* can only have the form (**comp A** (**comp A** ... (**comp A** $\mathbf{Pf_{final}}$))) where $\mathbf{Pf_{final}}$ is either **Id** or **A**. Indeed, for any (**comp** $\mathbf{Pf_1}$ $\mathbf{Pf_2}$), $\mathbf{Pf_2}$ can only be either (**comp A** $\mathbf{Pf_3}$) which just adds another level of nesting, or **Id** or **A** which is $\mathbf{Pf_{final}}$.

Now, note that $\mathbf{Pf_{final}}$ cannot be **A** since rewrite rule 9 would be applicable. Therefore, our *Pf* must have the form (**comp A** (**comp A** ... (**comp A Id**))), but this expression can be generated from $\mathbf{G_2}$ by productions 6' and 7'. Contradiction.

Production 4. (**and D D Ds**). In this case, rule 6 in Table A.1 would still be applicable. Contradiction.

Part 3: Proof of equivalence. Suppose we obtained a terminology *T'* after exhaustively applying transformations in Table A.1 to a terminology *T*. We have to prove that $T \models E < F$ if and only if $T' \models E < F$ for any subsumption constraint $E < F$.

---

[30] We use subscripts in (**comp** $\mathbf{Pf_1}$ $\mathbf{Pf_2}$) in order to distinguish between different **Pf** non-terminals.

Consider the process of applying rules from Table A.1. It is sufficient to prove that for every step of the process (which we know terminates), when a terminology T_1 is rewritten into a terminology T_2 after applying just *one* rule from Table A.1, $T_1 \models E < F$ if and only if $T_2 \models E < F$ for any subsumption constraint $E < F$.

We are going to use the following three methods to prove that an application of a rule to a constraint of a terminology produces an "equivalent" terminology.

Method 1. Consider a subsumption constraint S_1 in T_1 and let us denote the rest of the constraints by T_{const} (i.e. $T_1 = \{S_1\} \cup T_{const}$). Also, consider a *set* S_2 of the constraints that are produced by some rule applied to S_1 (i.e. $T_2 = S_2 \cup T_{const}$).

To prove that $T_1 \models E < F$ *if* $T_2 \models E < F$ for any subsumption constraint $E < F$, it is sufficient to prove that $\{S_1\} \models S_2$ (i.e. terminology $\{S_1\} \models S$, for any $S \in S_2$). Indeed, we can conclude that if $T_2 \models E < F$ then

$$T_1 = \{S_1\} \cup T_{const} \models S_2 \cup T_{const} = T_2 \models E < F.^{31}$$

The method can be also applied in the reverse direction: to prove that that $T_1 \models E < F$ *only if* $T_2 \models E < F$ for any subsumption constraint $E < F$, it is sufficient to prove that $S_2 \models S_1$, since if $T_1 \models E < F$ then

$$T_2 = S_2 \cup T_{const} \models \{S_1\} \cup T_{const} = T_1 \models E < F.^{31}$$

Method 2. If a rule replaces a constraint $G < H$ by a constraint $G < H'$, then it is sufficient to prove that $H^{\mathbf{I}} = H'^{\mathbf{I}}$ for any interpretation $\mathbf{I}$. Indeed, directly from definitions of subsumption and logical implication, $H^{\mathbf{I}} = H'^{\mathbf{I}}$ imply that $\{G < H\} \models G < H'$ and $\{G < H'\} \models G < H$; and therefore, by Method 1, $T_1 \models E < F$ if and only if $T_2 \models E < F$ for any constraint $E < F$.

In addition, since interpretation of a description only depends on the interpretations of its components, if a rule replaces a component of a description by another component with the same interpretation, the interpretation of the whole description will be unchanged. Thus, it is sufficient to prove the equality of the interpretations of only the components that changed.

Method 3. This method is essentially a proof by contradiction. Proving that $T_1 \models E < F$ if $T_2 \models E < F$ for any constraint $E < F$ means verifying that any interpretation $\mathbf{I}$ that satisfies every

[31] Directly from the definition of logical implication, it follows that if a terminology T_1 logically implies every subsumption constraint of terminology T_2, and if $T_2 \models S$ for some subsumption constraint S, then $T_1 \models S$.

member of T_1 also satisfies constraint $E < F$ given that any interpretation **J** that satisfies every member of T_2 also satisfies $E < F$. Then we would assume that this is *not* true and uncover a contradiction. In other words, we would assume that there is a constraint $E < F$ such that

(a) any interpretation **J** that satisfies every member of T_2 also satisfies $E < F$;
(b) there is an interpretation **I** that satisfies every member of T_1; and
(c) **I** does *not* satisfy $E < F$.

Using the methods above, we can now prove the equivalence for every rule in Table A.1:

Rule 1: Replace $C <$ (**all Id** D) by $C < D$.

By Method 2, and from the definitions of the interpretation function,

$$(\textbf{all Id } D)^{\mathbf{I}} = \{x \in \Delta \mid x \in D^{\mathbf{I}}\} = D^{\mathbf{I}}.$$

Rule 2: Replace $C_1 <$ (**all** (**comp** $Pf_1\ Pf_2$) D) by $C_1 <$ (**all** $Pf_1\ C_2$) and $C_2 <$ (**all** $Pf_2\ D$).

Part (a): $T_1 \models E < F$ *if* $T_2 \models E < F\ \forall\ E < F$.

By Method 3, assume that there is a constraint $E < F$ such that

(i) any interpretation **J** that satisfies every member of T_2 also satisfies $E < F$;
(ii) there is an interpretation **I** that satisfies every member of T_1; and
(iii) **I** does *not* satisfy $E < F$.

Let us consider an interpretation **J** that is the same as the interpretation **I** for all concepts and attributes of T_1 and with interpretation of C_2 being

$$C_2^{\mathbf{J}} = (\textbf{all } Pf_2\ D)^{\mathbf{I}}.$$

Since, by the assumption of the proposition and by condition of the rule 2, C_2 is a new concept that does not appear anywhere in E or F, **J** does *not* satisfy $E < F$. On the other hand, since **J** is the same as **I** except for $C_2^{\mathbf{J}}$, it satisfies all the constraints in T_{const}. It also satisfies

$$C_2 < (\textbf{all } Pf_2\ D)$$

by definition of $C_2^{\mathbf{J}}$. (Note that (**all** $Pf_2\ D)^{\mathbf{I}} = (\textbf{all } Pf_2\ D)^{\mathbf{J}}$ since C_2 is a new concept not occurring in Pf_2 or D, and thus, interpretations of Pf_2 and D remain unchanged by the construction of **J**.)

Again, since C_2 is a new concept, **J** continues to satisfy

$$C_1 < (\textbf{all}\ (\textbf{comp}\ Pf_1\ Pf_2)\ D).$$

Finally, **J** also satisfies

$$C_1 < (\textbf{all}\ Pf_1\ C_2),$$

since otherwise, there would be an object o_1 in $C_1^{\mathbf{J}}$ such that $o_2 = Pf_1^{\mathbf{J}}(o_1) \notin C_2^{\mathbf{J}}$. However, since **J** satisfies $C_1 < (\textbf{all}\ (\textbf{comp}\ Pf_1\ Pf_2)\ D)$, we know that $Pf_2^{\mathbf{J}}(Pf_1^{\mathbf{J}}(o_1)) \in D^{\mathbf{J}}$, i.e. $Pf_2^{\mathbf{J}}(o_2) \in D^{\mathbf{J}}$; and since $C_2^{\mathbf{J}} = (\textbf{all}\ Pf_2\ D)^{\mathbf{I}} = (\textbf{all}\ Pf_2\ D)^{\mathbf{J}}$, we know that $o_2 \in C_2^{\mathbf{J}}$ which leads to contradiction.

Therefore, we found an interpretation **J**, which satisfies every member of T_2 but does *not* satisfy $E < F$. This contradicts to (i).

Part (b): $T_1 \models E < F$ *only if* $T_2 \models E < F\ \forall\ E < F$.

By Method 1, it suffices to show that

$$\{C_1 < (\textbf{all}\ Pf_1\ C_2),\ C_2 < (\textbf{all}\ Pf_2\ D)\} \models C_1 < (\textbf{all}\ (\textbf{comp}\ Pf_1\ Pf_2)\ D).$$

By definitions of interpretation, we have

$$C_1^{\mathbf{I}} \subseteq \{x \in \Delta \mid Pf_1^{\mathbf{I}}(x) \in C_2^{\mathbf{I}}\}, \text{ and}$$
$$C_2^{\mathbf{I}} \subseteq \{y \in \Delta \mid Pf_2^{\mathbf{I}}(y) \in D^{\mathbf{I}}\}\ \forall\ \mathbf{I}.$$

Therefore,

$$C_1^{\mathbf{I}} \subseteq \{x \in \Delta \mid Pf_1^{\mathbf{I}}(x) \in \{y \in \Delta \mid Pf_2^{\mathbf{I}}(y) \in D^{\mathbf{I}}\}\} = \{x \in \Delta \mid Pf_2^{\mathbf{I}}(Pf_1^{\mathbf{I}}(x)) \in D^{\mathbf{I}}\} =$$
$$\{x \in \Delta \mid (\textbf{comp}\ Pf_1\ Pf_2)^{\mathbf{I}}(x) \in D^{\mathbf{I}}\} = (\textbf{all}\ (\textbf{comp}\ Pf_1\ Pf_2)\ D)^{\mathbf{I}}\ \forall\ \mathbf{I}.$$

Rules 3 and 4: Both rules 3 and 4 are instances of the following more general rule: Replace $C_1 <$ (**all** $A\ D$) by $C_1 <$ (**all** $A\ C_2$) and $C_2 < D$. Therefore, it is sufficient to prove the equivalence for this more general rule.

Part (a): $T_1 \models E < F$ *if* $T_2 \models E < F\ \forall\ E < F$.

By Method 3, assume that there is a constraint $E < F$ such that

(i) any interpretation **J** that satisfies every member of T_2 also satisfies $E < F$;
(ii) there is an interpretation **I** that satisfies every member of T_1; and
(iii) **I** does *not* satisfy $E < F$.

Let us consider an interpretation **J** that is the same as the interpretation **I** for all the concepts and attributes of T_1 and with interpretation of C_2 being

$$C_2^{\mathbf{J}} = D^{\mathbf{I}}.$$

Since, by the assumption of the proposition and by condition of the rules 3 and 4, C_2 is a new concept that does not appear anywhere in E or F, **J** does *not* satisfy $E < F$. On the other hand, since **J** is the same as **I** except for $C_2^{\mathbf{J}}$, it satisfies all the constraints in T_{const}. It also satisfies

$$C_2 < D$$

by definition of $C_2^{\mathbf{J}}$. (Note that $D^{\mathbf{I}} = D^{\mathbf{J}}$ since C_2 is a new concept not occurring in D, and thus, interpretation of D remains unchanged by the construction of **J**.)

Again, since C_2 is a new concept, **J** continues to satisfy

$$C_1 < (\mathbf{all}\ A\ D).$$

Finally, **J** also satisfies $C_1 < (\mathbf{all}\ A\ C_2)$ since $C_2^{\mathbf{J}} = D^{\mathbf{I}} = D^{\mathbf{J}}$, and thus, $(\mathbf{all}\ A\ C_2)^{\mathbf{J}} = (\mathbf{all}\ A\ D)^{\mathbf{J}}$.

Therefore, we found an interpretation **J**, which satisfies every member of T_2 but does *not* satisfy $E < F$. This contradicts to (i).

Part (b): $T_1 \models E < F$ *only if* $T_2 \models E < F\ \forall\ E < F$.

By Method 1, it suffices to show that

$$\{C_1 < (\mathbf{all}\ A\ C_2),\ C_2 < D\} \models C_1 < (\mathbf{all}\ A\ D).$$

By definitions of interpretation, we have

$$C_1^{\mathbf{I}} \subseteq \{x \in \Delta \mid A^{\mathbf{I}}(x) \in C_2^{\mathbf{I}}\},\ \text{and}\ C_2^{\mathbf{I}} \subseteq D^{\mathbf{I}}\ \forall\ \mathbf{I}.$$

Therefore, $C_1^{\mathbf{I}} \subseteq \{x \in \Delta \mid A^{\mathbf{I}}(x) \in D^{\mathbf{I}}\} = (\mathbf{all}\ A\ D)^{\mathbf{I}}\ \forall\ \mathbf{I}$.

Rule 5: Replace $C < (\mathbf{all}\ A\ (\mathbf{and}\ D_1 \ldots D_k))$ by $C < (\mathbf{all}\ A\ D_1), \ldots, C < (\mathbf{all}\ A\ D_k)$.

By Method 1, it suffices to prove that

$\{ C < (\textbf{all}\ A\ (\textbf{and}\ D_1 \ldots D_k)) \} \models \{ C < (\textbf{all}\ A\ D_1), \ldots, C < (\textbf{all}\ A\ D_k) \}$ and
$\{ C < (\textbf{all}\ A\ D_1), \ldots, C < (\textbf{all}\ A\ D_k) \} \models \{ C < (\textbf{all}\ A\ (\textbf{and}\ D_1 \ldots D_k)) \}$.

Both statements follow directly from the fact that by definitions of interpretation and subsumption, for any interpretation **I**, **I** satisfies $C < (\textbf{all}\ A\ (\textbf{and}\ D_1 \ldots D_k))$, i.e.

$$C^{\mathbf{I}} \subseteq (\textbf{all}\ A\ (\textbf{and}\ D_1 \ldots D_k))^{\mathbf{I}} = \{x \in \Delta \mid A^{\mathbf{I}}(x) \in D_1{}^{\mathbf{I}} \cap \ldots \cap D_k{}^{\mathbf{I}}\},$$

if and only if **I** satisfies constraints $C < (\textbf{all}\ A\ D_1)$ through $C < (\textbf{all}\ A\ D_k)$, i.e.

$$C^{\mathbf{I}} \subseteq (\textbf{all}\ A\ D_1)^{\mathbf{I}} = \{x \in \Delta \mid A^{\mathbf{I}}(x) \in D_1{}^{\mathbf{I}}\},$$
$$\ldots$$
$$C^{\mathbf{I}} \subseteq (\textbf{all}\ A\ D_2)^{\mathbf{I}} = \{x \in \Delta \mid A^{\mathbf{I}}(x) \in D_k{}^{\mathbf{I}}\}.$$

Rule 6: Replace $C < (\textbf{and}\ D_1 \ldots D_k)$ by $C < D_1, \ldots, C < D_k$.

By Method 1, it suffices to prove that

$$\{C < (\textbf{and}\ D_1 \ldots D_k)\} \models \{ C < D_1, \ldots, C < D_k\} \text{ and}$$
$$\{ C < D_1, \ldots, C < D_k\} \models \{C < (\textbf{and}\ D_1 \ldots D_k)\}.$$

Both statements follow directly from the fact that by definitions of interpretation and subsumption, for any interpretation **I**, **I** satisfies $C < (\textbf{and}\ D_1 \ldots D_k)$, i.e.

$$C^{\mathbf{I}} \subseteq (\textbf{and}\ D_1 \ldots D_k)^{\mathbf{I}} = D_1{}^{\mathbf{I}} \cap \ldots \cap D_k{}^{\mathbf{I}},$$

if and only if **I** satisfies constraints $C < D_1$ through $C < D_k$, i.e.

$$C^{\mathbf{I}} \subseteq D_1{}^{\mathbf{I}}, \ldots, C^{\mathbf{I}} \subseteq D_k{}^{\mathbf{I}}.$$

Rule 7: Replace $C_1 < (\textbf{fd}\ C_3\ Pfs_1\ A\ Pfs_2)$ by $C_1 < (\textbf{fd}\ C_3\ Pfs_1\ (\textbf{comp}\ A\ \textbf{Id})\ Pfs_2)$.

By Method 2, since the only component that changed is A replaced by $(\textbf{comp}\ A\ \textbf{Id})$, it suffices to show that $A^{\mathbf{I}} = (\textbf{comp}\ A\ \textbf{Id})^{\mathbf{I}}$. However, by definitions,

$$(\textbf{comp}\ A\ \textbf{Id})^{\mathbf{I}} = \{ (x, y) \mid x \in \Delta \text{ and } \mathbf{Id}^{\mathbf{I}}(A^{\mathbf{I}}(x)) = y \} = \{ (x, y) \mid x \in \Delta \text{ and } A^{\mathbf{I}}(x) = y \} = A^{\mathbf{I}}.$$

Rule 8: Replace $(\textbf{comp}\ \textbf{Id}\ Pf)$ by Pf.

By Method 2, it is sufficient to show that $(\textbf{comp}\ \textbf{Id}\ Pf)^{\mathbf{I}} = Pf^{\mathbf{I}}$. Indeed,

$$(\mathbf{comp}\ \mathbf{Id}\ Pf)^{\mathbf{I}} = \{\ (x, y) \mid x \in \Delta \text{ and } Pf^{\mathbf{I}}(\mathbf{Id}^{\mathbf{I}}(x)) = y\ \} = \{\ (x, y) \mid x \in \Delta \text{ and } Pf^{\mathbf{I}}(x) = y\ \}.$$

However, since by definition, for any $x \in \Delta$, $Pf^{\mathbf{I}}(x) = y$ where $(x, y) \in Pf^{\mathbf{I}}$,

$$Pf^{\mathbf{I}} = \{\ (x, y) \mid x \in \Delta \text{ and } Pf^{\mathbf{I}}(x) = y\ \}.$$

Rule 9: Replace (**comp** $Pf\,A$) by (**comp** Pf (**comp** A **Id**)).

By Method 2, it is sufficient to show that (**comp** $Pf\,A)^{\mathbf{I}}$ = (**comp** Pf (**comp** A **Id**))$^{\mathbf{I}}$. Indeed,

$$(\mathbf{comp}\ Pf\,A)^{\mathbf{I}} = \{\ (x, y) \mid x \in \Delta \text{ and } A^{\mathbf{I}}(Pf^{\mathbf{I}}(x)) = y\ \} = \{\ (x, y) \mid x \in \Delta \text{ and } \mathbf{Id}^{\mathbf{I}}(A^{\mathbf{I}}(Pf^{\mathbf{I}}(x))) = y\ \} =$$
$$\{\ (x, y) \mid x \in \Delta \text{ and } (\mathbf{comp}\ A\ \mathbf{Id})^{\mathbf{I}}(Pf^{\mathbf{I}}(x)) = y\ \} = (\mathbf{comp}\ Pf\ (\mathbf{comp}\ A\ \mathbf{Id}))^{\mathbf{I}}.$$

Rule 10: Replace (**comp** (**comp** $Pf_1\ Pf_2$) Pf_3) by (**comp** Pf_1 (**comp** $Pf_2\ Pf_3$)).

By Method 2, it is sufficient to show that

$$(\mathbf{comp}\ (\mathbf{comp}\ Pf_1\ Pf_2)\ Pf_3)^{\mathbf{I}} = (\mathbf{comp}\ Pf_1\ (\mathbf{comp}\ Pf_2\ Pf_3))^{\mathbf{I}},$$

and indeed,

$$(\mathbf{comp}\ (\mathbf{comp}\ Pf_1\ Pf_2)\ Pf_3)^{\mathbf{I}} = \{\ (x, y) \mid x \in \Delta \text{ and } Pf_3^{\mathbf{I}}((\mathbf{comp}\ Pf_1\ Pf_2)^{\mathbf{I}}(x)) = y\ \} =$$
$$\{\ (x, y) \mid x \in \Delta \text{ and } Pf_3^{\mathbf{I}}(Pf_2^{\mathbf{I}}(Pf_1^{\mathbf{I}}(x))) = y\ \}, \text{ and}$$

$$(\mathbf{comp}\ Pf_1\ (\mathbf{comp}\ Pf_2\ Pf_3))^{\mathbf{I}} = \{\ (x, y) \mid x \in \Delta \text{ and } (\mathbf{comp}\ Pf_2\ Pf_3)^{\mathbf{I}}(Pf_1^{\mathbf{I}}(x)) = y\ \} =$$
$$\{\ (x, y) \mid x \in \Delta \text{ and } Pf_3^{\mathbf{I}}(Pf_2^{\mathbf{I}}(Pf_1^{\mathbf{I}}(x))) = y\ \}.$$

Appendix B

Simplifying Terminology for the General Logical Implication Problem

We prove Proposition 5.1 that relates to a mapping presented in Chapter 5 for constructing so-called *molecular terminologies*. As in Appendix A, we begin by reviewing the proposition, starting with the relevant grammars and rewrite rules.

General grammar:

(1) **D** ::= **C** | **V**
(2) | **(all Pf D)**
(3) | **(fd D Pfs Pf)**
(4) | **(and D D Ds)**
(5) | **(same-as Pf Pf)**

(6) **Pf** ::= **A**
(7) | **Id**
(8) | **(comp Pf Pf)**

(9) **Pfs** ::= ε
(10) | **Pfs Pf**

(11) **Ds** ::= ε
(12) | **Ds D**

Simplified grammar:

(1') **D** ::= **C** | **V**
(2') | **(all A D)**
(3') | **(fd D Pfs Pf)**
(4') | **(and D D Ds)**
(5') | **(same-as Pf Pf)**

(6') **Pf** ::= **Id**
(7') | **(comp A Pf)**

Rule number	Replace	By
1	$C < D_1$ and $C < D_2$	$C <$ (**and** D_1 D_2)
2	(**all Id** D)	D
3	(**all** (**comp** Pf_1 Pf_2) D)	(**all** Pf_1 (**all** Pf_2 D))
4	(**fd** D Pfs_1 A Pfs_2)	(**fd** D Pfs_1 (**comp** A **Id**) Pfs_2)
5	(**same-as** A Pf)	(**same-as** (**comp** A **Id**) Pf)
6	(**same-as** Pf A)	(**same-as** Pf (**comp** A **Id**))
7	(**comp Id** Pf)	Pf
8	(**comp** Pf A)	(**comp** Pf (**comp** A **Id**))
9	(**comp** (**comp** Pf_1 Pf_2) Pf_3)	(**comp** Pf_1 (**comp** Pf_2 Pf_3))

Table B.1: Molecular simplification of terminology in Chapter 5 (revisited).

(8′) **Pfs** ::= ε
(9′) | **Pfs Pf**

(10′) **Ds** ::= ε
(11′) | **Ds D**

Let us call the general grammar $\mathbf{G_1}$ and the simplified grammar $\mathbf{G_2}$. We say a description D is *generated* by a *grammar* ($\mathbf{G_1}$ or $\mathbf{G_2}$) if, after replacing every primitive concept name in D by **C**, every view name by **V**, and every primitive attribute name by **A**, the resulting description can be generated by the grammar. We say a constraint $D_1 < D_2$ (or $D_1 \equiv D_2$) is generated by a grammar if both D_1 and D_2 are generated by the grammar. Finally, we say a *terminology is generated by a grammar* if every constraint in the terminology is generated by the grammar.

In addition, for every rewrite rule in Table B.1, we call the string to be replaced the *left-hand-side* of the rule, and the string that the left-hand-side is replaced with the *right-hand-side* of the rule.

Proposition 5.1 (*revisited*) *Let T_1 denote a terminology and $D_1 < D_2$ a subsumption constraint. Then an exhaustive application of the rewrite rule 1 in Table B.1 to constraints in terminology T_1 followed by exhaustive application of the rewrite rules 2 through 9 in Table B.1 to constraints inside the resulting terminology and to constraint $D_1 < D_2$ terminates and obtains a terminology T_2 and a constraint $D_1' < D_2'$ respectively that can be generated from the simplified grammar and for which $T_1 \models D_1 < D_2$ if and only if $T_2 \models D_1 < D_2$, and $T_1 \models D_1 < D_2$ if and only if $T_1 \models D_1' < D_2'$. In addition, no two constraints in T_2 have the same left-hand-side.*

Proof. First, consider an exhaustive application of the rewrite rule 1 to a terminology T_1. Since every application of the rule decrements a (finite) number of constraints in the terminology, the

process of exhaustive application of the rule terminates and obtains some terminology T_1'. Moreover, at the end of the process, no two constraints in T_1' have the same primitive concept on the left-hand-side, since otherwise, rule 1 would still be applicable. This property is also preserved by applications of other rewrite rules, since they only rewrite descriptions inside constraints and do not create any new constraints. Finally, T_1 and T_1' have the same consequences, since analogously to Method 1 described in the equivalence part of the proof in Appendix A,

$$\{C < (\textbf{and } D_1\ D_2)\} \models \{C < D_1, C < D_2\} \text{ and } \{C < D_1, C < D_2\} \models \{C < (\textbf{and } D_1\ D_2)\}.$$

Finally, note that T_1' can still be generated by $\mathbf{G_1}$. Therefore, it is sufficient to show that the following modified proposition that concentrates on the second exhaustive application process holds:

Proposition B.1 *Let T_1 denote a terminology and $D_1 < D_2$ a subsumption constraint. Then an exhaustive application of the rewrite rules 2 through 9 in Table B.1 to constraints inside the terminology T_1 and to constraint $D_1 < D_2$ terminates and obtains a terminology T_2 and a constraint $D_1' < D_2'$ respectively that can be generated from the simplified grammar and for which $T_1 \models D_1 < D_2$ if and only if $T_2 \models D_1 < D_2$, and $T_1 \models D_1 < D_2$ if and only if $T_1 \models D_1' < D_2'$.*

As in Appendix A, the proof of the proposition consists of three parts: termination, translation, and equivalence. In other words, we prove that the process of exhaustive rule application terminates; that every description generated by $\mathbf{G_1}$ is transformed into a description that can be generated by $\mathbf{G_2}$; and that $T_1 \models D_1 < D_2$ if and only if $T_2 \models D_1 < D_2$, and $T_1 \models D_1 < D_2$ if and only if $T_1 \models D_1' < D_2'$.

Part 1: Proof of termination. To prove that an exhaustive application of the rewrite rules 2 through 9 in Table B.1 terminates, we assign finite non-negative *costs* to attribute and concept descriptions. Then, we show that application of any of the rules 2 through 9 to a description decreases the cost of the description, and therefore, there is at most a finite number of times that the rules can be applied to a description. Termination then is a trivial consequence of the facts that the rules are applied to descriptions inside the constraints independently, and the number of descriptions in a terminology is finite.

Let us consider a function, *Cost*[], that "charges" the following amounts for the expressions generated by non-terminals of $\mathbf{G_1}$[32]:

---

[32] Note that by *C*, *V*, and *A* we mean any symbol in *D* that is replaced by **C**, **V**, and **A** respectively before its derivation from $\mathbf{G_1}$. We use other *italic* symbols here in place of the **bold** non-terminals to indicate strings

$Cost[C] = 0$; $Cost[V] = 0$;
$Cost[(\textbf{all}\ Pf\ D)] = 1 + Cost[Pf] + Cost[D]$;
$Cost[(\textbf{fd}\ D\ Pfs\ Pf)] = 3 + Cost[D] + Cost[Pfs] + Cost[Pf] + Cost_{sp}[Pf]$;
$Cost[(\textbf{and}\ D_1\ D_2\ Ds)] = 1 + Cost[D_1] + Cost[D_2] + Cost[Ds]$;
$Cost[(\textbf{same-as}\ Pf_1\ Pf_2)] = 1 + Cost[Pf_1] + Cost_{sp}[Pf_1] + Cost[Pf_2] + Cost_{sp}[Pf_2]$;
$Cost[A] = 0$;
$Cost[\textbf{Id}] = 1$;
$Cost[(\textbf{comp}\ Pf_1\ Pf_2)] = 2 + 4 \cdot Cost[Pf_1] + Cost[Pf_2] + Cost_{sp}[Pf_2]$;
$Cost[Pfs\ Pf] = Cost[Pfs] + Cost[Pf] + Cost_{sp}[Pf]$;
$Cost[\varepsilon] = 0$;
$Cost[Ds\ D] = Cost[Ds] + Cost[D]$;

where $Cost_{sp}[Pf]$ is 10 if Pf is a primitive attribute A, and 0 otherwise. It is not hard to see that for any description generated by $\mathbf{G_1}$, its cost is finite and non-negative (as it is computed as a finite number of additions and multiplications of non-negative numbers).

Also note that after applying any of the rules 2 through 9 in Table B.1 to a description generated by grammar $\mathbf{G_1}$, the resulting description can also be generated by $\mathbf{G_1}$. This fact follows from the observation that the right-hand-sides of the rules can be generated from the grammar assuming their components (such as D, A, Pf, Pfs_1, Pfs_2, Pf_1, and Pf_2) can. However, since right-hand-sides only use components of the left-hand-sides, and since left-hand-sides can be generated from $\mathbf{G_1}$ by assumption, so can the resulting description. Therefore, to prove that the process of the exhaustive rule application to a description (generated by $\mathbf{G_1}$) terminates, it is sufficient to show that an application of any of the rewrite rules 2 through 9 to any description decreases its (finite, integer and non-negative) cost.

Next, note that rewrite rules 2 through 6 deal with rewriting concept descriptions, while the last three rules rewrite attribute descriptions. Since attribute descriptions do not contain concept descriptions, and since the cost of every inner concept description is always added to the total cost of the outer (concept) description by our cost function definition, it is sufficient to show that the cost of the right-hand-side of each of the rules 2 through 6 is less than the cost of the corresponding left-hand-side.

On the other hand, note that the cost of an inner attribute description is either added (and possibly modified by a constant positive factor) to the cost of the outer description by itself or with addition of the special cost. However, both left- and right-hand-sides of rewrite rules 8 and 9 as well as any attribute description containing them have special costs of 0. It follows that in order to prove that an application of any of these rules decreases the total cost of a description, it

---

generated from the corresponding non-terminals, and we use subscripts to distinguish between different strings generated from the same non-terminal.

is also sufficient to show that the cost of the right-hand-side of the rule is less than the cost of the corresponding left-hand-side.

Consider rule 7 now. An attribute description *Pf'* that might contain the left-hand-side of the rule, (**comp Id** *Pf*), computes its cost using a product of a positive constant factor and either the cost of the left-hand-side of the rule by itself or in addition to its special cost. However, since *Pf'* and any "higher-level" descriptions containing *Pf'* have special costs of 0, it is sufficient to show that the cost of the right-hand-side, $Cost[Pf]$, of rule 7 is less than the cost of the left-hand-side, $Cost[(\textbf{comp Id } Pf)]$, as well as that the cost of the right-hand-side plus its special cost, $Cost_{sp}[Pf]$, is less than the cost of the left-hand-side plus its special cost $Cost_{sp}[(\textbf{comp Id } Pf)]$.

The following shows the sufficient differences of the costs of the right- and left-hand-sides of the rewrite rules 2 through 9. All of the equalities follow directly from the definitions of costs.

(2) $Cost[(\textbf{all Id } D)] - Cost[D] = 2 + Cost[D] - Cost[D] > 0;$

(3) $Cost[(\textbf{all } (\textbf{comp } Pf_1\ Pf_2)\ D)] - Cost[(\textbf{all } Pf_1\ (\textbf{all } Pf_2\ D))] =$
$1 + (2 + 4 \cdot Cost[Pf_1] + Cost[Pf_2] + Cost_{sp}[Pf_2]) + Cost[D] -$
$(1 + Cost[Pf_1] + (1 + Cost[Pf_2] + Cost[D])) =$
$1 + 3 \cdot Cost[Pf_1] + Cost_{sp}[Pf_2] > 0;$

(4) $Cost[(\textbf{fd } D\ Pfs_1\ A\ Pfs_2)] - Cost[(\textbf{fd } D\ Pfs_1\ (\textbf{comp } A\ \textbf{Id})\ Pfs_2)] =$
$3 + Cost[D] + Cost[Pfs_1] + 10 + Cost[Pfs_2] -$
$(3 + Cost[D] + Cost[Pfs_1] + (2 + 1) + Cost[Pfs_2]) = 7 > 0;$

(5) $Cost[(\textbf{same-as } A\ Pf)] - Cost[(\textbf{same-as } (\textbf{comp } A\ \textbf{Id})\ Pf)] =$
$1 + 10 + Cost[Pf] + Cost_{sp}[Pf] - (1 + (2 + 1) + Cost[Pf] + Cost_{sp}[Pf]) =$
$7 > 0;$

(6) $Cost[(\textbf{same-as } Pf\ A)] - Cost[(\textbf{same-as } Pf\ (\textbf{comp } A\ \textbf{Id}))] =$
$1 + Cost[Pf] + Cost_{sp}[Pf] + 10 - (1 + Cost[Pf] + Cost_{sp}[Pf] + (2 + 1)) =$
$7 > 0;$

(7) $Cost[(\textbf{comp Id } Pf)] - Cost[Pf] = 2 + 4 \cdot 1 + Cost[Pf] + Cost_{sp}[Pf] - Cost[Pf] =$
$6 + Cost_{sp}[Pf] > 0;$

$Cost[(\textbf{comp Id } Pf)] + Cost_{sp}[(\textbf{comp Id } Pf)] - Cost[Pf] - Cost_{sp}[Pf] =$
$2 + 4 \cdot 1 + Cost[Pf] + Cost_{sp}[Pf] + 0 - Cost[Pf] - Cost_{sp}[Pf] = 6 > 0;$

(8) $Cost[(\textbf{comp}\ Pf\ A)] - Cost[(\textbf{comp}\ Pf\ (\textbf{comp}\ A\ \textbf{Id}))] =$
$2 + 4 \cdot Cost[Pf] + 10 - (2 + 4 \cdot Cost[Pf] + (2 + 1)) = 7 > 0;$

(9) $Cost[(\textbf{comp}\ (\textbf{comp}\ Pf_1\ Pf_2)\ Pf_3)] - Cost[(\textbf{comp}\ Pf_1\ (\textbf{comp}\ Pf_2\ Pf_3))] =$
$2 + 4 \cdot (2 + 4 \cdot Cost[Pf_1] + Cost[Pf_2] + Cost_{sp}[Pf_2]) + Cost[Pf_3] + Cost_{sp}[Pf_3] -$
$(2 + 4 \cdot Cost[Pf_1] + (2 + 4 \cdot Cost[Pf_2] + Cost[Pf_3] + Cost_{sp}[Pf_3])) =$
$10 + 16 \cdot Cost[Pf_1] + 4 \cdot Cost[Pf_2] + 4 \cdot Cost_{sp}[Pf_2] + Cost[Pf_3] + Cost_{sp}[Pf_3] -$
$(4 + 4 \cdot Cost[Pf_1] + 4 \cdot Cost[Pf_2] + Cost[Pf_3] + Cost_{sp}[Pf_3]) =$
$6 + 12 \cdot Cost[Pf_1] + 4 \cdot Cost_{sp}[Pf_2] > 0.$

Part 2: Proof of translation. Next, we prove that after an exhaustive application of the rules 2 through 9 in Table B.1 to a description D generated by $\mathbf{G_1}$, one obtains a description D' that can be generated by $\mathbf{G_2}$.

As pointed out in the proof of termination, if left-hand-side of a rewrite rule can be generated by $\mathbf{G_1}$, so can the right-hand-side; therefore, D' can be generated from $\mathbf{G_1}$. Thus, consider the following inductive proof on the number of productions 1 through 5 applied during the generation of D' by $\mathbf{G_1}$. (Validity of the induction is based on the fact that rewrite rules 2 through 9 in Table B.1 apply to and modify a concept description independently of whether it is inside of other descriptions.) At every step, we assume that D' cannot be derived from $\mathbf{G_2}$ and obtain a contradiction. In the base case, D' must be generated from G_1 either by production 1 or 5.

Production 1. Production **D** ::= **C** | **V** also exists in $\mathbf{G_2}$ (production 1'), and therefore, we would be able to generate D' from $\mathbf{G_2}$. Contradiction.

Production 5. Production **D** ::= (**same-as Pf Pf**) also exists in $\mathbf{G_2}$ (production 5'). Therefore, we must have an attribute description Pf generated from some **Pf** that cannot be generated by $\mathbf{G_2}$.

Pf cannot be **Id** as $\mathbf{G_2}$ would be able to generate it with production 6'. Pf cannot be an attribute **A** either since a rewrite rule 5 or 6 in Table B.1 would be applicable. The only other choice we have for Pf is production 8 of $\mathbf{G_1}$. Thus, Pf has the form (**comp** $\mathbf{Pf_1}$ $\mathbf{Pf_2}$)[33].

Note that for *any* (**comp** $\mathbf{Pf_1}$ $\mathbf{Pf_2}$) expression, $\mathbf{Pf_1}$ cannot generate **Id** (by production 7 of $\mathbf{G_1}$) since rewrite rule 7 would be applicable. Analogously, $\mathbf{Pf_1}$ cannot generate (**comp** $\mathbf{Pf_3}$ $\mathbf{Pf_4}$) (by production 8 of $\mathbf{G_1}$) since the rule 9 would be applicable. Therefore, $\mathbf{Pf_1}$ must generate **A** (by production 6 of $\mathbf{G_1}$).

Since, as we just showed, any (**comp** $\mathbf{Pf_1}$ $\mathbf{Pf_2}$) can only generate (**comp A** $\mathbf{Pf_2}$), Pf can only have the form (**comp A** (**comp A** ... (**comp A** $\mathbf{Pf_{final}}$))) where $\mathbf{Pf_{final}}$ is either **Id** or **A**. Indeed,

[33] We use subscripts in (**comp** $\mathbf{Pf_1}$ $\mathbf{Pf_2}$) in order to distinguish between different **Pf** non-terminals.

for any (**comp** $\mathbf{Pf_1}$ $\mathbf{Pf_2}$), $\mathbf{Pf_2}$ can only be either (**comp A** $\mathbf{Pf_3}$) which just adds another level of nesting, or **Id** or **A** which is $\mathbf{Pf_{final}}$.

Next, note that $\mathbf{Pf_{final}}$ cannot be **A** since rewrite rule 8 would be applicable. Therefore, our *Pf* must have the form (**comp A** (**comp A** ... (**comp A Id**))), but this expression can be generated from $\mathbf{G_2}$ by productions 6′ and 7′. Contradiction.

For the inductive step, let us assume that any description obtained by an exhaustive application of the rewrite rules 2 through 9 in Table B.1 that can be generated by $\mathbf{G_1}$ by a sequence of at most k ($k \geq 1$) productions 1 through 5 can be also generated from $\mathbf{G_2}$. Let us also assume that there is a description D' obtained by an exhaustive application of the rewrite rules 2 through 9 in Table B.1 that (a) can be generated by $\mathbf{G_1}$ by $k+1$ applications of productions 1 through 5; and (b) cannot be generated by $\mathbf{G_2}$. Since $k+1 \geq 2$, the first production applied during the generation of D' from $\mathbf{G_1}$ is either 2, 3, or 4. However, a contradiction can then be obtained in all three cases as follows.

Production 2. (**all Pf D**). In this case, we should be able to generate D' from (**all A D**). Indeed, if **Pf** did *not* generate **A**, it would have to generate either (**comp Pf Pf**), in which case rewrite rule 3 in Table B.1 would still be applicable contrary to our assumption that all rules were already exhaustively applied; or **Id**, in which case rule 2 would still be applicable. By inductive assumption however, since it takes no more than k productions 1 through 5 to generate **D** by $\mathbf{G_1}$, it can also be generated by $\mathbf{G_2}$. Contradiction.

Production 3. (**fd D Pfs Pf**). Note that $\mathbf{G_2}$ has the same production (3′), and $\mathbf{G_1}$ and $\mathbf{G_2}$ have the same productions for **Pfs** non-terminal. Therefore, we must have an attribute description *Pf* generated from some **Pf** that cannot be generated by $\mathbf{G_2}$ (recall that by induction, **D** can be generated by $\mathbf{G_2}$, since generation of description **D** from $\mathbf{G_1}$ uses at most k productions 1 through 5).

Pf cannot be **Id** as $\mathbf{G_2}$ would be able to generate it with production 6′. *Pf* cannot be an attribute **A** either, since the rewrite rule 4 in Table B.1 would be applicable. The only other choice we have for *Pf* is production 8 of $\mathbf{G_1}$. Thus, *Pf* has the form (**comp** $\mathbf{Pf_1}$ $\mathbf{Pf_2}$).

Then however, a contradiction can be obtained in exactly the same way as it was done when we discussed the (base) case of production 5.

Production 4. (**and D D Ds**). In this case, the description can be generated by $\mathbf{G_2}$ by production 4′ and due to the fact that all of the "inner" descriptions are derived by $\mathbf{G_1}$ with no more than k productions 1 through 5. Contradiction.

Part 3: Proof of equivalence. Suppose we obtained a terminology T' (resp. constraint E', either of the form $D_1' < D_2'$ or $D_1' \equiv D_2'$) after exhaustively applying rewrite rules 2 through 9 in Table B.1 to a terminology T (resp. some constraint E, either of the form $D_1 < D_2$ or $D_1 \equiv D_2$). We will prove that $T \models E$ if and only if $T' \models E$, and $T \models E$ if and only if $T \models E'$.

Consider an application of one of the rules 2 through 9 in Table B.1 to some description D_1 which replaces a description H inside it with a description F to obtain the new top-level description D_2. By the compositional nature of the definition of the interpretation function (i.e. by the fact that interpretation of a description directly depends only on the interpretation of its "parts"), it follows that if $F^{\mathbf{I}} = H^{\mathbf{I}}$ for any interpretation **I**, then $D_1^{\mathbf{I}} = D_2^{\mathbf{I}}$ for any interpretation **I**.

Moreover, if $D_1^{\mathbf{I}} = D_2^{\mathbf{I}}$ for any interpretation **I**, given a terminology T with constraint S of the form $C < D_1$, a terminology T' with constraint S' of the form $C < D_2$ in place of S, and some constraint S'', $T \models S''$ if and only if $T' \models S''$. Indeed, by definitions of satisfiability and logical implication, T logically implies every constraint in T' and vice versa. Therefore, if a terminology T (resp. T') logically implies every constraint of terminology T' (resp. T), and if $T' \models S''$ (resp. $T \models S''$) for some constraint S'', then $T \models S''$ (resp. $T' \models S''$).

Completely analogous reasoning applies if S (resp. S') has the form $C \equiv D$ (resp. $C \equiv D'$).

In addition, if we consider an application of one of the rules 2 through 9 in Table B.1 to S'' that obtains a constraint S''' such that both left- and right-hand-sides of S'' and S''' have the same interpretation for an arbitrary interpretation **I**, the reasoning analogous to the argument above allows us to deduce that for any terminology T, $T \models S''$ if and only if $T \models S'''$. (Consider a terminology containing only constraint S'' or S'''.)

It follows that in order to prove the equivalence, it is sufficient to show that for every rule 2 through 9 in Table B.1 that replaces its left-hand-side H with the right-hand-side F, $H^{\mathbf{I}} = F^{\mathbf{I}}$ for an arbitrary interpretation **I**.

Rule 2: Replace (**all Id** D) by D.

$$(\textbf{all Id } D)^{\mathbf{I}} = \{x \in \Delta \mid x \in D^{\mathbf{I}}\} = D^{\mathbf{I}}.$$

Rule 3: Replace (**all** (**comp** Pf_1 Pf_2) D) by (**all** Pf_1 (**all** Pf_2 D)).

$$(\textbf{all } (\textbf{comp } Pf_1\ Pf_2)\ D)^{\mathbf{I}} = \{\, x \in \Delta \mid Pf_2^{\mathbf{I}}(Pf_1^{\mathbf{I}}(x)) \in D^{\mathbf{I}} \,\} =$$
$$\{\, x \in \Delta \mid Pf_1^{\mathbf{I}}(x) \in (\textbf{all } Pf_2\ D)^{\mathbf{I}} \,\} = (\textbf{all } Pf_1\ (\textbf{all } Pf_2\ D))^{\mathbf{I}}.$$

Rules 4, 5, 6 and 8: A is replaced by (**comp** A **Id**).

$$(\textbf{comp } A\ \textbf{Id})^{\mathbf{I}} = \{\, (x, y) \mid x \in \Delta \text{ and } \textbf{Id}^{\mathbf{I}}(A^{\mathbf{I}}(x)) = y \,\} =$$

$$\{ (x, y) \mid x \in \Delta \text{ and } A^{\mathbf{I}}(x) = y \} = A^{\mathbf{I}}.$$

Rule 7: Replace (**comp Id** Pf) by Pf.

$$(\mathbf{comp\ Id}\ Pf)^{\mathbf{I}} = \{ (x, y) \mid x \in \Delta \text{ and } Pf^{\mathbf{I}}(\mathbf{Id}^{\mathbf{I}}(x)) = y \} =$$
$$\{ (x, y) \mid x \in \Delta \text{ and } Pf^{\mathbf{I}}(x) = y \} = Pf^{\mathbf{I}}.$$

Rule 9: Replace (**comp** (**comp** Pf_1 Pf_2) Pf_3) by (**comp** Pf_1 (**comp** Pf_2 Pf_3)).

$$(\mathbf{comp}\ (\mathbf{comp}\ Pf_1\ Pf_2)\ Pf_3)^{\mathbf{I}} = (\mathbf{comp}\ Pf_1\ (\mathbf{comp}\ Pf_2\ Pf_3))^{\mathbf{I}} \text{ since}$$

$$(\mathbf{comp}\ (\mathbf{comp}\ Pf_1\ Pf_2)\ Pf_3)^{\mathbf{I}} = \{ (x, y) \mid x \in \Delta \text{ and } Pf_3^{\mathbf{I}}((\mathbf{comp}\ Pf_1\ Pf_2)^{\mathbf{I}}(x)) = y \} =$$
$$\{ (x, y) \mid x \in \Delta \text{ and } Pf_3^{\mathbf{I}}(Pf_2^{\mathbf{I}}(Pf_1^{\mathbf{I}}(x))) = y \}, \text{ and}$$

$$(\mathbf{comp}\ Pf_1\ (\mathbf{comp}\ Pf_2\ Pf_3))^{\mathbf{I}} = \{ (x, y) \mid x \in \Delta \text{ and } (\mathbf{comp}\ Pf_2\ Pf_3)^{\mathbf{I}}(Pf_1^{\mathbf{I}}(x)) = y \} =$$
$$\{ (x, y) \mid x \in \Delta \text{ and } Pf_3^{\mathbf{I}}(Pf_2^{\mathbf{I}}(Pf_1^{\mathbf{I}}(x))) = y \}.$$

Bibliography

[AHU74] A.V. Aho, J.E. Hopcroft, and J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley Publishing Co., Reading, MA, 1974.

[AM86] H. Arisawa, and T. Miura, "On the Properties of Extended Inclusion Dependencies," *Proc. 12th Int'l Conf. on Very Large Data Bases*, pp. 449–456, Aug. 1986.

[Arm74] W.W. Armstrong, "Dependency Structures of Data Base Relationships," *Information Processing 74*, North-Holland Pub. Co., Amsterdam, pp. 580–583, 1974.

[ASU79] A.V. Aho, Y. Sagiv, and J.D. Ullman, "Efficient Optimization of a Class of Relational Expressions," *ACM Trans. Database Systems*, vol. 4, no. 4, pp. 435–454, Dec. 1979.

[BB79] C. Beeri, and P.A. Bernstein, "Computation Problems Related to the Design of Normal Form Relational Schemas," *ACM Trans. Database Systems*, vol. 4, no. 1, pp. 30–59, Mar. 1979.

[BBM89] A. Borgida, R.J. Brachman, D.L. McGuinness, and L.A. Resnick, "CLASSIC: A Structural Data Model for Objects," *Proc. ACM SIGMOD Int'l Conf. Management of Data*, pp. 58–67, 1989.

[BDB79] J. Biskup, U. Dayal and P.A. Bernstein, "Synthesizing Independent Database Schemas," *Proc. ACM SIGMOD Int'l Conf. Management of Data*, pp. 143–151, 1979.

[BDN94] M. Buchheit, F.M. Donini, W. Nutt, and A. Schaerf, "Refining the Structure of Terminological Systems: Terminology = Schema + Views," *Proc. Twelfth Nat'l Conf. Artificial Intelligence*, pp. 199–204, 1994.

[Ber76] P.A. Bernstein, "Synthesizing Third Normal Form Relations from Functional Dependencies," *ACM Trans. Database Systems*, vol. 1, no. 4, pp. 277–298, Dec. 1976.

[BGN89] H.W. Beck, S.K. Gala, S.B. Navathe, "Classification as a Query Processing Technique in the CANDIDE Semantic Data Model," *Proc. First Int'l Conf. Data Engineering*, pp. 572–581, 1989.

[Bis89] J. Biskup, "Boyce-Codd Normal Form and Object Normal Forms," *Information Processing Letters*, vol. 32, no. 1, pp. 29–33, July 1989.

[BJN94] M. Buchheit, M.A. Jeusfeld, W. Nutt, and M. Staudt, "Subsumption between Queries to Object-Oriented Databases," *Information Systems*, vol. 9, no. 1, pp. 33–54, 1994.

[BL84] R.J. Brachman, and H.J. Levesque, "The Tractability of Subsumption in Frame-Based Description Languages," *Proc. Nat'l Conf. Artificial Intelligence*, pp. 34–37, 1984.

[BL89] G. Di Battista, and M. Lenzerini, "A Deductive Method for Entity-Relational Modeling (Extended Abstract)," *Proc. Fifteenth Int'l Conf. on Very Large Data Bases*, pp. 13–22, 1989.

[Bom96] M.F. van Bommel, *Path Constraints for Graph-Based Data Models*, doctoral dissertation, Univ. of Waterloo, Dept. Computer Science, Waterloo, Ontario, Canada, 1996.

[Bor95] A. Borgida, "Description Logics in Data Management," *IEEE Trans. Knowledge and Data Eng.*, vol. 7, no. 5, pp. 671–682, 1995.

[BP94] A. Borgida, and P.F. Patel-Schneider, "A Semantics and Complete Algorithm for Subsumption in the CLASSIC Description Logic," *J. Artificial Intelligence Research*, pp. 277–308, 1994.

[BS92] S. Bergamaschi, and C. Sartori, "On Taxonomic Reasoning in Conceptual Design," *ACM Trans. Database Systems*, vol. 17, no. 3, pp. 385–422, Sept. 1992.

[BW94] M.F. van Bommel, and G.E. Weddell, "Reasoning about Equations and Functional Dependencies on Complex Objects," *IEEE Trans. Knowledge and Data Eng.*, vol. 6, no. 3, pp. 455–469, 1994.

[BW97] A. Borgida and G.E. Weddell, "Adding Uniqueness Constraints to Description Logics (Preliminary Report)," *Proc. Fifth Int'l Conf. Deductive and Object-Oriented Databases*, pp. 85–102, Dec. 1997.

[BWW98] P. Buneman, W. Fan, and S. Weinstein, "Path Constraints on Semistructured and Structured Data," *Proc. 17th ACM SIGACT-SIGMOD-SIGART Symp. Principles of Database Systems*, pp. 129–138, 1998.

[Cal96] D. Calvanese, "Reasoning with Inclusion Axioms in Description Logics: Algorithms and Complexity," *Proc. 12th European Conf. Artificial Intelligence*, pp. 303–307, 1996.

[CFP82] M.A. Casanova, R. Fagin, and C.H. Papadimitriou, "Inclusion Dependencies and Their Interaction with Functional Dependencies," *Proc. First ACM SIGACT-SIGMOD Symp. Principles of Database Systems*, pp. 171–176, 1982.

[CFP84] M.A. Casanova, R. Fagin, and C.H. Papadimitriou, "Inclusion Dependencies and Their Interaction with Functional Dependencies," *J. Computer and System Sciences*, vol. 28, no. 1, pp. 171–176, Feb. 1984.

[CGL95] D. Calvanese, G. De Giacomo, and M. Lenzerini, "Structured Objects: Modeling and Reasoning," *Proc. Fourth Int'l Conf. on Deductive and Object-Oriented Databases*, pp. 229–246, Dec. 1995.

[Cha97] P.K.C. Chan, *Optimizing OQL on Legacy Main-Memory Data Structures with Existential Graphs*, master's thesis, Univ. of Waterloo, Dept. Computer Science, Waterloo, Ontario, Canada, 1997.

[CK85] S.S. Cosmadakis, and P.C. Kanellakis, "Equational Theories and Database Constraints," *Proc. 17th Ann. Symp. Theory of Computing*, pp. 273–284, 1985.

[CK86] S.S. Cosmadakis, and P.C. Kanellakis, "Functional and Inclusion Dependencies: a Graph Theoretic Approach," *Advances in Computing Research*, vol. 3, pp. 163–184, 1986.

[CKV90] S.S. Cosmadakis, P.C. Kanellakis, and M.Y. Vardi, "Polynomial-Time Implication problems for Unary Inclusion Dependencies," *J. Assoc. for Computing Machinery*, vol. 37, no. 1, pp. 15–46, Jan. 1990.

[Cod70] E.F. Codd, "A Relational Model of Data for Large Shared Data Banks," *Comm. ACM*, vol. 13, no. 6, pp. 377–387, pp. 33–64, June 1970.

[Cod72] E.F. Codd, "Further Normalization of the Data Base Relational Model," *Data Base Systems*, Courant Computer Science Symposium 6, R. Rustin, ed., Prentice-Hall, Englewood Cliffs, N.J., 1972

[CV83] M.A. Casanova, and V.M.P. Vidal, "Towards a Sound View Integration Methology," *Proc. Second ACM SIGACT-SIGMOD Symp. Principles of Database Systems*, pp. 36–47, 1983.

[DD89] L.M.L. Delcambre, and K.C. Davis, "Automatic Validation of Object-Oriented Database Structures," *Proc. Fifth Int'l Conf. Data Engineering*, pp. 2–9, 1989.

[DLN91] F.M. Donini, M. Lenzerini, D. Nardi, and W. Nutt, "Tractable Concept Languages," *Proc. 12th Int'l Joint Conf. Artificial Intelligence*, pp. 458–463, Aug. 1991.

[DST80] P.J. Downey, R. Sethi, and R.E. Tarjan, "Variations on the Common Subexpression Problem", *J. Assoc. for Computing Machinery*, vol. 27, no. 4, pp. 758–771, Oct. 1980.

[DW89] S.K. Debray, and D.S. Warren, "Functional Computations in Logic Programs," *ACM Trans. Programming Languages Systems*, vol. 11, no. 3, pp. 451–481, July 1989.

[Fag81] R. Fagin, "A Normal Form for Relational Databases that is Based on Domains and Keys," *ACM Trans. Database Systems*, vol. 6, no. 3, pp. 387–415, Sept. 1981.

[GL95] G. De Giacomo, and M. Lenzerini, "What's in an Aggregate: Foundations for Description Logics with Tuples and Sets," *Proc. 14th Int'l Joint Conf. Artificial Intelligence*, pp. 801–807, Aug. 1995.

[GM91] R. Godin, and R. Missaoui, "Semantic Query Optimization Using Inter-Relational Functional Dependencies," *Proc. 24th Ann. Hawaii Int'l Conf System Sciences*, vol. 3, pp. 368–375, 1991.

[HD99] C.S. Hara, and S.B. Davidson, "Reasoning about Nested Functional Dependencies," *Proc. 18th ACM SIGACT-SIGMOD-SIGART Symp. Principles of Database Systems*, pp. 91–100, 1999.

[HZ80] M. Hammer, and S.B. Zdonik, "Knowledge-Based Query Processing," *Proc. Sixth Int'l Conf. on Very Large Data Bases*, pp. 137–147, Oct. 1980.

[IN93] M. Ito, and M. Nakanishi, "On Deriving Specialization Constraints over Complex Objects," *Proc. Third Int'l Symp. Database Systems for Advanced Applications*, pp. 335–342, Apr. 1993.

[IW94] M. Ito, and G.E. Weddell, "Implication Problems for Functional Constraints on Databases Supporting Complex Objects," *J. Computer and System Sciences*, vol. 49, no. 3, pp. 726–768, Dec. 1994.

[KCV83] P.C. Kanellakis, S.S. Cosmadakis, and M.Y. Vardi, "Unary Inclusion Dependencies have Polynomial Time Inference Problems," *Proc. 15th Ann. Symp. Theory of Computing*, pp. 264–277, 1983.

[Kin81] J.J. King, "QUIST: A System for Semantic Query Optimization in Relational Databases," *Proc. Seventh Int'l Conf. on Very Large Data Bases*, pp. 510–517, Sept. 1981.

[LB87] H.J. Levesque, and R.J. Brachman, "Expressiveness and Tractability in Knowledge Representation and Reasoning," *Computational Intelligence*, vol. 3, no. 2, pp. 78–93, May 1987.

[LS91] M. Lenzerini, and A. Schaerf, "Concept Languages as Query Languages," *Proc. Ninth Nat'l Conf. Artificial Intelligence*, pp. 471–476, 1991.

[LSK95] A.Y. Levy, D. Srivastava, T. Kirk, "Data Model and Query Evaluation in Global Information Systems," *J. Intelligent Information Systems*, vol. 5, pp. 121–143, 1995.

[Men85] A.O. Mendelzon, "Functional Dependencies in Logic Programs," *Proc. 11th Int'l Conf. on Very Large Data Bases*, pp. 324–330, Aug. 1985.

[MG93] D. McAllester and R. Givan, "Taxonomic Reasoning for First Order Inference," *J. Assoc. for Computing Machinery*, vol. 40, no. 2, pp. 246–283, Apr. 1993.

[Mit83] J.C. Mitchell, "The Implication Problem for Functional and Inclusion Dependencies," *Information and Control*, vol. 56, no. 3, pp. 154–173, Mar. 1983.

[MW91] A.O. Mendelzon, and P.T. Wood, "Functional Dependencies in Horn Clause Queries," *ACM Trans. Database Systems*, vol. 16, no. 1, pp. 31–55, Mar. 1991.

[Neb88] B. Nebel, "Computational Complexity of Terminological Reasoning in BACK," *Artificial Intelligence*, vol. 34, no. 3, pp. 371–383, Apr. 1988.

[NO77] G. Nelson, and D.C. Oppen, "Fast Decision Algorithms Based on Union and Find," *Proc. 18th Symp. Foundations of Computer Science*, pp. 114–119, Oct. 1977.

[NO80] G. Nelson, and D.C. Oppen, "Fast Decision Procedures Based on Congruence Closure," *J. Assoc. for Computing Machinery*, vol. 27, no. 2, pp. 356–364, Apr. 1980.

[Sch89] M. Schmidt-Schauβ, "Subsumption in KL-ONE is Undecidable," *Proc. First Int'l Conf. Principles of Knowledge Representation and Reasoning*, pp. 421–431, 1989.

[SSM96] D. Simmen, E. Shekita, and T. Malkemus, "Fundamental Techniques for Order Optimization," *Proc. ACM SIGMOD Int'l Conf. Management of Data*, pp. 57–67, 1996.

[TF82] D.–M. Tsou and P.C. Fischer, "Decomposition of a Relation Scheme into Boyce-Codd Normal Form," *SIGACT News*, vol. 14, no. 3, pp. 23–29, 1982.

[WC90] G.E. Weddell, and N. Coburn, "A Theory of Specialization Constraints for Complex Objects," *Proc. Third Int'l Conf. on Database Theory*, pp. 229–244, Dec. 1990.

[Wed89] G.E. Weddell, "A Theory of Functional Dependencies for Object-Oriented Data Models," *Proc. First Int'l Conf. on Deductive and Object-Oriented Databases*, pp. 165–184, Dec. 1989.

[Wed92] G.E. Weddell, "Reasoning about Functional Dependencies Generalized for Semantic Data Models," *ACM Trans. Database Systems*, vol. 17, no. 1, pp. 32–64, Mar. 1992.

[Wij99] J. Wijsen, "Temporal FDs on Complex Objects," *ACM Trans. Database Systems*, vol. 24, no. 1, pp. 127–176, Mar. 1999.

[WWB93] J. Wright, E.S. Weixelbaum, K. Brown, G.T. Vesonder, S.R. Palmer, J.I. Berman, and H.H. More, "A Knowledge-Based Configurator that Supports Sales, Engineering, and Manufacturing at AT&T Network Systems," *Proc. Conf. Industrial Applications of Artificial Intelligence*, pp. 183–193, 1993.

[YP82] M. Yannakakis, and C.H. Papadimitriou, "Algebraic Dependencies," *J. Computer and System Sciences*, vol. 25, no. 1, pp. 2–41, Aug. 1982.