Consistency Management for Object Databases

Dissertation

zur Erlangung des akademischen Grades

Doktoringenieur

(Dr.-Ing.),

angenommen durch die Fakultät für Informatik
der Otto-von-Guericke-Universität Magdeburg

von

MSc. Hussien Oakasha

geboren am 25. März 1967
in Dokki, Giza, Ägypten

Gutachter: Prof. Dr. Gunter Saake
Prof. Dr. Stefan Conrad
Dr. Michael Gertz

Magdeburg, 30. Oktober 2000

The author dedicates this work to the spirit of his mother who left our world on 19.9.1999

Zusammenfassung

Der Integritätsaspekt existierender objektorientierter Datenbankmanagementsysteme ist heutzutage nur unzureichend unterstützt. Integritätsbedingungen wie zum Beispiel Inter-Objekt-Bedingungen müssen entweder durch anwendungsorientierte Techniken oder durch ECA-Regelmechanismen der Systeme spezifiziert und verwaltet werden. Beide Techniken haben verschiedene Nachteile, wodurch die Gewährleistung der Datenbankintegrität unvollständig bleibt. Insbesondere die Änderung von Integritätsbedingungen ist schwierig.

Die vorliegende Arbeit präsentiert einen neuartigen Ansatz für die Konsistenzsicherung in Objektdatenbanken. Hierbei werden Integritätsbedingungen gleichwertig zu Anwendungsobjekten strukturiert und in einer Metadatenbank, dem "*Constraint Catalog*", gespeichert. Wenn ein Objekt erzeugt wird, so werden Integritätsbedingungen zu diesem Objekt aus dem Constraint Catalog ermittelt und Beziehungen zwischen dem Objekt und den Integritätsbedingungen hergestellt.

Die Struktur, die wir für Integritätsbedingungen entwickeln, hat eine Reihe von Eigenschaften, welche die Konsistenzsicherung für Objektdatenbanken verbessern und in konventionellen Ansätzen nicht in zufriedenstellender Weise umgesetzt sind. Dieser Ansatz unterstützt die Überwachung von Objektkonsistenz auf verschiedenen Ebenen der Update-Granularität, Integritätsunabhängigkeit, Effizienz der Überwachung von Integritätsbedingungen und die Kontrolle von inkonsistenten Objekten. Weiterhin wird die globale Aktivierung und Deaktivierung von Integritätsbedingungen für alle Objekte einer Datenbank sowie lokal für einzelne Objekte und die Deklaration von Integritätsbedingungen für einzelne Objekte ermöglicht. Diese Eigenschaften werden auf der Basis grundlegender Konzepte von Objektdatenmodellen bereitgestellt.

Abstract

The aspect of semantic integrity in the mainstay object-oriented database management systems (OODBMSs) today is generally weak. Constraints like inter-object constraints are specified and maintained either by application-oriented techniques or using event-condition-action (ECA) rules facilities of these systems. Both techniques have many disadvantages that makes database integrity incomplete. In particular, modifying constraints is a hard task.

This thesis presents a novel approach for consistency management in object databases. In this approach constraints are structured as first class citizens and stored in a meta-database called the constraint catalog. When an object is created, constraints restricting that object are retrieved from the constraint catalog and relationships between these constraints and the object are established.

The structure of constraints has several features that enhance consistency management for object databases which do not exist in conventional approaches in a satisfactory way. These features are monitoring objects consistency at different levels of update granularity, integrity independence, efficiency of constraints maintenance, and controlling inconsistent objects. Furthermore, these features include the capabilities for enabling and disabling of constraints globally to all objects of database as well as locally to individual objects, and the possibility of declaring constraints on individual objects. All these features are provided by means of basic concepts of object data models.

Acknowledgments

I am grateful (as always) to my supervisor Prof. Dr. Gunter Saake for giving me the opportunity to do this thesis in his research group and for all of his encouragement and support throughout this work. I am also grateful to Prof. Dr. Stefan Conrad (University of Munich) and Dr. Michael Gertz (University of California, Davis) for accepting to work as reviewers for this thesis.

I would like to thank the commission of scholarships at Magdeburg University and the German Federal State Sachsen-Anhalt for their financial support to my grant from (10.6.1997) to (31.5.2000). I would also like to thank the department of Technical and Business Information Systems, Computer Science Faculty, Magdeburg University, for allowing me to use all of the available facilities.

Special thanks to my colleagues Dr. Kai-Uwe Sattler and Eike Schallehn for reading parts of the thesis and valuable discussions and helps on technical issues concerning O_2 object-oriented database system, Oracle8 object-oriented features and JDBC database access with Java.

I am also grateful for the support and assistance I receive from the following persons: Dr. Ingo Schmitt, Nasreddine Aoumeur, Jörg Fischer, Sören Balko, Dr. Meike Hollatz, Stephan Dassow, Martin Endig (ITI-DB group), Dr. Michael Höding, Eyk Hildebrandt, Dr. Can Türker, Dr. Kerstin Schwarz (previous members of ITI-DB group), Frau Claudia Bethge, Frau Kerstin Lange (ITI Secretariat), Herr Gerd Lange, Herr Fred Kreutzmann, Herr Steffen Thorhauer (ITI Technical staff), Prof. Dr. Claus Rautenstrauch (Wirtschafts-Informatik group), Prof. Dr. Ralf Hofestädt (Bio-Informatik group), Dr. Georg Paul (Ingenieur-systeme group), Frau Wölke (Abteilung Studentensekretariat/Weiterbildung), Frau Renate Hotz (Dean Secretariat) and Frau Jutta Timme (Prüfungs- und Praktikantenamt).

Magdeburg, October 2000 *Hussien Oakasha*

Contents

1 Introduction **1**

1.1 Semantic Integrity . . . 1

1.2 Consistency Management: The Problem . . . 3

1.3 The Constraint Catalog: Our Approach . . . 4

1.4 Related Work . . . 6

1.5 Thesis Outline . . . 8

2 Object-Oriented Databases **9**

2.1 Some Concepts of Data Abstractions . . . 10

2.2 Limitations of Relational Databases . . . 11

2.3 The Core Model . . . 12

2.4 Object Persistence . . . 15

2.4.1 Explicit Persistence . . . 15

2.4.2 Persistence by Reachability . . . 15

2.5 A Database Example . . . 17

2.6 A Formal Object Database Model . . . 19

2.7 Assumptions: Transactions and Relationships . . . 26

3 Consistency Constraints **29**

3.1 Constraint Design . . . 31

3.1.1 Users' Requirements Collection and Analysis . . . 31

3.1.2 Conceptual Database Design . . . 31

3.1.3 Logical Database Design . . . 32

3.2 The Quality of Constraints . . . 34

3.3 Constraint Specification Language . . . 35

3.3.1 Syntax of the Language . . . 35

3.3.2 Semantics of the Language . . . 38

3.4 Path Expressions 39
3.5 Modification Operations 41
3.6 Inverse Relationships 44
3.7 Well-Formed Constraints 49
3.8 Constraint Examples 51

4 Constraint Catalog: The IC Class 55

4.1 Canonical Constraints 56
4.2 Constrained Classes 57
4.3 Constrained Paths and Persistent Roots 58
4.4 Transformation into Canonical Constraints 63
4.4.1 Path-Flattened 63
4.4.2 Removing Inter-Paths 66
4.5 Structure of the IC Class 67
4.5.1 Constraint Name 68
4.5.2 Constraint Canonical Specification 70
4.5.3 Constraint Status 70
4.5.4 Constraint Mode 70
4.5.5 Constrained Classes 71
4.5.6 Constrained Paths and Persistent Roots 72
4.5.7 Constraint Shells 74

5 Constraint Catalog: The Shell Class 77

5.1 Simplified Form: Motivating Example 78
5.2 Constraint Boolean Form 80
5.3 Simplified Forms 82
5.3.1 Outermost Quantification 83
5.3.2 Non Outermost Quantification 84
5.4 Observations 86
5.4.1 Object Roles 86
5.4.2 Existential Constraints 87
5.4.3 Key Constraints 88
5.4.4 Uniqueness of Boolean Forms 88
5.5 Structure of the Shell Class 90
5.5.1 Constraint Attribute 91

5.5.2 Simplified Form 92
5.5.3 Class Attribute 93
5.5.4 Paths and Persistent Roots 93
5.5.5 Range of Simplified Forms 94
5.5.6 Interrelated Objects and their Shells 95

6 Constraint Structure 99
6.1 Consistency Maintenance by Using Shells 100
6.2 Linking Objects and Shells 102
6.3 The *Kernel* Class 107
6.4 The *Constraint* Class 111
6.5 States of Constraint Instances 112
6.5.1 Flat Constraint Instances 113
6.5.2 Nested Constraint Instances 113
6.5.3 The Life Cycle of a Constraint Instance 113
6.6 Role of Class Extension *Constraints* 115
6.7 The Initiating of the *hasConstraints* Attribute 116
6.8 The Constraint Catalog: An Overview 118

7 Consistency Management 121
7.1 Checking Constraint Instances 122
7.1.1 Flat Constraint Instance 122
7.1.2 Inner and Core Constraint Instances 122
7.1.3 Nested Constraint Instance 124
7.2 Different Levels of Consistency Checking 127
7.2.1 The *check*() Method of the *Constraint* Class 127
7.2.2 The *check*() Method of the Root Class *Any* 128
7.2.3 The *check*() Method of the *Transaction* Class 129
7.3 Maintaining Constraint Kernels 129
7.3.1 Maintaining the *lastCheck* Attribute 130
7.3.2 Maintaining the *lastUpdate* Attribute 130
7.3.3 Methods *checked*() and *unchecked*() 132
7.4 Enabling and Disabling Constraint Instances 132
7.4.1 Global Status 134
7.4.2 Local Status 134

7.5 Consistency Maintenance 135
7.6 Control Methods of the Root Class 137
7.7 Control Methods of Individual Classes 138
7.7.1 Atomic Type 138
7.7.2 Set-Structured Type 141
7.7.3 Tuple-Structured Type 141
7.8 Control Methods of the *Transaction* class 143
7.9 Consistency Control: Application Example 144
7.10 Constraint Manipulation 146
7.10.1 Adding a Constraint to the Constraint Catalog 146
7.10.2 Removing a Constraint from the Constraint Catalog 147

8 Conclusions **149**
8.1 Contributions 149
8.2 Future Work 152

A Syntax for Class Definition **155**

B Syntax for Method Implementation **157**

Bibliography **159**

Symbol Index **173**

Index **175**

List of Figures

2.1 The UNIVERSITY database schema. 16

3.1 Objects of the UNIVERSITY database. 40
3.2 Side affects to objects of Figure 3.1 by update unit of Example 3.5.1. 43

4.1 Structure of the IC Class. 68
4.2 States of IC objects $IC(W1)$ and $IC(W2)$. 69
4.3 Relationship between the two classes of the constraint catalog. 74

5.1 Recursive relationship of the *Person* class. 86
5.2 Structure of the Shell class. 90
5.3 Shells $S(W1|pr)$, $S(W1|se)$, and $S(W1|co)$ of the constraint $W1$. 97

6.1 Relationships between the classes *Any*, *Constraint*, *Kernel*, Shell, and IC. . . 100
6.2 A set of interrelated objects of *Professor*, *Section* and *Course* classes. 102
6.3 The flat shell $S(W1|pr)$, and the nested shells $S(W1|se)$ and $S(W1|co)$. . . 103
6.4 Structure of the *Kernel* class. 107
6.5 Kernels of Example 6.3.1 according to the update scenario of Table 6.1. . . . 108
6.6 Kernels and shells of the constraint $W1$ and objects of Figure 6.3. 109
6.7 Structure of the *Constraint* class. 111
6.8 Relationship between the root class *Any* and the *Constraint* class. 112
6.9 The life cycle of a constraint instance. 114
6.10 The constraint catalog and its relationship to the object schema. 120

7.1 The integration of Figures 6.2 and 6.6. 123
7.2 The maintenance of kernels of Figure 7.1. 133
7.3 The *Professor* class 138
7.4 Named object *deletedShell*. 147

List of Tables

2.1 Classes and persistent roots of the UNIVERSITY database schema. 18

2.2 Types associated with classes of the UNIVERSITY database schema. 23

2.3 Relationships of the UNIVERSITY database schema. 26

6.1 An update scenario to objects of Figure 6.3. 105

6.2 The method *conIni*() of the root class *Any*. 117

7.1 The *check*() method of the *Constraint* class. 126

7.2 The *check*() method of the root class *Any*. 128

7.3 The *check*() method of the *Transaction* class. 129

7.4 The *setLastUpdate*() method of the *Constraint* class. 131

7.5 The *setEnabled*() method of the *Constraint* class. 136

7.6 The template for the control method of atomic attributes. 140

7.7 The template for the control method $A+()$ of attribute A of a set type. . . 142

7.8 The template for control method $r+()$ of persistent root r of a set type. . . 145

7.9 An example of safe transaction. 146

Chapter 1

Introduction

1.1 Semantic Integrity

A database has semantic integrity if the data it stores is correct as perceived from the semantics of the modeled application domain. Full integrity is not guaranteed if the data obey only the data structures declared in the database schema. Essentially, a database schema is an interpolation to the intention of an application domain. This due to the fact that no data model can capture all potential semantics that may be present in database applications. As such, the schema of a database may accept data to be populated in the database extension. However this data causes the presence of some data configurations that have no counterpart in the application domain.

In response to data modeling limitations, the notion of integrity constraint is introduced and the definition of database integrity is adapted consequently. Integrity constraints are application-dependent conditions that govern object states and behavior. In terms of integrity constraints, database integrity is the requirement of constraint satisfaction by each object state and/or update operations that leads to it.

It is worth mentioning, that database integrity as adapted here only takes the database correctness and not the database completeness into account. Database completeness means that all correct information of the application domain is included in the database. This is in contrast with database correctness, which means that all incorrect information of the application domain is excluded from the database. Defining database integrity to be important properties of database completeness and correctness is introduced in [Mot89]. However, for feasibility considerations, database completeness is hard, if not impossible, to obtain. Thus, there is a common agreement among researchers in the area of database integrity

that database integrity should only consider database correctness as it can be determined by integrity constraints.

Throughout the lifetime of a database, integrity is maintained by checking integrity constraints every time updates are made to the database. If a check of these constraints results in a violation of at least one of them, the updates are either rejected or an appropriate action may be taken such as repairing the errors resulting by these updates and then restarting the constraint checking. Otherwise, the updates are allowed. Thus, database integrity in this latter sense is a matter of protecting a database against requests for updating database states with incorrect information of the application domain. Invalid updates may be caused by errors in data entry introduced by the users and application programs.

The notion of database integrity presented above is called *semantic integrity*, for two simple reasons. First, this definition emphasizes the role of integrity constraints which are nothing but a description of part of the semantics of a database. Second, semantic integrity is one way in which the accuracy of data in a database may be guaranteed. However, in addition to semantic integrity, there are some other areas of database that are responsible for database integrity [GA93, BM88, ÖV91, EN94]:

- *Concurrency control:* Consistency of a database may be compromised because of improper control of access to shared data by multiple concurrent transactions. For this reason, concurrency control mechanisms based on serializability principles have been developed such that an interleaved execution of concurrent transactions is correct [BHG87, EN94].

- *Security:* Database security may be compromised due to an administration error or non-reliable database security mechanism. As such, consistency of a database may be corrupted maliciously by making unauthorized changes. For instance, the database administrator (DBA) of a university database may, by accident, give permission to a teaching assistant to access and change parts of the database the access to which should be protected against users of rank teaching assistant. This teaching assistant may then change his rank in the university database from teaching assistant to professor. This unauthorized change can then violate an integrity constraint such as the one that says "*a professor must have a Ph.D. degree*".

- *Data reliability:* Reliability of the database system may be compromised due to a failure in hardware or software. Examples of hardware failures are disk read/write head crash or malfunctions in the systems main memory. In these cases invalid data

may result because some block on the disk or content of the main memory lose its data. Examples of software failures are malfunctions of the operating system or (database management system).

There has been a considerable amount of work on concurrency control, database security mechanisms and database recovery techniques. The principles behind these mechanisms and techniques are described in most textbooks, including [Ull88, EN94, ÖV91, Dat90, SH99]. For more in-depth coverage of concurrency control and database security mechanisms we refer the reader to the textbooks [BHG87, CFMS95].

In this thesis we are only concerned with semantic integrity. Thus, we assume the environment in which the database will be used is optimal. That means that a total control on data reliability, security, and concurrency is guaranteed by the database management system and the DBA. Hence, the only way to violate database integrity is by violating the integrity constraints that are defined for a database.

1.2 Consistency Management: The Problem

The aspect of semantic integrity in object data models underlying the mainstay object-oriented database management systems (OODBMSs) today [LV97, Kim95] is generally weak although it has improved over the one of the pure relational data model. Apart from integrity constraints like domain constraints, cardinality constraints, referential constraints and key constraints, other complex types of state constraints like inter-object constraints [GGJ93] are specified and maintained either by encoding them to application programs or methods of classes [BS97, BS96] (the so-called application-oriented techniques) or using event-condition-action (ECA) rules facilities of these OODBMS [BMP91].

Disadvantages of application-oriented techniques are redundancy, i.e., a constraint must be specified in every transaction that might violate it, and understandability of the semantics of constraints, since they are encoded in statements of a general-purpose programming language. Another disadvantage is that modifying constraints is a hard task. This is due to the fact that modifying constraints is done manually by users and through modifying transactions and application programs. Finally, features like handling objects inconsistency and disabling and enabling of constraints are absent.

A possible approach to avoid some problems of application-oriented techniques like disabling and enabling of constraints is to maintain integrity using ECA rules [WC96]. However, modifying constraints is still a problem. First, apart from general problems like confluence

and termination, the specification of constraints is in general not mapped one-to-one to an ECA rule: a given constraint is represented by many ECA rules with different events and different actions. Second, events of ECA rules are considered to be calls of methods of classes which have side effects that may violate the condition parts of rules. Thus, ECA rules will be sensitive to modification of class methods. In addition, to modify a constraint, users are required to determine which ECA rules represent the constraint. If it is determined that an ECA rule is relevant to the modified constraint then the user must modify this rule accordingly. As there is no overall control on the process of modification, this increases the possibility of inconsistencies; both by modifying rules which are irrelevant to the modified constraints and missing rules which should be modified.

1.3 The Constraint Catalog: Our Approach

We believe that a consistency management subsystem (CMS) for OODBMSs should have the following features which are not provided by either techniques mentioned above in a satisfactory way:

- *Object-orientation.* CMS should work with basic principles of object-orientation such as inheritance and encapsulation.
- *Specification.* Constraints should be specified declaratively and structured as first class citizens.
- *Interrelated Objects.* CMS should be able to maintain consistency of a large number of interrelated objects with complex structures.
- *Transactions.* CMS should work with advanced types of transactions such as interactive and long duration transactions.
- *Efficiency.* Constraint checking should rely on optimization techniques for improving constraint evaluation.
- *Integrity Independence.* CMS should be capable to change constraint specifications without changing application programs and update transactions.
- *Inconsistency.* CMS should control inconsistencies among objects.
- *Persistence Style.* CMS should maintain consistency of all objects, persistent and non-persistent ones, and regardless of the persistence style of OODBMSs.

- *Disabling and Enabling of Constraints.* CMS should provide tools for enabling and disabling constraints at various levels of abstraction like the whole database, or a class or a specific object.

- *Update Granularity.* CMS should work with different levels of update granularity of objects such as updating a simple or complex attribute or the whole state of an object.

In this thesis we propose an approach for consistency management in OODBMS that supports the features listed above. The main aspects of our approach are the *constraint catalog* and a novel technique for constraint structuring.

The constraint catalog is an object oriented meta-database acting as a central repository for constraint specifications. The main purpose of the constraint catalog is to decouple constraint specifications from transactions and application programs and hence to provide the feature of integrity independence to our approach. The constraint catalog consists of the two classes: IC and Shell.

Every user-defined constraint is compiled into an object of the IC class. To facilitate this process, we propose a constraint specification language and express a user-defined constraint as a closed well-formed formula (wff) of that language. Then the constraint is transformed into a canonical form. We define a constraint canonical form such that it covers most of known types of state constraints.

For efficient constraint evaluation, we propose an optimization technique. Given a constraint, the optimization technique derives a set of simplified forms. Every simplified form is complied into an object of the second class of the constraint catalog, the Shell class.

To provide the other features, we consider constraints as first class citizens. Every constraint W defined for a database is represented in the object base by a set of constraint objects O_W. The relationship that an individual object o is "*subject to*" the constraint W is represented by a link between the object o and a constraint object $i \in O_W$. The constraint object $i \in O_W$ is structured as an aggregation of two objects. The first object is called the *kernel* of the constraint W w.r.t. i. The kernel stores local information of the constraint W that concerns the link of the constraint object i and the object o. The second object is an object of the Shell class called the *shell* of the constraint W. The shell stores global information of the constraint W that concerns simplified forms of the constraint W.

In addition to features listed above, a main advantage of the constraint catalog and the structure of constraint objects is that our approach to consistency management is seamlessly integrated into the dynamic part of the object schema.

1.4 Related Work

Database integrity has received a large interest in relational databases where several different goals have been pursued [CKS, GA93]. For instance, improving integrity checking has been investigated for relational databases [Nic82, MH89, QW86, QS87, Oak95, RSS96, Orm98, OS99, OS98a], but also for deductive ones [BM86, KSS87]. A large number of different issues have been covered, for instance semantic integrity constraint design [BEST98], transaction correctness w.r.t. integrity constraint [SS89, BM88, Gre93] and repair of constraint violation [Ger96a, ML91]. For active databases, compiling constraints into ECA rules has extensively been investigated in [CW90, CFPT92, CFPT94]. Most of the work deals with state constraints. For dynamic constraints some of these topics have been addressed in e.g. [SL88, Saa91, Cho92, GL93, LGS94, GL95]. Another direction of research is the usage of integrity constraints for semantic query optimization [GGGM98].

Several approaches have been proposed in the literature for consistency management in object-oriented databases. This has been done in the context of passive databases and persistent programming languages [SB92, EJK93, NNJ94, GD94, BD95, TSS97, BS97, JQ92, EGB93], active databases [BMP91, GJ91, Día92, UKN92], deductive databases [JJ91, BCB97, CF97], and object-oriented data models [BV94, BdBZ93, FG95, FMP97, BM91, Tar92, NNJ94]. These approaches can be divided into two groups.

The first group takes the approach that constraints should be encoded in application programs. Their main motivation is efficiency of consistency checking. Work done in this group provides techniques either to generate checking code for constraints specified declaratively [EGB93, BLR92, BD95] or explicitly specified in the conceptual schema that is described by using the entity relationship model or one of its extension [EJK93, NNJ94, Tar92] or to prove correctness of methods of classes and transactions w.r.t. integrity constraints [BS96, BS97, SB97].

Work in this direction is applicable to specific types of integrity constraints, such as intra-object, domain, cardinality and key constraints, for otherwise the generated checking code will be too complex and needs further manual optimization as it is the case of approaches provided in [BS96, BS97]. For general constraints, they should first be compiled into an intermediate language before the actual generation of code, for instance, the approach of [BM91] where an intermediate language based on semantic nets is proposed. In some approaches the language in which the generated code is specified is not an object-oriented language but merely a declarative one. For example in [EGB93] the generated methods are specified in Prolog. Moreover, apart from the problems mentioned above, work in this direction inherits

the problems of application-oriented techniques mentioned in Section 1.2

The second group takes the approach that constraints should not be specified in applications programs. The motivation is to avoid problems of application-oriented techniques. This group can be further divided into three subgroups:

- The first subgroup takes the approach that constraints should be specified in class declaration. Although constraints are specified as part of the class structure in the Ode object-oriented database management system using the language O++, (1) "*its association with the class is merely a notational convenience*" and (2) "*the constraint facility provided in O++ is an intra-object in that when an object is updated only the constraints associated with it, through its class definition, are checked*" [JQ92]. In this approach, update granularity is fixed to insertion and deletion of objects from a class extension which are the only means of objects persistence. The problem of how constraints are potentially violated by member functions of classes is not considered.

- The second subgroup follows the direction that constraints should be specified as (ECA) rules. Work in this group provides techniques for deriving ECA rules from constraints specified declaratively [Día92, BMP91, UKN92]. Most of this work is directed to a specific OODBMS such as [GJ91] (see [WC96] for more details). The problems of this direction, i.e., consistency management via ECA rules have been mentioned in the introductory section.

- The third subgroup, oriented to deductive databases, takes the view that constraints should be specified as rules such as those of DataLog. Work in this group adopts optimization techniques proposed for relational databases to object-oriented databases [JJ91, BCB97, CF97]. As for the previous subgroup, most of the work deals with a specific OODBS, e.g. [JJ91] for the ConceptBase system, and [BCB97] for the Chimera system.

In contrast to most of the approaches mentioned before we aim at a system-independent conceptual infrastructure for integrity checking in OODBMS. Our approach does not require an OODBMS to provide deductive or active capabilities. Furthermore, our approach is neither restricted to certain kinds of static integrity constraints nor to specific kinds of transactions. In particular, we support integrity checking also in presence of *ad hoc* style (or *interactive*) transactions.

Approaches for consistency management have also been proposed in other domains such as software engineering. The features listed in the introductory section and provided by our

approach meet requirements of these applications. The main difference between the work presented, for instance, in [TC98] and ours is that our approach provides these features only by means of basic notations of object data models and hence on an abstract level rather than on the implementation level.

1.5 Thesis Outline

Apart from this introductory chapter and the conclusions, the thesis is organized into three main parts:

Chapters 2 and 3: In these chapters the basic definitions and notations that will be used throughout the rest of the thesis are presented. In Chapter 2, we briefly present the basic principles of object databases as described in the OODBS manifesto [ABD+89]. Then we present the formal object data model that will be used in thesis. In Chapter 3, we present a constraint specification language, the definition of path expressions, a set of primitive modification operations, a canonical form for integrity constraints and how user-defined constraint can be transformed into canonical ones.

Chapters 4 and 5: The structure of the constraint catalog is presented in these chapters. In Chapter 4, we present the first class of the constraint catalog, the IC class and show how canonical constraints can be compiled into objects of the IC class. In Chapter 5, we first present an optimization technique for deriving simplified forms from a given constraint. Then we present the second class of the constraint catalog, the Shell class. Finally, we show how simplified forms of integrity constraint can be compiled into objects of the Shell class.

Chapters 6 and 7: These chapters concern the structuring of constraints into objects and how they are managed. In Chapter 6, we first motivate the proposed structure for constraints and then we present that structure. Chapter 7 is devoted to answer questions of what the tasks of consistency management are and how they can be realized in our approach.

Finally, the thesis has two appendices; Appendix A lists the syntax for class definitions, and Appendix B describes the syntax of the formal language for method implementation that will be used in the thesis.

Chapter 2

Object-Oriented Databases

The relational data model does not offer the full flexibility to support new requirements of advanced database applications [EN00, LV97, Cat91]. Examples of advanced applications are computer-aided design (CAD), computer-aided manufacturing (CAM), computer-aided software engineering (CASE), intelligent offices, office information systems, engineering databases, artificial intelligence applications, medical images systems, geographical information processing systems, and biological information systems. For discussion of why these advanced applications need database management system (DBMS) to be one of their components we refer the reader to [EL90, Cat91].

The characteristics and requirements of advanced database applications are different from those of traditional database applications such as booking, payroll and banking systems. Advanced applications mange a large number of complex data structures storing small numbers of large objects. Data structures of advanced applications are highly constrained since they are heavily interrelated. Another characteristic of these applications is that – as software systems – they are developed by using an object-oriented programming language such as C++ or Java.

In addition to requirements of traditional applications such as persistence, concurrency, recovery, and ad hoc queries, advanced applications have new requirements. These new requirements are that in the underlying DBMS complex data structures should be available, new data types should be possible to introduce, and long duration transactions should be supported.

Object databases have been developed to address the requirements of advanced applications, and to the fact that object-orientation employed in several areas in computing field. Object databases introduced to the database field the notions of types, data abstraction,

inheritance, object identifiers, and integration of data and code.

Until the beginning of the last decade, object databases were characterized by the lack of a common reference model. Currently, several efforts develop a standard for object databases. Most notably, the ODMG 3.0 standard proposed by a consortium of more than ten of ODBMS vendors [CB00]; and SQL3 standard of ISO/ANSI standardization committees. In this thesis, we use a generic data model that provides all features of the core object model presented in the OODBMS manifesto [ABD+89].

This chapter is organized as follows. In Section 2.1, we present two concepts of data abstraction. These two concepts show us that there is some semantics in the domain of discourse that cannot be modeled easily in relational databases. This will be discussed in Section 2.2. In Section 2.3, we present the core object model of the OODBMS manifesto [ABD+89]. The core object model consists of a set of features that an OODBMS must satisfy. These features are the minimum requirements for a database system to be declared as an OODBS. There are two approaches to object persistence. In Section 2.4, we present these approaches and select the general one, known as persistence by reachability, to be used in this thesis. In Section 2.6, we present a formal object database model that will be used in the rest of this thesis. Finally, in Section 2.7 we conclude the chapter by making two assumptions concerning update transactions and inverse relationships.

2.1 Some Concepts of Data Abstractions

Aggregation. Aggregation is an abstraction concept for grouping several different entities, called the components, to build a composite entity. As such we have two cases. The first one is that of grouping attributes to define an entity type. The second case and most important one is the *part-of relationship*. This relationship specifies that an entity type (a component) is a part of another entity (the composite entity). For example, *PCSet* can be modeled as an aggregation of *MotherBoard*, *Monitor* and *DiskDrive*. The part-of relationships may be nested to form a hierarchy of aggregation(s). For example, *Window* is a part of *Room*, *Room* is a part of *Apartment*, *Apartment* is a part of *BuildingBlock*. Usually, "*aggregation*" is used to refer to the part-of relationship [KM94].

Generalization and Specialization. Generalization is an abstraction concept of specifying similar properties of several different entities types (subtypes) as a generic entity type (supertype). The relationship between subtypes and supertype is called the *subtyping relationship*. For example, *MainFrame*, *Workstation*, and *PCSet* entity types have similar

properties such that we can unified them into one type, the *Computer* entity type. In this case there is one subtyping relationship from each of these subtypes to the supertype *Computer*. The converse of generalization is specialization by which a type of objects can be classified into several different subtypes. Using the same example, *Computer* type can be classified into subtypes *MainFrame*, *Workstation*, and *PCSet*.

Generalization introduces the concept of *inheritance*, which means that in subtyping relationship, a subtype has all attributes of its supertype. For instance all subtypes *MainFrame*, *Workstation*, and *PCSet* have the same attribute *processor* of the supertype *Computer*. Similar to the part-of relationship, the subtyping relationship may span over several levels to form an inheritance or a generalization hierarchy. For example, *Workstation* is a subtype of *Computer* and *Computer* is a subtype of *ElectricalDevice*.

2.2 Limitations of Relational Databases

Relational databases cannot fully address requirements of advanced database applications for two reasons. First, there is a limitation in the concepts they offer for data modeling. Second, there is the problem of record versus set-orientation, i.e., the problem of impedance mismatch.

Data Modeling Limitations. In relational databases, data is organized into relations with specified attributes. The values of attributes, by the assumption of the first-normal form, are limited to be of atomic types, such as numeric and string types. Non-atomic attributes such as composite and set-valued attributes are not supported. Thus important data modeling concepts such as aggregations and generalization are not provided. Consequently relational databases are unable to easily represent complex data structure directly. This means that an entity that has a complex structure will be represented by several relations with many semantics constraints imposed on them.

In relational databases, tuple identification is provided through keys. Keys are users assigned and defined on the values of attributes and hence they are changeable. Thus the uniqueness property of keys must be maintained by the application programs or the system. This complicates the writing of updating application programs because they have to take care of referential constraints emerging of using keys as tuples references to implement relationships between relations.

Impedance Mismatch. SQL, the standard database language for commercial relational DBMSs, is not a computationally complete language. That is, it does not provide control structures of general purpose programming languages such as while-loops. For instance, the transitive closure of a relation cannot be evaluated using an SQL query. The solution is provided by embedding SQL queries into a programming language such as C or PL/1. This solution complicates writing of application programmers and makes their semantics unclear. On the one hand, query languages in general are declarative whereas on the other hand conventional programming languages are procedural. Another shortcoming, is the problem of *impedance mismatch*: the difference in the format in which data is stored in the databases and the one used by programming languages. This means that a lot of time and coding effort can be consumed for type and data conversion between two different formats to allow correct data manipulation. This problem becomes harder if programming languages used are object-oriented programming language such as C++, Smalltalk or Java.

It is worth mentioning that to avoid some of these shortcomings and to meet the market demands, object-relational databases have evolved by extending relational DBMS technology with various object-oriented concepts such as object identifiers, user-defined types, methods, (also called stored procedures), and inheritance [SM96]. These concepts have also been included in the current version of the standard SQL, called SQL3 [EN00, LV97]. Moreover SQL3 is a computationally complete language [Coo97].

2.3 The Core Model

The purpose of the OODBS manifesto [ABD$^+$89] is to propose a core model having generally accepted features that object databases should satisfy. The OODBMS manifesto describes what are these features but not how they should be implemented. Features of the core model are separated into three categories; mandatory, optional and open features. In this thesis, we use a generic object database model that provides all mandatory features.

The mandatory features are classified into database features and object-oriented features. Database features are the requirements for a system in order to be declared as a database system: persistency, secondary storage management, concurrency, recovery and ad hoc query facility. Object-oriented features are the minimum requirements for a database system to be declared as an object database system: complex objects, object identity, encapsulation, types and classes, class or type hierarchies, overriding, and late binding.

In this section we review all these features except object persistence, which will be discussed in Section 2.4.

Objects and Object Identifiers. Each entity relevant to the universe of discourse is modeled as an *object*. Each object is associated with a unique *identifier* which is system generated, invisible to users and hence it is unchangeable. As such, an object identifier is fixed and only valid throughout the object's lifetime. Each object has a state (or value) and a behavior (or methods). An object *state* consists of values for the attributes of the object. The object *behavior* is the set of methods which operate on the object state.

Complex Values and Types. The *value* of an attribute can be either an atomic value or a complex value. Atomic values are float numbers, alphanumeric strings, boolean constants and object identifiers. Complex values are constructed by recursively applying tuple and set constructors to values. Other value constructors are possible, e.g., bags, list and arrays, but in this thesis, only tuple and set constructors will be considered. Considering object identifiers as values, an object can refer to itself or to other objects, these creating a set of interrelated objects that can be seen as a network (or a graph) of objects. This is a desirable feature because in this case object sharing is supported which has an advantage on object updates; the side effects for updating an object are propagated to all objects that refer to it.

Classes. Objects that have the same complex values with identical internal structure and which share the same set of methods can be grouped into a class. A *class* serves as a basis for instantiation (i.e., an object is an instance of a class) and therefore it consists of two parts. The first part is the *type* specifying the structure of values of objects of the class. The second part is the specification of a set of methods which operates on objects of the class. Class attributes are allowed to reference other classes which is one of several methods to model binary relationships between classes – see Page 26.

A *method* has a signature and an implementation. A *signature* is a complete specification necessary to invoke the method. This information is the name of the method and the types and order of input and output arguments. The *implementation* of a method is the code written in the programming language that is supported by the underlying OODBMS.

There are three sorts of methods: primitive, observer and mutator methods [KM94]. *Primitive* methods are those which create new instances of classes. We assume that each class has a primitive method named *new* that creates a new object of that class. *Observer* methods retrieve the whole or parts of the state of the object on which they are invoked. *Mutator* methods change the whole or parts of the state of the object on which they are invoked. Thus, observer methods have no side effects on object states, i.e., they do not

change object states, whereas mutator methods have.

Encapsulation. *Encapsulation* is the property of decoupling the internal structure of object state and method implementations from application programs and users. The rationale behind the adaption of this principle is maintainability, i.e., certain modifications of the internal structure of objects and method implementations can be made without changing any of the application programs that use them. Encapsulation can be achieved through restricting the manipulation and observation of a state of an object to be only done through applying methods that are defined in the class to which that object belongs. There is a common agreement that in many situations a strict encapsulation is not desirable, e.g., ad hoc queries, as maintainability is an irrelevant issue in these cases [ABD+89].

Class Hierarchies. Generalization and specialization relationships between classes are modeled by defining a partial order relation, called ISA, on all classes. The *ISA relationship* among classes can be represented as a rooted directed acyclic graph, called *ISA hierarchy* or simply *class hierarchy*. The unique root of the class hierarchy is a virtual class named *Any*. The semantics of the class hierarchy taken in this thesis is that of multiple inheritance. A class, called subclass, inherits all attributes and methods of its direct or indirect ancestors, called superclasses. As such, a subclass is a specialization of superclasses and a superclass is a generalization of subclasses, and the root class *Any* is a superclass of all classes.

Inheritance and Dynamic Binding. *Inheritance* implies that a subclass c' inherits every method m of superclass c. In some situations, it is convenient to redefine the implementation of m in c'. In this case, the implementation of m is said to be *overridden* by the implementation of m in c. This also introduces the concept of method *overloading*, which means that the same method name is corresponding to different implementation depending on object types.

There are two approaches to select which implementation of an overloaded method to execute. In a strongly typed system, the object type is known at compile time. In this case the system selects which implementation of the method at compile time. In some other systems, the object type is known at run-time. In this case, it is undecidable to identify the implementation of overloaded methods at compile time. As such this is done at run time. This approach is called *late* or *dynamic binding*.

2.4 Object Persistence

Objects are created either during the execution of application programs or through working with a standard OODBMS interface using the database. The created objects are either transient or persistent objects. Transient objects are the ones that are no longer used after the termination of a program and then are discarded. Persistent objects are the ones that are stored in the database and thus may last several executions of several programs. In object database systems, object persistence should be orthogonal and transparent. *Orthogonal* means that every object can be persistent regardless of its type. *Transparent* means that users should not see any differences in the processing of transient and persistent objects [LV97]. There are two approaches for an object to be persistent: explicit persistence and persistence by reachability [Coo97]. In the following we describe these approaches.

2.4.1 Explicit Persistence

In this approach, objects are explicitly stated to be persistent. This can be done by making persistency a property of objects or classes. The former style is called *persistence by object naming* and the latter one is called *persistence by class extent.*

In persistence by object naming, for an object to be persistent, it must be named by a unique persistent name. Persistent objects are identified and retrieved in application programs by using their names. The disadvantages of this style is that it becomes impractical if a massive number of objects are required to be persistent.

In persistence by class extent, users declare some classes as persistent classes. For an object to persist, it should be a member of the extension of a persistent class. As such, entry points to access a database are class extensions. Also, deletion of objects occurs explicitly in the same way as deletion of a tuple from the extension of a relation. Thus referential constraints should be maintained to avoid dangling object references.

2.4.2 Persistence by Reachability

In this approach, for an object or value to persist, at least one of two conditions must be satisfied: (1) the object or value is explicitly stated as persistent, or (2) the object or value has a relationship to an explicitly stated persistent object or value. Objects and values explicitly stated as persistent are called *persistent roots.* In this approach, persistence is a transitive relation. Every object that is reachable from – i.e., referenced by – another persistent object is persistent. In other words, if an object a refers to an object b and b refers

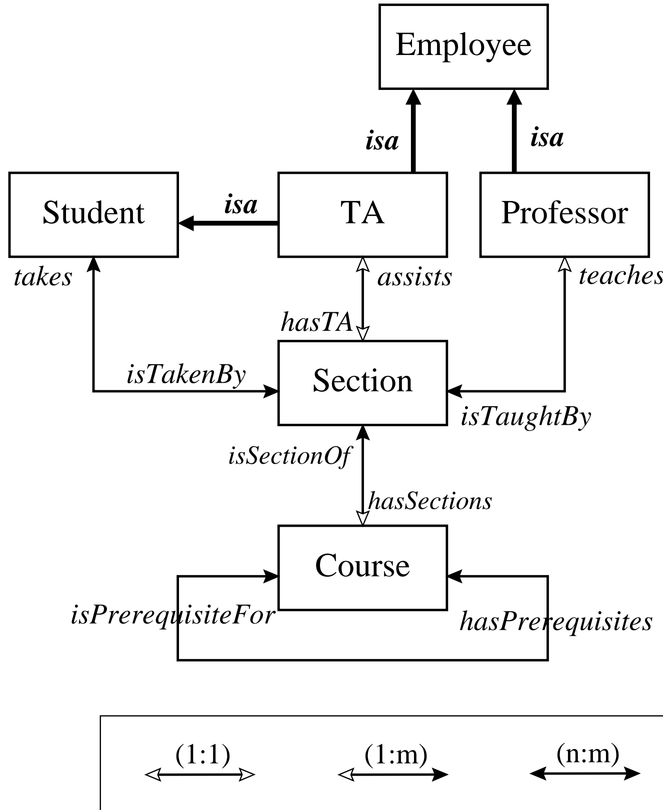

Figure 2.1: The UNIVERSITY database schema.

to an object c; or equivalently if c is reachable from b and b is reachable from a, then if a is a persistent then so are objects b and c. For this reason, this style is also called *transitive persistence* [CB97].

In persistence by reachability, persistent roots are entry points to database access. Also, objects cannot be deleted explicitly from the database but are garbage collected when no longer referenced. This means that problems of dangling object identifiers do not happen and thus referential constraints are guaranteed to be valid. Although this is an advantage to programmers it is a disadvantage from the consistency management point of view. An object may be no longer reachable from persistent roots leading to integrity violations however there is no explicit deletion of that object that indicates that an integrity violation happen (see Page 49 for an example).

It follows from the definition of persistence by reachability that persistence by reachability includes both styles of the explicit persistency. Thus, for our approach of consistency management to be applicable to all of these styles, in this thesis we will consider the general approach: persistence by reachability.

2.5 A Database Example

Throughout the rest of the thesis we draw examples from an example database given in [CB97]; the UNIVERSITY database. The schema of the UNIVERSITY database is shown in Figure 2.1. In this figure, each rectangle represents a class. Classes of the UNIVERSITY database are *Employee*, *Student*, *Professor*, *Section*, *Course*; and *TA*, which denotes the class of teaching assistants. Thin arrows represent relationships between classes. The relationships are bidirectional and consists of five relationships:

(1) (1:m) relationship from *Employee* to *Section*, meaning an employee teaches a set of sections and conversely, a section is taught by at most one professor.

(2) (1:1) relationship from *Section* to *TA*, meaning a teaching assistant assists at most one section and conversely, a section has at most one teaching assistant.

(3) (n:m) relationship from *Section* to *Student*, meaning a student takes a set of sections and conversely, a section is taken by a set of students.

(4) (1:m) relationship from *Section* to *Course*, meaning a section is of at most one course and conversely, a course has a set of sections.

(5) (n:m) relationship from *Course* to itself, meaning a course has a set of prerequisite courses and it is a prerequisite for a set of courses.

Finally, thick arrows indicate ISA relationships between classes. In the example, a professor is an employee and a teaching assistant is an employee and a student.

The declarations of the classes in the UNIVERSITY database is shown in Table 2.1. The syntax used for a class declaration is presented in Appendix A. Each class has a name, followed by an optional list of names of its superclasses. Each class is of tuple type. Attributes of a class are either atomic or of a structured type. Examples of atomic attributes are *id* in the *Student* class and *salary* in the the *Empolyee* class , both of them are of *integer* type. A structured type is either a set or a tuple type. For example, the attribute *teaches* of the *Professor* class is of set type $\{Section\}$ and the attribute *dateOfBirth* of the *Employee* class is of tuple type: $[day : integer, month : integer, year : integer]$. Finally, the UNIVERSITY database has eight persistent roots. Each persistent root has a name and associated with a type. For example, *Employees* is a persistent root of type $\{Employee\}$. The other persistent roots are shown in Table 2.1.

Classes

Class $Course$ [
$name$: $string$,
$number$: $string$,
$hasSections$: $\{Section\}$,
$hasPrerequisites$: $\{Course\}$,
$isPrerequisiteFor$: $\{Course\}$]

Class $Section$ [
$number$: $string$,
$isTaughtBy$: $Professor$,
$hasTA$: TA,
$isSectionOf$: $Course$,
$isTakenBy$: $\{Student\}$]

Class $Employee$ [
$name$: $string$,
id : $integer$,
$salary$: $integer$,
age : $integer$,
$address$: $[city : string, street : string]$]

Class $Professor : Employee$ [
$rank$: $string$,
$teaches$: $\{Section\}$]

Class $Student$ [
$name$: $string$,
id : $integer$,
$address$: $[college : string,$
$roomNumber : string]$,
$takes$: $\{Section\}$]

Class $TA : Employee, Student$ [
$assists$: $Section$]

Persistent Roots

name $Employees$: $\{Employee\}$
name $Courses$: $\{Course\}$
name $PrimaryCourses$: $\{Course\}$
name $Students$: $\{Student\}$
name TAs : $\{TA\}$
name $Professors$: $\{Professor\}$
name $VisitingProfessors$: $\{Professor\}$
name $Sections$: $\{Section\}$

Table 2.1: Classes and persistent roots of the UNIVERSITY database schema.

2.6 A Formal Object Database Model

In this thesis, we use a formal object database model as presented in Chapter 21 of [AHV95]. In particular, this model will be used in Chapter 3 to present a logic-based constraint specification language, and a formalization for path expressions and primitive modification operations. The model has all features of the core model presented in Section 2.3, such as object identity, complex values, object classification and typing as well as multiple inheritance. The model is very similar to the one underlying the O_2 OODBS, as presented in Chapters 3 and 4 of [Deu91].

Formal definitions of the model are based on the assumption that the following notations and sets are given:

- Atomic types names are *integer*, *string*, *float*, *boolean* and their pairwise disjoint corresponding domains.
- The set of atomic values, denoted **dom**, is the union of domains of the atomic types.
- A set **att** of attribute names.
- A set **class** of class names.
- An infinite set **oid** $= \{o_1, o_2, \ldots\}$ of object identifiers.
- A special value *nil* represents the undefined value.

The sets **dom**, **att**, **class** and **oid** are pairwise disjoint sets.

In the following definition, we present how complex values are constructed by recursively applying tuple and set constructors to atomic values, object identifiers and *nil*.

Definition 2.6.1 (Values) Given a set $\mathbf{I} \subseteq$ **oid**, the family of values over $\mathbf{I}$, denoted by **val**($\mathbf{I}$), is defined as follows:

(1) The undefined value, *nil*, is a value over $\mathbf{I}$.

(2) The object identifiers, $\mathbf{I}$, are values over $\mathbf{I}$.

(3) The atomic values **dom** are values over $\mathbf{I}$.

(4) If $v_1, \ldots, v_n$ are values over $\mathbf{I}$ and $a_1, \ldots, a_n$ are distinct attributes names, then the tuple $[a_1 : v_1, \ldots, a_n : v_n]$ is a value over $\mathbf{I}$.

(5) If $v_1, \ldots, v_n$ are distinct values over **I** then the set $\{v_1, \ldots v_n\}$ is a value over **I**.

The set of all values over the set of all identifiers, **val**(**oid**), will be denoted **val**. □

Two remarks about the definition of values: First, in (2) objects identifiers are considered as values. Thus an object can refer to itself or to other objects. Second, other constructors for values, e.g., a list constructor can be included, but in this thesis we will only consider the tuple and set constructors.

Now, by using sets of object identifiers **oid** and values **val** we can define an objects follows.

Definition 2.6.2 (Object) An object is a pair $< i, v >$ where $i \in$ **oid** represents an object identifier and $v \in$ **val** represents the object value. □

▶ **Example 2.6.3 (Values and Objects)** The following are examples of values:

– $[day : 25, month, \text{"May"}, year : 2000]$

– $\{\text{"red"}, \text{"blue"}, \text{"green"}, \text{"white"}\}$

– $[name : \text{"Math."}, number : \text{"G32"}, hasSections : \{\#33, \#34\},$
  $hasPrerequisites : \{nil\}, isPrerequisiteFor : \{\#21, \#22\}]$

An example of an object is

$$\begin{aligned} < \#71, [\ & name : \text{"Pluto"}, id : 4722, \\ & address : [college : \text{"Informatik"}, roomNumber : \text{"G311"}], \\ & takes : \{\#33\}] > \end{aligned}$$

■

Similar to the definition of values which are defined w.r.t. to a set of object identifiers, types of values and objects are defined w.r.t. a given set of class names.

Definition 2.6.4 (Type) Given a set $\mathbf{C} \subseteq$ **class**, the family of types over **C** is defined as follows:

(1) The atomic types *integer*, *string*, *float*, and *boolean* are types over **C**.

(2) The class names, **C**, are types over **C**.

(3) If $\tau_1, \ldots, \tau_n$ are types over **C** and $a_1, \ldots, a_n$ are distinct attributes names in **att**, then $[a_1 : \tau_1, \ldots, a_n : \tau_n]$ is a *tuple* type over **C**.

(4) If τ is a type over $\mathbf{C}$ then $\{\tau\}$ is a *set* type over $\mathbf{C}$. □

To allow class attributes to refer to their classes or to other classes, in (2) class names are considered as types.

The notion $\mathbf{type}(\mathbf{C})$ will be used to denote the family of types over $\mathbf{C}$ together with the special class name *Any*. The class *Any* is not used in constructing the family of types over $\mathbf{C}$. This means that "*Any*" does not appear in the definition of any type over $\mathbf{C}$.

▶ **Example 2.6.5 (types)** Examples of types over the classes of the UNIVERSITY database are listed below with names associated with them:

– $Date : [day : integer, month, string, year : integer]$

– $Colors : \{string\}$

– $CourseType : [name : string, Number : string, hasSections : \{Section\},$
  $hasPrerequisities : \{Course\}, isPrerequisiteFor : \{Course\}]$

– $Address : [city : string, street : string]$

– $EmployeeType : [name : string, id : integer, salary : integer,$
  $age : integer, address : Address]$

– $ProfessorType : [name : string, id : integer, salary : integer,$
  $age : integer, address : Address, rank : string, teaches : \{Section\}]$

The three values given in Examples 2.6.3 are of types *Date*, *Colors*, and *CourseType* respectively. ■

Recall from Section 2.3 that ISA relationships among classes can be represented as a rooted directed acyclic graph, called a class hierarchy. Formally, a class hierarchy consists of a set of class names, the types of these classes, and the description of ISA relationships between these classes.

Definition 2.6.6 (Class Hierarchy) A class hierarchy is a triple $(\mathbf{C}, \sigma, \prec)$, where $\mathbf{C}$ is a finite set of class names, σ is a mapping from $\mathbf{C}$ to $\mathbf{type}(\mathbf{C})$, and $\prec$ is a partial order on $\mathbf{C}$. □

Recall from the informal description of object-oriented database concepts that in a class hierarchy a subclass inherits all attributes and methods from its direct and indirect superclasses. Thus the type associated with a subclass should include the types associated with its superclasses. To formulate this semantics in the model, the subtyping relationship, denoted $\leq$, is introduced and describes when one type includes another.

Definition 2.6.7 (Subtyping Relationship) Given a class hierarchy $(\mathbf{C}, \sigma, \prec)$. The subtyping relationship $\leq$ on $\mathbf{type}(\mathbf{C})$ is the smallest partial order satisfying the following conditions:

(1) If $c \prec c'$, then $c \leq c'$.

(2) If $\tau_i \leq \tau_i'$ for each i, $1 \leq i \leq n$ and $n \leq m$, then $[a_1 : \tau_1, \ldots, a_m : t_m] \leq [a_1 : \tau_1', \ldots, a_n : \tau_n']$.

(3) If $\tau \leq \tau'$, then $\{\tau\} \leq \{\tau'\}$.

(4) For each τ, $\tau \leq Any$. □

A class hierarchy is well formed if for every two classes c and c', if c is a subclass of c' then, the type associated with c is a subtype of the type associated with c'. Formally:

$$c \prec c' \Longrightarrow \sigma(c) \leq \sigma(c').$$

▶ **Example 2.6.8 (Well Formed Class Hierarchy)** Let $(\mathbf{C}, \sigma, \prec)$ be the class hierarchy of the UNIVERSITY schema. The set of class names $\mathbf{C}$ is

$$\mathbf{C} = \{Course, Section, Employee, Professor, Student, TA\}.$$

In the UNIVERSITY schema, we have that the *Professor* class is a subclass of the *Employee* class, and the *TA* class is a subclass of both of the *Employee* class and the *Student* class. Thus, the description of ISA relationships between these classes by the relation $\prec$ is

$$\begin{aligned} &Professor \prec Employee, \\ &TA \prec Employee \text{ and} \\ &TA \prec Student. \end{aligned}$$

The types associated with the classes in the UNIVERSITY schema are shown in Table 2.2. From this table and the definition of the subtyping relationship we have

$$\begin{aligned} &\sigma(Professor) \leq \sigma(Employee), \\ &\sigma(TA) \leq \sigma(Employee), \text{ and} \\ &\sigma(TA) \leq \sigma(Student). \end{aligned}$$

$$\begin{aligned}
\sigma(Employee) = \ & [name : string, id : integer, salary : integer, \\
& age : integer, address : Address] \\
\sigma(Professor) = \ & [name : string, id : integer, salary : integer, \\
& age : integer, address : Address, \\
& rank : string, teaches : \{Section\}] \\
\sigma(TA) = \ & [name : string, id : integer, salary : integer, \\
& age : integer, address : Address, \\
& rank : string, teaches : \{Section\}, assists : Section] \\
\sigma(Student) = \ & [name : string, id : integer, address : Address, takes : \{Section\}] \\
\sigma(Section) = \ & [number : string, isTaughtBy : Professor, hasTA : TA, \\
& isSectionOf : Course, isTakenBy : \{Student\}]
\end{aligned}$$

Table 2.2: Types associated with classes of the UNIVERSITY database schema.

As such, the class hierarchy $(\mathbf{C}, \sigma, \prec)$ of the UNIVERSITY schema is a well formed class hierarchy. ■

In the following definition we present the semantics of class hierarchies. Informally, in this semantics classes are associated with sets of oids by using an oid assignment such that

- oids associated with two classes that are not related by an ISA relationship are disjoint; and
- the set of oids associated with a superclass is a superset of oids of its subclasses.

Therefore, an object of a subclass is also an object of its superclasses.

Definition 2.6.9 (Semantics of a Class Hierarchy) Let $(\mathbf{C}, \sigma, \prec)$ be a well formed class hierarchy. An *oid assignment* is a function π mapping each class name in $\mathbf{C}$ to a finite set of oids, such that for every c and c' in $\mathbf{C}$, if $c \neq c'$ then $\pi(c) \cap \pi(c') = \emptyset$. Given the oid assignment π, the *extension* of c is denoted by $\pi^*(c)$ and defined as follows,

$$\pi^*(c) = \bigcup_{\mathbf{c'} \in \mathbf{K(c)}} \pi(c')$$

where

$$\mathbf{K(c)} = \{c' | c' \in \mathbf{C}, c' \prec c\}.$$

□

Now, we can define the semantics of types. Since class names are types, the semantics of types is defined according to the class hierarchy $(\mathbf{C}, \sigma, \prec)$ and an oid assignment π.

Definition 2.6.10 (Semantics of Types) Given an oid assignment, the *disjoint interpretation* of a type τ, denoted $dom(\tau)$, is defined as follows.

(1) If $\mathbf{I} = \bigcup_{c \in \mathbf{C}} \pi(c)$, then $dom(Any) = \mathbf{val}(\mathbf{I})$.

(2) For each atomic type τ, $dom(\tau)$ is defined as the usual interpretation of that type.

(3) For each $c \in \mathbf{C}$, $dom(c) = \pi^*(c) \cup \{nil\}$.

(4) $dom(\{\tau\}) = \{\{v_1, \dots, v_n\} | n \geq 0, \text{ and } v_i \in dom(\tau), i \in [1, n]\}$.

(5) $dom([A_1 : \tau_1, \dots, A_k : \tau_k]) =$
$\{[A_1 : v_1, \dots, A_k : v_k, A_{k+1} : v_{k+1}, \dots A_n : v_n] \mid v_i \in dom(\tau_i), i \in [1, k],$
$v_j \in \mathbf{val}(\mathbf{I}), j \in [k+1, n]\}$. □

According to Definition 2.6.10, the domain of a subtype is a subset of the domain of its supertype. This semantics of types is called domain-inclusion semantics [Car88]. Formally, for every type τ and τ' in $\mathbf{type}(\mathbf{C})$,

$$\tau \leq \tau' \Longrightarrow dom(\tau) \subseteq dom(\tau').$$

So far we described how components of the structural part of an object database can be formalized. It remains to show how the behavior part, i.e., methods, can be formalized. We first need to define the concept of method signatures formally. Recall that the signature of a method consists of the name of that method, and types and order of input and output arguments to that method. Formally, let $(\mathbf{C}, \sigma, \prec)$ be a well formed class hierarchy. Let c be a class in $\mathbf{C}$. Each method defined in a class c has a signature $m : c \times \tau_1 \times \dots \times \tau_n \rightarrow \tau$ where m is the name of the method, and $\tau_1, \dots, \tau_n, \tau$ are types in $\mathbf{type}(\mathbf{C})$.

Recall that inheritance implies that a subclass inherits methods of its superclasses. Also, sometimes the implementation of a method of a superclass is overridden by the implementation of that method at one of its subclasses. This should be done without type conflicts or ambiguity. Formally, let $\mathbf{M}$ be the set of method signatures that are associated with the classes in a well formed class hierarchy $(\mathbf{C}, \sigma, \prec)$. The set $\mathbf{M}$ is well formed if it satisfies the following rules.

Covariance: If the method m is defined in both of classes c, c' and c is a subclass of c', then $m(c, \tau_1, \dots, \tau_n) : t$ and $m(c', \tau'_1, \dots, \tau'_n) : \tau'$, and τ_i is subtype of τ'_i, $i \in [1, n]$ and τ is subtype of τ'.

Unambiguity: If c is a subclass of c' and c'' and there is a definition of the method m at c' and c'' then there is a definition of m in a subclass of c' and c'' that is either c itself or a superclass of c.

We now present formal definitions for database schemas and instances. A database schema consists of a well formed class hierarchy, a well formed set of signatures and set of names that act as persistent roots.

Definition 2.6.11 (Database Schema) A database schema is a 5-tuple $(\mathbf{C}, \sigma, \prec, \mathbf{M}, \mathbf{G})$, such that

- $(\mathbf{C}, \sigma, \prec)$ is a well formed class hierarchy,
- $\mathbf{M}$ is a well formed set of method signatures, and
- $\mathbf{G}$ is a set of names disjoint from $\mathbf{C}$ with a type associated to each name. □

Examples of persistent roots of the UNIVERSITY schema are: *Employees*, referring to a set of employees objects; and *Courses* and *PrimaryCourses*, referring to sets of course objects. Formally, the persistent roots of the the UNIVERSITY schema are

$$\begin{aligned}\mathbf{G} = \{\ & \mathit{Employees} : \{\mathit{Employee}\},\ \mathit{VisitingProfessors} : \{\mathit{Professor}\},\ \mathit{TAs} : \{\mathit{TA}\},\\ & \mathit{PrimaryCourses} : \{\mathit{Course}\},\ \mathit{Students} : \{\mathit{Student}\},\ \mathit{Sections} : \{\mathit{Section}\},\\ & \mathit{Professors} : \{\mathit{Professor}\},\ \mathit{Courses} : \{\mathit{Course}\}\}.\end{aligned}$$

A database instance is defined as an assignment that assigns oids to classes, values to oids, values to persistent roots, and partial functions to method signatures.

Definition 2.6.12 (Database Instances) An instance of schema $(\mathbf{C}, \sigma, \prec, \mathbf{M}, \mathbf{G})$ is a 4-tuple $I = (\pi, \nu, \gamma, \mu)$ where

- π is an oid assignment and $\mathbf{I} = \bigcup_{c \in \mathbf{C}} \pi(c)$.
- ν maps each oid in $\mathbf{I}$ to a value in $\mathbf{val}(\mathbf{I})$ such that for each $c \in \mathbf{C}$ and $o \in \pi(c)$, $\nu(o) \in \mathit{dom}(\sigma(c))$.
- γ associates with each name in $\mathbf{G}$ of type τ a value in $\mathit{dom}(\tau)$.
- μ assigns semantics to method names in agreement with the method signatures in $\mathbf{M}$. For each signature $m : c \times \alpha \to \tau$, $\mu(m : c \times \alpha \to \tau)$ is a partial function from $\mathit{dom}(c \times \alpha)$ to $\mathit{dom}(\tau)$. □

#	c_A	a	τ_A	c_B	b	τ_B
1	*Professor*	*teaches*	$\{Section\}$	*Section*	*isTaughtBy*	*Professor*
2	*TA*	*assists*	*Section*	*Section*	*hasTA*	*TA*
3	*Student*	*takes*	$\{Section\}$	*Section*	*isTakenBy*	$\{Student\}$
4	*Course*	*hasSections*	$\{Section\}$	*Course*	*isSectionOf*	*Course*
5	*Course*	*hasPrerequisites*	$\{Course\}$	*Course*	*isPrerequisitesFor*	$\{Course\}$

Table 2.3: Relationships of the UNIVERSITY database schema.

From the definition it follows that method implementations are considered extensionally in the model. However, the semantics to method implementations is actually given intensionally. This can be done by coding method implementations in some programming language. In Appendix B we present the syntax of a formal language for method implementations.

We conclude this chapter by making two assumptions concerning update transactions and relationships.

2.7 Assumptions: Transactions and Relationships

We assume that transactions are objects of a class named *Transaction* and that there is an attribute of the *Transaction* class named *updatedObjects* of set type $\{Any\}$.

We assume that relationships are bidirectional. As such, in this thesis the binary relationships between classes are modeled as follows:
Let a and b be reference attributes for the relationship R in the class c_A and c_B, respectively; thus b is an inverse attribute of a, and vice versa.

(1) If R is a (1:1) relationship, then both attributes a and b are single-valued of types c_B and c_A, respectively.

(2) If R is a (1:m) relationship then the attribute a is a set-valued attribute of type $\{c_B\}$ and the attribute b is a single-valued attribute of type c_A.

(3) If R is a (n:m) relationship, then both attributes a and b are set-valued of types $\{c_B\}$ and $\{c_A\}$ respectively.

In Section 3.6, we present how inverse relationships in our approach are maintained.

The inverse relationships in the UNIVERSITY schema of Figure 2.1 are represented by making thin arrows, which represent relationships between classes, bidirectional. The semantics

of relationships of the UNIVERSITY schema are presented in Section 2.5. The description of how these relationships and their inverses are implemented is summarized in Table 2.3.

In Table 2.3, for each tuple $< c_A, a, \tau_A, c_B, b, \tau_B >$, the attribute a in class c_A of type τ_A has the inverse attribute b in class c_B of type τ_B, and vice versa. For example, the first tuple represents the (1:m) relationship from the *Employee* class to the *Section* class, in which the *isTaughtBy* attribute in the *Section* class of type *Professor* is an inverse attribute of the attribute *teaches* in the *Professor* class of type $\{$*Section*$\}$, and vice versa. In Table 2.3, tuples 2 through 5 represent respectively relationships (2) through (5), that are listed in Section 2.5.

Chapter 3

Consistency Constraints

Consistency constraints represent laws that govern states and transitions in an application domain. There are three types of integrity constraints that are defined according to how many database states are subject to the constraint. These types are referred to as state, transition and dynamic constraints.

State constraints are specified over objects of a single database state. They describe properties that a database has to satisfy at each moment in time in which the database is stable. For that reason, this type of constraints is also called *static constraints*. This type of constraints defines valid database states. A typical example is the constraint that says that "*every section must be taught by a professor*". The validation of this constraint consists of ensuring that, for each object o of the *Section* class in the current database state, the value of the *isTaughtBy* attribute is not null.

There are several types of state constraints defined in the literature that have specific names such as domain, non null, key, referential integrity, to name a few [GA93, Deß93, Ger96b]. It is seldom to find a database application that has none of these types of constraints associated with schema. This shows why most work in the literature devoted to semantic database integrity is addressed for state constraints. In fact, in addition to database integrity, state constraints are used in other areas of the database field such as database design, semantic query optimization, development of new concurrency control protocols and schema transformation from one data model to another [FG95].

Transition constraints are specified over objects of two (logically) consecutive database states: the current database state and the one obtained by an update transaction to the current state. They describe properties that a database has to satisfy at every transition from one database state to another. This type of constraints defines valid transitions. A

much used example of this type is the constraint that says that "*salary of an employee should not be decreased*". The validation of this constraint consists of ensuring that for each transition from the current state, say c, to a new state, say n, in which the *salary* attribute of an object o of *Employee* class is changed, the value of $o.salary$ in the state c is less than that of $o.salary$ in the state n.

Transition constraints are specified in an ad hoc way, as we did in the example above or by adding operators such *old* and *new* to a logic-based constraint language such as the one used in the IDEA methodology for designing database applications [CF97].

Dynamic constraints (also called temporal constraints [GA93, SL88, Cho94]) represent a general case of state and transition constraints. This type of constraints is specified over a sequence of more than two database states. Usually dynamic constraints are used in temporal and historical databases. An example of dynamic constraints is the constraint that says that "*once an employee is fired from a company, she/he can never be hired again*". This means that adding a new employee to the current state of the database is accepted only if the new employee does not appear in any of the old states of the database. Thus the constraint is not only specified on the state before and after the modification but also on the history of the database. There is a major concerns about the relevance of dynamic constraints to the practice and the efficiency of approaches proposed for their maintenance [GA93].

In this thesis we deal only with state constraints. This chapter is devoted to some topics concerning state constraints. The chapter is organized as follows. In Section 3.1, we present an overview of the process of constraint design. In Section 3.2, we present an assumption concerning constraint quality. In Section 3.3, we present a logical specification language for state constraints. In Section 3.4, we present a formal definition for path expressions. In our approach we consider object consistency at the lowest update granularity. As such, in Section 3.5, we present definitions of primitive modification operations. Section 3.6 is devoted to a special type of state constraints, namely constraints that arise due to the presence of inverse relationships in our object database model. In Section 3.7, we first motivate well formedness of state constraints adapted in this thesis and then we present the definition of this formulation, namely range restricted well formed formulas. Finally, in Section 3.8 we present a set of constraint examples that covers most of the constraint classes discussed in the literature.

3.1 Constraint Design

Constraint design is a subtask of database design that transforms rules of an application domain either to notions of the data model that is underlying the DBMS (*implicit constraints*) or to formulas of some formal language (*explicit constraints*). Before the implementation, the set of explicit constraints is constructed through three stages. Each stage corresponds to a phase of the database design process. Following the description of database design process given in [HS97, Dat90, EN94], the three stages are users' requirements collection and analysis, the conceptual database design, and the logical database design.

3.1.1 Users' Requirements Collection and Analysis

After database designers have carefully completed the phase of database users' requirements collection and analysis, database designers identify a part of data requirements they believe to be the initial set of integrity constraints. Thus, the starting point of integrity constraints design is the database user' requirements. Constraints in this phase, as most of other data requirements, are described in a textual form e.g.; "*each department is managed by a professor*". At this early stage of database design, database designers cannot exactly identify database requirements that will be implemented as database constraints. In fact, in this phase, designers need not to be aware of doing that, but their work are centered around understating the collected users' requirements so that they can analyse it and obtain what is relevant to database design and capable for implementation. The identification of constraints is usually left to the next phase of the database design process, the conceptual database design.

3.1.2 Conceptual Database Design

The next phase of the database design process is the conceptual database design. In this phase, database designers transform database requirements into a conceptual database schema. Database designers do the transformation by using notations of a high-level conceptual data models, such as the Entity-Relationship (ER) model [Che76] or one of its extensions [FG95]. Compared to the phase of analysis of user' requirements, database designers can identify database constraints easily. The identification of database constraints relies on two facts:

(1) Some database requirements are captured in the conceptual schema but they are neither defined as entity types nor as relationship types.

(2) Some database requirements cannot be described in the notation of the used conceptual data model.

Database requirements of the first category are called *implicit constraints*. Examples of implicit constraints in the (ER) model are those that imposed on a relationships between entity types, namely *cardinality ratio* and *participation constraints*. An example of a cardinality ratio is the constraint "*a professor may teach many courses*". An example of the participation constraints is the constraint that says "*each student takes at least one section*".

Database requirements of the second category are called *explicit constraints*. Another characteristics of explicit constraints is that they are general statements that have not specific patterns to be included as a part of the notations of data models. An example of explicit constraints is the constraint "*the salary of an employee is less than that of his department's manager*". Usually explicit constraints retain their textual form of requirements collection and analysis phase unless the conceptual data model has a formal or a semi-formal specification language which is not the case in the (ER) data model. But some formalizations of the (ER) data model have this feature such as the one proposed in [Gog94].

3.1.3 Logical Database Design

The goal of logical database design is to transform the conceptual database schema into a logical schema of some implementable data model such as the relational data model or one of the object-oriented data models such as O_2 [Deu91]. In this transformation, some implicit constraints of the conceptual database schema are transformed to explicit constraints of the logical database schema. The reason behind this is that conceptual data models have more notations for describing data than logical data models. This indicates why conceptual data models are attributed as high-level data models. Thus explicit constraints can be used to measure whether one data model is more semantically powerful than another.

The formal specification of explicit constraints in logical database design is necessary for two reasons. First, some explicit constraints are used to improve the quality of the resulting logical database schema such that anomalies of data redundancies are avoided. This approach is taken for relational database schemata. Also, for object-oriented schemata, some explicit constraints are used to confirm that the resulting schema is correct according to database designers' interpretation of users' data requirements. Often these classes of constraints are expressed using an ad hoc notation. Second, before the implementation of explicit constraints they must be addressed against the issues of satisfiability and redundancy. These issues need constraints to be described as well-formed formulas of some logic-based

specification language.

Relational Databases. In relational databases, a well known class of constraints that influences the design of logical database schema is called *dependencies*. Several forms of dependencies have been introduced in the literature, such as functional dependencies, key dependencies, join dependencies, multivalued dependencies, and inclusion dependencies [Ull88, Tha91]. In the seventies and eighties, a considerable amount of work has been devoted to study dependencies in depth. The motivation was to obtain a good logical database schema. Here "good"means that a database schema satisfies certain normal forms, where each normal form has certain desirable properties that enhance data manipulation. A normal form restricts dependencies that a relation schema may have. Although dependencies are well-studied they are often described in ad hoc notation. An early attempt to a logic-based formulation of dependencies is due to [Nic78]. An interesting fact of this topic is that they are special cases of a closed wff of first-order logic having a certain pattern, known as *generalized dependency* [GJ82]. For a more in-depth coverage of dependencies and normalization we refer reader to [HS97, EN94, Ull88]. Also a coverage of formal treatments of dependencies can be found in [AHV95, Tha91].

Object Databases. More recently, object normal forms and dependency constraints for object databases have been introduced in [TSS97]. In this approach, functional and multivalued dependencies are extended to reflect the complexity of object-oriented schemata such that dependencies are not only imposed on attributes of the same class (as in the relational schemata) but also on attributes of different components of an object-oriented schema. The result of this adaption are three classes of dependencies.

The first class, called path dependencies, is imposed at the schema level and expresses an inter-object relationships. An example of path dependencies is the constraint that says "*all sections taught by a professor are assisted by the same teaching assistant*". This constraint is validated by evaluating all paths that have a professor as a source object and a teaching assistant as a destination object. The constraint then is valid if for every two paths that pass by two sections, each taught by the same professor, the two paths have the same teaching assistant as a destination.

The other two classes, called the local and global dependencies, are imposed at the object level and express the intra-relationships between objects and their components. Whereas local dependencies apply to single instances only, global dependencies apply to all instances of the same class.

The definition of object-oriented normalization introduced in this approach is that a class is in a normal form if user's data requirements, expressed as a set of global dependencies constraints, are derivable from those that are implicitly expressed in the object-oriented schema. For this purpose, a set of inference axioms for each type of dependency mentioned above is introduced. Normalization in this sense is the matter of confirming that the logical database schema is correct with respect to the database designer interpretation of users' data requirements.

To this end, the result of the logical database design is a logical database schema and a set of explicit constraints. Before the actual implementation of constraints on a specific DBMS, their quality must be tested first.

3.2 The Quality of Constraints

The quality of the constraints resulting from logical database design can be defined according to three criteria, *correctness*, *consistency*, *minimality*.

Constraints are *correct* if they are

- syntactically correct with respect to the syntax of the used specification language, and
- semantically correct with respect to the logical database schema to be associated with.

The second condition is necessary to avoid cases in which a constraint is syntactically correct but semantically is not. For instance, it may happen that a constraint refers to a class or an attribute that does not exist in the schema or compares two attributes of incompatible domains.

A constraint set is *finitely satisfiable* if it can be satisfied by a finite database state. A constraint set must be finitely satisfiable. Otherwise, in the case of unsatisfiability, every update transaction would be rejected or, in the case of infinitely satisfiable, the content of the database would be infinite. Due to feasibility considerations, finite satisfiability is almost ignored in constraint design; satisfiability is undecidable and for decidable cases the problem is NP-hard [Man90]. The problem is formally addressed in [BM86] and an interactive prototype to assist in the design of finitely satisfiable constraints, called SIC, is proposed in [BEST98]. This is done in the context of relational databases. For object-oriented databases we are not aware of any similar approaches.

A constraint set is *minimal* if it has no redundancies. A constraint set IC is redundant if, in the logic terminology, for some constraint $W \in IC$, the constraint set $IC - \{W\}$ logically

implies W (or equivalently W can be derived from $IC - \{W\}$). Except the dependency constraints mentioned above, minimality is almost ignored for general constraints because of feasibility issues. However, this can be done using tools of theorem prover under the supervision of a person that may have experience or good knowledge of the logic.

It is worth pointing out that we can address satisfiability and minimality of constraint sets of object-oriented schemata by mapping the object schema and integrity constraints to one of logical formalizations of object-oriented databases such as F-logic [KL89] or one of knowledge representation languages such as Telos [JJ91].

In this thesis we assume that the constraint sets are guaranteed to be correct, minimal and non-redundant. In the following section we present a logic-based language for constraint specification.

3.3 Constraint Specification Language

In this section, we present a logic based constraint specification language. The language is strongly influenced by calculus-based query languages for complex values [AB95], for OODBs [AHV95] and the identity query language (IQL) [BDK92]. The syntax and semantics of the language are presented respectively in Subsection 3.3.1 and Subsection 3.3.2.

3.3.1 Syntax of the Language

The language is defined by adapting the syntax of first-order, many-sorted languages [End72] to components of a given database schema S. The language is denoted by $\mathcal{L}_S$ to emphasize the role of the database schema S in the definition of the language. Let S be the database schema $(C, \sigma, \prec, M, G)$. Sorts of $\mathcal{L}_S$ are the set of types, $\mathbf{type}(C)$, as described in Definition 2.6.4. As $\mathcal{L}_S$ is a many-sorted language, every constant and every variable is associated with a type. Every function and predicate is associated with a signature. The signature of a k-ary predicate is a k-tuple of types. For a k-ary function, the signature is a $k+1$-tuple of types. In the following, we define the alphabet, terms and well-formed formulas of $\mathcal{L}_S$ respectively.

Alphabet. The alphabet of $\mathcal{L}_S$, denoted by $\mathcal{A}$, consists of the following symbols:

- *Names:* For every name g in G, g is a member of $\mathcal{A}$.
- *Atomic constants:* *nil* and every $x \in \mathbf{oid} \cup \mathbf{dom}$ are members of $\mathcal{A}$.
- *Variables:* For every $\tau \in \mathbf{type}(C)$, there is a countable infinite set of variables in $\mathcal{A}$.

- *Predicates:* Binary predicates of $\mathcal{A}$ are:
  - For every type $\tau \in \mathbf{type}(C)$, there is a predicate symbol in $\mathcal{A}$ denoted by $=_\tau$ (equality) with signature $\tau \times \tau$.
  - For every type $\tau \in \mathbf{type}(C)$, there is a predicate symbol in $\mathcal{A}$ denoted by $\in_\tau$ (membership) with signature $\tau \times \{\tau\}$.
  - For every type $\tau \in \mathbf{type}(C)$, there is a predicate symbol in $\mathcal{A}$ denoted by $\subseteq_\tau$ (subset) with signature $\{\tau\} \times \{\tau\}$.
  - For every type $\tau \in \{integer, float\}$, there is a predicate symbol in $\mathcal{A}$ denoted by $<_\tau$ (less than) with signature $\tau \times \tau$.
  - For every type $\tau \in \{integer, float\}$, there is a predicate symbol in $\mathcal{A}$ denoted by $>_\tau$ (greater than) with signature $\tau \times \tau$.
- *Functions:* Function symbols of $\mathcal{A}$ are:
  - A dereferencing function $\uparrow$ of signature $\mathbf{oid} \times Any$.
  - For every attribute A of type τ and each type τ' such that $[A : \tau] \leq \tau'$ there is a function $A : \tau' \rightarrow \tau$.
  - For every attribute $A_1, \ldots, A_n$ and types $\tau_1, \ldots, \tau_n$ there is a tuple constructor $[]_{A_1:\tau_1,\ldots,A_n:\tau_n}$.
  - For every n and every type τ there is a set constructor $\{\}^n_\tau$.
- *Logical connectives:* $\neg$ (not), $\vee$ (or), $\wedge$ (and), $\rightarrow$ (implication).
- *Quantifiers:* For each type $\tau \in \mathbf{type}(C)$, there are quantifiers $\forall_\tau$ (for all) and $\exists_\tau$ (there exists).
- *Auxiliary symbols:* ")", "(".

Terms. The terms of $\mathcal{L}_S$ are the following:

- Atomic constants, variables and elements of G are terms of their corresponding types.
- If t is a term of type $\mathbf{oid}$ then $\uparrow(t)$ is a term of type $\sigma(c)$ for some $c \in C$.
- If $t_1, \ldots, t_n$ are terms of types $\tau_1, \ldots, \tau_n$ respectively, then $[]_{A_1:\tau_1,\ldots,A_n:\tau_n}(t_1, \ldots, t_n)$ is a term of type $[\tau_1, \ldots, \tau_n]$.

- If $v_1, \ldots, v_n$ are terms of type τ then $\{\}^n_\tau(v_1, \ldots, v_n)$ is a term of type $\{\tau\}$.
- If A is an attribute of type τ, t is a term of type τ', and $[A : \tau] \leq \tau'$ then $A(t)$ is a term of type τ.

Well Formed Formulas. Well formed formulas, (wffs), of $\mathcal{L}_S$ are constructed as follows:

- If t and t' are terms of type τ then $t =_\tau t'$ is a formula.
- If t and t' are terms of types τ and $\{\tau\}$ respectively then $t \in_\tau t'$ is a formula.
- If t and t' are terms of type $\{\tau\}$ then $t \subseteq_\tau t'$ is a formula.
- If t and t' are terms of type $\tau \in \{integer, float\}$ then $t <_\tau t'$ and $t >_\tau t'$ are formulas.
- If ϕ is a formula then so is $\neg\phi$.
- If ϕ and ψ are formulas then so are $\phi \wedge \psi$ and $\phi \vee \psi$.
- If ϕ is a formula then so are $(\exists_\tau x)\phi$ and $(\forall_\tau x)\phi$, where x is a variable of type τ.

▶ ***Notation 3.3.1*** In the rest of the thesis we use the following conventions and notations:

- We denote the application of tuple constructor $[]_{A_1:\tau_1,\ldots,A_n:\tau_n}$ to terms $t_1 \ldots, t_n$ as $[A_1 : t_1, \ldots, A_n : t_n]$, and the application of set constructor $\{\}^n_\tau$ to terms $t_1 \ldots, t_n$, as $\{t_1 \ldots, t_n\}$.
- Applying the attribute A to the tuple t, $A(t)$, is denoted as $t.A$ to obtain the value of component A of tuple t. The dereferencing of an object o is denoted as $o\uparrow$.
- For simplicity, the quantifications $(\forall_\tau x)$ and $(\exists_\tau x)$ will be written as $(\forall x : \tau)$ and $(\exists x : \tau)$, respectively.
- Types of constants and variables, and signatures of function and predicates are sometimes omitted when they can be inferred from the context or an irrelevant. □

A positive *literal* is an atomic formula of the form $t = t'$, $t \in t'$, or $t \subseteq t'$. A negative literal is a negation of a positive literal.

An occurrence of a variable in a wff is *free* if it is not governed by a quantifier (in that formula) containing that variable. A *closed* formula is a formula in which there are no free

occurrences of any variables. An *open* formula is a formula in which there is at least one free occurrence of a variable.

We adopt some assumptions made on quantified variables in [BM88], and which will be used in the thesis. Quantified variables must occur free in the scope of their quantifiers. This implies that, the formula $(\forall x)\psi[y]$ is not accepted. A quantified variable cannot also be a free variable in the formula. For example, the formula: $\phi[x,y] \wedge (\exists x)(\psi[x,y] \wedge \zeta[x])$, has to be written in the form: $\phi[x,y] \wedge (\exists z)(\psi[z,y] \wedge \zeta[z])$. A variable is quantified at most once. For example the formula $(\exists z)(\phi[x,y] \wedge \zeta[z]) \wedge (\exists x)(\phi[x,x])$, has to be written in the form: $(\exists z)(\phi[x,y] \wedge \zeta[z]) \wedge (\exists t)(\phi[t,t])$.

3.3.2 Semantics of the Language

The syntax for the language is defined in Subsection 3.3.1. Thus given a string of symbols one can decide whether it is a wff or not. However there is no way of giving any meaning to wffs. The meaning of wffs is defined when the semantics of the language is given. The semantics of the language involves the definition of interpretations and models for wffs of the language.

The semantics of $\mathcal{L}_S$ is defined according to the components of an instance of the schema S. Let $I = (\pi, \nu, \mu, \gamma)$ be an instance of S, as defined in Definition 2.6.12. The semantics of $\mathcal{L}_S$ is defined by assigning each type τ to $dom(\tau)$ and each term of type τ to a value in $dom(\tau)$ as follows:

- $I(v) \in dom(\tau)$ if v is a variable of type τ.
- $I(g) = \gamma(g)$ if g is a name in G.
- $I(t) = t$ if t is an atomic constant.
- $I(o\uparrow) = \nu(o)$.
- $I([A_1 : t_1, \dots, A_n : t_n]) = [A_1 : I(t_1), \dots, A_n : I(t_n)]$.
- $I(\{t_1 \dots, t_n\}) = \{I(t_1) \dots, I(t_n)\}$.
- $I(t.A) = u$, where u is the value associated with the component A of tuple t.

The closed wffs can be evaluated to **T** (true) or **F** (false) according to the following rules:

- $t = t'$ is evaluated to **T** iff $I(t) = I(t')$.

- $t < t'$ is evaluated to **T** iff $I(t) < I(t')$.
- $t \in t'$ is evaluated to **T** iff $I(t) \in I(t')$.
- $\neg\psi$ is evaluated to **T** iff ψ is evaluated to **F**.
- $\phi \wedge \psi$ is evaluated to **T** iff ϕ and ψ are both evaluated to **T**.
- $(\forall_\tau x)\psi[x]$ is evaluated to **T** iff for every $v \in I(\tau)$ $\psi[x/v]$ is evaluated to **T**, where $\psi[x/v]$ is the wff obtained by substituting each occurrence of x in ψ by constant v.

The truth value for the other connectives can be deduced from the equivalences:

$$\begin{array}{lcl} x < y & \equiv & y > x \\ \phi \vee \psi & \equiv & \neg(\neg\phi \wedge \neg\psi) \\ \phi \rightarrow \psi & \equiv & \neg\phi \vee \psi \\ (\exists_\tau x)\psi & \equiv & \neg((\forall_\tau x)\neg\psi)) \\ (t \subseteq t') & \equiv & (\forall x)(x \in t \rightarrow x \in t') \end{array}$$

Finally, if a wff W is true in a given interpretation I, then this interpretation is said to be a *model* of W; we will denote this by $I \models W$.

3.4 Path Expressions

In this section we first present the definition of path expressions and then discuss some notations and definitions concerning their usage in this thesis.

Definition 3.4.1 (Path Expressions) The set of path expressions, denoted by **PE**, is a subset of the set of terms of the language $\mathcal{L}_S$ defined as follows:
If A is an attribute of type τ, v is a term of type τ', and $[A : \tau]$ is a subtype of τ' then $v.A \in$ **PE**. □

The definition of terms of language $\mathcal{L}_S$ restricts the application of the dot operator "."– in fact attributes, see Notation 3.3.1– to be applied only to tuple structured terms. Hence in the path expression $v.A_1.\cdots.A_n$, term v and every attribute A_i, $i \in [1, n[$, are never be of a set type. As such, path expressions in **PE** sometimes are referred to as *linear* path expressions. More general, but also more complicated, formal definition for path expression is given in [KMP92, LV97]. In these attempts, the authors first present general forms of

$pr :< \#93, [rank : \text{"associate"}, teaches : \{\#55\}, \ldots] >$

$st :< \#23, [\ldots, address : [college : \text{"Informatik"}, roomNumber : \text{"D23"}], \ldots] >$

$se :< \#55, [\ldots, isTaughtBy : \#93, \ldots] >$

$co :< \#11, [\ldots, hasSections : \{\#55\}, \ldots] >$

Figure 3.1: Objects of the UNIVERSITY database.

path expressions, second what are conditions under which these general forms are valid with respect to the type system of the underlying database, and then they present semantics for path expressions. Here, our definition presents what is a path expression but not how it is constructed nor what is its semantics. This is simply because of the construction and semantics of path expressions are included in the syntax and semantics of terms of language $\mathcal{L}_S$.

▶ **Example 3.4.2 (Path Expressions)** Let pr, st, se, and co be terms denoting objects of the classes *Professor*, *Student*, *Section* and *Course*, respectively. In the UNIVERSITY schema, the following are path expressions in **PE** :

$$P1 : st \uparrow .address.college$$

$$P2 : se \uparrow .isTaughtBy \uparrow .rank$$

$$P3 : co \uparrow .hasSections$$

The expression $pr \uparrow .teaches \uparrow .hasSections.hasTA$ is not a path expression since *hasSections* is of type $\{Section\}$ and attribute *hasTA* does not apply to a set structured type. For the same reason, the expression $pr \uparrow .teaches \uparrow .hasSections \uparrow .hasTA$ is not a path expression. Suppose that objects pr, st, se, and co have the states as shown in Figure (3.1). Then the semantics of paths *P1*, *P2*, and *P3* are values "Informatik", "associate", and $\{\#55\}$ respectively. ■

▶ ***Notation 3.4.3*** In the remainder of the thesis we use the following notations and definitions:

- Let $p : v.A_1.\cdots.A_n$ be a path expression in **PE**. The term v is said to be a *path prefix* of p and the sequence $A_1 \ldots A_n$ a *path suffix*.

- We will denote by $\mathbf{PE}_\tau$ the set of all paths $v.A_1.\cdots.A_n$ in $\mathbf{PE}$ such that v is a variable and A_n is an attribute of type τ.
- Sometimes we refer to path suffixes as "path expressions".
- For sake of simplicity, we will remove the deference operator "↑"from path expressions of the form $o \uparrow .A_1.\cdots.A_n$. For instance, paths *P1*, *P2* and *P3* of Example 3.4.2 will be written as shown below:

$$P1 : st.address.college$$
$$P2 : se.isTaughtBy.rank$$
$$P3 : co.hasSections$$

- We will denote by $\mathbf{V}_\tau$ and $\mathbf{P}_\tau$ the set of all variables and persistent roots of type τ respectively. □

Wffs of language $\mathcal{L}_S$ have no side-effects. In other words, by using paths from **PE** we cannot change object states but only observe information stored in them. The language $\mathcal{L}_S$ lacks this feature because it is essentially designed for constraint specification. Recall from Chapter 2 that object states are changed through invocation of methods, called *mutators*, on object states. Mutators do that in turn through carrying out primitive modification operations on object states.

3.5 Modification Operations

The modification operations are those primitive operations of the underlying database system which make it possible to change the state of objects. There are three families of modification operations that can be used to change the internal states of objects. Each operation in one of these families is defined to modify variables, objects and persistent roots of a certain type. These families of operations are named assign, add, and remove. Operations of the *assign* family, are used to assign values to variables, object attributes and persistent roots. Operations of the *add* family are used to insert a value into set valued variables, attributes and persistent roots. Operations of the *remove* family are used to delete a value from set valued variables, attributes and persistent roots. In addition to this basic semantics, the operations are defined such that when a part of an object state can be observed through a path expression, then we can modify this part through the same path expression. The definition of modification operations are presented below.

Assign Operations

For each type τ, there is an assign operation, denoted $\longleftarrow_\tau$, and has the form

$$v \longleftarrow_\tau c$$

where $v \in (\mathbf{PE}_\tau \cup \mathbf{V}_\tau \cup \mathbf{P}_\tau)$ and c is a term of type τ.

Let I be the current state, c_{old}, c_{new} be terms of type τ and $I \models (v = c_{old})$. The operation $v \longleftarrow_\tau c_{new}$ changes the state I to a state I' such that $I' \models (v = c_{new})$.

Add Operations

For each type τ, there is an add operation, denoted $\stackrel{+}{\longleftarrow}_\tau$, and has the form

$$v \stackrel{+}{\longleftarrow}_\tau c$$

where $v \in (\mathbf{PE}_{\{\tau\}} \cup \mathbf{V}_{\{\tau\}} \cup \mathbf{P}_{\{\tau\}})$ and c is a term of type τ.

Let I be the current state, c_{old}, c_{new} be terms of type $\{\tau\}$ and τ respectively, and $I \models (v = c_{old})$. The operation $v \stackrel{+}{\longleftarrow}_\tau c_{new}$ changes the state I to a state I' such that $I' \models (v = c_{old} \cup \{c_{new}\})$.

Remove Operations

For each type τ, there is a remove operation, denoted $\stackrel{-}{\longleftarrow}_\tau$, and has the form

$$v \stackrel{-}{\longleftarrow}_\tau c$$

where $v \in (\mathbf{PE}_{\{\tau\}} \cup \mathbf{V}_{\{\tau\}} \cup \mathbf{P}_{\{\tau\}})$ and c is a term of type τ.

Let I be the current state, c_{old}, c_{new} be terms of type $\{\tau\}$ and τ respectively, and $I \models (v = c_{old})$. The operation $v \stackrel{-}{\longleftarrow}_\tau c_{new}$ changes the state I to a state I' such that $I' \models (v = c_{old} - \{c_{new}\})$.

In the remainder of the thesis we will use the term *update unit* to refer to any non-empty *sequence* of modification operations.

$pr : < \#93, [rank : \text{"professor"}, teaches : \{nil\}, \ldots] >$

$pr' : < \#95, [rank : \text{" "}, teaches : \{\#55\}, \ldots] >$

$st' : < \#25, [\ldots, address : [college : \text{"Informatik"}, roomNumber : \text{"D23"}], \ldots] >$

$se : < \#55, [\ldots, isTaughtBy : \#95, \ldots] >$

$co : < \#11, [\ldots, hasSections : \{\#55\}, \ldots] >$

Figure 3.2: Side affects to objects of Figure 3.1 by update unit of Example 3.5.1.

▶ **Example 3.5.1 (Update Unit)** Consider the update unit U given below.

$$\begin{aligned} U : \{&Professors \stackrel{+}{\longleftarrow} pr', \\ &pr'.teaches \longleftarrow pr.teaches, \\ &pr.teaches \longleftarrow \{nil\}, \\ &pr.rank \longleftarrow \text{" professor"}, \\ &se.isTaughtBy \longleftarrow pr', \\ &Students \stackrel{+}{\longleftarrow} st', \\ &st'.address \longleftarrow st.address, \\ &Students \stackrel{-}{\longleftarrow} se\}. \end{aligned}$$

The side affects of applying U to the UNIVERSITY database with objects shown in Figure 3.1, is presented in Figure 3.2. ■

The definition of modification operations as presented above does not take a basic principle of object orientation into account, namely encapsulation, that is only operations of objects have the right to alter the object state. This comes from the fact that through a certain sort of path expressions an object can alter the internal state of another object. Thus, to preserve encapsulation we restrict path expressions that may occur on the left hand side of modification operations to intra paths. The definition of intra paths is presented below.

Definition 3.5.2 (Intra Path) Let o be an object. A path expression $o.A_1.\ldots.A_n$ is an intra path if either $n = 1$ or $n > 1$ and none of the attributes A_i, $i \in [1, n[$ is a reference attribute. □

Intuitively, an intra path is a path expression by which an object o *cannot* reach the state of another object o' and hence none of o''s attributes can be accessed or modified from the object o. It follows from the definition that for each attribute A of object o, $o.A$ is an intra path. For example, paths *P1* and *P3* are intra paths, whereas *P2* is not. In *P2* the attribute *isTaughtBy* of *Section* class refers to an object of *Professor* class. Thus, through the path *P2*, we can modify the attribute *rank* of *Professor* objects from objects of *Section*. Consider, for instance the following update:

$$se.isTaughtBy.rank \longleftarrow \text{"associate"}$$

If the update statement shown above occurs in a method of the *Section* class then this method will break encapsulation.

To preserve the principle of encapsulation, we assume in this thesis that all paths that appear on the left hand side of modification operations are intra paths.

3.6 Inverse Relationships

Let c_A and c_B be classes participating in a relationship R. Let a and b be reference attributes of the relationship R in the class c_A and c_B, respectively. In our approach, the inverse of the relationship R is maintained by considering the attribute b an inverse attribute to a and vice versa. In this section first, we present the definition of inverse attributes. Then we show how the inverse relationships in our approach are maintained.

The definition that a and b are inverse attributes depends on the cardinality of R. Thus we have four cases corresponding to whether the cardinality of R is (1:1), (1:m), (m:n), or (m:1).

- If R is a (1:1) relationship, then for every object o_1 of type c_A and o_2 of type c_B:

$$o_1.a = nil \text{ or } o_1.a.b = o_1, \text{ and } o_2.b = nil \text{ or } o_2.b.a = o_2$$

- If R is a (1:m) relationship, then for every object o_1 of type c_A and o_2 of type c_B:

$$o_1.a = \emptyset \text{ or for every } o' \in o_1.a,\ o'.b = o_1, \text{ and}$$
$$o_2.b = nil \text{ or } o_2 \in o_2.b.a$$

- If R is a (m:n) relationship, then for every object o_1 of type c_A and o_2 of type c_B:

$$o_1.a = \emptyset \text{ or for every } o' \in o_1.a,\ o_1 \in o'.b, \text{ and}$$
$$o_2.b = \emptyset \text{ or for every } o' \in o_2.b,\ o_2 \in o'.a$$

- If R is a (m:1) relationship, then for every object o_1 of type c_A and o_2 of type c_B:

$$o_1.a = nil \text{ or } o_1 \in o_1.a.b, \text{ and}$$
$$o_2.b = \emptyset \text{ or for every } o' \in o_2.b,\ o_2 = o'.a$$

From the definition of inverse attributes given above, it follows that to maintain the inverse of relationships of cardinality (1:1), (1:m), (m:n) and (m:1) the following logical equivalences should be maintained.

(1) R is a (1:1) relationship:

$$(\forall o_1 : c_A)(\forall o_2 : c_B)(o_1.a = o_2 \leftrightarrow o_2.b = o_1). \tag{3.1}$$

(2) R is a (1:m) relationship:

$$(\forall o_1 : c_A)(\forall o_2 : c_B)(o_2 \in o_1.a \leftrightarrow o_2.b = o_1). \tag{3.2}$$

(3) R is a (m:n) relationship:

$$(\forall o_1 : c_A)(\forall o_2 : c_B)(o_2 \in o_1.a \leftrightarrow o_1 \in o_2.b). \tag{3.3}$$

(4) R is a (m:1) relationship:

$$(\forall o_1 : c_A)(\forall o_2 : c_B)(o_2 = o_1.a \leftrightarrow o_1 \in o_2.b). \tag{3.4}$$

Now we present a transformation, denoted by Υ, that maps every update unit into a safe one that maintains the equivalences (3.1) – (3.4). Let U be an update unit, Υ maps every update up in U as follows:

Case 1: If up modifies a reference attribute a to a relationship R of which b is an inverse attribute then Υ maps up into a sequence of updates.

Case 2: Otherwise, Υ is the identity mapping, i.e.; $\Upsilon(up) = up$.

If up is of the first case, then up matches the pattern $o_1.a\theta v$, where o_1 is an object of type c_A, θ is a modification operator; and v is value. Also, in this case, the definition of $\Upsilon(up)$ depends on the cardinality of the relationship R and thus we have four cases corresponding to whether the cardinality of R is (1:1), (1:m), (m:n) or (m:1).

R **is a (1:1) relationship.** In this case, θ is an assignment operator, $\longleftarrow$, and v is either an object o_2 of type c_B or *nil*. In this case, the equivalence (3.1) is maintained by the following definition of Υ:

$$\Upsilon(o_1.a \longleftarrow o_2) = \{o_1.a \longleftarrow o_2, o_2.b \longleftarrow o_1\} \tag{3.5}$$

$$\Upsilon(o_1.a \longleftarrow nil) = \{o_1.a \longleftarrow nil, o_2.b \longleftarrow nil\} \tag{3.6}$$

R **is a (1:m) relationship.** In this case, θ is either the add operator, $\overset{+}{\longleftarrow}$, the remove operator $\overset{-}{\longleftarrow}$, or the assignment operator $\longleftarrow$, and v is either an object o_2 of type c_B or a set of objects S (possibly empty) of type $\{c_B\}$. In this case, the equivalence (3.2) is maintained by the following definition of Υ, in which o is an object variable and S' is a variable of a set type:

$$\Upsilon(o_1.a \overset{+}{\longleftarrow} o_2) = \{o_1.a \overset{+}{\longleftarrow} o_2, o_2.b \longleftarrow o_1\} \tag{3.7}$$

$$\Upsilon(o_1.a \overset{-}{\longleftarrow} o_2) = \{o_1.a \overset{-}{\longleftarrow} o_2, o_2.b \longleftarrow nil\} \tag{3.8}$$

$$\Upsilon(o_1.a \longleftarrow \emptyset) = \left\{ \begin{array}{l} S' \longleftarrow o_1.a, \\ \underline{\texttt{for}}\ \underline{\texttt{each}}\ o \in S'\ \underline{\texttt{do}} \\ \qquad o.b \longleftarrow nil \\ \underline{\texttt{endfor}}, \\ o_1.a \longleftarrow \emptyset \end{array} \right\} \tag{3.9}$$

$$\Upsilon(o_1.a \longleftarrow S) = \left\{ \begin{array}{l} \Upsilon(o_1.a \longleftarrow \emptyset), \\ o_1.a \longleftarrow S \\ \underline{\texttt{for}}\ \underline{\texttt{each}}\ o \in S\ \underline{\texttt{do}} \\ \qquad o.b \longleftarrow o_1 \\ \underline{\texttt{endfor}} \end{array} \right\} \tag{3.10}$$

R **is a (n:m) relationship.** In this case, θ is either the add operator, $\overset{+}{\longleftarrow}$, the remove operator $\overset{-}{\longleftarrow}$, or the assignment operator $\longleftarrow$, and v is either an object o_2 of type c_B or a set of objects S (possibly empty) of type $\{c_B\}$. In this case, the equivalence (3.3) is maintained by the following definition of Υ, in which o is an object variable and S' is a variable of a set

type:

$$\Upsilon(o_1.a \overset{+}{\longleftarrow} o_2) = \{o_1.a \overset{+}{\longleftarrow} o_2, o_2.b \overset{+}{\longleftarrow} o_1\} \tag{3.11}$$

$$\Upsilon(o_1.a \overset{-}{\longleftarrow} o_2) = \{o_1.a \overset{-}{\longleftarrow} o_2, o_2.b \overset{-}{\longleftarrow} nil\} \tag{3.12}$$

$$\Upsilon(o_1.a \longleftarrow \emptyset) = \left\{ \begin{array}{l} S' \longleftarrow o_1.a, \\ \underline{\texttt{for}}\ \underline{\texttt{each}}\ o \in S'\ \underline{\texttt{do}} \\ \qquad o.b \overset{-}{\longleftarrow} o_1 \\ \underline{\texttt{endfor}}, \\ o_1.a \longleftarrow \emptyset \end{array} \right\} \tag{3.13}$$

$$\Upsilon(o_1.a \longleftarrow S) = \left\{ \begin{array}{l} \Upsilon(o_1.a \longleftarrow \emptyset), \\ o_1.a \longleftarrow S \\ \underline{\texttt{for}}\ \underline{\texttt{each}}\ o \in S\ \underline{\texttt{do}} \\ \qquad o.b \overset{+}{\longleftarrow} o_1 \\ \underline{\texttt{endfor}} \end{array} \right\} \tag{3.14}$$

R is a (m:1) relationship. In this case, θ is an assignment operator, $\longleftarrow$, and v is either an object o_2 of type c_B or *nil*. In this case, the equivalence (3.4) is maintained by the following definition of Υ:

$$\Upsilon(o_1.a \longleftarrow o_2) = \{o_1.a \longleftarrow o_2, o_2.b \overset{+}{\longleftarrow} o_1\} \tag{3.15}$$

$$\Upsilon(o_1.a \longleftarrow nil) = \{o_1.a \longleftarrow nil, o_2.b \overset{-}{\longleftarrow} o_1\} \tag{3.16}$$

In this thesis we assume that the inverse relationship is maintained by our object database model automatically. Thus, we assume that the semantics of modification operations, given in Section 3.5, is overridden by that of the transformation Υ. Examples of such a model are the one underlying the ObjectStore System [LLOW91] and the object model of ODMG 3.0 [CB00].

We conclude this section by giving an example that shows how inverse relationships are maintained by using the transformation Υ.

▶ **Example 3.6.1 (Inverse Relationships)** In the UNIVERSITY database the relationship between the *Professor* class and the *Section* class is (1:m). Reference attributes of that relationship are the attribute *teaches* in the *Professor* class of type $\{Section\}$, and the attribute *isTaughtBy* in the *Section* class of the type *Professor*. Thus, attributes *teach* and

isTaughtBy are inverse attributes. Thus, according to the definition of inverse attributes of (1:m) relationships, for every professor *pr* and section *se*:

$$pr.teaches = nil \text{ or for every } se' \in pr.teaches,\ se'.isTaughtBy = pr; \text{ and}$$
$$se.isTaughtBy = nil \text{ or } se \in se.isTaughtBy.teaches$$

Using the equivalence (3.2), this particular relationship can be formulated as follows:

$$(\forall pr : Professor)(\forall se : Section)(se \in pr.teaches \leftrightarrow pr = se.isTaughtBy). \tag{3.17}$$

Let U be an update unit. Let $up : pr1.teaches \xleftarrow{+} se1$ be an update in U. From the semantics of add operations, the update up adds a relationship that goes from *pr1* to *se1* but not the inverse relationship that goes from *se1* to *pr1*. As such, U violates (3.17) if there is no update in U of the form $se1.isTaughtBy \longleftarrow pr1$. This can be avoided if we replace up in U by the sequence $\Upsilon(up)$ defined below:

$$\Upsilon(up) = \{pr1.teaches \xleftarrow{+} se1, se1.isTaughtBy \longleftarrow pr1\}$$

Suppose that in U there is an update $up' : pr2.teaches \longleftarrow \emptyset$. Let $'S$ be the set of sections in *pr3.teaches* before carrying out up'. Carrying out up' removes relationships that go from *pr2* to sections in *pr2.teaches* but not their inverse relationships. Thus, to maintain (3.17), we first have to remove relationships that go from sections in S' to *pr2*. This can be done by using the transformation $\Upsilon(up)$ defined below:

$$\Upsilon(pr2.teaches \longleftarrow \emptyset) = \left\{ \begin{array}{l} S' \longleftarrow pr2.teaches, \\ \underline{\texttt{for}}\ \underline{\texttt{each}}\ se \in S'\ \underline{\texttt{do}} \\ \qquad se.isTaughtBy \longleftarrow nil \\ \underline{\texttt{endfor}}, \\ pr2.teaches \longleftarrow \emptyset \end{array} \right\}$$

Suppose that in U there is an update $up'' : pr3.teaches \longleftarrow S$, where S is a set of sections. Let S' be the set of sections in *pr3.teaches* before carrying out up''. Carrying out up'' (1) removes relationships that go from *pr3* to sections in S'; and (2) adds new relationships that go from *pr3* to sections of S. Carrying out up'' does not (3) remove relationships that go from sections in S' to *pr3*; nor (4) adds relationships that go from sections in S to *pr3*. Thus to maintain (3.17), we have to carry out updates of (1) through (4), in the following order, (1),(3),(2),(4). This can be done by using transformation $\Upsilon(up'')$ defined below, in which,

updates (1) and (3) in that order, are done by $\Upsilon(pr2.teaches \longleftarrow \emptyset)$.

$$\Upsilon(pr3.teaches \longleftarrow S) = \left\{ \begin{array}{l} \Upsilon(pr3.teaches \longleftarrow \emptyset), \\ pr3.teaches \longleftarrow S \\ \underline{\texttt{for}}\ \underline{\texttt{each}}\ se \in S\ \underline{\texttt{do}} \\ se.isTaughtBy \longleftarrow pr3 \\ \underline{\texttt{endfor}} \end{array} \right\}$$

Similarly, the relationship between *TA* class and *Section* class, which is (1:1), and the relationship between *Section* class and *Student* class, which is (m:n), can be specified by using the equivalences (3.1) and (3.3) respectively, as follows:

$$(\forall se : Section)(\forall ta : TA)(ta = se.assists \leftrightarrow se = ta.hasTA)$$
$$(\forall se : Section)(\forall st : Student)(se \in st.hasSections \leftrightarrow st \in se.isTakenBy)$$

■

3.7 Well-Formed Constraints

Essentially, a constraint is any closed formula of $\mathcal{L}_S$. However, such a definition has a serious drawback. Some closed wffs, when considered as constraints, have the undesirable property that their evaluation in two instances of a database may change, although instances of types mentioned in these wffs are not updated. Such a class of formulas causes many problems when they are considered as constraints. Most notably is that the naive approach to their evaluation is not acceptable as some quantified variables may run over an infinite domain. Moreover, other problems caused by these wffs with regard to integrity checking can be illustrated by a simple example as follows. Consider an instance I that satisfies the constraint

$$(\exists pr : Professor)(pr.age > 100). \tag{3.18}$$

Suppose that there is one and only one object, say pr, in $dom(Professor)$. Suppose that in the instance I object pr is persistent through its relationship with a persistent object se of type *Section*. Suppose that object se, in turn, is persistent by being attached to the persistent root *PrimarySections* of type $\{Section\}$. Suppose that I' is a database instance obtained from I by removing object se from *PrimarySections*. Therefore in the new state, I', the object pr is no longer reachable, that is, $pr \notin dom(Professor)$ and hence $I' \not\models W$. Thus, due to the principle of reachability, there is no way to associate such a modification to the constraint to determine whether the update might lead to an invalid state or not.

To avoid the above problems, wffs considered as acceptable constraints (and queries) are only wffs that restrict their own reference domains. Formulas with this property are called domain independent [Var81, Dem92]. A formula W is said be *domain independent* iff W satisfies the following property for any two instances I and I' such that I' is obtained from I by adding a new constant to one of the domains of types mentioned in W.

$$I \models W \iff I' \models W.$$

Intuitively, a domain-independent wff is a formula whose evaluation in a given database state does not depend on domains of types of variables occurring in it. For instance, the problem with formula (3.18) can be avoided if its domain independent version is used, in which objects of type *Professor* run over the persistent root *Professors* of type $\{Professor\}$:

$$(\exists pr : Professor)(pr \in Professors \wedge Pr.age > 100).$$

The class of domain independent formulas cannot be defined syntactically. It has been shown in [Var81] that the class of domain-independent formulas is undecidable. This means that there is no efficient way to decide whether a given wff is domain independent or not.

There have been several attempts to introduce syntactically defined subclasses of domain independent formulas [Nic82, Rei80] (also, see [Dem92, GMN84] for an overview). In the field of integrity constraints, Nicolas [Nic82] introduced the notion of range restricted formulas to characterize a subset of prenex conjunctive normal forms which are domain independent. There are several adaptations to range restricted formulas that cope with OODBs [JJ91, AHV95]. In the thesis, we use the one presented in [JJ91]. The definition is adapted to our constraint specification language $\mathcal{L}_S$ and the approach of object persistency by reachability.

Definition 3.7.1 (Range Restricted Constraints) A range restricted constraint is a closed wff matching one of the patterns:

$$\begin{array}{l} (\exists o_1 : \tau_1) \ldots (\exists o_n : \tau_n)(o_1\ \theta\ E \wedge L_1 \wedge \ldots L_m \wedge R) \\ (\forall o_1 : \tau_1) \ldots (\forall o_n : \tau_n)(o_1\ \theta\ E \wedge L_1 \wedge \ldots L_m \rightarrow R) \end{array} \tag{3.19}$$

where

- $\theta \in \{\in, =\}$, E is a persistent root and $L_1 \ldots L_m$ are positive literals.
- Each variable o_i, $i \in]1, n]$ occurs in a literal, L_k, $k \in [1, m]$; and every L_k matches one of the patterns $o_i\ \theta_i\ E_i$ or $o_i\ \theta_i\ o_j.p$, where $j \in [1, i[$, $\theta_i \in \{\in, =\}$, E_i is a persistent root, and p is a path expression.

- The subformulas of R are either quantifier-free wffs or again in one of formats (3.19).
- if one of the variables $o_1 \ldots o_n$ occurs in R then it is free.

The literals $L_1 \ldots L_m$ in patterns (3.19) are called range literals. Range literals are defined such that the range of each variable o is either restricted by a persistent root, $o \; \theta \; E$, or by object(s) that are reachable through path p from object(s) that restrict the range of another variable o', $o \; \theta \; o'.p$. In the definition we considered without loss of generality that the range of the first variable in pattern (3.19) is restricted by object(s) of a persistent root E.

3.8 Constraint Examples

There are several classifications proposed in the literature for integrity constraints. We have already classified constraints into state, transition, and dynamic constraints in the introduction to this chapter. Another often used classification is the one by which constraints are classified as inherent, implicit and explicit [EN94].

Inherent constraints are those imposed on databases by the data model. An example of this type is the constraint that every relation in the relational model must be in first normal form. In our object database model, an inherent constraint is the one that the type of classes must be a tuple type.

Implicit constraints are user-defined constraints for a database application that can be specified by using notions of the data model used. For example, the ISA relationship in OODBs can be seen as a constraint of this type. Another example is that objects of subclasses are disjoint subsets of objects of a superclass. As such, in our model, the constraint that "*an employee cannot be a teaching assistant and a professor at the same time*" is an implicit constraint.

Explicit constraints are user-defined constraints for a database application that cannot be specified directly by using notions of the used data model. As such, almost all work done in the literature concerns this type of constraints. The question of whether a user-defined constraint is an implicit or explicit constraint depends on the concepts offered by data models. For instance, in some semantic data models such as the one proposed in [BdBZ93], the constraint example just mentioned before is an explicit constraint whereas in our model it is an implicit one.

In the literature, there are several criteria by which constraints can further be classified, for example [GA93, Ger96b, Deß93, Tür99]. Two remarks can be stated about these criteria. First they are not standard. Criteria proposed by one researcher are different from those

proposed by the others. Second, classes of constraints that are defined according to these criteria are non-orthogonal; the same constraint can be in more than one class. As such, in the following we list a set of constraint classes that are stated in almost all the literature we aware of. Each of them has a particular name which shows that there is a consensus about them.

Domain Constraints. Domain constraints restricts values that may be added or assigned to an attribute. An example of a domain constraint is the one that states that "*every employee has one of the science degrees, Bachelor of Science, or Master of Science, or Doctor of Philosophy*":

$$(\forall em)(em \in \mathit{Employees} \rightarrow em.\mathit{degree} \in \{\text{MSc.}, \text{PhD}, \text{BSc.}\}). \tag{3.20}$$

Non Null Constraints. Non null constraints are similar to the domain constraints but they have an additional function. By using this type of constraints mandatory relationships between classes can be specified. For example, to let the relationship between the *Professor* class and the *Section* class be mandatory, the non null constraint example that says "*every professor teaches at least one section*" should be specified:

$$(\forall pr)(pr \in \mathit{Professors} \rightarrow pr.\mathit{teaches} \neq \mathit{nil}). \tag{3.21}$$

In most OODBSs such as the O_2, non null constraints are implicit constraints, i.e., they can be specified in the object schema.

Referential Constraints. Referential constraints restrict removing objects that are referenced by other objects to avoid the problem of dangling oids. An example is the constraint that states that "*every professor teaches a primary section*":

$$(\forall pr)(pr \in \mathit{Professors} \rightarrow (\exists se)(se \in pr.\mathit{teaches} \wedge se \in \mathit{PrimarySections})). \tag{3.22}$$

Key Constraints. In OODBs every object has a unique identifier. However, key constraints are still used for object identification as object identifiers are hidden from users. An example of a key constraint is the one that says "*teaching assistance can be identified by their names*":

$$(\forall ta1)(\forall ta2)(ta1 \in \mathit{TAs} \wedge ta2 \in \mathit{TAs} \rightarrow (ta1.\mathit{name} = ta2.\mathit{name} \rightarrow ta1 = ta2)). \tag{3.23}$$

Inclusion Constraints. An inclusion constraint restricts objects of one set to be subset of another set. It can be stated between attributes of the same class or two different classes; and between persistent roots of set type. An example of an inclusion constraint is the one that says, "*every visiting professor is a professor*":

$$(\forall pr)(pr \in \mathit{VisitingProfessors} \rightarrow pr \in \mathit{Professors}). \tag{3.24}$$

Intra-Object Constraints. Inter-object constraints restrict states of objects of the same type. An example is the constraint that says "*no course is prerequisite for itself*":

$$(\forall co)(co \in \mathit{Courses} \rightarrow co \notin co.\mathit{isPrerequisiteFor}). \tag{3.25}$$

Inter-Object Constraints. The class of inter-object constraints is the most general type of constraints. It restricts objects of interrelated set of classes. An example is the one that says: "*Every professor teaches a section such that the course of this section has not prerequisite courses*":

$$(\forall pr)(pr \in \mathit{Professors} \rightarrow (\exists se)(se \in pr.\mathit{teaches} \wedge se.\mathit{isSectionOf}.\mathit{hasPrerequisites} = \emptyset)). \tag{3.26}$$

We conclude this section by pointing out that all constraint examples mentioned above are sometimes referred to as non-aggregation constraints. An aggregation constraint is one that specifies a condition on objects by using one of the aggregations functions provided by the query language of the underlying DBMS. A well know set of aggregations functions are *count*, *average*, *sum*, *min* and *max*. Arguments to these aggregate functions are multisets of elements. An example of an aggregation constraint, in which the *sum* function is involved, is the constraint that says "*the total of salaries of employees in a project should not exceed the budget assigned to that project*". In this thesis we deal only with non-aggregation constraints. For more in-depth coverage and formal treatment of aggregation constraints we refer the reader to [RSSS98].

By the end of this chapter the first part of the thesis is completed. In the second part, Chapters 4 and 5, we present the constraint catalog.

Chapter 4

Constraint Catalog: The IC Class

It is essential for any consistency management approach to provide the feature of integrity independence [Cod90], that is, the ability to change constraint specifications without changing application programs and transactions. To incorporate this feature into our approach we must separate constraint specifications from transactions and class methods [OS98b, OCS99a]. For this, we use a meta-database called the *constraint catalog* [OCS00, OCS99b]. The constraint catalog is a central repository for all information about constraints of a database.

The information stored in the constraint catalog is divided into two categories. The first category is concerned with the canonical specification of constraints. This kind of information will be stored in a class named IC. The second category of information describes simplified forms of constraints. This type of information will be stored in a class named Shell.

In this chapter we present the structure of the class IC. The function of the class IC is to store as objects complete specifications of constraints that are defined for an object database. By complete specifications we mean that on one hand objects of the class IC can be seen as a documentation of the integrity constraints defined for an object database. On the other hand, by using this specification we can (1) maintain integrity of the object base, (2) determine constraints imposed on a given class(es) and/or paths and vice versa, and (3) if we want, ignore the presence of some constraints during integrity checking.

For every constraint, there is an object of the IC class. To facilitate the compilation of constraints into objects of the class IC, constraints are first transformed into a canonical form. The properties of the constraint canonical form provide us with grounds upon which we guarantee the correctness of the compilation.

This chapter is organized as follows. In Section 4.1, we present the definition of the

constraint canonical form. Sections 4.2 and 4.3 present some properties of the constraint canonical form by which we can determine from a given constraint, the classes, path expressions and persistent roots that are subject to that constraint. In Section 4.4, we present how user-specified constraints can be transformed into canonical ones. Finally, in Section 4.5 the structure of the IC class and the semantics of its attributes are presented.

4.1 Canonical Constraints

In the definition of a canonical constraints we refer to two sorts of path expressions. The first one has been presented in Definition 3.5.2, called *intra* path, and the second one is presented below, called *flat* path.

Definition 4.1.1 (Flat Path) A path expression $o.A_1.\dots.A_n$ is called a *flat* path if the attribute A_n, $n \geq 1$ is not of tuple type. □

Examples of flat paths are paths *P1*, *P2* and *P3* of Example 3.4.2. An example of a non-flat path is the path *em.dateOfBirth*, where *em* is an object of *Employee* class and *dateOfBirth* is an attribute of tuple type:

$$[day : integer, month : integer, year : integer].$$

We now present the definition of the constraint canonical form or for sake of simplicity "*canonical constraint*".

Definition 4.1.2 (Canonical Constraint) A *canonical constraint* is a closed range restricted wff W such that for each quantified object variable o in W and each path $o.A_1 \dots A_n$ that appears in W, $o.A_1 \dots A_n$ is an *intra and flat* path. □

Examples of canonical constraints are the constraint examples (3.20)–(3.25) of Section 3.8. First they are range restricted constraints. Second, all paths that occur in them are intra and flat paths. On the one hand, these paths are intra because each of them has exactly one attribute. On the other hand, these paths are flat because their attributes are not of a tuple type.

An example of a non canonical constraint is the constraint example (3.26) of Page 53. This constraint is range restricted but not canonical. It has the path expression

$$P : se.isSectionOf.hasPrerequisites$$

According to Definition 3.5.2 of intra paths, the path expression P is not an intra path. This is due to the presence of two reference attributes; the *isSectionOf* attribute which refers to an object of *Course* class and *hasPrerequisites* attribute which refers to a set of objects of *Course* class.

It is worth pointing out why we consider paths of canonical constraints to be intra and flat. The rationale behind that will be presented in Sections 4.2 and 4.3 respectively. In Section 4.4 we will present a method to transform a user specified constraint into a canonical one.

4.2 Constrained Classes

In Section 4.5 we show that among the information to be included in the result of compiling a constraint into an object of IC class are the names of classes that are subject to that constraint. We call these classes *constrained classes*. In this section, we first present a property by which we can determine all constrained classes of a constraint. Then we show by an example that without the assumption that every path that appears in a constraint is an intra path, this property is no longer valid.

► **Lemma 4.2.1** Let W be a canonical constraint. Let o be an object to be updated by an update unit U. If none of the classes mentioned in the quantifications of W or any of their subclasses is the type of o, then the evaluation of the constraint W remains the same as before the update unit U is executed. □

Proof of Lemma 4.2.1 follows from the definition of domain independent wffs and the semantics we adapted to inheritance as set inclusion, i.e., objects of a subclass are also objects of its superclasses.

Now we present why we considered all paths of canonical constraints are intra paths. In fact, in the constraint specification language $\mathcal{L}_S$, presented in Section 3.3, attributes are defined as function symbols and hence their compositions are considered as terms. As such, terms of $\mathcal{L}_S$ contain path expressions of the form

$$o.A_1 \dots A_n$$

such that more than one attribute in the suffix of the path is a reference attribute to some relationship, where $n \geq 2$. A serious shortcoming due to this fact is that some classes may be subject to a constraint however they are not mentioned explicitly in quantifications of the

constraint. This means that Lemma 4.2.1 would not be always valid if we would not assume that all paths of constraints are intra paths. The following example shows this fact.

▶ **Example 4.2.2 (Inter Path Expression)** Consider the constraint example *W1* which specifies that "*every professor teaches a section such that the course of that section has no prerequisite courses*". The formula (4.1) shown below is a range restricted logical specification of the constraint *W1*.

$$\begin{aligned} W1 : & (\forall pr : Professor)\ (pr \in Professors \rightarrow \\ & (\exists se : Section)(se \in pr.teaches \wedge se.isSectionOf.hasPrerequisites = \emptyset)). \end{aligned} \tag{4.1}$$

Classes stated explicitly in wff (4.1) are *Professor* and *Section*. Also none of them is a superclass of any class in the UNIVERSITY schema. If we apply Lemma 4.2.1 to the constraint *W1* but by considering specification (4.1) instead of canonical one, then we obtain the following result. Constraint *W1* is not affected by any update unit that updates objects type of which is neither the *Professor* class nor the *Section* class.

However the careful investigation of the types of path expressions of (4.1) shows that the constraint *W1* may also be affected if objects of the *Course* class are modified. This is due to the fact that the *Course* class is implicitly restricted by the constraint through the presence of the path expression $se.isSectionOf.hasPrerequisites$ in the specification (4.1). We will refer to such path expressions as *inter path expression.* ■

We conclude this section by pointing out that in Subsection 4.4.2 we present a method to transform an inter path into a set of intra paths.

4.3 Constrained Paths and Persistent Roots

Lemma 4.2.1 presents an interesting property of canonical constraints. Thereby we can exactly identify from quantifications of a constraint and the class hierarchy the constrained classes of that constraint. This can be used to avoid unnecessary integrity checking which is a right step towards efficient integrity maintenance. However, we need more steps in this direction. Lemmas 4.3.1–4.3.3, stated below, present criteria by which we can identify updates may occur to objects of constrained classes that do not violate constraints.

In comparison to relational databases, there are no default modify operations to object databases but user-defined methods, called mutators. If we consider the user-defined methods as causes for constraint violations then we lose the feature of integrity independence, which is one of our major targets, and hence we get all disadvantages of application-oriented

techniques as mentioned in the introductory chapter. To resolve these problems, we consider integrity at the lowest update level, namely at primitive modification operations presented in Section 3.5.

▶ **Lemma 4.3.1** Let W be a valid canonical constraint in prenex conjunctive normal form. Let o be an object of a class restricted by W to be updated by $up : o.A_1 \ldots A_n \overset{+}{\longleftarrow} v$. If the following conditions are satisfied, then W is valid after the update.

(1) None of the negative literals in W matches the pattern $\neg(v' \in o'.A_1 \ldots A_n)$.

(2) None of the positive literals in W matches the pattern $S \subseteq o'.A_1 \ldots A_n$.

(3) None of the literals in W matches the pattern $o'.A_1 \ldots A_n = S$. □

Proof:.
There are similar criteria to those of the lemma used in [MH89, Nic82] for relational schemata. The following theorem is proved in [MH89]:
"*Suppose that W is a range restricted constraint in prenex conjunctive normal form and consider an insert into (resp. delete from) a relation R. If W has no negated (resp. positive) occurrences of "R", and if W is valid before the insertion (resp. deletion) then W is valid after the insertion (resp. deletion).*"
We will exploit the theorem stated above to prove the lemma. Thus, we present an equivalent relational representation of constraints and modify operations. This can be done by translating persistent roots and attributes into unary and binary predicates respectively. Let $\rho(W)$ be the translated wff of the constraint W. $\rho(W)$ is obtained by applying three manipulations steps to literals in W. The first and second steps are transformations of literals of W, denoted as (TF1) and (TF2) below. The third step, denoted as (TF3), is the transformation of the resulting wff into $\rho(W)$. The descriptions of the three steps are presented below.

(TF1) Each literal $S \subseteq o'.A_1 \ldots A_n$ in W is rewritten as

$$(\forall x)(\neg(x \in S) \vee (x \in o'.A_1 \ldots A_n))$$

where x is a new variable.

(TF2) Each literal $o'.A_1 \ldots A_n = S$ in W is rewritten as

$$(\forall x)(\neg(x \in o'.A_1 \ldots A_n) \vee (x \in S) \wedge$$
$$(\forall y)(\neg(y \in S) \vee (y \in o'.A_1 \ldots A_n))$$

where x and y are new variables.

(TF3) Each literal $v' \in o'.A_1 \ldots A_n$ in W is translated into a conjunction

$$(\forall v_1) \ldots (\forall v_n) A_1(o, v_1) \ldots \wedge \ldots A_n(v_{n-1}, v')$$

where $v_1, \ldots v_{n-1}$ are new variables.

The translation of a path expression in (TF3) sometimes is called a strong translation [Día92]. The details of the translations of other terms is immaterial. In fact there are two alternatives for the translation other terms, the first one is taken for flatten nested relations, see Chapter 20 of [AHV95] and the other one is presented in [JJ91] for deductive databases. Here the translation of terms other than those mentioned above follows [AHV95].

Consider the constraint example that says that "*sections taught by every professor belong to the persistent root Sections*". The constraint can be formulated transformed by (TF1), and translated into a relational representation as follows:

$$\begin{aligned} W :& (\forall pr)(pr \in Professors \rightarrow pr.teaches \subseteq Sections) \\ \text{TF1}(W) :& (\forall pr)(pr \in Professors \rightarrow (\forall x)(\neg(x \in pr.teaches) \vee (x \in Sections))) \\ \rho(W) :& (\forall pr)(Professors(pr) \rightarrow (\forall x)(\neg teaches(pr, x) \vee Sections(x))) \end{aligned}$$

Now we prove the lemma. In what follows we assume that $\rho(W)$ is in prenex conjunctive normal form. Let o be an object to be updated by $up : o.A_1 \ldots A_n \overset{+}{\longleftarrow} v$. The translation of up, denoted by $\rho(up)$ can be seen as a set of relational modification operations, namely the *Insert* operation:

$$\{Insert(A_1, < o, v_1 >), \ldots, Insert(A_n, < v_{n-1}, v_n >)\}$$

where the values $v_1, \ldots v_n$ are defined as follows:

$$\begin{aligned} v_1 &= o.A_1 \\ &\vdots \\ v_n &= v_{n-1}.A_n \end{aligned}$$

If W satisfies the conditions of the lemma, then according to steps (TF1), (TF2) and (TF3), wff $\rho(W)$ has no negated occurrences of predicates $A_1, \ldots, A_n$. Since up consists only of insert operations, it follows from the theorem that if $\rho(W)$ is valid before update $\rho(up)$ then $\rho(W)$ remains valid after the update $\rho(up)$ and hence the lemma is valid. ■

Similarly we can prove the following Lemma:

▶ **Lemma 4.3.2** Let W be a valid canonical constraint in prenex conjunctive normal form. Let o be an object of a class restricted by W to be updated by $up : o.A_1 \ldots A_n \overset{-}{\longleftarrow} v$. If the following conditions are satisfied then, W is valid after the update.

(1) None of the positive literals in W matches the pattern $(v' \in o'.A_1 \dots A_n)$.

(2) None of the negative literals in W matches the pattern $\neg(S \subset o'.A_1 \dots A_n)$.

(3) None of the literals in W matches the pattern $o'.A_1 \dots A_n = S$. □

In Lemma 4.3.1 and Lemma 4.3.2, we considered the modify operations add and remove respectively to examine when they do not violating constraints. Thus it remains to examine the assign operation. This is presented in Lemma 4.3.3.

▶ **Lemma 4.3.3** Let W be a valid canonical constraint in prenex conjunctive normal form. Let o be an object of a class restricted by W to be updated by $up : o.A_1 \dots A_n \longleftarrow v$. If $A_1 \dots A_n$ is not a subpath of any of the paths mentioned in W, then W is valid after the update. □

Proof:.
We recall that we restricted the application of modification operations only to a subset of path expressions, namely intra paths. Also, by definition of a canonical constraint, all paths in W are intra and flat paths. Thus none of the paths mentioned in W is a subpath of $A_1 \dots A_n$. Now if $A_1 \dots A_n$ is not a subpath of any of the paths mentioned in W, then all conditions of Lemma 4.3.1 and 4.3.2 are satisfied. Also, from the semantics of $\longleftarrow$, $\rho(up)$ can be seen as a sequence of relational delete operations followed by a sequence of relational insert operations. As such, the lemma follows using the same arguments presented in the proof of Lemma 4.3.1. ■

It is worth stressing that without assuming that all paths in canonical constraint specification are flat, Lemma 4.3.3 is no longer valid. This point is illustrated in the following example.

▶ **Example 4.3.4 (Non-Flat Path)** Consider the following constraint example which restricts the range of employees' date of birth.

$$\begin{aligned} \mathit{W2} :& (\forall em)(\forall d)(\forall m)(\forall y)(em \in \mathit{Employees} \wedge em.\mathit{dateOfBirth} = [d,m,y] \rightarrow \\ & [d,m] \notin \{[30,2][31,2]\} \wedge 1 \leq d \leq 31 \wedge 1 \leq m \leq 12 \wedge 1900 \leq y \leq 2000) \end{aligned} \tag{4.2}$$

Let em' be an object in the extension $\mathit{Employees}$. Suppose that one of the following updates

$$em'.\mathit{dateOfBirth}.\mathit{day} \leftarrow dd, \text{ or}$$
$$em'.\mathit{dateOfBirth}.\mathit{month} \leftarrow mm, \text{or}$$
$$em'.\mathit{dateOfBirth}.\mathit{year} \leftarrow yy$$

occurs in an update unit U. The type of *dateOfBirth* has three components labeled by *day*, *month*, and *year*. As such, any update that occurs to one of the these components results in an update to *dateOfBirth*. Therefore constraint *W2* may be violated by the update U. However, none of the paths

$$dateOfBirth.day$$
$$dateOfBirth.month$$
$$dateOfBirth.year$$

is a subpath of *dateOfBirth*. This means that there are some kinds of updates that may violate *W2* but cannot be captured by the criteria of Lemma 4.3.3. This is due to the fact that components *day*, *month*, and *year* are implicitly restricted by the constraint through the presence of the path expression *se.dateOfBirth* in wff (4.2). We will refer to this sort of path expression as *non-flat path expression.*

The problems of non-flat paths of constraint specification (4.2) mentioned above can be avoided if the constraint *W2* is specified as follows:

$$\begin{aligned} W2 :& (\forall em)(\forall d)(\forall m)(\forall y)(em \in Employees \wedge em.dateOfBirth.day = d \wedge \\ & em.dateOfBirth.month = m \wedge em.dateOfBirth.year = y \rightarrow \\ & [d, m] \notin \{[30, 2][31, 2]\} \wedge 1 \leq d \leq 31 \wedge 1 \leq m \leq 12 \wedge 1900 \leq y \leq 2000). \end{aligned} \tag{4.3}$$

■

The following lemma shows that similar criteria like those presented in Lemma 4.3.1–4.3.3 are also valid for persistent roots. We recall that a persistent root may be a named object of a set of objects.

▶ **Lemma 4.3.5** Let W be a valid canonical constraint in prenex conjunctive normal form. Let P be a persistent root of type c, where c is a constrained class of W. If P is to be updated by $up : P \overset{+}{\longleftarrow} v$ and the following conditions are satisfied, then W is valid after the update up.

(1) None of the negative literals in W matches the pattern $\neg(v' \in P)$.

(2) None of the positive literals in W matches the pattern $S \subseteq P$.

(3) None of the literals in W matches the pattern $P = S$. □

▶ **Lemma 4.3.6** Let W be a valid canonical constraint in prenex conjunctive normal form. Let P be a persistent root of type c, where c is a constrained class of W. If P is to be updated by $up : P \overset{-}{\longleftarrow} v$ and the following conditions are satisfied, then W is valid after the update up.

(1) None of the positive literals in W matches the pattern $(v \in P)$.

(2) None of the negative literals in W matches the pattern $\neg(P \subseteq S)$.

(3) None of the literals in W matches the pattern $P = S$. □

▶ **Lemma 4.3.7** Let W be a valid canonical constraint in prenex conjunctive normal form. Let P be a persistent root of type c, where c is a constrained class of W. If P is to be updated by $up : P.A_1 \ldots A_n \longleftarrow v$ and $P.A_1 \ldots A_n$ is not a subpath of any of the paths mentioned in W, then W is valid after the update up. □

A consequence of lemmas 4.3.1–4.3.3 and lemmas 4.3.5–4.3.7 is that, given an update unit U, a constraint is potentially violated by U if there is at least an update $up \in U$ that does not satisfy at least one of the conditions of the lemmas.

4.4 Transformation into Canonical Constraints

We now present a method to transform a given constraint W into a canonical constraint. The method contains two steps. In the first step, non-flat path expressions are flattened. Then, in the obtained wff, inter path expressions are removed. In the sequel we present the first and second steps respectively.

4.4.1 Path-Flattened

In this subsection, we present a transformation, denoted as Φ, that maps each wff into an equivalent one in which all paths are flat. First, we define Φ for literals and then for wffs.

The definition of canonical constraints does not impose conditions on path expressions the prefix of which are non-object variables. As such, the transformation Φ becomes the identity map for literals in which all path expressions are of this sort. So it remains to define Φ for literals in which there is a path expression $o.A_1 \ldots A_n$ such that o is an object variable.

Let ℓ be a literal in which $o.A_1 \ldots A_n$ occurs. Thus ℓ matches one of the following patterns.

$$o.A_1 \ldots A_n = t \text{ or } t = o.A_1 \ldots A_n \tag{4.4}$$

$$o.A_1 \ldots A_n \in t \tag{4.5}$$

$$t \in o.A_1 \ldots A_n \tag{4.6}$$

$$t_1 \; \theta \; o.A_1 \ldots A_n \text{ or } o.A_1 \ldots A_n \; \theta \; t_2 \tag{4.7}$$

where t, t_1 and t_2 are terms and θ is one of the operators $\{<, \leq, >, \geq, \subseteq\}$.

(1) If ℓ matches one of the patterns (4.4) or (4.5) and A_n is an attribute of an atomic type or of a set type, then $o.A_1 \ldots A_n$ is a flat path and thus there is nothing to do: $\Phi(\ell) = \ell$.

(2) If ℓ matches one of the patterns in (4.4) and A_n is an attribute of a tuple type $[k_1 : \tau_1, \ldots, k_m : \tau_m]$, then
$\Phi(\ell) = \Phi(o.A_1 \ldots A_n.k_1 = t.k_1) \wedge \ldots \wedge \Phi(o.A_1 \ldots A_n.k_m = t.k_m)$.

(3) If ℓ matches the pattern (4.5) and A_n is an attribute of a tuple type $[k_1 : \tau_1, \ldots, k_m : \tau_m]$ then $\Phi(\ell) = (\exists u)\Phi(o.A_1 \ldots A_n = u) \wedge (u \in t)$, where u is a new variable.

(4) If ℓ matches one of the patterns (4.6) or (4.7), then all paths occurring in ℓ are flat and hence there is nothing to do: $\Phi(\ell) = \ell$

Now we extend the definition of the transformation Φ to wffs as follows. Let U_1 and U_2 be wffs, $\psi \in \{\wedge, \vee, \rightarrow\}$, and $Q \in \{\forall, \exists\}$. Then

- $\Phi(U_1 \; \psi \; U_2) = \Phi(U_1) \; \psi \; \Phi(U_2)$
- $\Phi(\neg U) = \neg\Phi(U)$
- $\Phi((Qo)M) = (Qo)\Phi(M)$

▶ **Example 4.4.1 (Path-Flattened)** Consider the following part of the *Professor* class:

class *Professor*[
  ⋮ ...
  institution : *Institution*,
  degrees : {*Degree*},
  ⋮ ...].

The attribute *institution* describes the institution at which a professor holds a professorship chair and *degree* decribes details of a professor's degrees. The types of these attributes are below. Attributes in these types are self-explanatory:

type *Institution* [
  name : *string*,
  address : *Address*]

type *Address* [
  country : *string*,
  city : *string*]

type *Degree* [
  name : *string*,
  year : *integer*,
  institution : *Institution*]

Consider the following example constraint which specifies that "*a professor cannot be promoted to a professor in the institution from which he/she rewarded the Habilitation degree*"

$$\begin{aligned} W3 : & (\forall pr)(\forall u)(pr \in Professors \wedge u \in pr.degrees \wedge u.name = \text{"Habilitation"} \rightarrow \\ & \neg(u.institution = pr.institution)) \end{aligned}$$

The path expression $pr.institution$ is not a flat path, the attribute *institution* is of a tuple type and *pr* is an object variable of the type *Professor*. As such, the specification of the constraint, *W3*, above is not canonical. The application of the tranformation Φ to *W3* is shown below.

$$\begin{aligned} \Phi(W3) = \; & (\forall pr)(\forall u)\Phi(pr \in Professors \wedge u \in pr.degrees \wedge \\ & u.name = \text{"Habilitation"} \rightarrow \neg(u.institution = pr.institution)) \\ = \; & (\forall pr)(\forall u)\Phi(pr \in Professors) \wedge \Phi(u \in pr.degrees) \wedge \\ & \Phi(u.name = \text{"Habilitation"}) \rightarrow \Phi(\neg(u.institution = pr.institution)) \\ = \; & (\forall pr)(\forall u)(pr \in Professors) \wedge (u \in pr.degrees) \wedge \\ & (u.name = \text{"Habilitation"}) \rightarrow \neg\Phi(u.institution = pr.institution) \\ = \; & (\forall pr)(\forall u)(pr \in Professors) \wedge (u \in pr.degrees) \wedge \\ & (u.name = \text{"Habilitation"}) \rightarrow \\ & \neg(\Phi(u.institution.name = pr.institution.name) \wedge \\ & \Phi(u.institution.address = pr.institution.address)) \end{aligned}$$

$$\begin{aligned} = \ & (\forall pr)(\forall u)(pr \in \mathit{Professors}) \wedge (u \in pr.degrees) \wedge \\ & (u.name = \text{"Habilitation"}) \rightarrow \\ & \neg(u.institution.name = pr.institution.name \wedge \\ & \Phi(u.institution.address.country = pr.institution.address.country) \wedge \\ & \Phi(u.institution.address.city = pr.institution.address.city)) \end{aligned}$$

$$\begin{aligned} = \ & (\forall pr)(\forall u)(pr \in \mathit{Professors}) \wedge (u \in pr.degrees) \wedge \\ & (u.name = \text{"Habilitation"}) \rightarrow \\ & \neg(u.institution.name = pr.institution.name \wedge \\ & u.institution.address.country = pr.institution.address.country \wedge \\ & u.institution.address.city = pr.institution.address.city) \end{aligned}$$

■

4.4.2 Removing Inter-Paths

Let W be a constraint in which all paths are flat paths. Let ℓ be a literal in W, in which the path $o.A_1 \ldots A_n$ appears. From the definition of range restricted constraints, Definition 3.7.1, W can be considered as follows:

$$\mathbf{L}\ \psi_1\ (Qo)(Qo_1)\ldots(Qo_q)\ \ell \wedge \ell_1 \wedge \ldots \ell_p\ \psi_2\ \mathbf{R} \tag{4.8}$$

where $\mathbf{L}$ and $\mathbf{R}$ are wffs that match one of patterns (3.19), $Q \in \{\forall, \exists\}$ and $\psi_1, \psi_2 \in \{\wedge, \rightarrow\}$.

To remove inter path expressions from W, we apply the following steps. If $o.a_1 \ldots a_n$ is an inter path in a literal ℓ of W then:

(1) Replace the constraint (4.8) with the following wff:

$$\mathbf{L}\ \psi_1\ (Qo)(Qo_1)\ldots(Qo_q)(\forall o')\ o' = o.a_1 \ldots a_m \rightarrow \ell\gamma \wedge \ell_1\gamma \wedge \ldots \ell_p\gamma\ \psi_2\ \mathbf{R}\gamma \tag{4.9}$$

where $o.a_1 \ldots a_m$ is the *longest* proper subpath of $o.a_1 \ldots a_n$ such that a_m is a reference attribute to a relationship, o' is a new object variable that does not occur in W, and γ is the substitution $\{o.a_1 \ldots a_m / o'\}$.

(2) If no inter paths in (4.9) exist, then wff (4.9) is the canonical specification of W. Otherwise, we apply the same step (1) but this time we use the wff (4.9) instead of the constraint W.

To prove the correctness of the transformation steps given above, it is sufficient to show that formulas (4.8) and (4.9) are equivalent. This can be shown by using the following equivalence:

$$(Qo)F[o] \equiv (Qo)(\forall o)(o = o' \rightarrow F[o]\{o/o'\})$$

where, $Q \in \{\forall, \exists\}$, $F[o]$ is wff in which o is a free variable, and the variable o' does not occur in F.

We conclude this section by presenting an example to transform a user specified constraint into a canonical constraint specification.

▶ **Example 4.4.2 (Removing Inter Path)** Consider the user specified constraint (4.1):

$$\begin{aligned} \mathit{W1} :& (\forall pr : \mathit{Professor})\ (pr \in \mathit{Professors} \rightarrow (\exists se : \mathit{Section})(se \in pr.\mathit{teaches} \wedge \\ & se.\mathit{isSectionOf}.\mathit{hasPrerequisites} = \emptyset)). \end{aligned}$$

Constraint *W1* is a range restricted wff, but not a canonical constraint. This is due to the presence of the inter path $se.\mathit{isSectionOf}.\mathit{hasPrerequisites}$. To remove this inter path from *W1*, we apply the transformation steps given above. The attribute *isSectionOf* is a reference attribute to the relationship between the *Section* class and the *Course* class. As such, $se.\mathit{isSectionOf}$ is the longest proper subpath of $se.\mathit{isSectionOf}.\mathit{hasPrerequisites}$ that satisfies the condition of step (1) of the transformation. Therefore, the substitution γ is $\{se.\mathit{isSectionOf}/co\}$ where co is a new variable of the type *Course*. According to step (1), *W1* is transformed to the following wff:

$$\begin{aligned} \mathit{W1} :& (\forall pr : \mathit{Professor})\ (pr \in \mathit{Professors} \rightarrow (\exists se : \mathit{Section})(se \in pr.\mathit{teaches} \wedge \\ & (\forall co : \mathit{Course})(co = se.\mathit{isSectionOf} \rightarrow co.\mathit{hasPrerequisites} = \emptyset))). \end{aligned} \tag{4.10}$$

According to step (2), the constraint (4.10) is a canonical one since all paths are intra paths.

■

4.5 Structure of the IC Class

The IC class is one of the two classes of the constraint catalog. The structure of IC class is shown in Figure (4.1). An object of IC describes a *complete specification* of an integrity constraint defined for the database. By complete specification, we mean that by using this specification we can (1) maintain integrity of the object base, (2) determine constraints

$$
\textbf{Class } \mathsf{IC} \; [\; \begin{array}{l} name: \; string, \\ wff: string, \\ status: string, \\ mode: string, \\ classes: \{string\}, \\ paths\&roots: \{string\}, \\ shells: \; \{\mathsf{Shell}\} \;] \end{array}
$$

Figure 4.1: Structure of the IC Class.

imposed on a given class(es) and/or paths and vise versa, and (3) if we want, ignore the presence of some constraints during integrity checking.

For each constraint W, there is exactly one object, denoted $IC(W)$, of the IC class. The object $IC(W)$ persists as long as it is attached to a persistent root named ICs of a set type {IC}; namely the extension of the IC class. We assume that newly instantiated IC objects are implicitly added to the extension ICs. Also, we do not allowed explicit manipulations of the extension ICs, e.g., adding an object to ICs, and removing an object from ICs. We will discuss the topic of constraints manipulation in detail in Section 7.10.

In the following, we describe the information the object $IC(W)$ contains about the constraint W. Henceforth, we use the term *IC object* to refer to an object of the IC class. Also, as a running example, we will use the constraint $W1$ of Example 4.2.2 in its canonical form, wff (4.10). The state of the IC object $IC(W1)$ is shown in Figure 4.2 on Page 69.

4.5.1 Constraint Name

For tasks like constraint maintaining, querying, and manipulating, there must be a way to identify constraints. Therefore, each constraint must have a unique name. The attribute *name* of type *string* describes the name of a constraint. A constraint name should be meaningful. For instance, instead of naming a constraint on the range of employees age as W it will be more helpful to name it as `EmployeeAgeRange`. This is due to the fact that the latter name coded useful information about what is the subject matter of the constraint. However, for sake of simplicity, we will consider the value of the attribute *name* for the IC object $IC(W)$ of the constraint W as `W`.

name	*wff*	*status*	*mode*	*classes*	*paths&roots*	*Shells*
`W1`	Wff: (4.10)	*enabled*	*deferred*	{*Professor*, *Section*, *Course*}	{+*Professors*, −*teaches*, *isSectionOf*, +*hasPrerequisites*, −*hasPrerequisites*, *hasPrerequisites*}	{#50, #51, #52}
`W2`	wff: (4.3)	*enabled*	*deferred*	{*Employee*, *TA*, *Professor*}	{+*Employees*, *dateOfBirth.day*, *dateOfBirth.month*, *dateOfBirth.year*}	{#55}

Figure 4.2: States of IC objects *IC*(*W1*) and *IC*(*W2*).

4.5.2 Constraint Canonical Specification

The canonical specification of constraints is an essential requirement for facilitating the process of compiling constraints into objects of the IC class. As we will show, all information described by the other attributes of the IC object $IC(W)$ can be derived from the canonical specification of W. As such, the criteria upon which we can establish our judgment of whether information stored in $IC(W)$ is correct, is the canonical specification of W. In addition, canonical constraints are added to the information that is stored in the constraint catalog for the actual evaluation of the constraint.

The attribute *wff* of type *string* describes the canonical specification of a constraint. For instance, the value for the attribute *wff* of the IC object $IC(W1)$ is the canonical specification of the constraint $W1$, that is, wff (4.10).

4.5.3 Constraint Status

By the status of a constraint we mean whether or not this constraint is applied to objects of constrained classes of that constraint. If a constraint is applied, then every object of a constrained class of that constraint must obey the constraint. In this case, the status of the constraint is said be *enabled*. Otherwise, the constraint is considered as if it would not exist. In this case, the status of the constraint is said to be *disabled*. To provide this capability to users, the status of a constraint is added to the structure of IC class.

The attribute *status* of type *string* describes the status of a constraint. The initial value of *status* for each newly created IC object is *enabled*.

4.5.4 Constraint Mode

The mode of a constraint specifies at which point within an update unit, *enabled* constraints, that may be potentially violated by this update unit, should be checked. There are two standard modes for constraints, that are almost used by all constraint management approaches we are aware of, particularly those which are based on ECA rules. The first mode, called *immediate*, corresponds to constraints where checking should be done immediately after an update that might violate them. Typical examples of constraints of this category are domain constraints which restrict values of an attribute to be a specific set of values or to be non null such as:

$$(\forall pr : Professor)(pr \in Professors \rightarrow pr.rank \in \{\text{"full"}, \text{"associate"}, \text{"assistant"}\})$$
$$(\forall st : Student)(st \in Students \rightarrow st.takes \neq \emptyset)$$

The second mode, called *deferred*, corresponds to constraints where checking should be deferred until the update unit commits. A typical example of constraints of this category are key constraints. The attribute *mode* of type *string* describes the mode of the constraint.

In our opinion, a constraint mode should not be determined from the syntax of the constraint alone, but the semantics of update units should be taken into account. Sometimes deferred checking of domain constraints can be better for some update units than other. Consider a simple attribute that is modified several times during the course of an update unit. All modifications but the last one will not be recorded. As such, an immediate check of a domain constraint imposed on this attribute is useless for all modifications but the last one. It has been shown that deferred checking is more appropriate for engineering databases such as CAD databases than immediate checking [Laf82], which is one of our target database applications. For that reason, we assume that all constraints have the deferred mode, unless users specify explicitly the immediate mode. For the IC object $IC(W1)$, the value of *mode* is *deferred*. In Section 7.2, we show that immediate checking of deferred constraints can also be provided.

4.5.5 Constrained Classes

A constraint restricts possible objects of one or more classes to admissible ones. Thus, to exploit results stated by Lemma 4.2.1 in integrity maintenance, names of constrained classes must be considered as part of the constraint specification. Constrained classes of a constraint can be divided into two categories. The first category contains classes that are mentioned explicitly in the quantification of the canonical constraint. The second category contains all subclasses of classes of the first category. The correctness of considering these classes as one means of constraint violations is proved in Lemma 4.2.1.

The attribute *classes* of set type $\{string\}$ contains names of classes that are subject to the constraint. A rewriting of Lemma 4.2.1 in terms of attributes of IC class is shown below, in which $type(o)$ denotes the type of object o:

$$\mathbf{RC} = \{ic.wff \mid type(o) \in ic.classes \wedge ic \in \mathsf{ICs}\} \tag{4.11}$$

If $\mathbf{RC} = \emptyset$ then the evaluation of constraints defined for the database remains the same as before the update unit takes place. We call the set **RC** the set of *relevant constraints* for the update unit U.

For the IC object $IC(W1)$, the value of attribute *classes* are all classes that appear in the quantification of the canonical specification of $W1$, wff (4.10) and its subclasses.

Classes that appear in the quantification of wff (4.10) are *Professor*, *Section*, and *Course*. From the UNIVERSITY schema, none of these classes has a subclass. Thus, the value for the attribute *classes* is the set $\{Professor, Section, Course\}$. For constraint (4.3), the value of $IC(W2).classes$ is $\{Employee, TA, Professor\}$ containing all subclasses of *Employee*. The state of IC object $IC(W2)$ is shown in Figure 4.2 on Page 69.

4.5.6 Constrained Paths and Persistent Roots

Criterion (4.11) alone is not sufficient for efficient integrity maintenance. There are some constraints in **RC** that cannot be violated by certain assignments. Lemmas 4.3.1 through 4.3.7 stated criteria by which we can detect whether a modify operation may violate constraints. Given an update unit, these criteria can be used to filter constraints that might be violated by that update unit from those which cannot be violated. Thus to exploit these results in integrity maintenance by using the constraint catalog, an attribute of the IC class must be added to describe constrained paths and persistent roots. This attribute is named *paths&roots* and is of set type $\{string\}$. These criteria are different – one for each modify operator. Therefore, if we store these constrained paths as they are appearing in the constraint, then constraint filtration will be inefficient, particularly if that filtration process is carried out at run time. As such, we adapt a definition, given below, for constrained paths such that the criteria of Lemmas 4.3.1 through 4.3.7 become one unified criterion.

Definition 4.5.1 (Constrained Paths) Let W be a canonical constraint. Let $Pcn(W)$ denotes the prenex conjunctive normal form of W. The set of *constrained paths and persistent roots* of constraint W, denoted **CPR**, is defined as follows.

Paths: For each path $o.A_1 \ldots A_n$ occurring in W, there is a subset w of **CPR**. The elements of w are strings defined as follows:

- If A_n is a simple or reference attribute then $w = \{A_1 \ldots A_n\}$.
- If A_n is of a set type and $o.A_1 \ldots A_n$ occurs in a negative literal of $Pcn(W)$ of the form $\neg(v' \in o'.A_1 \ldots A_n)$, then $w = \{+A_1 \ldots A_n\}$.
- If A_n is of a set type and $o.A_1 \ldots A_n$ occurs in a positive literal of $Pcn(W)$ of the form $(S \subseteq o'.A_1 \ldots A_n)$, then $w = \{-A_1 \ldots A_n\}$.
- If A_n is of a set type and $o.A_1 \ldots A_n$ occurs in a positive literal of $Pcn(W)$ of the form $(v' \in o'.A_1 \ldots A_n)$, then $w = \{-A_1 \ldots A_n\}$.

- If A_n is of a set type and $o.A_1 \ldots A_n$ occurs in a negative literal of $Pcn(W)$ of the form $\neg(S \subseteq o'.A_1 \ldots A_n)$, then $w = \{+A_1 \ldots A_n\}$.
- If A_n is of a set type and $o.A_1 \ldots A_n$ occurs in a positive or a negative literal of $Pcn(W)$ of the form $(S = o'.A_1 \ldots A_n)$, then $w = \{A_1 \ldots A_n, +A_1 \ldots A_n, -A_1 \ldots A_n\}$.

Persistent Roots: For each persistent root P occurring in W there is a subset w of **CPR**. The elements of w are strings defined as follows:

- If P is of a set type and P occurs in negative literal of $Pcn(W)$ of the form $\neg(o \in P)$, then $w = \{+P\}$.
- If P is of a set type and P occurs in a positive literal of $Pcn(W)$ of the form $(S \subseteq aP)$, then $w = \{-P\}$.
- If P is of a set type and P occurs in a positive e literal of $Pcn(W)$ of the form $\neg(o \in P)$, then $w = \{-P\}$.
- If P is of a set type and P occurs in a negative literal of $Pcn(W)$ of the form $(S \subseteq aP)$, then $w = \{+P\}$.
- If P is of a set type and P occurs in a positive or a negative literal of $Pcn(W)$ of the form $(S = P)$, then $w = \{P, +P, -P\}$. □

For example, the value of $IC(W1).paths\&roots$ is shown below.

$$IC(W1).paths\&roots = \{+Professors, -teaches, isSectionOf, +hasPrerequisites, hasPrerequisites, -hasPrerequisites\}$$

where the prenex conjunctive normal of canonical specification of $W1$ is the following wff.

$$\begin{aligned} Pcn(W1) : & (\forall pr : Professor)(\exists se : Section)(\forall co : Course) \\ & (\neg pr \in Professors \vee se \in pr.teaches) \wedge \\ & (\neg pr \in Professors \vee \neg co = se.isSectionOf \vee co.hasPrerequisites = \emptyset). \end{aligned}$$

Now we can rewrite Lemmas 4.3.1–4.3.7 in terms of attributes of IC class as shown below: Let

$$\mathbf{VC} = \{ic.wff \mid ic \in \mathsf{ICs} \wedge type(o) \in ic.classes \wedge (\exists cp)(cp \in ic.paths\&roots \wedge up' \sqsubseteq cp)\} \tag{4.12}$$

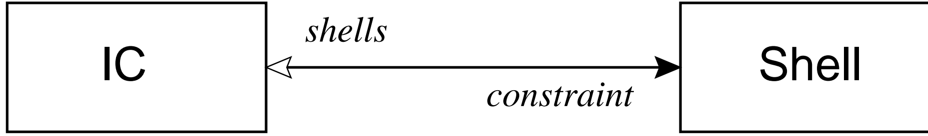

Figure 4.3: Relationship between the two classes of the constraint catalog.

if the evaluation of the query (4.12) is an empty set, then the update $up : o.A_1 \ldots A_n\ \theta\ v$, $\theta \in \{\longleftarrow, \overset{+}{\longleftarrow}, \overset{-}{\longleftarrow}\}$ does not violate database integrity, where the symbol $\sqsubseteq$ denotes the substring operator and up' is a string defined according to operator θ in the update up as shown below:

$$up' = \begin{cases} A_1 \ldots A_n, & \text{if } \theta \text{ is } \longleftarrow \\ +A_1 \ldots A_n, & \text{if } \theta \text{ is } \overset{+}{\longleftarrow} \\ -A_1 \ldots A_n & \text{if } \theta \text{ is } \overset{-}{\longleftarrow} \end{cases}$$

We call the set **VC** the set of potentially *violated constraints* by the update *up*.

The reader can easily prove that the set (4.12) is a subset of (4.11). This in turn shows that the set (4.12) fits the requirements for efficient integrity maintenance better than the set (4.11).

4.5.7 Constraint Shells

There is of course a trade-off in using canonical constraints instead of their original forms. This trade-off is between stating explicitly classes that are subject to the constraint, and hence introducing new quantified variables, and the evaluation of the constraint. The direct evaluation of general forms is generally inefficient.

For a given constraint, there may be one or more simplified forms. In Chapter 5, we present how these simplified forms can be obtained and then compiled into objects of the second class of the constraint catalog, the Shell class. Objects of the Shell class describe all information concerning simplified forms of constraints. The description of the structure of the Shell class will be presented in Chapter 5.

Essentially, in our approach, all constraints defined for a database are compiled into objects of the IC class and all simplified forms of these constraints are compiled into objects of the Shell class. Thus, there is a relationship between the IC class and the Shell class. Since

for a given constraint there may be more than one simplified form, this relationship is (1:m) from the IC class to the Shell class, see Figure 4.3. Reference attributes to this relationship are:

- an attribute named *Shells* in the IC class of type {Shell}; and
- an attribute named *constraint* in the Shell class of type IC.

For the IC object *IC(W1)*, the value of *Shells* attribute is a set of three objects of the Shell class, each of them corresponds to a compilation of a simplified form of *W1* into an object of the Shell class. Details of the internal state of objects in *IC(W1).shells* will be presented in Chapter 5.

We conclude this chapter by pointing out that by its end we complete the description of the first class of the constraint catalog, namely, the IC class. So to complete the description of the constraint catalog, it remains to present its second class, namely the Shell class. This will be the topic of the next chapter in addition to introducing our method for optimizing integrity constraints evaluation.

Chapter 5

Constraint Catalog: The Shell Class

The cornerstone for any constraint management approach is the efficiency of integrity checking. By using the criteria stated in (4.12), we can detect constraints that are potentially violated by a given update unit. However, constraints of the set **VC** are general statements expressed as closed wff without any attempt to their optimization. Moreover, constraints of **VC** are canonical ones and hence we expect that they contain more literals than their original users specification. This is due to removal of inter paths and flatness of non flat paths. As such, evaluation of constraints in **VC** may access a considerable portion of the object database. This makes the process of integrity checking by using objects of the IC class directly inefficient.

In this chapter, we first present an optimization technique. The goal is to enhance constraint evaluation. For a given constraint, the method derives a set of simplified forms. The method is based on the incremental evaluation of constraints. As such, simplified forms in most cases are much easier to evaluate than original constraints in the sense that the evaluation cost (in time) of the simplified forms is less than that of original constraints assuming direct evaluation [OCS99b]. We will prove that it is sufficient to check simplified forms to ensure consistency of an object database.

Second, a class named Shell is added to the constraint catalog. The goal is to store simplified forms in the constraint catalog. We present how simplified forms are compiled into objects of the Shell class. The properties of objects of the Shell class we present in this chapter will be the grounds upon which a method of consistency maintenance by using shells is proposed in Chapter 6.

This chapter is organized as follows. In Section 5.1, we present a motivating example of simplified forms. To facilitate the derivation of simplified forms from constraints, constraints

first are transformed into boolean expressions that match a certain pattern. In Section 5.2, we present the definition of a constraint boolean form and describe how a canonical constraint can be transformed into boolean form. In Section 5.3, the optimization technique is presented. Then, in Section 5.4 some observations concerning the optimization method are discussed. Finally, in Section 5.5 the semantics of the Shell class is presented.

5.1 Simplified Form: Motivating Example

Consider the constraint *W1* of Example 4.2.2. The canonical specification of *W1* is stated in (4.10) as follows:

$$\begin{aligned} \mathit{W1} :& (\forall pr : \mathit{Professor})\ (pr \in \mathit{Professors} \rightarrow \\ & (\exists se : \mathit{Section})(se \in pr.teaches \wedge \\ & (\forall co : \mathit{Course})(co = se.isSectionOf \rightarrow co.hasPrerequisites = \emptyset))). \end{aligned}$$

Suppose that the constraint *W1* is a valid constraint in the current state of the object database. Therefore for each object *pr* in the persistent root *Professors*, the specialization of *W1* to the object *pr*, denoted $(\mathit{W1}|pr)$, is a valid wff:

$$\begin{aligned} (\mathit{W1}|pr) :& (pr \in \mathit{Professors} \rightarrow \\ & (\exists se : \mathit{Section})(se \in pr.teaches \wedge \\ & (\forall co : \mathit{Course})(co = se.isSectionOf \rightarrow co.hasPrerequisites = \emptyset))). \end{aligned}$$

Let se' be an object of *Section* class. If $se'.isTaughtBy \in \mathit{Professors}$, i.e., the professor teaching se' is already attached to the *Professors* root, then the formula obtained by replacing the object variable *pr* in $(\mathit{W1}|pr)$ with $se'.isTaughtBy$, denoted $(\mathit{W1}|pr)[pr/se'.isTaughtBy]$, is a valid wff. Let

$$\begin{aligned} \mathit{V1} :& (\forall se' : \mathit{Section})(\mathit{W1}|pr)[pr/se'.isTaughtBy] := \\ & (\forall se' : \mathit{Section})(se'.isTaughtBy \in \mathit{Professors} \rightarrow \\ & (\exists se : \mathit{Section})(se \in se'.isTaughtBy.teaches \wedge \\ & (\forall co : \mathit{Course})(co = se.isSectionOf \rightarrow co.hasPrerequisites = \emptyset))). \end{aligned}$$

Therefore

$$\mathit{W1} \Rightarrow (\forall se' : \mathit{Section})(se'.isTaughtBy \in \mathit{Professors} \rightarrow (\mathit{W1}|pr)[pr/se'.isTaughtBy]) \equiv \mathit{V1}$$

Unfortunately, *V1* does not imply *W1*. It may happen that *V1* is valid but *W1* is not. A typical example is the case in which there exists an object *pr* in *Professors* persistent root such that $pr.teaches = nil$. However, if *W1* is a valid constraint in the current state S of the object base and an object se' of *Section* class is modified, then in the new state, S', *W1* is valid if and only if $(V1|se')$ is also valid:

$$S' \models W1 \text{ iff } S' \models (V1|se') \tag{5.1}$$

where $(V1|se')$ is the specialization of *V1* to the object se':

$$\begin{aligned}(V1|se') :&(se'.isTaughtBy \in Professors \rightarrow \\ &(\exists se : Section)(se \in se'.isTaughtBy.teaches \wedge \\ &(\forall co : Course)(co = se.isSectionOf \rightarrow co.hasPrerequisites = \emptyset))).\end{aligned}$$

W1 implies *V1* and therefore to prove (5.1) it is sufficient to show that if $(V1|se')$ is valid then so is *W1*. Let $(V1|se')$ be a valid formula. Let *pr* be an object in *Professors* persistent root. Then we have two cases:

- The first case is the one in which $se' \notin pr.teaches$. In this case, the state of the object *pr* is the same in S and S'. Thus, in this case, $(W1|pr)$ is valid in S'.

- The second case is the one in which $se' \in pr.teaches$. By the assumption that the inverse relationship is automatically maintained by the underlying OODBS, we have $pr = se'.isTaughtBy$. In this case, the state of *pr* object in S is not the same as in S'. However, as $(V1|se')$ is valid in S' and $pr = se'.isTaughtBy$, then $(W1|pr) := (W1|se)[se'.isTaughtBy/pr]$ and hence $(W1|pr)$ is valid in S', too.

Since *pr* was arbitrary, then for each $pr \in Professors$, $(W1|pr)$ is valid in S'.

Now suppose that the set of prerequisite courses of an object co' of *Course* class are updated. In this case constraint *W1* is potentially violated. To check *W1*, we need not to check $(W1|pr)$ for each *pr* of *Professor* class, but only those that have links with object co' through objects of *Section* class. These objects of *Professor* class can be defined as follows:

$$\{pr \mid pr = se'.isTaughtby \wedge se' \in co'.hasSections\}. \tag{5.2}$$

From (5.1) and (5.2), we can check $(V1|se')$ for each se' in $co'.hasSections$ instead of checking $(W1|pr)$ for each *pr* defined as in (5.2). Let *V2* be the conjunction of $(V1|se')$, for each

course co' and for each section se' in $co'.hasSection$. Let $(V2|co')$ be the specialization of $V2$ to the object co':

$$V2 : (\forall co')(\forall se')(se' \in co'.hasSections \rightarrow (V1|se')).$$
$$(V2|co') : (\forall se')(se' \in co'.hasSections \rightarrow (V1|se')).$$

The above discussion shows that if $W1$ is a valid constraint in a database state S and an object co' of the *Course* class is modified, then in the new state S' $W1$ is valid if and only if $(V1|co')$ is valid too:

$$S' \models W1 \text{ iff } S' \models (V2|co'). \tag{5.3}$$

It is worth mentioning that in the constraint $W1$, objects of *Section* class are existentially quantified objects, and objects of the *Course* class are universally quantified objects but bounded by an existentially quantified one. As such the classical approach of optimizing $W1$ via specialization fails to obtain simplified form for objects of *Section* and *Course* classes. Therefore, in the case of modifying an object of the *Section* (resp. *Course*) class, we are forced to evaluate the whole constraint $W1$. However, as we have shown above, by exploiting semantics of relationships among interrelated objects and their inverse relationships, it is sufficient to evaluate $(V1|se)$ (resp. $(V2|co)$) and, of course, evaluating $(V1|se)$ (resp. $(V2|co)$) is more efficient than that of $W1$.

In Section 5.3, we present a method to derive a set of simplified forms from a given constraint that have the same properties such as those presented above. To simplify steps of this method, we first transform constraints into boolean forms and then we apply the steps of the method. In the following section, we first present the definition of boolean forms and then how the transformation of constraints into boolean forms can be done.

5.2 Constraint Boolean Form

It is convenient to write wffs as boolean expression by making variables in the quantification run over set valued persistent roots and/or path expressions. This enhances the readability of the constraint and also facilitates the derivation of their simplified forms. Now we present the definition of a constraint boolean form, or for sake of simplicity "*boolean constraint*". The reader will notice that the definition is a syntactic reformulation of Definition 4.1.2.

Definition 5.2.1 (Boolean Constraint) A range restricted constraint in a boolean form is a boolean expression matching one of the patterns:

$$\begin{aligned}&(\exists o_1 \in E_1)\ldots(\exists o_n \in E_n)(L_1 \wedge \ldots L_m \wedge R)\\&(\forall o_1 \in E_1)\ldots(\forall o_n \in E_n)(L_1 \wedge \ldots L_m \rightarrow R)\end{aligned} \tag{5.4}$$

where

(1) E_1 is a set valued persistent root or an expression of the form $\{P\}$, where P is a single valued persistent root.

(2) Each E_i, $i \in]1, n]$ either satisfies the condition (1) or is a set valued expression of the form $(o_j.p)$, $j < i$.

(3) $L_1 \ldots L_m$ are positive literals.

(4) The subformulas of R are either quantifier-free wffs or again in one of the patterns given in (5.4).

(5) If one of the variables $o_1 \ldots o_n$ occurs in R then it is free. □

Two issues concerning constraint boolean forms should be considered, namely existence and uniqueness. In the existence issue we address the question of whether or not there exists a boolean form for every constraint. In the uniqueness issue we address the question of if there are two boolean forms for the same constraint whether these forms are always identical. In the following, we consider the existence of boolean forms and in Subsection 5.4.4 we shall consider the issue of uniqueness.

For each constraint, there exists an equivalent constraint in boolean form. This form can be achieved by using the following syntactic abbreviations.

$$(\forall o)(o \in A \rightarrow F[o]) \equiv (\forall o \in A)F[o]. \tag{5.5}$$

$$(\exists o)(o \in A \wedge F[o]) \equiv (\exists o \in A)F[o]. \tag{5.6}$$

$$(\forall o)(o = A \rightarrow F[o]) \equiv (\forall o \in \{A\})F[o]. \tag{5.7}$$

$$(\exists o)(o = A \wedge F[o]) \equiv (\exists o \in \{A\})F[o]. \tag{5.8}$$

▶ **Example 5.2.2 (Boolean Form)** Consider the constraint *W1* of Example 4.2.2. The canonical specification of *W1* is stated in (4.10) as follows:

$$\begin{aligned}W1 :&(\forall pr : Professor)\ (pr \in Professors \rightarrow\\&(\exists se : Section)(se \in pr.teaches \wedge\\&(\forall co : Course)(co = se.isSectionOf \rightarrow co.hasPrerequisites = \emptyset))).\end{aligned}$$

The boolean form of the constraint *W1* is derived by using equivalences (5.7),(5.6), and (5.5) as follows:

$$
\begin{aligned}
W1 \equiv\ & (\forall pr : Professor)\ (pr \in Professors \rightarrow \\
& (\exists se : Section)(se \in pr.teaches \wedge \\
& (\forall co \in \{se.isSectionOf\})(co.hasPrerequisites = \emptyset))). \\
\\
\equiv\ & (\forall pr : Professor)\ (pr \in Professors \rightarrow \\
& (\exists se \in pr.teaches)(\forall co \in \{se.isSectionOf\}) \\
& (co.hasPrerequisites = \emptyset))). \\
\\
\equiv\ & (\forall pr \in Professors)(\exists se \in pr.teaches) \\
& (\forall co \in \{se.isSectionOf\})(co.hasPrerequisites = \emptyset).
\end{aligned} \tag{5.9}
$$

■

Henceforth, we assume, unless stated otherwise, that all constraints are canonical constraints in boolean form.

5.3 Simplified Forms

In this section, we present a method to optimize evaluation of constraints. For every quantification in a constraint, the method derives a simplified form of that constraint. Simplified forms have the same properties as those presented in Section 5.1. The method is based on the incremental evaluation of constraints. As such, simplified forms in most cases are much easier to evaluate than original constraints.

Let W be a constraint. The simplified form of W w.r.t. the quantification $Q_i o_i \in E_i$ is denoted by $(W|i)$. The simplified form $(W|i)$ is derived according to whether the quantification $(Q_i o_i \in E_i)$ can be the outermost quantification of W or not. We first present, in Subsection 5.3.1, the derivation of simplified forms for quantifications that can be the outermost of a constraint. Then, in Subsection 5.3.2, we present the derivation of simplified forms for quantifications that cannot be the outermost. Before doing that, first a remark should be made concerning when a quantification of a constraint can be the outermost one.

▶ ***Remark 5.3.1*** The quantification $(Q_i o_i \in E_i)$ in a constraint W can be the outermost one if the following conditions are satisfied:

- E_i is a set valued persistent root P or a set valued expression $\{P\}$ and P is a named object, and
- Q_i is a universal (resp. existential) quantifier unbounded by an existential (resp. universal) quantifier. Formally, for every Q_j $j < i$, Q_j is the same as Q_i. □

Consider the constraints shown below in which $M[pr, se]$ and $M'[pr, se]$ denote open wffs of free object variables pr and se. The reader can easily verify, through using the equivalences (5.5)–(5.8) that for every quantifier $Q \in \{\forall, \exists\}$, the second quantification of the formula (5.10) cannot be the outermost one, whereas for wff (5.11) this can be. In fact, in constraint (5.10), having the free variable pr, the path expression $pr.teaches$ cannot be evaluated without the variable pr being first bounded to an object.

$$(Q\ pr \in Professors)(Q\ se \in pr.teaches)\ M[pr, se]. \tag{5.10}$$

$$(Q\ se \in Sections)(Q\ pr \in Professors)\ M'[pr, se]. \tag{5.11}$$

We now present steps of the method where simplified forms corresponding to outermost quantifications are derived.

5.3.1 Outermost Quantification

Let $(Q_i o_i \in E_i)$ be the outermost quantification of the constraint W or a quantification of W that can be the outermost one. We can assume that W has the following pattern:

$$W :\ (Q_i o_i \in E_i) F[o_i]$$

where $F[o_i]$ is an open formula in which the variable o_i is the only free variable. Then, according to the definition of range-restricted constraints, Definition 5.4, E_i is a set value persistent root or an expression of the form $\{P\}$, where P is a single value persistent root. We have two cases:

Case 1: Q_i is the existential quantifier $\exists$. In this case the simplified form $(W|i)$ is the same as the constraint W.

Case 2: Q_i is the universal quantifier $\forall$. In this case $(W|i)$ is the open wff:

$$(W|i): \ (o_i \in E_i \rightarrow F[o_i]). \tag{5.12}$$

To complete the steps of the method, we have to show how simplified forms corresponding to non outermost quantifications are derived. This will be presented in the following subsection.

5.3.2 Non Outermost Quantification

Let $(Q_i o_i \in E_i)$ be one of the quantifications of W that cannot be the outermost one of W. We have three cases:

Case 1: E_i is a set valued persistent root. In this case, the simplified form $(W|i)$ is the same as the constraint W.

Case 2: E_i is a path expression of the form $o_j.A$ or $\{o_j.A\}$, and $(W|j)$ is the constraint W. In this case $(W|i)$ is the same as the constraint W.

Case 3: E_i is a path expression of the form $o_j.A$ or $\{o_j.A\}$ and $(W|j)$ is not the constraint W. Let A' be the inverse attribute of A. We have two subcases:

(1) If $o_i.A'$ is a single valued expression, then $(W|i)$ is the open wff

$$(W|i): \ (W|j)[o_j/o_i.A']. \tag{5.13}$$

(2) If $o_i.A'$ is a set valued expression, then $(W|i)$ is the open wff

$$(W|i): \ (\forall o_j \in o_i.A') \ (W|j)[o_j]. \tag{5.14}$$

In Case 2 and Case 3, $(W|j)$ is the simplified form of W w.r.t. quantification $(Q_j o_j \in E_j)$ and $(W|j)[o_j/o_i.A']$ is the wff obtained by substituting each occurrence of a free variable o_j in $(W|j)$ with $o_i.A'$.

▶ **Example 5.3.2 (Simplified Forms)** Consider the constraint *W1* of Example 4.2.2. The boolean form of *W1* is stated in Example 5.2.2 as follows:

$$\begin{aligned} W1 :& (\forall pr \in Professors)(\exists se \in pr.teaches) \\ & (\forall co \in \{se.isSectionOf\})(co.hasPrerequisites = \emptyset). \end{aligned}$$

There are three quantifiers in $W1$, stated below, that bind the object variables pr, se and co, respectively:

$$(\forall pr \in Professors) \tag{5.15}$$

$$(\exists se \in pr.teaches) \tag{5.16}$$

$$(\forall co \in \{se.isSectionOf\}) \tag{5.17}$$

As such there are three simplified forms denoted, according to the notation used in the method, as $(W1|pr)$, $(W1|se)$ and $(W1|co)$. The simplified form $(W1|pr)$ corresponds to the outermost quantifier (5.15). Thus, according to (Case 2) of Subsection 5.3.1, $(W1|pr)$ is as follows:

$$\begin{aligned}(W1|pr)\ :&(pr \in Professors \rightarrow (\exists se \in pr.teaches)\\ &(\forall co \in \{se.isSectionOf\})(co.hasPrerequisites = \emptyset)).\end{aligned} \tag{5.18}$$

The quantification (5.16) cannot be the outermost quantification of $W1$. The simplified form $(W1|pr)$ is not the constraint $W1$. The inverse of attribute $teaches$ is the attribute $isTaughtBy$, and $se.isTaughtBy$ is single valued expression. Thus, the simplified form $(W1|se)$ is derived according to the first subcase of (Case 3) of Subsection 5.3.2:

$$(W1|se) : (W1|pr)[pr/se.isTaughtBy]. \tag{5.19}$$

The quantification (5.17) cannot be the outermost quantification of $W1$. The simplified form $(W1|se)$ is not the constraint $W1$. The inverse of attribute $isSectionOf$ is the attribute $hasSections$, and $co.hasSections$ is a set valued expression. Thus, the simplified form $(W1|co)$ is derived according to the second subcase of (Case 3) of Subsection 5.3.2:

$$(W1|co) : (\forall se \in co.hasSections)(W1|se). \tag{5.20}$$

The reader can notice that, up to the renaming of variables se' to se and co' to co, the wffs (5.18),(5.19), and (5.20) are the boolean forms of wffs $(W1|pr)$, $(V1|se')$ and $(V2|co')$ respectively, defined in Section 5.1. ■

The correctness of the simplified forms is stated in the following lemma.

▶ **Lemma 5.3.3** Let W be a valid constraint in the current state S of the object base. Let $(Q_i o_i \in E_i)$ be a quantification in W and the object variable o_i is of type c. If an object o of class c is modified, then in the new state S' the constraint W remains valid if and only if $(W|i)[o_i/o]$ is valid

$$S' \models W \text{ iff } S' \models (W|i)[o_i/o].$$

□

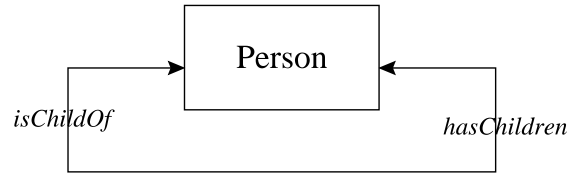

Figure 5.1: Recursive relationship of the *Person* class.

Proof:.
According to the simplification method, if the simplified form $(W|i)$ is not the constraint itself then it matches one of the patterns (5.12), (5.13), or (5.14). If $(W|i)$ matches the pattern (5.12), then the lemma follows from the fact that in this case $(W|i)$ is a specialization of the constraint W. If $(W|i)$ matches the pattern (5.13) or (5.14), then the lemma follows by using the same arguments as those presented in Section 5.1 to prove claims (5.1) and (5.3).

■

5.4 Observations

In this section, we present four observations concerning the derivation of simplified forms. In the first observation we discuss why we derive simplified forms on the object level rather than class level. The second observation is about existential constraints. In the other two observations, we discuss some issues that should be taken into account when we derive simplified forms of two special types of constraints.

5.4.1 Object Roles

Simplified forms are derived on the object level and not on the class level. Each simplified form corresponds to one and only one quantified object variable in the constraint. A constraint may have more than one object variable of the same type. Thus, on the class level, a class c may have more than one simplified form, $(W|1), \ldots, (W|n)$ for the same constraint W. Each $(W|i), i \in [1, n]$ corresponds to a *role* an object of the class c plays in a relationship restricted by the constraint W. A typical example of this sort of constraint is a constraint imposed on a recursive relationship. Consider the following class definition for class *Person*.

$$Person[name : string, age : integer, hasChildren : \{Person\}, isChildOf : \{Person\}]$$

In the class *Person* a recursive (n:m) relationship is implemented through the attributes *hasChildren* and *isChildOf* of set type $\{Person\}$. Each object participating in this relationship plays two roles: one as a child and the other one as a parent. Now consider the constraint example which states that "*parents should always be older than their children*".

$$W5 : (\forall pa \in Persons)(\forall ch \in Persons)(ch \in pa.hasChildren \rightarrow ch.age < pa.age) \quad (5.21)$$

Intuitively, to maintain *W5* we have to check every time when a person's age is modified that

(1) if the person plays the role of a parent then his/her children are younger than him/her; and

(2) if the person plays the role of a child then his/her parents are older than him/her.

This means that we have to check the following wffs:

$$(W5|pa) : (\forall ch \in Persons)(ch \in pa.hasChildren \rightarrow ch.age < pa.age). \quad (5.22)$$

$$(W5|ch) : (\forall pa \in Persons)(ch \in pa.hasChildren \rightarrow ch.age < pa.age). \quad (5.23)$$

If we would have derived a simplified form on the class level, then we would consider only one of the simplified forms above but not both. Hence the maintenance of constraint *W5* in this case would be incomplete.

This shows that we should consider the derivation of simplified forms of constraints on the object level as we have done in the simplification method and not on class level, otherwise integrity maintenance would be incomplete.

5.4.2 Existential Constraints

In the second case of non outermost quantifiers, the simplified form $(W|i)$ is the constraint W itself. This case corresponds to existential constraints, that is, constraints in which the first quantifier is the existential quantifier '$\exists$'. Our method as all simplification methods that are based on specialization technique, fails to obtain simplified forms for existential constraints. For that reason, if W is an existential constraint we consider $(W|i)$ is the constraint W itself.

5.4.3 Key Constraints

Considering the derivation of simplified forms of constraints on the object level has one drawback that can easily be avoided. It may happen that two simplified forms of a constraint are identical up to the permutation of the disjunctions, a permutation of literals and renaming of variables. In this case, the two simplified forms are equivalent and therefore it is sufficient to consider only one of them. A typical example, is the key constraint. Consider the following constraint that restricts names of teaching assistant to be unique:

$$W6 : (\forall ta1 \in TAs)(\forall ta2 \in TAs)(ta1.name = ta2.name \rightarrow ta1 = ta2)).$$

According to the simplification method, there are two simplified forms the constraint *W6*:

$$(W6|ta1) : (\forall ta2 \in TAs)(ta1.name = ta2.name \rightarrow ta1 = ta2).$$
$$(W6|ta2) : (\forall ta1 \in TAs)(ta1.name = ta2.name \rightarrow ta1 = ta2).$$

The reader can easily verify that $(W6|ta1)$ and $(W6|ta2)$ are identical up to applying the substitution $\{ta2/x, ta1/y\}$ to $(W6|ta1)$ and the substitution $\{ta2/y, ta1/x\}$ to $(W6|ta2)$; and using of the commutativity of equality symbol '=':

$$(W6|y) : (\forall x \in TAs)(y.name = x.name \rightarrow y = x).$$

This shows that for maintenance of key constraints it is sufficient to consider only one of its simplified forms. It is worth mentioning that this problem has been addressed before for relational data model in the steps of the Nicolas method for improving integrity checking and our solution is nothing but an adaptation of the solution proposed in [Nic82].

5.4.4 Uniqueness of Boolean Forms

The transformation of a constraint into a boolean form is not unique. For a given constraint there may be more than one boolean form. This is due to the fact that an object variable may be restricted by a *conjunction* of range literals. Consider the constraint stated below, which is a slight modification of the constraint *W1* from Section 5.1:

$$\begin{aligned} W7 : (\forall pr : Professor)\ & (pr \in Professors \rightarrow \\ & (\exists se : Section)(se \in pr.teaches \wedge se \in Sections \wedge \\ & (\forall co : Course)(co = se.isSectionOf \rightarrow co.hasPrerequisites = \emptyset)). \end{aligned} \tag{5.24}$$

In the constraint (5.24) range literals referring to the variable se are $se \in pr.teaches$ and $se \in Sections$. As such there are two possible boolean forms for that constraint:

$$\begin{aligned} W7 : (\forall pr \in Professors)(\exists se \in pr.teaches)(se \in Sections \wedge \\ (\forall co \in \{se.isSectionOf\})(co.hasPrerequisites = \emptyset)). \end{aligned} \tag{5.25}$$

$$\begin{aligned} W7 : (\forall pr \in Professors)(\exists se \in Sections)(se \in pr.teaches \wedge \\ (\forall co \in \{se.isSectionOf\})(co.hasPrerequisites = \emptyset)). \end{aligned} \tag{5.26}$$

As we have seen in the simplification method, the quantifier structure of a constraint determines simplified forms of that constraint. Now the question is if there is more than one boolean form for a constraint, what is the criterion upon which we select the boolean form of that constraint that best fit the simplification method. We have two criteria.

(1) Let o_j be a universally (resp. existentially) quantified variable that is bounded by an existentially (resp. universally) quantified variable o_i in the canonical constraint W. If there is more than one range literal for the object variable o_i, then in the transformation of W into a boolean form select the range literal which represent a relationship between o_i and o_j, otherwise select any range literal.

(2) Let o_i be a universally quantified variable not bounded by an existentially quantified one in the canonical constraint W. If there is more than one range literal for the object variable o_i, then in the transformation of W into a boolean form select the range literal which is a named value to be the range of o_i in the transformed boolean form, otherwise select any range literal.

The following two examples show the usage of the two criteria above on two different example constraints.

▶ Example 5.4.1 (Selection of Boolean Forms: Criterion (1))
Consider the constraint $W7$, stated above. If we select the boolean form (5.26), then according to (Case 1) of Subsection 5.3.2, $(W7|se)$ is the constraint $W7$ itself. Thus no optimization has been achieved. But if we select the boolean form (5.25), as criterion (1) above suggests, then, according to the first subcase of (Case 3) of Subsection 5.3.2, $(W7|se)$ is as follows:

$$(W7|pr)[pr/se.isTaughtBy]. \tag{5.27}$$

Of course the evaluation of (5.27) is more efficient than $W7$. ■

▶ **Example 5.4.2 (Selection of Boolean Forms: Criterion (2))**
Consider the constraint *W5* given in Subsection 5.4.1. *W5* has two boolean forms, one is given in (5.21) and the other is as follows:

$$W5 : (\forall pa \in Persons)(\forall ch \in pa.hasChildren)(ch \in Persons \rightarrow ch.age < pa.age). \quad (5.28)$$

If we select the boolean form (5.28), then according to the second subcase of (Case 3) of Subsection 5.3.2 $(W5|ch)$ is as follows

$$(\forall pr \in ch.isChildOf)(W5|pr). \quad (5.29)$$

But if we select the boolean form (5.21), as criterion (2) above suggests, then according to (Case 1) of Subsection 5.3.1, $(W7|se)$ is the simplified form (5.23). Again the evaluation of (5.23) is more efficient than (5.27) since (5.23) is a specialization of *W5* to one object, *ch*, and not to set of objects, *ch.isChildFor* as in (5.27). ■

5.5 Structure of the Shell Class

The second class of the constraint catalog, in addition to the IC class, is the Shell class. The structure of the Shell class is shown in Figure 5.2. Objects of the Shell class store information that is necessary for efficient consistency maintenance, essentially information concerning simplified forms of integrity constraints that are defined for the database.

Class Shell [*constraint* : IC,
form : *string*,
class : {*string*},
paths&roots : {*string*},
range : *string*,
objects : *string*,
shell : Shell]

Figure 5.2: Structure of the Shell class.

For every simplified form $(W|i)$ of a constraint W, there is an object, denoted $S(W|i)$, of the Shell class called the *shell* of the constraint W w.r.t. object variable o_i. The object $S(W|i)$ is a persistent object as long as it is attached to a persistent root named Shells of set type

{Shell}; namely the extension of Shell class. We assume that every newly instantiated Shell object is implicitly added to the extension Shells. Also, we do not allow explicit manipulations of the extension Shells, e.g., adding an object into Shells or removing an object from Shells. We will discuss the topic of constraint manipulation in detail in Section 7.10.

In the sequel we assume that the constraint W is compiled into the IC object $IC(W)$ and $(W|i)$ is a simplified form of W that is defined according to the quantification $(Q_i o_i \in E_i)$ of the boolean form of W. Also, as a running example, we will use the constraint example $W1$ and its simplified forms $(W1|pr)$, $(W1|se)$, and $(W1|co)$ as they have been defined in Example 5.3.2. The states of the shells of $W1$ are shown in Figure 5.3 on Page 97.

The rest of this section is devoted to the description of how the simplified form $(W|i)$ is compiled into the shell $S(W|i)$. The shell $S(W|i)$ describes two kinds of information. The first group is described by the attributes, *constraint*, *form*, *class* and *paths&roots*. These attributes are applicable to all patterns of simplified forms and thus they are presented first. The second group is described by the attributes, *range*, *objects* and *shell*. These attributes are applicable only if $(W|i)$ matches certain patterns of simplified forms and are presented in the last two subsections of this section.

5.5.1 Constraint Attribute

In our approach, all constraints specified for a database are compiled into objects of the IC class and all simplified forms of these constraints are compiled into objects of the Shell class. Thus, there is a relationship between the classes IC and Shell. The relationship is a one-to-many from IC to Shell. This comes from the fact that for a given constraint there may be more than one simplified form. The relationship is represented by two reference attributes. The first attribute exists in the IC class, named *Shells*, and is of set type {Shell}. The second attribute exists in the Shell class, named *constraint*, and is of type IC. The relationship is depicted in Figure 4.3 on Page 74.

The relationship between the classes IC and Shell is bidirectional and represented in the constraint catalog for the following reason:

- The criteria upon which we can establish a judgment of whether information stored in the shell $S(W|i)$ is correct is the IC object $IC(W)$. As we will show, all information described by the other attributes of object $S(W|i)$ is derived either directly from attributes of $IC(W)$ or indirectly through their manipulation. Thus in general, the relationship between the classes IC and Shell must be represented in the constraint catalog.

- Representing the direction of the relationship from IC class to Shell class facilitates the deletion of a constraint object and its shells from the constraint catalog and hence from the object database. Moreover, given a constraint name, we can obtain its simplified forms as well as all information stored about these simplified forms in the extension Shells. For instance to obtain general information that is stored in an attribute of shells of a constraint named W, we can formulate the query below against the constraint catalog, in which 'A' is a parameter denoting an attribute of Shell.

$$\mathbf{GI}(\mathbf{A}) = \{S.A \mid S \in \mathit{Shells} \wedge S.\mathit{constraint.name} = \mathtt{W}\}.$$

- Shells of the same constraint object share the same status and mode of the constraint. This information is stored in the IC class. By representing the direction of the relationship from Shell class to IC class, we do not need to repeat this information for each shell. For a given shell we can reach this information through the relationship that goes from it to its IC object. By avoiding this kind of redundancy we can guarantee that no two shells of the same IC object have two different values for their status or mode.

For the shell $S(W1|pr)$, the value of the attribute *constraint* is the IC object $IC(W1)$ that has oid #25 – see Figure 5.3.

5.5.2 Simplified Form

To improve integrity checking, we check an optimized form of a constraint rather than its canonical form. The attribute *form* of type *string* describes a simplified form of a constraint. For shell $S(W1|pr)$, the value of the attribute *form* is the simplified form $(W1|pr)[pr/self]$ that is the open wff obtained by replacing all occurrences of *pr* in wff (5.18) by the parameter *self*:

$$(W1|pr)[pr/self] :(self \in Professors)(\exists se \in self.teaches)$$
$$(\forall co \in \{se.isSectionOf\})(co.hasPrerequisites = \emptyset).$$

Here the parameter *self* denotes an object of the *Professor* class. Of course, at the implementation stage, wff $(W1|pr)[pr/self]$ must be translated into a statement of the query language of the underlying OODBMS. This point will be discussed in more detail in Section 7.6.

5.5.3 Class Attribute

We recall from Subsection 4.5.6 the following: If the evaluation of the query

$$\mathbf{VC} = \{ic.\mathit{wff} \mid ic \in \mathsf{ICs} \wedge type(o) \in ic.classes \wedge (\exists cp)(cp \in ic.paths\&roots \wedge up' \sqsubseteq cp)\}$$

is an empty set, then the update $up : o.A_1 \ldots A_n \; \theta \; v$, $\theta \in \{\longleftarrow, \overset{+}{\longleftarrow}, \overset{-}{\longleftarrow}\}$ does not violate database integrity. Thus if **VC**, the set of potentially violated constraints by the update up, is a none empty set, then we must check each wff in **VC**. In this case, by Lemma 5.3.3, it is sufficient to check for each $W \in \mathbf{VC}$ the simplified form of W w.r.t object variable of the same type as object o. To obtain these simplified forms from the extension Shells, each object $S(W|i)$ of Shell stores the name of the class of the object variable o_i and of the names of its subclasses. An attribute named *class* of type $\{string\}$ stores this information such that $S(W|i).class \subseteq IC(W).classes$. By this way, we can obtain simplified forms of potentially violated constraints by addressing the query below against the extension Shells.

$$\mathbf{SVC} = \{S.form \mid S \in \mathsf{Shells} \wedge type(o) \in S.class \wedge S.constraint.\mathit{wff} \in \mathbf{VC}\}. \tag{5.30}$$

For instance, the value for the attribute *class* of shell $S(W1|pr)$ only contains the class name *Professor*, that is, $S(W1|pr).class = \{Professor\}$. This is due to the fact that no subclass of the *Professor* class exists in the UNIVERSITY schema.

5.5.4 Paths and Persistent Roots

Query (5.30) is a nested query. To obtain **SVC** we first have to obtain **VC** by evaluating the query (4.12). To optimize the evaluation of this query, the Shell class has an attribute named *paths&roots* of set type $\{string\}$. For every shell $S(W|i)$, the value for *paths&roots* is a subset of the constrained paths and persistent roots that are stored in the IC object $IC(W).paths\&roots$ such that

- every path in $S(W|i).paths\&roots$ is a valid path for objects of classes stored in $S(W|i).class$, and
- every persistent root of $S(W|i).paths\&roots$ is of type c or $\{c\}$ and $c \in S(W|i).class$.

For instance, for shell $S(W1|pr)$, the value of attribute *paths&roots* consists of the persistent root *Professors* and the path *teaches*.

Now the set **SVC** can be obtained directly from objects of Shells as follows:

$$\begin{aligned} \mathbf{SVC} = \; & \{S.form \mid S \in \mathsf{Shells} \wedge type(o) \in S.class \wedge \\ & (\exists cp)(cp \in S.paths\&roots \wedge up' \sqsubseteq cp)\} \end{aligned} \tag{5.31}$$

where up' is a string as defined for the query (4.12). The evaluation of **SVC** by query (5.31) rather than (5.30) is desirable. First, in (5.31) we access one class rather than two classes as in (5.30). Second, because in general for each W, $S(W|i).paths\&roots \subseteq IC(W).paths\&roots$, the search space for up' in the query (5.31) is a subset of that in the query (5.30) or in the worst case equal.

In the following, we show that by storing some additional information about simplified forms that match one of the patterns (5.12)–(5.14) we can avoid – under certain condition – their checking even if they are in **SVC**.

5.5.5 Range of Simplified Forms

If the simplified form $(W|i)$ matches the pattern (5.12):

$$(W|i): \; (o_i \in E_i \rightarrow F[o_i])$$

then we call E_i the *range* of $(W|i)$. If $(W|i)$ is in **SVC**, simplified forms of potentially violated constraints, because of an updating to the object o, then we can avoid evaluation of $(W|i)$ if object o does not belong to the range of $(W|i)$. In other words, if $o \notin E_i$, then $(W|i)$ evaluates to true. If $(W|i)$ does not match pattern (5.12), then the range of the simplified form is undefined. In this case, if $(W|i)$ is in **SVC** for some object o, then we have to check $(W|i)$.

The attribute *range* of type *string* describes the range of the simplified forms. If the range of the simplified forms is undefined, then the value stored in $S(W|i).range$ is the empty string. Otherwise the value of $S(W|i).range$ is the expression E_i.

For example, the simplified form $(W1|pr)$ matches (5.12). The range of $(W1|pr)$ is the persistent root *Professors*, and thus $S(W1|pr).range = Professors$. Neither the simplified form $(W1|se)$ nor $(W1|co)$ matches the pattern (5.12) and therefore the range of each of them is undefined. This means that the value stored in both $S(W1|se).range$ and $S(W1|co).range$ is the empty string.

5.5.6 Interrelated Objects and their Shells

If the simplified form $(W|i)$ matches the patterns (5.13) or (5.14) :

$$(W|i) : (W|j)[o_j/o_i.A'] \text{ or}$$
$$(W|i) : (\forall o_j \in o_i.A')\ (W|j)[o_j]$$

then $(W|i)$ can be written as the following conjunction.

$$(W|i)[o_i] \equiv \bigwedge_{o_j \theta o_i.A'} (W|j)[o_j] \tag{5.32}$$

where θ is '$=$' (resp. '$\in$') if $(W|i)$ matches the pattern (5.13) (resp. (5.14)).

Intuitively, the conjunction (5.32) says that truth evaluations of simplified forms $(W|i)$ and $(W|j)$'s are non-orthogonal and this due to the fact that the object o_i and o_j's are interrelated objects. Let $o_i.A' = \{o_{j_1}, \ldots, o_{j_k}\}$. Suppose that o_i and $o_{j_1}, \ldots, o_{j_k}$ are modified by updates up_i and $up_{j_1}, \ldots, up_{j_k}$ respectively. Let $\mathbf{SVC}_i$ and $\mathbf{SVC}_{j_1}, \ldots, \mathbf{SVC}_{j_k}$ be sets of simplified forms of constraints that are potentially violated by up_i and $up_{j_1}, \ldots, up_{j_k}$ respectively. If $(W|i)[o_i]$ in $\mathbf{SVC}_i$ and $(W|j)[o_{j_l}] \in \mathbf{SVC}_{j_l}$, for some $l \in [1, k]$, then, from (5.32), the evaluation of each $(W|j)[o_{j_l}] \in \mathbf{SVC}_{j_l}$ is redundant.

In Chapter 6 we will show that by introducing the concept of constraint kernel this kind of redundancy can be avoided. To do that, we first have to reflect the conjunction (5.32) in the structure of the Shell class. For that reason two kinds of information are added to the shell $S(W|i)$:

(1) A valid path expression for objects of types stored in $S(W|i).class$: For that purpose, the shell $S(W|i)$ has an attribute named *objects* of type *string*.

(2) A reference to the shell that is associated with the object(s) determined by the path stored in $S(W|i).objects$: For that purpose, the shell $S(W|i)$ has an attribute named *shell* of type Shell.

The values of $S(W|i).objects$ and $S(W|i).shell$ are defined as follows:

- If $(W|i)$ matches the pattern (5.12), then the value of $S(W|i).objects$ is the empty string, and the value of $S(W|i).shell$ is *nil*.

- If $(W|i)$ matches the pattern (5.13), then the value of $S(W|i).objects$ is the set valued expression "$\{self.A'\}$" and the value of $S(W|i).shell$ is a reference to the shell of $(W|j)$, that is $S(W|j)$.

- If $(W|i)$ matches the pattern (5.14) then the value of $S(W|i).objects$ is the set valued expression "$self.A'$"; and the value of $S(W|i).shell$ is a reference to the shell of $(W|j)$, that is $S(W|j)$. Here *self* is a parameter denoting an object of one of the classes in $S(W|i).class$.

Now if $(W|i)$ matches one of the patterns (5.13) or (5.14), then (5.32) can be rewritten in terms of the Shell attributes as shown in (5.33). In this case, the shell $S(W|i)$ is said to be a *nested shell* . If $(W|i)$ matches the pattern (5.12), the shell $S(W|i)$ is said to be a *flat shell.*

$$S(W|i).form \equiv \bigwedge_{o \in S(W|i).objects} S(W|j).form[o]. \tag{5.33}$$

For example, the simplified forms $(W1|pr)$, $(W1|se)$ and $(W1|co)$

$$\begin{aligned}
(W1|pr) :& (pr \in Professors \rightarrow (\exists se \in pr.teaches)\\
& (\forall co \in \{se.isSectionOf\})(co.hasPrerequisites = \emptyset))\\
(W1|se) :& (W1|pr)[pr/se.isTaughtBy]\\
(W1|co) :& (\forall se \in co.hasSections)
\end{aligned}$$

match the patterns (5.12), (5.13), and (5.14) respectively. Thus, values for the *objects* and *shell* attributes of shells $S(W1|pr)$, $S(W1|pr)$, and $S(W1|pr)$ are as follows:

$$\begin{array}{ll}
S(W1|pr).objects ='' \ '' & S(W1|pr).shell = nil\\
S(W1|se).objects ='' \{self.isTaughtBy\}'' & S(W1|se).shell = S(W1|pr)\\
S(W1|co).objects ='' self.hasSections'' & S(W1|co).shell = S(W1|se)
\end{array}$$

Thus $S(W1|pr)$ is a flat shell whereas $S(W1|se)$ and $S(W1|se)$ are nested shells.

The complete compilation of simplified forms of constraint *W1* into objects of the class Shell is presented in Figure 5.3.

We conclude this chapter by pointing out that by the end of this chapter the second part of the thesis is completed, in which we presented the structure of the constraint catalog. In Section 6.8 we will present an overview on the structure of the constraint catalog and its relationship with the object schema.

In the third part, Chapters 6 and 7, we first present how constraints are considered as first class citizen in our approach and then how they are managed.

oid	*constraint*	*class*	*form*	*range*	*paths&roots*	*objects*	*shell*
#50	#25	*Professor*	$(W1 \mid pr)$	*Professors*	$\{+Professors, -teaches\}$	" "	*nil*
#51	#25	*Section*	$(W1 \mid se)$		$\{isSectionOf, isTaughtBy\}$	"$\{self.isTaughtBy\}$"	#50
#52	#25	*Course*	$(W1 \mid co)$		$\{+hasPrerequisites, -hasPrerequisites, hasPrerequisites, hasSections\}$	"$self.hasSections$"	#51

Figure 5.3: Shells $S(W1 \mid pr)$, $S(W1 \mid se)$, and $S(W1 \mid co)$ of the constraint $W1$.

Chapter 6

Constraint Structure

In Chapter 5, we have shown, under the condition that the current state of an object base is valid, that simplified forms and constraints are equivalent. This has been proved in Lemma 5.3.3. Thus, simplified forms of constraints and hence shells representing them are sufficient to maintain consistency of an object base. On that grounds, in this chapter we first propose a general method for consistency maintenance by using shells and address three drawbacks of that method. The first drawback is run-time transaction overhead due to accessing the constraint catalog. This drawback arises because there is no link between individual objects and shells.

The two other drawbacks are the missing of some features that we want to provide to our approach:

- *Efficiency.* Although shells store simplified forms of constraints, there is a drawback of redundant checking. The same simplified form can be checked unnecessarily more than one time for the same or different objects.

- *Interrelated Objects.* Consistency maintenance of a large number of interrelated objects with complex structures would be impractical with the presence of the kind of redundant checking mentioned in the previous point.

- *Disabling and Enabling of Constraints.* All information stored in shells is applied to all objects of a specific class. Thus by using shells we can only enable and disable constraints for all object of a class but not for specific object(s) of that class.

These drawbacks result from the fact that shells are sharable among individual objects. In conclusion, shells alone are not adequate to represent constraints. This leads us to the introduction a new class named *Kernel*. The aggregation that consists of the Shell class

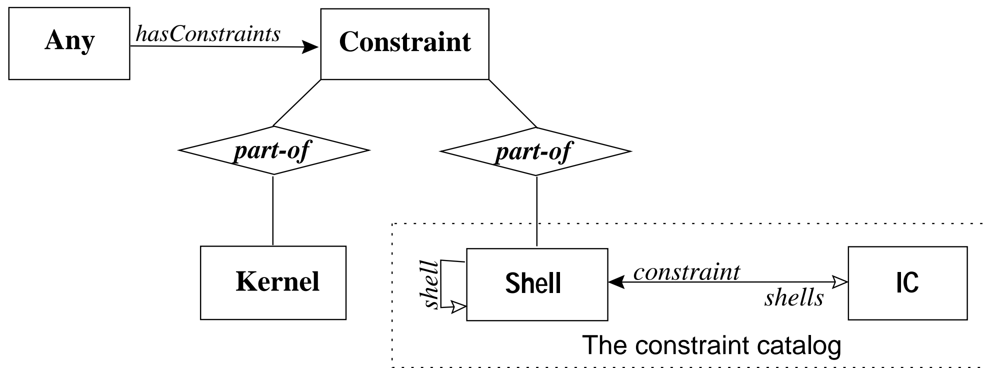

Figure 6.1: Relationships between the classes *Any*, *Constraint*, *Kernel*, Shell, and IC.

and the *Kernel* class introduce another class named *Constraint*. Then the *Constraint* class is linked to the root class *Any* and hence objects of the *Constraint* class provide the link between shells and individual objects. We will show that objects of the *Constraint* class overcome the mentioned drawbacks of the proposed method and provide the missing features stated above.

This chapter is organized as follows. In Section 6.1, we sketch the steps of a method for consistency maintenance by using shells and identify three drawbacks. In Section 6.2, the first drawback will be addressed and we show through an example that the proposed solution does not overcome the other drawbacks. In Sections 6.3, and 6.4, we introduce the *Kernel* and the *Constraint* classes respectively. Then in Sections 6.5 through 6.7 three issues related to objects of the *Constraint* class are discussed, namely the life cycle of objects of the *Constraint* class, the role of the extension of the *Constraint* class, and the linkage between objects of the *Constraint* class and individual objects.

6.1 Consistency Maintenance by Using Shells

To maintain integrity of an object base it is sufficient to consider simplified forms that are stored in shells. If the current state of an object is consistent and then the state is changed, in order to check whether the new state is still consistent, it is sufficient to check ground instances of forms that are stored in the shells of possibly violated constraints. This point has been discussed in detail in Subsection 5.5.3. On that grounds, a general method of integrity maintenance using shells can be introduced. In the following we present the individual steps of the method and then examine three drawbacks of it. The method is described below.

Let o be an object. The maintenance of the consistency of o by using shells consists of two parts. First, every time o is modified by an update up, the following steps are performed:

(1) The shells of potentially violated constraints **SPVC** are retrieved from the Shells extension by using the query:

$$\mathbf{SPVC} = \{S \mid S \in \mathsf{Shells} \wedge type(o) \in S.class \wedge (\exists cp)(cp \in S.paths\&roots \wedge up' \sqsubseteq cp)\} \tag{6.1}$$

where up' is defined as for query (4.12).

(2) Every ground instance of the form of an immediate constraint of a shell in **SPVC** is checked. If any of them is violated, then the modification to o by up is undone and the update is rejected.

(3) Every ground instance of the form of a deferred constraint of a shell in **SPVC** is stored temporary.

Finally, when a check is issued to test consistency of the o object, every simplified form stored in step (3) above is checked. If any of them is violated then all modifications to o are undone.

Compared to the consistency maintenance by using application-oriented techniques and ECA rules, the method has three advantages that stem from the fact that the method is based on the constraint catalog. First, constraint specifications are totally separated from update transactions and application programs. Thus the method meets the requirement of integrity independence; changing constraints does not change application programs and vice versa. Second, the method checks ground instances of simplified forms of constraints and not the original constraints themselves. Third, the method meets the requirement that "*control in object-oriented database is localized rather than centralized*"[JQ92]. In this method, each object handles validity of constraints it must obey.

However, the method also has three drawbacks. First, there is a run-time transaction overhead due to the constraint catalog accessing each time an update up is made to o. Second, the method does not exploit any other information stored in shells than those needed to obtain the set **SPVC**. Thus, the method treats flat shells and nested shells in the same way. Nested shells should be treated differently to avoid redundant checking as discussed in Subsection 5.5.6. Third, all information stored in a shell is global in the sense that it is applied to all objects of the class described in that shell. Hence we cannot enable and disable constraints locally for particular object(s).

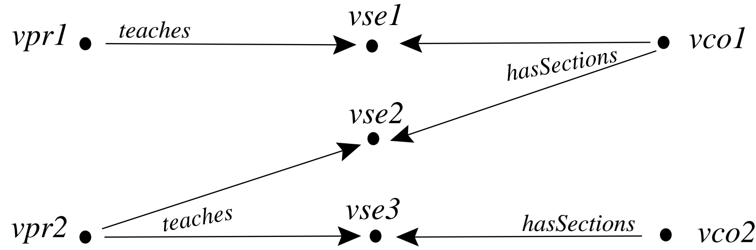

Figure 6.2: A set of interrelated objects of *Professor*, *Section* and *Course* classes.

In the next section we show that the first drawback above can be avoided by attaching each object with shells of constraints that restrict that object.

6.2 Linking Objects and Shells

To avoid run-time transaction overhead due to constraint catalog access, we can exploit that shells are objects and thus they can be shared among individual objects. This can be done by establishing a (1:m) relationship from the root class *Any* to the Shell class. The relationship can be implemented by introducing a new attribute *hasShells*, say to the root class *Any*. Now, shells of o can be identified and assigned to the *hasShells* attribute by using the update statement given below.

$$o.hasShells \longleftarrow \{S \mid S \in \mathsf{Shells} \wedge type(o) \in S.class\} \tag{6.2}$$

In the update (6.2) we only need to know the type of the object o. This leads to two remarks. First, the execution of (6.2) is carried out one time throughout the life of object o, namely at the time of creating o. Second, all objects of the same type will share the same set of shells. This last point is illustrated in the following example.

▶ **Example 6.2.1 (Shells of Objects of the Same Type)** Consider the interrelated objects shown in Figure 6.2. Objects *vpr1* and *vpr2* are of the *Professor* class; objects *vse1*, *vse2* and *vse3* are of the *Section* class; and objects *vco1* and *vco2* are of the *Course* class. Suppose that these objects are subject to only one constraint, the constraint *W1*. Recall from Section 5.5 that *W1* has three shells, the flat shell $S(W1|pr)$ and the nested shells $S(W1|se)$ and $S(W1|co)$. In Figure 5.3, the values of the *class* attribute of shells $S(W1|pr)$, $S(W1|se)$, and $S(W1|co)$ are the name of classes *Professor*, *Section* and *Course*, respectively.

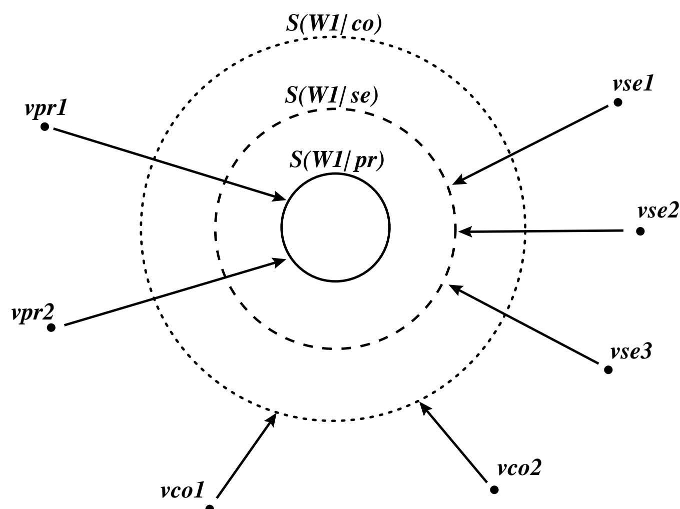

Figure 6.3: The flat shell $S(W1|pr)$, and the nested shells $S(W1|se)$ and $S(W1|co)$.

Linking shells with objects, as proposed above, means that for each object *vpr* of the *Professor* class, *vse* of the *Section* class and *vco* of the *Course* class the values of the *hasShells* attributes are as follows.

$$vpr.hasShells = \{S(W1|pr)\}$$
$$vse.hasShells = \{S(W1|se)\}$$
$$vco.hasShells = \{S(W1|co)\}$$

Thus, the *vpr1* and *vpr2* object share the flat shell $S(W1|pr)$, the *vse1*, *vse2* and *vse3* objects share the nested shell the $S(W1|se)$, and *vco1*, and *vco2* objects share the nested shell $S(W1|co)$. This point is illustrated in Figure 6.3. ■

The direct link between objects and shells overcomes the problem of run-time transaction overhead due to the constraint catalog access each time an update is carried out to an object. Instead of accessing the constraint catalog we can evaluate the set **SPVC** locally by replacing the *Shells* extension in the query (6.1) with the set valued expression *o.hasShells*. Moreover, the evaluation of **SPVC** in that manner is better than before as *o.hasShells* is a subset of Shells.

In this method, each object checks ground instances of a form in **SPVC** locally. A ground instance of the form stored in a nested shell, such as $(W1|se)$, is the conjunction of ground instances of the form stored in another shell, such as $(W1|pr)$. Because of this,

checking ground instances of a form of nested shell locally without paying any attention to ground instances of forms of related shells introduces a redundant checking. This kind of redundancy has been discussed in Subsection 5.5.6 and will be illustrated by the following example.

▶ **Example 6.2.2 (Nested Shells)** Consider the scenario given in Figure 6.1 which tabulates a series of time instances at which either a modification to one of the objects in Figure 6.2 is carried out or a form of the constraint *W1* w.r.t. one of these objects is *successfully* checked.

Consider the snapshot of the scenario that is delimited by the time instances t_0 and t_3. In that snapshot, objects *vpr1* and *vse1* are modified at time instances t_0 and t_1, respectively. Then ground instances of the forms stored in the shells $S(W1|pr)$ and $S(W1|se)$ are checked at time instances t_3 and t_4, respectively.

The checking of $S(W1|pr).form[self/vpr1]$ is carried out at time instance t_2. This happens *after* all modifications to *vpr1* and *vse1* objects have been carried out. Thus, at instance t_2 the object *vpr1* obeys the constraint *W1*.

In Example 5.3.2, we have shown that the form of *W1* w.r.t. objects of *Section* class, $(W1|se)$, is derived from the form of *W1* w.r.t. objects of *Professor* class, $(W1|pr)$. This is stated in (5.19). This yields the following equivalences between forms stored in shells to *vpr1* and *vse1* objects, and *vpr2* and *vse2* objects:

$$S(W1|se).form[self/vse1] \equiv S(W1|pr).form[self/vpr1] \tag{6.3}$$

$$S(W1|se).form[self/vse2] \equiv S(W1|pr).form[self/vpr2] \tag{6.4}$$

Because of the equivalence (6.3) the *vse1* object at instance t_2 also obeys constraint *W1*. No modification occurs to the *vse1* object after the time instance t_2. Thus, checking

$$S(W1|se).form[self/vse1]$$

at time instance t_3 is redundant.

Consider the snapshot of the scenario that is delimited by the time instances t_4 and t_8. In that snapshot, *vco1*, *vse2* and *vpr2* objects are modified at t_4, t_5 and t_6, respectively. Then the ground instances $S(W1|pr).form[self/vpr2]$ and $S(W1|co).form[self/vco1]$ are checked at t_8 and t_7 time instances, respectively.

In Example 5.3.2, we have also shown that the form of *W1* w.r.t. objects of *Course* class, $(W1|co)$, is derived from the form of *W1* w.r.t. objects of *Section* class, $(W1|se)$. This is stated in (5.20). This yields the following equivalence among, on the one hand, the

time	*vpr1*	*vse1*	*vco1*	*vse2*	*vpr2*	Violated/Checked form
t_0	update					$(W1\,\vert pr)$
t_1		update				$(W1\,\vert se)$
t_2	check					$(W1\,\vert pr)[self/vpr1]$
t_3		check				$(W1\,\vert se)[self/vse1]$
....	......	......	......	......	......	....
t_4			update			$(W1\,\vert co)$
t_5				update		$(W1\,\vert se)$
t_6					update	$(W1\,\vert pr)$
t_7					check	$(W1\,\vert pr)[self/vpr2]$
t_8			check			$(W1\,\vert co)[self/vco1]$

Table 6.1: An update scenario to objects of Figure 6.3.

conjunction of forms of shells of objects *vse1* and *vse2* and, on the other hand, the object *vco1*.

$$S(W1\,|co).form[self/vco1] \equiv S(W1\,|se).form[self/vse1] \wedge S(W1\,|se).form[self/vse2] \quad (6.5)$$

Because of the equivalence (6.4) the checking of $S(W1\,|co).form[self/vco1]$ implies the checking of $(W1\,|se)[self/vpr1]$ and $(W1\,|se)[self/vpr1]$. Checking of $S(W1\,|pr).form[self/vpr2]$ is carried out at t_7, that is, *after* each update to objects *vco1*, *vse2* and *vpr2* has been carried out. Thus the object *vpr2* obeys the constraint *W1*. The last update of *vco1* occurred at time instance t_4. This happens after the last valid check at object *vpr1*; that is at time instance t_2. Thus we are not sure whether object *vpr1* still obeys the constraint *W1* after update of abject *vco2* at t_4. This means that checking of $S(W1\,|se).form[self/vco1]$ at t_9 is partially redundant as the form $S(W1\,|se).form[self/vpr2]$ is valid. In other words, to check $S(W1\,|se).form[self/vco1]$ at t_8 it is sufficient to check only $S(W1\,|se).form[self/vpr1]$. ■

In Example 6.2.2 we inferred equivalences among forms of shells of interrelated objects on the grounds of the semantics of the simplified forms. However, we can infer these equivalences from shells directly.

▶ **Example 6.2.3 (Inferring Equivalences among Forms from Shells)** Objects of the configuration, given in Figure 6.3, are interrelated in the following sense. Professor *vpr1* teaches section *vse1* of course *vco1*; and professor *vpr2* teaches section *vse2* of course *vco1*,

and section *vse3* of course *vco2*. This semantics can be stated by the following equalities:

$$\begin{aligned} vco1.hasSections &= \{vse1, vse2\} \\ vse1.isTaughtBy &= vpr1 \\ vse2.isTaughtBy &= vpr2 \end{aligned} \tag{6.6}$$

If S is a nested shell then upon the semantics given in Chapter 5 to attributes *form*, *objects* and *shell* of Shell class, the form of the shell S is equivalent to the conjunction stated in (5.33):

$$S(W|i).form \equiv \bigwedge_{o \in S(W|i).objects} S(W|j).form[o]$$

The result of the compilation of the simplified forms $(W1|pr)$, $(W1|se)$, and $(W1|co)$ into shells is shown in Figure 5.3. Thus, we have for each professor *vpr*, section *vse* and course *vco* in Figure 6.2, the following equivalences:

$$S(W1|se).form[self/vse] \equiv \bigwedge_{vpr \in S(W1|se).objects} S(W1|pr).form[self/vpr] \tag{6.7}$$

$$S(W1|co).form[self/vco] \equiv \bigwedge_{vse \in S(W1|co).objects} S(W1|se).form[self/vse] \tag{6.8}$$

Thus, from (6.6), (6.7) and (6.8) it follows:

$$\begin{aligned} (W1|se)[self/vse1] &\equiv (W1|pr)[self/vpr1] \\ (W1|se)[self/vse2] &\equiv (W1|pr)[self/vpr2] \\ (W1|co)[self/vco1] &\equiv (W1|se)[self/vse1] \wedge (W1|se)[self/vse2] \end{aligned}$$

The reader may notice that the last three equivalences are the same as the equivalences (6.3)–(6.5) but here we infer them directly from the semantics of shells and not as we have done in Example 6.2.2, where equivalences are inferred from the semantics of simplified forms. ■

In Example 6.2.2, we examined redundant checking by using the following observations:

(1) Equivalences that may exist among ground instances of forms stored in shells of inter-related objects.

(2) A series of time instances at which either a modification to an object is carried out and potentially violates a constraint or a ground instance of the form of that constraint is successfully checked, i.e., valid.

$$\begin{array}{l}\textbf{Class } Kernel\ [\\ \quad lastUpdate: string,\\ \quad lastCheck: string\ ]\end{array}$$

Figure 6.4: Structure of the $Kernel$ class.

In Example 6.2.3, we have shown that we can infer directly equivalence of (1) from states of shells. However, without information needed to maintain a a series of time instances of (2) we cannot exploit them to avoid redundant checking. The information of (2) is not a part of the specification of the Shell class. This is due to two reasons. On the one hand, this information is local to individual objects. On the other hand, each shell represents a simplified form. Thus, each shell is sharable among all objects of the same type and hence a shell represents global information of objects. In conclusion, this shows that using shells as the only means to represent constraints does not help in avoiding the kind of redundant checking illustrated tin Example 6.2.2.

In Section 6.3, we introduce a new class, named *kernel*, and we show that by using objects that consists of the aggregation of a constraint shell and a kernel instead of objects of the Shell class we can the avoid the drawbacks mentioned so far.

6.3 The *Kernel* Class

To avoid shortcomings of considering shells as the only means to represent constraints, shells are associated with another kind of objects called *constraint kernels*. For every constraint W and object o such that o is subject to W there is a kernel, denoted $K(W|o)$, that stores local information about

(1) the last time the object o has been modified and this modification potentially violates the constraint W and

(2) the last time the form stored in the shell, associated with the object o, has been successfully checked.

The class of kernels, named *Kernel*, has the structure shown in Figure 6.4 and consists of two attributes, *lastUpdate* and *lastCheck*. Each of theses attributes is of type $string$. The semantics of *lastUpdate* and *lastCheck* attributes correspond to (1) and (2) above.

time	$K(W1 \mid vpr1)$		$K(W1 \mid vse1)$		$K(W1 \mid vco1)$		$K(W1 \mid vse2)$		$K(W1 \mid vpr2)$		Violated/ Checked form
	last−Update	*last−Check*	*last−Update*	*last−Check*	*last−Update*	*last−Check*	*last−Update*	*last−Check*	*last−Update*	*last−Check*	
t_0	t_o	$t_{-\infty}$	$t_{-\infty}$	$t_{-\infty}$	$t_{-\infty}$	$t_{-\infty}$	$t_{-\infty}$	$t_{-\infty}$	$t_{-\infty}$	$t_{-\infty}$	$(W1 \mid pr)$
t_1	$\vdots$	$\vdots$	t_1	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$(W1 \mid se)$
t_2	$\vdots$	t_2	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$(W1 \mid pr)[self/vpr1]$
t_3	$\vdots$	$\vdots$	$\vdots$	t_3	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$(W1 \mid se)[self/vse1]$
....	....	......	....	......	....	......	....	......	....	......	....
t_4	$\vdots$	$\vdots$	$\vdots$	$\vdots$	t_4	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$(W1 \mid co)$
t_5	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	t_5	$\vdots$	$\vdots$	$\vdots$	$(W1 \mid se)$
t_6	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	t_6	$\vdots$	$(W1 \mid pr)$
t_7	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	t_7	$(W1 \mid pr)[self/vpr2]$
t_8	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	t_8	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$(W1 \mid co)[self/vco1]$

Figure 6.5: Kernels of Example 6.3.1 according to the update scenario of Table 6.1.

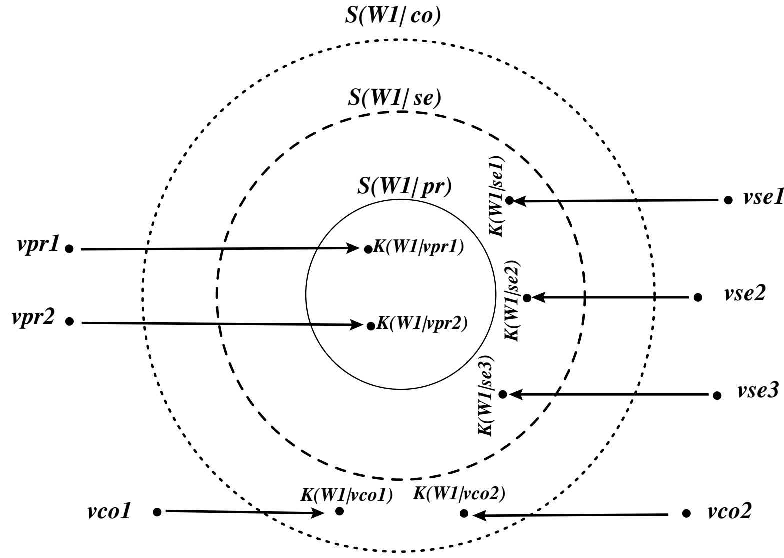

Figure 6.6: Kernels and shells of the constraint *W1* and objects of Figure 6.3.

▶ **Example 6.3.1 (Constraint Structure = Kernel + Shell)** Consider the objects in Example 6.2.2. Suppose that for each object o of Figure 6.2 and each shell in $o.hasShells$ we create a kernel $K(W1|o)$ and associate it with that shell. This situation is depicted in Figure 6.6. In addition, suppose that the attributes *lastUpdate* and *lastCheck* of each kernel are maintained throughout the whole scenario, which is given in Figure 6.1. The result of this maintenance is shown in Figure 6.5.

Let *unchecked*(*vse*, *W1*) be the set of all professors *vpr* such that

(1) the section *vse* is taught by the professor *vpr* and

(2) the last time an update to the section *vse* potentially violating *W1* is performed after the last time of a successful check to the simplified form $S(W1|pr).form$ at the object *vpr* has been done.

This can be stated in terms of attributes of the shell $S(W1|se)$ and the kernel $K(W1|vse)$ as follows:

$$\begin{aligned} unchecked(vse, W1) = \{vpr \mid vpr \in S(W1|se).objects[vse] \wedge \\ K(W1|vpr).lastCheck < K(W1|vse).lastUpdate\} \end{aligned} \tag{6.9}$$

where $S(W1|se).objects[vse]$ is the set valued expression obtained by replacing the parameter *self* with the object *vse* in the expression stored in $S(W1|se).objects$, that is, the set value expression $\{vse.isTaughtBy\}$.

It follows from (6.9) that $unchecked(vse, W1)$ is a subset of $S(W1|se).objects[vse]$. Let

$$checked(vse, W1) = S(W1|se).objects[vse] - unchecked(vse, W1)$$

Let vpr' be a professor such that $vpr' \in checked(vse, W1)$. Thus, vpr' is related to vse and hence $(W1|pr)[self/vpr']$ and $(W1|se)[self/vse]$ are equivalent and $K(W1|vpr').lastCheck$ is greater than $K(W1|vse).lastUpdate$ and hence $(W1|se)[self/vse]$ is valid. From (6.7), we have

$$S(W1|se).form[self/vse] \equiv \bigwedge_{vpr \in S(W1|se).objects} S(W1|pr).form[self/vpr]$$

Thus

$$\begin{aligned} S(W1|se).form[self/vse] &\equiv \bigwedge_{vpr \in checked(vse, W1) \cup unchecked(vse, W1)} S(W1|pr).form[self/vpr] \\ &\equiv \bigwedge_{vpr \in checked(vse, W1)} S(W1|pr).form[self/vpr] \wedge \\ &\quad \bigwedge_{vpr \in unchecked(vse, W1)} S(W1|pr).form[self/vpr] \end{aligned}$$

According to the definition of $checked(vse, W1)$ we have

$$\bigwedge_{vpr \in checked(vse, W1)} S(W1|pr).form[self/vpr] \equiv \mathbf{T}$$

This shows that the following wff is equivalent to (6.7):

$$S(W1|se).form[self/vse] \equiv \bigwedge_{vpr \in unchecked(vse, W1)} S(W1|pr).form[self/vpr] \tag{6.10}$$

Now at time instance t_3 of the scenario, $unchecked(vse1, W1)$ is the empty set. Thus, the redundant check of the first snapshot of the scenario can be avoided if we use (6.10) instead of (6.7).

Similarly the redundancy of the second snapshot can be avoided if we use the following equivalence instead of (6.8):

$$\begin{aligned} S(W1|co).form[self/vco] &\equiv \bigwedge_{vse \in unchecked(vco, W1)} S(W1|se).form[self/vse] \\ &\equiv \bigwedge_{vse \in unchecked(vco, W1)} \bigwedge_{vpr \in unchecked(vse, W1)} S(W1|pr).form[self/vpr] \end{aligned} \tag{6.11}$$

Here $unchecked(vco, W1)$ is the set of all sections vse such that

Class *Constraint* [
shell : *Shell*,
kernel : *Kernel*,
localStatus : *string*]

Figure 6.7: Structure of the *Constraint* class.

(1) *vse* is a section of the course *vco* and

(2) the last time an update to the course *vco* potentially violating *W1* is performed after the last time a successful check to the simplified form of $S(W1|se)$ at the object *vse* has been done.

$$unchecked(vco, W1) = \{vse \mid vse \in S(W1|co).objects[vco] \wedge K(W1|vse).lastCheck < K(W1|vco).lastUpdate\} \quad (6.12)$$

According to the scenario given in Figure 6.1, the reader can easily verify that, at time instance t_8, $unchecked(vco1)$ is $\{vse1\}$. ■

This shows that by gathering a constraint shell and a kernel in one frame and making this frame a representation of constraints, we can avoid redundant checks that may arise because of treating nested shells in the same way as flat shells.

6.4 The *Constraint* Class

To combine a constraint shell and a kernel in one frame we introduce a new class named *Constraint*. The structure of the *Constraint* class is presented in Figure 6.7. It consists of an aggregation of a shell and a kernel, and an attribute named *localStatus*.

The shell is represented by the attribute *shell* of type Shell and describes global information about a particular constraint to all objects restricted by that constraint.

The kernel is represented by the attribute *kernel* of type *Kernel* and describes information about a particular constraint relative to object(s) restricted by that constraint.

The *localStatus* attribute is of type *string* and indicates whether a constraint is enabled or disabled w.r.t. the object it is associated with. It is introduced so that users can handle situations in which some objects are exceptions for some constraints, that is, they violate constraints but they are stored in the database.

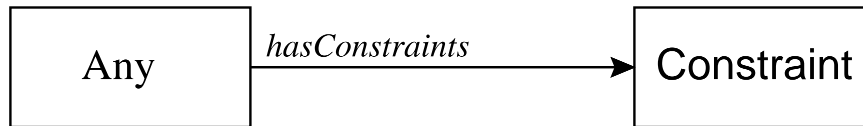

Figure 6.8: Relationship between the root class *Any* and the *Constraint* class.

To make the frame consisting of a shell and a kernel a representation for constraints, we establish a relationship between the root class *Any* and the *Constraint* class as shown in Figure 6.8.

The relationship is (1:m) and unidirectional from the root class *Any* to the *Constraint* class. The relationship is implemented through adding to the root class *Any* the attribute *hasConstraints* of set type {*Constraint*}. This means that each class in the database schema inherits this attribute. The relationship is one-to-many because in general an object is restricted by many integrity constraints and hence that object is associated with many shells. The relationship is of one direction from *Constraint* class to *Any* class, so that the persistence of objects is orthogonal to that of object constraints. Otherwise, all objects would be persistent as long as they are reachable from object constraints. This comes from the style of persistence by reachability that we adapted in this thesis.

Henceforth, we use the term a *constraint instance* to refer to an object of the *Constraint* class.

6.5 States of Constraint Instances

The evaluation of a constraint instance can be determined from values stored in the attributes *lastCheck* and *lastUpdate* of its kernel. Each constraint instance either is in the *checked* or *unchecked* state. In this section, we present *what* is meant by the *checked* and *unchecked* states and describe the life cycle of a constraint instance between the two states. In Chapter 7, we will discuss in details *how* constraints instances are actually checked.

In this section, we assume that i is a constraint instance in $o.hasConstraints$ and that $i.shell.constraint.name = \texttt{W}$.

The semantics of states of a constraint instance depends on whether the constraint instance has a nested or flat shell.

6.5.1 Flat Constraint Instances

Flat constraint instances are those which have flat shells. The flat constraint instance i is characterized by the fact that $i.shell.objects$ is an empty string and $i.shell.shell$ is nil. For instance, all constraint instances that have the shell $S(W1|pr)$ are flat.

The flat constraint instance i has one and only one of the following states at each point in time of its lifetime:

- *unchecked* state iff $i.kernel.lastUpdate > i.kernel.lastCheck$.
- *checked* state iff $i.kernel.lastUpdate < i.kernel.lastCheck$.

6.5.2 Nested Constraint Instances

Nested constraint instances are those which have nested shells. A nested constraint instant i is characterized by (1) the evaluation of $i.shell.shell$ is a reference to another shell and (2) the evaluation of $i.objects$ is a string that describes a set valued expression. The nested constraint instance i has one and only one of the following states at each point in time of its lifetime:

- *unchecked* state iff $i.kernel.lastUpdate > j.kernel.lastCheck$, for some j and o' such that $j.shell = i.shell.shell$, $o' \in i.shell.objects$ and $j \in o'.hasConstraints$.
- *checked* state iff $i.kernel.lastUpdate < j.kernel.lastCheck$ for every j and o' such that $j.shell = i.shell.shell$, $o' \in i.shell.objects$ and $j \in o'.hasConstraints$.

6.5.3 The Life Cycle of a Constraint Instance

We now we describe the life cycle of constraint instances by using the transition diagram shown in Figure 6.9.

The initial state. The initial state of each constraint instance i is the *unchecked* state. A constraint instance i remains in this state as long as one of two cases occur. First, if i is unsuccessfully checked, i.e., the ground instance $i.shell.form[self/o]$ is unsatisfied. Second, if an update is performed and potentially violates i. This update may occur to the state of the object o or to one of the objects that are related to it by the relationship described by the expression stored in $i.shell.objects$. This is illustrated in Figure 6.9 by the label *unsuccessful check or relevant update* on the arc loop of the *unchecked* node.

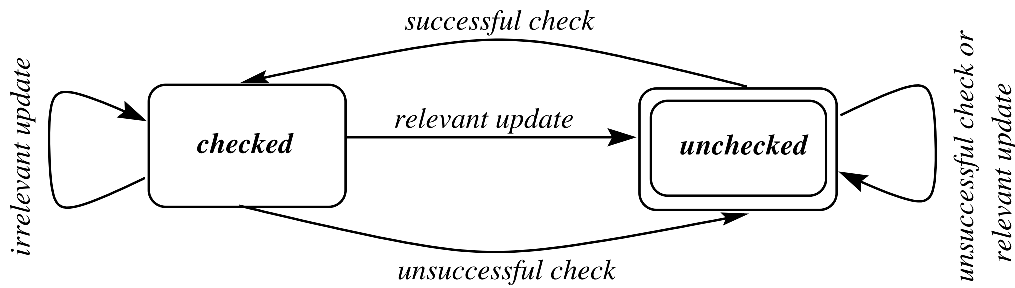

Figure 6.9: The life cycle of a constraint instance.

A transition from the *unchecked* state to the *checked* state. A transition of i from the *unchecked* state to the *checked* state occurs only in one case, namely when i is successfully checked. This is illustrated by the label *successful check* on the arc that goes from the *unchecked* node to the *checked* node. If i is a flat constraint instance, then the check occurs to the ground instance of the form stored in it, that is, $i.shell.form[self/o]$. If i is a nested constraint instance, its check occurs for constraint instances of W that are associated with objects related to the object o by the relationship stored in $i.shell.objects$. In Chapter 7 we discuss the topic of constraint instances checking in detail.

A constraint instance i remains in the *checked* state as long as no update is made to its state or to one of the objects related to it by the relationship described in $i.shell.objects$. This is indicated by the label *irrelevant update* on the arc loop of the *checked* node.

A transition from the *checked* state to the *unchecked* state. A transition of a constraint instance i from the *checked* state to the *unchecked* state only occurs in two cases. The first case is the one in which an update is performed and potentially violates the ground instance $i.shell.form[self/o]$. This update may occur to the state of the object o or to one of its related objects. This is indicated by the label *relevant update* on the arc going from *checked* node to *unchecked* node. The second case is the one in which the constraint instance is unsuccessfully checked, i.e., the ground instance $i.shell.form[self/o]$ is unsatisfied.

According to the semantics given above, the *unchecked* state can be seen as a *disjunction* of two states, namely the *unknown* state and the *invalid* state. Instances of the unknown state are cases in which it is unknown whether a constraint is satisfied or not. These cases occur for a (1) newly created constraint instances, and (2) *potentially violated* deferred constraint instances. However, we cannot consider these two states explicitly in our approach since the derivation of simplified forms is essentially based on the specialization technique

which in turn assumes that constraints are valid before an update.

6.6 Role of Class Extension *Constraints*

According to the semantics given to kernels and shells, the base for constraint instantiating is that for a given constraint, objects of the same type have different kernels but the same shell. Two kinds of constraints are exceptions to that base, namely key constraints and simplified forms that are the same as the constraint itself.

A key constraint restricts a *set* of objects and not an individual object. The violation of a key constraint by one object means that the whole set violates the key constraint. Thus, each instance of a key constraint associated with an object of the set should have the same state, that is either *cheched* or *unchecked*. Hence each constraint instance of a key constraint should have the same kernel. The characterization of a key constraint cannot be done automatically and thus it should be done manually by the users.

According to the method presented in Section 5.3, each simplified form of a constraint derived according to (Case 1) of Subsection 5.3.1 or (Case 1) or (Case 2) of Subsection 5.3.2 is the same as the constraint itself. Thus all constraint instances that have shells of this kind should have the same state and hence they should share the same kernel. A shell, S, of this kind can be characterized by the fact that wff stored in $S.form$ is the same as the boolean form of wff stored in $S.constraint.wff$.

We will refer the two kind of constraint instances mentioned above as *odd* constraint instances.

To take odd constraint instances into account in the process of constraint instantiating, we define an extensions named *Constraints* of set type $\{Constraint\}$ for the class *Constraint*. For each shell S in *Shells* that corresponds to a simplified form of a constraint W, there is a constraint instance i in the extension *Constraints* such that $i.shell = S$. The kernel of i is defined as follows:

(1) If S corresponds to a key constraint then, $i.kernel$ is a newly created kernel.

(2) If $S.form$ is the boolean form of the constraint W then $i.kernel$ is the kernel K, where K is the kernel of all odd instances j in *Constraints* such that $j.shell.constraint.name =$ `W`.

(3) If i is not an odd constraint instance, then $i.kernel$ is nil.

The initiation of kernels of constraint instances of cases (1) and (2) are performed by a method, named *conIni*, of the root class *Any*. This method will be presented in detail in Section 6.7.

We recall that each constraint instance has a global and a local status. The value of a global and local status of a constraint is either *enabled* or *disabled*. The global status of a constraint instance i is registered in $i.shell.constraint.status$ and its local status in $i.localStatus$. At the beginning of the life cycle of a constraint instance i, the local and global status must coincide. Thus, for each instance i of $o.hasConstraints$, the initial value of $i.localStatus$ is $i.shell.constraint.status$. One exception to that rule is that all constraint instances in the extension *Constraints* that have explicit range are disabled locally. Formally, for each $i \in Constraints$, if $i.shell.range \neq nil$ then $i.localStatus = disabled$. The rationale behind that will be explained in Section 7.8.

6.7 The Initiating of the *hasConstraints* Attribute

Let C be a class. After the creation of a new object o of the class C, o must be associated with constraint instances stored in the extension *Constraints* which represent the constraints the object o must obey. This can be done by initiating the *hasConstraints* attribute of o with those constraints instances. This can be achieved by adding a method named *conIni* to the root class *Any* with the signature $conIni(nil) : nil$. The method is presented in Table 6.2. In the following, we describe the method in detail.

First, we evaluate the set *ConstraintInstances* of instances that are stored in the extension *Constraints* and which restrict objects of the same type as the object o. This is carried out by step [4].

Second, for each instance i in *ConstraintInstances*, we create a new constraint instance j and initiate the attributes of j as follows. The initial value of $j.shell$ is the same as that of $i.shell$. This is carried out by step [8]. The initial local status of each newly created constraint instance is the same as $i.localStatus$. This is carried out by step [9]. The initial value of $j.kernel$ depends on whether $i.kernel$ is nil. This is carried out by steps [10]–[12]. If $i.kernel$ is nil, this means that i is not an odd constraint instance and hence j should not share the kernel of i. In this case, we create a new kernel and assign it to $j.kernel$. This is carried out by step [10]. If $i.kernel$ is nil, this means that i is an odd constraint instance and hence j should share the kernel of i. In this case, we assign the kernel of i to $j.kernel$. This is carried out by step [11]. The initial state of each newly created constraint instance is the *unchecked* state. Then by the end of the execution of the method, each

```
     conIni(nil) : nil
[1]  begin
/* variable declarations */
[2]  i, j : Constraint;
[3]  constraintInstances : {Constraint};
/* retrieve instances from the extension */
[4]  constraintInstances ⟵ {i | i ∈ Constraints∧
[5]                            type(o) ∈ i.shell.class};
[6]  for each i ∈ constraintInstances do
        /* create a new instance and initiate its shell and local status */
[7]     j ⟵ new() Constraint;
[8]     j.shell ⟵ i.shell;
[9]     j.localStatus ⟵ i.localStatus;
        /* initiate the kernel of the new instance */
[10]    if i.kernel = nil then j.kernel ⟵ new() Kernel
[11]    else  j.kernel ⟵ i.kernel
[12]    endif;
        /* initiate the state of the new instance to unchecked */
[13]    j.kernel.lastCheck ⟵ currentTime();
[14]    j.kernel.lastUpdate ⟵ currentTime();
        /* associate the new instance with self */
[15]    self.hasConstraints ⟵+ j;
[16] endfor
[17] end.
```

Table 6.2: The method *conIni*() of the root class *Any*.

j in *self.hasConstraints* should satisfy $j.kernel.lastUpdate > j.kernel.lastCheck$. For that reason, we assign the value $currentTime()$ to $j.kernel.lastCheck$ at step [13] and then to $j.kernel.lastUpdate$ at step [14].

Finally at the end of the loop, step [15], the newly created constraint instance j is added to *self.hasConstraints*, where *self* denotes the object to which the method *conIni* is applied.

The discussion presented so far motivates and presents a complete description of the semantics of the method $conIni()$ of the root class *Any*. Steps of the $conIni()$ method is presented in Table 6.2.

6.8 The Constraint Catalog: An Overview

In our approach, integrity constraints that are defined for an object database are considered as first class citizen, i.e., by representing them as objects. This is achieved by considering the *Constraint* class the link that on one side is connected with the constraint catalog and on the other side with the object schema. The structure of the constraint catalog and its relationship with the object schema is depicted in Figure 6.10.

The constraint catalog consists of two classes. The first class is the IC class which has the extension ICs. The second class is the Shell class which has the extension Shells. The extension ICs stores the results of compiling integrity constraints into IC objects. The extension Shells stores the results of compiling simplified forms of constraints into shells.

The link between the constraint catalog and the object schema is made indirectly via the *Constraint* class. On the one hand, there is a (1:1) relationship from the *Constraint* class to the Shell class. On the other hand there is a (1:m) relationship from the *Constraint* class to the root class *Any* of the object schema. In addition, there is a (1:1) relationship from the *Constraint* class to the *Kernel* class. This makes the main part of the structure of constraint instances consisting of an aggregation of a kernel and a shell.

The class *Constraint* has the extension *Constraints*. For every shell stored in the extension Shells, there is exactly one constraint instance in the extension *Constraints*. The *Kernel* class has no extension. A kernel is persistent as long as it is attached to a constraint instance.

The safe object schema is the one obtained by linking its root class *Any* with the *Constraint* class. Constraints restricting an object are represented by objects stored in the attribute *hasConstraints* of that object.

Constraint instances attached to individual objects that represent a non odd constraint have different kernels from those in the extension *Constraints*, whereas constraint instances

attached to individual objects that represent an odd constraint have the same kernel as those in the extension *Constraints*.

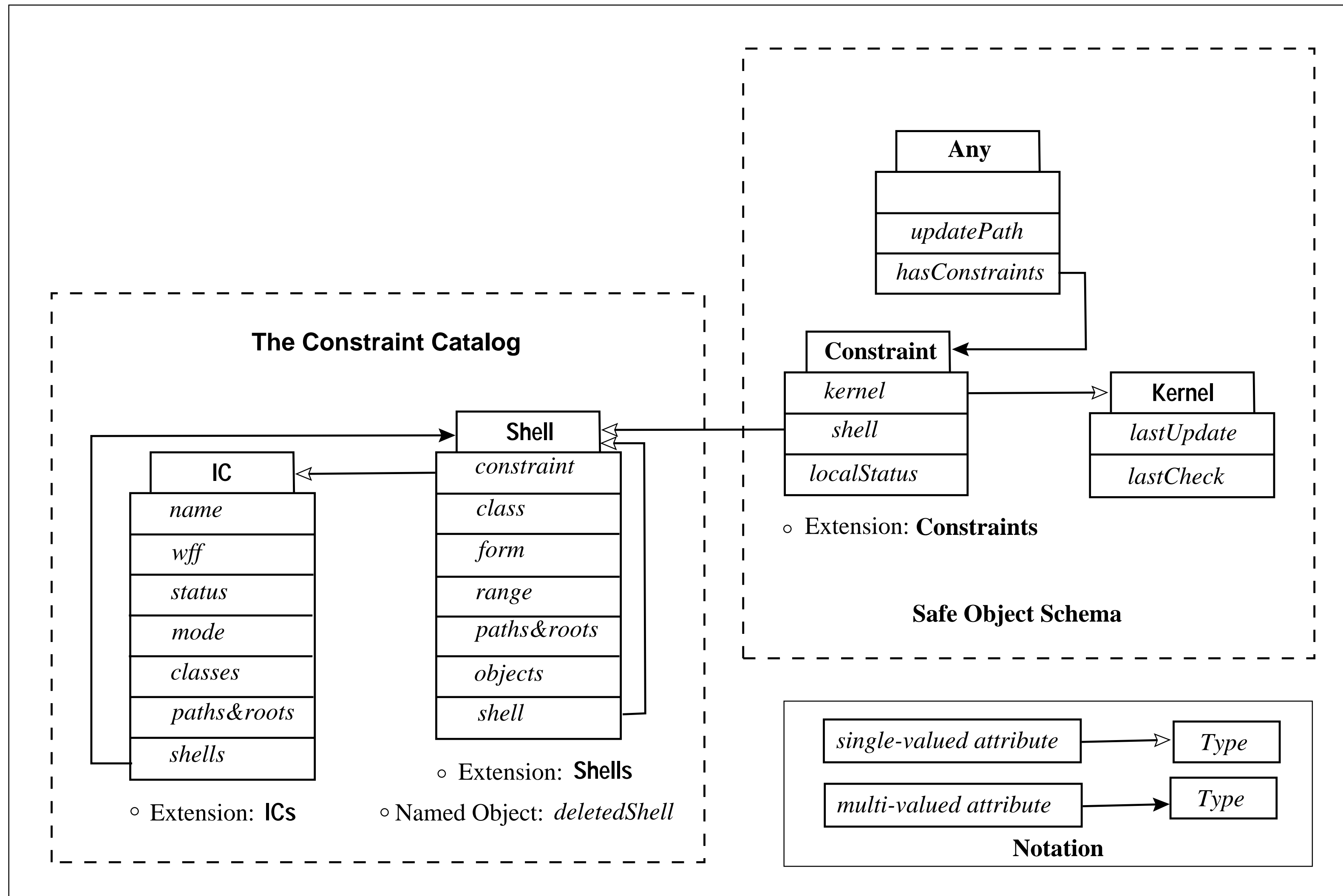

Figure 6.10: The constraint catalog and its relationship to the object schema.

Chapter 7

Consistency Management

In our approach, consistency management consists of the following tasks: (1) linking individual objects with constraint instances; (2) checking constraint instances of individual objects; (3) maintaining constraint kernels; (4) enabling and disabling of constraints; (5) consistency maintenance; and (6) adding and deleting constraints to and from the constraint catalog.

In Section 6.7, the task (1) above has been discussed in detail. There we have presented the method *conIni*() for initiating the attribute *hasConstraints* of individual objects. In this chapter, we present how tasks (2) through (6) are carried out and can be integrated seamlessly into the dynamic part of the object schema.

This chapter is organized as follows. In Section 7.1, we present grounds upon which constraint instances are checked. In our approach, users have a means to check consistency at three different levels, namely, at the level of the constraint instance, an object, or a set of objects that are updated in a transaction. This will be presented in Section 7.2. Sections 7.3 through 7.8 are devoted to answer the question of how tasks (3) through (5) above are handled in our approach. Then, in Section 7.9, we present an application. Finally, in Section 7.10, we discuss in detail the topic of adding and removing constraints from the constraint catalog.

As a running example, we will use the set of interrelated objects of the *Professor*, *Section* and *Course* classes shown in Figure 6.2. Shells and kernels of constraint instances of the constraint *W1* for those objects are shown in Figure 6.6. To facilitate the presentations of examples in this chapter we integrate Figures 6.2 and 6.6 into Figure 7.1.

7.1 Checking Constraint Instances

We check the validity of a constraint instance according to whether its shell is nested or flat. Thus, we have two cases. The first case, corresponding to flat instances, is presented in Subsection 7.1.1. The second case, corresponding to nested instances, is presented in Subsection 7.1.3.

7.1.1 Flat Constraint Instance

Flat constraint instances are those which have flat shells. Flat constraint instances can be characterized by the fact that the shell attribute of their shell part, *shell.shell* has the value *nil*. Let i be a flat instance. To check i, we evaluate the ground instance $i.shell.form[self/o]$, which is obtained by replacing the parameterized variable *self* with object o in the form stored in $i.shell.form$.

For example, object *vpr1* in Figure 7.1 has the constraint instance that consists of the kernel $K(W1|vpr1)$ and the shell $S(W1|pr)$. This constraint instance is labeled by *i3*. From Figure 5.3, we have $S(W1|pr).shell = nil$. From the update scenario 6.5, at time instance t_2, $K(W1|vpr1).lastCheck < K(W1|vpr1).lastUpdate$. Therefore, *i3* is an unchecked flat instance. Thus, to check *i3* we evaluate the ground instance $i3.shell.form[self/vpr1]$:

$$(vpr1 \in Professors) \rightarrow (\exists se \in vpr1.teaches)$$
$$(\forall co \in \{se.isSectionOf\})(co.hasPrerequisites = \emptyset).$$

Before we present how nested constraint instances can be checked, we need some definitions and notations to facilitate the presentation.

7.1.2 Inner and Core Constraint Instances

In this subsection, we first introduce the concepts of inner and core shells and then introduce the concepts of inner and core constraint instances.

Definition 7.1.1 (Inner and Core Shell) Let i be a nested constraint instance. A shell j is said to be an *inner shell* of i if

- $i.shell.shell = j$, or
- $k.shell.shell = j$ for some instance k such that $i.shell.shell = k$.

If S is an inner shell of i such that $S.shell = nil$ then S is said to be a *core shell* of i. □

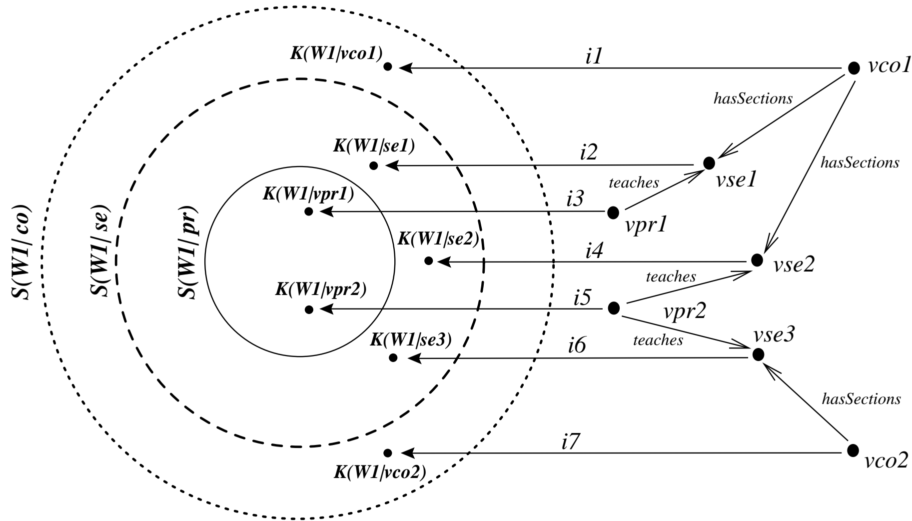

Figure 7.1: The integration of Figures 6.2 and 6.6.

For example, in Figure 7.1 the constraint instances labeled by *i1*, *i2*, *i4*, *i6*, and *i7* are nested. Instances *i1* and *i7* have the shell $S(W1|co)$. From Figure 5.3, we have that $S(W1|co).shell$ is $S(W1|se)$, thus $S(W1|se)$ is an inner shell of *i1* and *i7*. Similarly, $S(W1|pr)$ is an inner shell of *i1*, *i2*, *i4*, *i6* and *i7*. Also, we have $S(W1|pr).shell = nil$. Thus, $S(W1|pr)$ is a core shell of each of the constraint instances *i1*, *i2*, *i4*, *i6*, and *i7*.

Definition 7.1.2 (Inner Instance) Let $i \in o.hasConstraint$ be a nested instance. If for some instance $j \in o'.hasConstraints$, j has an inner shell of i and o' is related to o by the relationship that is described by the string stored in $i.objects$, then j is said to be an *inner instance* of i. The set of all inner instances of i is denoted by $allInner(o, i)$ and defined as follows:

$$allInner(o,i) = \{< o', j > \mid o' \in i.shell.objects[self/o] \wedge j \in o'.hasConstraints \wedge i.shell.shell = j.shell\}$$

□

For example, in Figure 7.1, the constraint instance *i3* is an inner instance of *i2*, and *i2* is an inner instance of *i1*. Constraint instance *i6* is not an inner instance of *i1* because *vse3* and *vco1* objects are not interrelated. In contrast, *i6* is an inner instance of *i7* because *vse3* object is related to *vco1* object by the relationship described in *i7.shell.objects*. In Figure 5.3 on Page 97, this relationship is "*hasSections*".

Definition 7.1.3 (Core Instance) Let $i \in hasConstraints$. If an instance j is a inner instance of i and the shell of j is a core shell of i then j is said to be a *core instance* of i. The set of all core instances of i is denoted by $allCore(o, i)$ and defined as follows:

$$allCore(o,i) = \begin{cases} \{< o, i >\}, & \text{if } i \text{ is flat;} \\ \bigcup_{< o',j > \in allInner(o,i)} allCore(o', j) & \text{if } i \text{ is nested.} \end{cases}$$

□

For example, in Figure 7.1 constraint instance *i3* is a core instance of *i2* and *i1*. Also *i5* is a core instance of *i4*, *i6*, *i1*, and *i7*.

7.1.3 Nested Constraint Instance

We recall that nested constraint instances are those which have nested shells. The nested constraint instances can be characterized by the fact that the shell attribute of their shell part refers to another shell, that is $shell.shell \neq nil$. As we have shown in Section 6.3, checking of nested instances in the same way as of flat instances leads to redundancy. For that reason, we introduced the *Kernel* class and used its attributes *lastUpdate* and *lastCheck* to avoid redundant checking. This has been shown in Example 6.3.1. This is also the rationale behind the way we propose to check nested constraints in the following lemma.

▶ **Lemma 7.1.4 (Checking Constraint Instances)** Let i be an unchecked nested constraint instance and o be an object such that $i \in o.hasConstraints$. Let $Inst(o, i)$ denote the ground instance $i.shell.form[self/o]$. Let $unchkCore(o, i)$ be the set of pairs $< o', j >$ in $allCore(o, i)$ such that $j.kernel.lastCheck < i.kernel.lastUpdate$. Then

$$Inst(o,i) \equiv \bigwedge_{< o',j > \in unchkCore(o,i)} j.shell.form[self/o'] \tag{7.1}$$

□

It other words, Lemma 7.1.4 states that in order to check a nested instances that have the state *unchecked*, it is sufficient to check each core instance j of i such that the time instance stored in $i.kernel.lastUpdate$ comes *after* the time instance stored in $j.kernel.lastCheck$. Formally, this can be stated as follows:

$$j.kernel.lastCheck < i.kernel.lastUpdate.$$

▶ **Example 7.1.5 (Checking Nested Instances)** The set of unchecked core instances of instance i, as defined in Lemma 7.1.4, depends on the time at which it is evaluated. In other words, $unchkCore(o, i)$ is not only a function of o and i but also of the time instance at which the evaluation is carried out. Let $unchkCore_t(o, i)$ denote the evaluation of $unchkCore(o, i)$ at time instance t. Consider the update scenario shown in Figure 6.5 on Page 108 for interrelated objects of Figure 7.1. From Figure 7.1, $i3$ and $i5$ are core instances of $i2$ and $i4$, respectively. Also $i2$ and $i4$ are inner instances of $i1$. Thus $i3$ and $i5$ are core instances of $i1$:

$$allInner(vco1, i1) = \{< vse1, i2 >, < vse2, i4 >\}$$
$$allCore(vco1, i1) = \{< vpr1, i3 >, < vpr2, i5 >\}$$

From the update scenario, at time instance t_8 we have the following:

$$i3.kernel.lastCheck < i1.kernel.lastUpdate$$
$$i5.kernel.lastCheck > i1.kernel.lastUpdate$$

Thus $unchkCore(vco1, i1) = \{< vpr1, i3 >\}$. Thus, to check the nested constraint instance $i1$ at time instance t_8 we evaluate the ground instance $i3.shell.form[self/vpr1]$ which is the same result as the one we obtained in Example 6.3.1. ■

Proof of Lemma 7.1.4

Let i be an unchecked nested constraint instance. From (5.33), $Inst(o, i)$ is equivalent to the conjunction of ground instances of all core instances of i:

$$Inst(o, i) \equiv \bigwedge_{<o',j> \in allCore(o,i)} j.shell.form[self/o'] \tag{7.2}$$

Let $chkCore(o, i) = allCore(o, i) - unchkCore(o, i)$. Now according to the semantics given to attributes of the *Kernel* class we have for each $< o', j >$ in $chkCore(o, i)$ that the time instance of last valid check of j comes *after* the time instance of the last an update occurred to o and potentially violate i:

$$j.kernel.lastCheck < i.kernel.lastUpdate.$$

Therefore

$$\bigwedge_{<o',j> \in chkCore(o,i)} j.shell.form[self/o'] \equiv \mathbf{T} \tag{7.3}$$

Hence, the lemma follows from (7.2) and (7.3). ■

This shows that to check a nested instance, we only need to check the unchecked core instances of it.

```
    check(o : Any, t : string) : boolean
[1]  begin
[2]  j : Constraint;
[3]  o' : Any;
[4]  if (self.checked() ∨ self.disabled()) then return(true);
[5]  if self.shell.shell = nil then  /* self is a flat instance */
[6]     if formEvaluation(self.shell.form) then
[7]        self.kernel.lastCheck ⟵ t;  /* mark self as checked */
[8]        return(true);  /* self is successfully checked */
[9]     else return(false);  /* self is a violated instance */
[10]    endif;
[11]  else
         /* self is a nested instance and */
         /* therefore find objects related to o */
[12]     for each o' ∈ queryEvaluation(self.shell.objects) do
            /* find core instances of self */
[13]        for each j ∈ o'.hasConstraints do
               /* check core instance j */
[14]           if (self.shell.shell = j.shell∧
[15]              j.kernel.lastCheck < self.kernel.lastUpdate) then
[16]              if ¬j.check(o', t) then return(false);
[17]           endif
[18]        endfor
[19]     endfor
         /* mark self as checked */
[20]     self.kernel.lastCheck ⟵ t;
         /* self is successfully checked */
[21]     return(true);
[22]  endif
[23]  end.
```

Table 7.1: The *check*() method of the *Constraint* class.

7.2 Different Levels of Consistency Checking

In our approach, users have a means to check database consistency at any point of time during the run of update transactions. This can be done at three different levels. At the lowest level, the validity of individual constraint instances can be checked. This is the task of a method of the *Constraint* class having the signature $check(Any, string) : boolean$. At the intermediate level, the consistency of individual objects can be checked. This is the task of a method of the root class *Any* having the signature $check(nil) : \{Constraint\}$. At the top level, consistency of objects that are updated during the run time of update transactions can be checked. At this level there is a method of the *Transaction* class having the signature $check(nil) : \{[Any, \{Constraint\}]\}$ that carries out this task.

The check method of the *Constraint* class is the main building block of the check method of the root class *Any* and the *Transaction* class. First we present the check method of the *Constraint* class and then the two other methods.

7.2.1 The *check*() Method of the *Constraint* Class

The check method of the *Constraint* class checks a given constraint instance i of an object o. The signature of the method is $check(Any, string) : boolean$. To verify the validity of i, the method is invoked on i with o and the method $currentTime()$ as values of the parameters $i.check(o, currentTime())$. The invocation of the method $i.check(o, currentTime())$ returns either *true*, which means that i is valid, or *false* meaning that i is violated.

The steps of the method are presented in Table 7.1. If i is in the *checked* state or it is disabled, then there is nothing to do and the method returns *true*. This is carried out by the first if-statement at step [4]. Otherwise, i is checked according to whether it is a flat, steps [5]–[10], or a nested instance, steps [12]–[21]. In both cases, if i is valid then its current state is changed to *checked* state. This is carried out by assigning the time instance at which the method is invoked on i to $i.kernel.lastCheck$. This is carried out either at step [7] or step [20]. If i is a nested instance, then each core instance j of i in $o'.hasConstraints$ is obtained and checked, where o' is an object related to o by the relationship described in $i.shell.objects$. This is carried out by the nested loop delimited by steps [12] and [19]. If $j.check(o', t)$ is *false*, then the method stops at this stage and returns *false* indicating that i is violated, step [16]. If $j.check(o', t)$ is *true*, then the state of j is changed implicitly to *checked* state inside $j.check(o', t)$ and the loop continues with another core instance of i until all core instances of i are successfully checked. In this case, the method arrives at step [21] and returns *true*,

```
    check(nil) :{Constraint}
[1]  begin
[2]  i : Constraint;
[3]  violatedConInsts : {Constraint};
[4]  t : string;
[5]  t ⟵ currentTime();
[6]  violatedConInsts ⟵ {nil};
[7]  for each i ∈ self.hasConstraints do
[8]      if ¬i.check(self, t) then violatedConInsts ⟵+ i;
[9]  endfor
[10] return(violatedConInsts);
[11] end.
```

Table 7.2: The *check*() method of the root class *Any*.

indicating that the nested instance is successfully checked.

In Table 7.1, we use the four methods, *checked*(), *disabled*(), *formEvaluation*(), and *queryEvaluation*(). The methods *checked*(), *disabled*() are used to observe the state and status of a constraint instance. The methods, *formEvaluation* and *queryEvaluation* evaluate the form and the relationship stored in attributes *form* and *objects*, respectively of a constraint instance. The four methods will be introduced in the sequel.

7.2.2 The *check*() Method of the Root Class *Any*

The check method of the root class *Any* checks *all* instances that are stored in the attribute *hasConstraints* of a given object *o*. The method has the signature *check*(*nil*) : {*Constraint*}. To check the consistency of *o*, the method is invoked on *o* with *o.check*(). Then *o.check*() returns either the empty set which means that all constraint instances in *o.hasConstraints* are valid, or a set of violated constraint instances. The steps of the method are presented in Table 7.2.

```
    check(nil) : {[Any, {Constraint}]}
[1] begin
[2] o : Any;
[3] inconsistentObjects : {[Any, {Constraint}]};
[4] for each o ∈ self.updatedObjects do
[5]     inconsistentObjects ←+ [o, o.check()];
[6] endfor
[7] return(inconsistentObjects);
[8] end.
```

Table 7.3: The *check*() method of the *Transaction* class.

7.2.3 The *check*() Method of the *Transaction* Class

The check method of the *Transaction* class checks consistency of all objects that are updated in a given transaction T. We recall that each transaction has an attribute named *updatedObjects* that stores all objects that are updated in that transaction. The signature of the check method is $check(nil) : \{[Any, \{Constraint\}]\}$. To check the consistency of all objects of $T.updateObjects$, the check method is invoked on T with $T.check()$. Then $T.check()$ returns either $\{[nil, \{nil\}]\}$, which means that objects of $T.updatedObjects$ are consistent, or a set of tuples $[o, A]$ each consisting of an inconsistent object o and a set A of violated constraint instances. The steps of the method is presented in Table 7.3.

7.3 Maintaining Constraint Kernels

As we have discussed in Sections 6.3 and 7.1, constraint kernels are employed to control the checking of nested instances such that the evaluation of necessarily valid core instances are avoided. Another function of constraint kernels, presented in Section 6.5, is that they are used to define the *checked* and *unchecked* states of constraint instances.

In this section we present how the values of *lastCheck* and *lastUpdate* attributes of kernels are maintained.

7.3.1 Maintaining the *lastCheck* Attribute

Maintaining attribute *lastCheck* is an easy task. Every time a constraint instance i is checked by invoking the method *check*() on i with o and *currentTime*() as values for parameters of the method, the time instance at which the method is invoked is assigned to *i.kernel.lastCheck* and to each *j.kernel.lastCheck* such that j is an inner instance of i. This is carried out directly before *i.check*(o, *currentTime*()) returns *true*. If *i.check*(o, *currentTime*()) returns *false*, then nothing happens, *i.kernel.lastCheck* remains the same as before invoking the check method on i, that is, i remains in the *unchecked* state.

7.3.2 Maintaining the *lastUpdate* Attribute

Maintaining the attribute *lastUpdate* of constraint instances needs some careful attention. Let j_k be an instance in o'_k.*hasInstances*, $k \in [1, n]$, and suppose that each j_k is an inner instance of i. According to the semantics of the check method of the *Constraint* class, if i is in the *checked* state then all inner instances of i are also in the *checked* state. Now suppose that the state of i is changed to *unchecked* because of an update to o occurred and potentially violates i. From Lemma 7.1.4, this implies that at least one instance of j_k, $k \in [1, n]$ is potentially violated by the update to o and thus its state must be changed accordingly to the *unchecked* state. Otherwise we have the situation that according to the values stored in the kernel, the constraint instance has the state *checked* whereas it is actually in the *unchecked* state.

Maintaining the attribute *lastUpdate* is the task of a method of the *Constraint* class named *setLastUpdate*. The signature of the method is *setLastUpdate*(*Any*, *string*) : *nil*. After each time an update is carried out to an object o and potentially violates an instance i in *o.hasConstraints*, the method is invoked on i with o and *currentTime*() as values for parameters of the method *i.setLastUpdate*(o, *currentTime*()).

The description of the method is presented in Table 7.4. The method maintains the attribute *lastUpdate* of the constraint instance i according to whether i is a flat, steps [5]–[6], or a nested instance, steps [7]–[12]. In both cases, the method sets the value of *lastUpdate* of i to be the time instance at which the method is invoked on i, step [4]. After this the method tests whether i is a flat or a nested instance, step [5]. If i is a flat instance then the method returns *nil*. If i is nested then the method evaluates each inner instance j of i one at a time; steps [8]–[10], and maintains the value of *j.kernel.lastUpdate* by invoking itself on j with *j.setLastUpdate*(o', t), step [11]. Thus, at the end of the method the value of the attribute *lastUpdate* of i and each of its inner instance is set to the same time instance t.

```
     setLastUpdate(o : Any, t : string) : nil
[1]  begin
[2]  j : Constraint;
[3]  o' : Any;
[4]  self.kernel.lastUpdate ⟵ t;
[5]  if self.shell.shell = nil then /*  self is a flat Instance */
[6]     return(nil);
[7]  else /* self is a nested Instance */
          /* find related objects to o */
[8]       for each o' ∈ queryEvaluation(self.objects[self/o]) do
              /* find inner instances of self */
[9]           for each j ∈ o'.hasConstraints
[10]              if j.shell = self.shell.shell then
                      /* maintain lastUpdate of the inner instance j */
[11]                  j.setLastUpdate(o', t);
[12]          endfor
[13]      endfor
[14]  endif
[15]  end.
```

Table 7.4: The *setLastUpdate*() method of the *Constraint* class.

▶ **Example 7.3.1 (Maintaining Kernels)** Consider the scenario of Example 6.2.2. In this scenario we maintain the times of the last update and the last check carried out on an object without paying attention to updates and checks that occurred to its related objects. This is shown in Figure 6.6. Thus, at time instance t_4, constraint instances *i3* of *vpr1* object and *i2* of *vse1* object are in the *checked* state. However, because *i3* and *i2* are inner instances of *i1* and that update occurred *after* the last check happened to *i2* and *i3*, the constraint instances *i2* and *i3* should both be in the *unchecked* state. This situation is rectified in Figure 7.2, in which constraint kernels are maintained according to the semantics of the methods *check*() and *setLastUpdate*() of the *Constraint* class. In Figure 7.2 primed time instances indicate the time and the object at which one of these methods is invoked. For example the symbol t'_4 means that at time instance t_4, the method *setLastUpdate* is invoked on the instance *i1* with *vco1* and t_4 as values for its parameters *i1*.*setLastUpdate*(*vco1*, t_4). Thus

according to the semantics of the *setLastUpdate*() method, the value of *lastUpdate* attribute of each inner instances of *i1* is set to t_4. Similarly t'_8 means that at time instance t_8, the *check*() method is invoked on the instance *i1* with *vco1* and t_8 as values for its parameters, *i1.check*(*vco1*, t_8), thus according to the semantics of the *check*() method, the value of the *lastCheck* attribute of each inner instance of *i1* is set to t_8. ■

7.3.3 Methods *checked*() and *unchecked*()

To observe the state of a given constraint instance, i.e., whether it is in the *checked* or *unchecked* state, the *Constraint* class has two methods:

(1) The first method is of the signature *checked*(*nil*) : *boolean*. Given a constraint instance i, i.*checked*() returns *true* if i is in the *checked* state, and *false* if i is in the *unchecked* state. The method *checked*() is a direct implementation of the semantics of *checked* state presented in Section 6.5.

(2) The second method is of the signature *unchecked*(*nil*) : *boolean*. Given a constraint instance i, i.*unchecked*() returns *true* if i is in the *unchecked* state, and *false* if i is in the *checked* state. The method *unchecked*() can be implemented by taking the the negation to *self.check*(), i.e., $\neg$*self.check*(), where *self* is the constraint instance on which the method *unchecked*() was invoked.

7.4 Enabling and Disabling Constraint Instances

Let W be an integrity constraint and $IC(W)$ the IC object in the extension ICs that represents W. Therefore, $IC(W).name = \texttt{W}$. The value stored in $IC(W).status$ is called the *global* status of the constraint W. The status $IC(W).status$ is global in the sense that if $IC(W).status$ is *enabled* (resp. *disabled*) then constraint W is (resp. is not) applied to each (resp. any) object of the classes in $IC(W).classes$.

Let i be a constraint instance in $o.hasConstraints$ such that $i.shell.constraint.name = \texttt{W}$. The value stored in $i.localStatus$ is called the *local* status of the constraint W w.r.t. o. The status $i.localStatus$ is local in the sense that if $i.localStatus$ is *enabled* (resp. *disabled*) then W is (resp. is not) applied to *the* object o.

Thus, a constraint has *one* global status but *many* local statuses. The manipulations of the global and local statuses are subject to the following rules:

Time	Constraint Instances									
	i3		*i2*		*i1*		*i4*		*i5*	
	$K(W1 \mid vpr1)$		$K(W1 \mid vse1)$		$K(W1 \mid vco1)$		$K(W1 \mid vse2)$		$K(W1 \mid vpr2)$	
	last−Update	*last−Check*	*last−Update*	*last−Check*	*last−Update*	*last−Check*	*last−Update*	*last−Check*	*last−Update*	*last−Check*
t_0	t_o	$t_{-\infty}$	$t_{-\infty}$	$t_{-\infty}$	$t_{-\infty}$	$t_{-\infty}$	$t_{-\infty}$	$t_{-\infty}$	$t_{-\infty}$	$t_{-\infty}$
t_1	t_1	⋮	t'_1	⋮	⋮	⋮	⋮	⋮	⋮	⋮
t_2	⋮	t_2	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
t_3	⋮	⋮	⋮	t_3	⋮	⋮	⋮	⋮	⋮	⋮
.....	....	......	....	......	....	......	....	......	....	......
t_4	t_4	⋮	t_4	⋮	t'_4	⋮	t_4	⋮	t_4	⋮
t_5	⋮	⋮	⋮	⋮	⋮	⋮	t'_5	⋮	t_5	⋮
t_6	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	t_6	⋮
t_7	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	t_7
t_8	⋮	t_8	⋮	t_8	⋮	t'_8	⋮	t_8	⋮	⋮

Figure 7.2: The maintenance of kernels of Figure 7.1.

rule 1 If the global status of W is *enabled* then this means that the constraint W is applied to *each* object of the classes that are restricted by W, except objects where the local status of W w.r.t. each of them is *disabled*. Thus, according to this rule, for some constraint instance i, it is possible that $i.localStatus$ is *disabled* whereas $i.shell.constraint.status$ is *enabled*.

rule 2 If the global status of W is *disabled*, then the constraint W is not applied to any of the objects of classes that are restricted by W, regardless of the local status of W w.r.t. any of them. Thus, according to this rule, for some constraint instance i, it is not possible that $i.localStatus$ is *enabled* whereas $i.shell.constraint.status$ is *disabled*.

7.4.1 Global Status

Two methods of the *Transaction* class are responsible for global enabling and disabling a constraint.

The first method has the signature $setEnabled(string) : nil$. Let T be a transaction. The constraint W can be enabled in the transaction T by invoking *setEnabled* on T with the name of the constraint W as value of the parameter of the method $T.setEnabled(\texttt{W})$.

The second method is of the signature $setDisabled(string) : nil$. Let T be a transaction. The constraint W can be disabled in the transaction T by invoking *setDisabled* on T with the name of the constraint W as value of the parameter, $T.setDisabled(\texttt{W})$.

To observe the global status of a constraint, the root class *Any* has two methods. The first method is of the signature $enabled(string) : boolean$. Given a constraint W with name `W`, $enabled(\texttt{W})$ returns *true* if $IC(W).status$ is enabled and *false* otherwise, where $IC(W)$ is the IC object in ICs that represents the constraint W. The second method is of the signature $disabled(string) : boolean$ and has the dual semantics of that of $enabled()$.

7.4.2 Local Status

For enabling and disabling a constraint instance, there are two methods named *setEnabled* and *setDisabled* of the *Constraint* class. The two methods have the following signatures respectively:

$$setEnabled(Any) : string$$
$$setDisabled(Any) : string$$

The semantics of these methods is subject to the two rules stated in the beginning of this section and based on whether the constraint instance is flat or nested. Let i be a constraint instance in $o.hasConstraints$. Then we have two cases:

- i is a flat instance. In this case, if $i.shell.constraint.status$ is *enabled*, then invoking *setEnabled*(o) on i changes the value of $i.localStatus$ to *enabled*; and invoking *setDisabled*(o) on i changes the value of $i.localStatus$ to *disabled*.

- i is a nested instance. In this case, if $i.shell.constraint.status$ is *enabled*, then invoking *setEnabled*(o) on i changes the value of $i.localStatus$ and the local status of each of its inner instances to *enabled*; and invoking *setDisabled*(o) on i changes the value of $i.localStatus$ and the local status of each of its inner instance to *disabled*.

The method $i.setEnabled(o)$ and $i.setDisabled(o)$ return an error message if any of rules (rule 1) or (rule 2) above is violated. The description of the method $i.setEnabled(o)$ is presented in Table 7.5. The steps of the method $i.setDisabled(o)$ is very similar.

To observe the local status of constraint instance, the *Constraint* class has two methods. The first method is of the signature $enabled(nil) : boolean$. Given a constraint instance i, $i.enabled()$ returns *true* if $i.localStatus$ is enabled and *false* otherwise. The second method is of the signature $disabled(nil) : boolean$ and has the dual semantics of that of *enabled*().

7.5 Consistency Maintenance

In Section 6.1, we have presented a general method for maintaining consistency of an object base by using constraint shells. In Chapter 6, we have also shown how drawbacks of the method are avoided by considering kernels to be part of constraint instances. On these grounds, a method for consistency maintenance of an object base can be introduced.

For every update $up : o.A_1 \ldots A_n \theta v$, $\theta \in \{\longleftarrow, \overset{+}{\longleftarrow}, \overset{-}{\longleftarrow}\}$, in a transaction T we do the following steps:

(1) The set of potentially violated constraints instances, **PVCI** is determined using the following query against $o.hasConstraints$:

$$\mathbf{PVCI} = \{i \mid i \in o.hasConstraints \wedge updatePath \in i.shell.paths\&roots\} \qquad (7.4)$$

where

$$updatePath = \begin{cases} A_1 \ldots A_n, & \text{if } \theta \text{ is } \longleftarrow \\ +A_1 \ldots A_n, & \text{if } \theta \text{ is } \overset{+}{\longleftarrow} \\ -A_1 \ldots A_n & \text{if } \theta \text{ is } \overset{-}{\longleftarrow} \end{cases}$$

```
    setEnabled(o : Any) : string
[1]  begin
[2]  j : Constraint;
[3]  o' : Any;
[4]  if self.shell.constraint.status = disabled then return(error message);
[5]  if self.shell.shell = nil then /* self is a flat instance */
[6]     self.localstatus ⟵ enabled;
[7]  else /* self is a nested Instance */
        /* find related objects to o */
[8]     for each o' ∈ queryEvaluation(self.objects[self/o]) do
           /* find inner instances of self */
[9]        for each j ∈ o'.hasConstraints
[10]          if j.shell = self.shell.shell then
              /* set the inner instance j to be enabled */
[11]             j.setEnabled(o');
[12]       endfor
[13]    endfor
[14] endif
[15] end.
```

Table 7.5: The *setEnabled*() method of the *Constraint* class.

(2) Each immediate instance i of **PVCI** is checked by $i.check(o, currentTime())$. If any of them is violated then the modification done by *up* to the object o is undone and the update is rejected.

(3) Each deferred constraint instance i of **PVCI** being in the state *checked* is moved to the state *unchecked* by $i.setLastUpdate(o, currentTime())$.

Finally, consistency of objects that are updated in the transaction T is checked by $T.check()$.

In our approach, the tasks of consistency control mentioned in the steps above can be integrated seamlessly with methods, application programs and update transactions. This is achieved by:

- adding an attribute named *updatePath* of type *string* to the root class *Any* (see the structure of class *Any* in Figure 6.10); and

- augmenting the dynamic part of the database schema with a set of methods called *control methods*.

Let o be an object. When an update of the form $up : o.A_1 \ldots A_n \theta v$ is carried out, $o.updatePath$ temporary stores the string σ :

$$\sigma = \begin{cases} A_1 \ldots A_n, & \text{if } \theta \text{ is } \longleftarrow \\ +A_1 \ldots A_n, & \text{if } \theta \text{ is } \overset{+}{\longleftarrow} \\ -A_1 \ldots A_n & \text{if } \theta \text{ is } \overset{-}{\longleftarrow} \end{cases}$$

Then the string stored in $o.updatePath$ will be used by the control methods to obtain constraint instances of $o.hasConstraints$ that are potentially violated by the update up.

There are three categories of control methods. Each category is characterized according to where its methods are located in the database schema. The methods of the first category are associated with the class root *Any*. Therefore, this category, is general and its control methods are inherited to all classes of the database schema. The control methods of the second category are associated with individual classes of the database schema. Signatures of control methods of this group are defined according to the structure of the class they are associated with. The third group is associated with the *Transaction* class and thus they are inherited to every update transaction. In Sections 7.6 through 7.8 we present control methods of the three categories.

7.6 Control Methods of the Root Class

Control methods of the root class *Any* provide the infrastructure for object manipulation and evaluation of queries or boolean formulas that are described as strings in the constraint catalog.

Object Manipulation For object manipulation, there are three control methods of signatures $assign(string, Any) : nil$, $add(string, Any) : nil$ and $remove(string, Any) : nil$. Let the path expression $A_1 \ldots A_n$, the value v and the object o be as defined in Section 3.5. The semantics of $o.assign(''A_1 \ldots A_n'', v)$ is the update statement $o.A_1 \ldots A_n \longleftarrow v$. The semantics of $o.add(''A_1 \ldots A_n'', v)$ is the update statement $o.A_1 \ldots A_n \overset{+}{\longleftarrow} v$. Analogously, the semantics of $o.remove(''A_1 \ldots A_n'', v)$ can be defined.

$$
\begin{array}{l}
\textbf{class} \quad Professor[\\
\quad \vdots \qquad \ldots \\
\quad salary : float, \\
\quad \vdots \qquad \ldots \\
\quad degree : [date : Date, \\
\qquad\qquad name : string, \\
\qquad\qquad university : [uniName : string, \\
\qquad\qquad\qquad\qquad address : [country : string, \\
\qquad\qquad\qquad\qquad\qquad\qquad city : string]], \\
\quad \vdots \qquad \ldots]
\end{array}
$$

Figure 7.3: The *Professor* class

Queries and Boolean Formulas The attributes *form* and *objects* of the Shell class are of type *string*. This means that we first have to convert the string stored in *form* and *objects* into a boolean expression and queries respectively so that we can evaluate them. For that purpose, there are two control methods of the root class *Any* of the following signatures:

$$formEvaluation(string) : boolean$$
$$queryEvaluation(string) : \{Any\}$$

Thus to evaluate the simplified form of a constraint instance $i \in o.hasConstraints$, we call $o.formEvaluation(i.shell.form)$. Similarly, to evaluate a query described in $i.shell.objects$ as a string, we call $o.queryEvaluation(i.shell.objects)$. In the evaluation task of both methods the parameter *self* in $i.shell.form$ is considered to be the object o.

7.7 Control Methods of Individual Classes

Control methods of this category *wrap* updates to object attributes with tasks of consistency control. The semantics of control methods depends on whether an attribute is of atomic or tuple or set type. In the following we present control methods of each of these types.

7.7.1 Atomic Type

For each atomic attribute A of a class C, there is one control method of C with the signature $A(v : t) : string$, where t is the type of A and $t \in \{integer, float, string, boolean\}$. The

template of the control method for atomic attributes is presented in Table 7.6. We recall that the root class *Any* has an attribute named *updatePath* of type *string*. The semantics of the control method $A(v)$ for the atomic attribute A consists of:

- constructing an update path by concatenating the string $'A'$ or $'.A'$ to the string stored in *self.UpdatePath*, steps [3]–[6];
- assigning a value v to the attribute A, step [7];
- obtaining potentially violated constraint instances, *PVCI*, step [9];
- checking immediate constraint instances of *PVCI* and undoing the update if one of them is violated, steps [11]–[15]; and
- setting the state of deferred constraint instances of *PVCI* to *unchecked*, step [16].

▶ **Example 7.7.1 (Control Methods of Atomic Attributes)** Consider the part of the structure of the *Professor* class shown in Figure 7.3. The attribute *degree* represents details of the scientific degree of a professor. The type of the attribute *degree* is a nested tuple-structured type. The components of these nested types are self-explanatory. For the atomic attribute *salary* of the *Professor* class there is a control method of the signature *salary*(*float*) : *string*. Also, for every atomic component of the nested tuple-structured attribute *degree*, there is a control method. These attributes are *name*, *uniName*, *country*, and *city* and their control methods are of the following signatures respectively:

$$name(string) : string$$
$$uniName(string) : string$$
$$country(string) : string$$
$$city(string) : string$$

The implementation of these methods can be generated automatically from the template that is presented in Table 7.6. For instance, the code of the method *salary*(*float*) : *string* can be obtained from that template by replacing "*A*" and "*atomicType*" by "*salary*" and "*float*" respectively. ■

```
     A(v : atomicType) : string
[1]  begin
[2]  updateTime : string, i : Constraint, PVCI : {Constraint};
/*  constructing update path  */
[3]  if self.updatePath =" "
[4]  then self.updatePath ⟵ self.updatePath +" A"
[5]  else self.updatePath ⟵ self.updatePath +" .A"
[6]  endif;
/*  assigning the value v to the atomic attribute A  */
[7]  self.assign(updatePath, v);
[8]  updateTime ⟵ currentTime();
 /* finding potentially violated constraint instances */
[9]  PVCI ⟵ {i | i ∈ self.hasConstraints ∧ updatePath ∈ i.shell.paths}
[10]  for each i ∈ PVCI do
        /*  checking immediate constaint instances  */
[11]     if i.mode = immediate then
[12]        if ¬i.check(self, t) then
[13]           undo();
[14]           return(error message);
[15]        endif
        /* Marking deferred constraint instances to unchecked  */
[16]     else i.setLastUpdate(t)
[17]     endif
[18]  endfor
[19]  self.updatePath =" ";
[20]  end.
```

Table 7.6: The template for the control method of atomic attributes.

7.7.2 Set-Structured Type

For each set-structured attribute A in a class C, there are *three* control methods of C with signatures $A{+}(v : t) : \mathit{string}$, $A{-}(v : t) : \mathit{string}$, and $A(v : \{t\}) : \mathit{string}$ where $\{t\}$ is the type of A. The semantics of $A(v)$ is the same as in the case of atomic attributes. The semantics of $A{+}(v)$ (resp. $A{-}(v)$) is the same as in the case of atomic attributes but

(1) the symbol "+" (resp. "−") is added to the prefix of the string stored in *updatePath* (see the definition of constrained paths in Subsection 4.5.6); and

(2) the assign operator in step [8] of Table 7.6 is replaced by the adding (resp. removing) operator.

The template of the control method of $A{+}(v : t) : \mathit{string}$ is presented in Table 7.7.

The rationale behind introducing three forms is to consider the three possible ways of updating set-valued attribute A, namely adding a value to A, removing a value from A and assigning a set of values to A. For instance, for the attribute *hasSections* of the *Course* class there are three control methods,

$$\mathit{hasSections}(\mathit{seSet} : \{\mathit{Section}\}) : \mathit{string}$$
$$\mathit{hasSections}{+}(\mathit{se} : \mathit{Section}) : \mathit{string}$$
$$\mathit{hasSections}{-}(\mathit{se} : \mathit{Section}) : \mathit{string}$$

Thus, for an object *vco* of class *Course*, we can update values stored in *vco.hasSections* by assigning a new set S, as in $\mathit{vco.hasSection}(S)$, or adding or deleting one section s, as in $\mathit{vco.hasSection}{+}(s)$ and $\mathit{vco.hasSection}{-}\,(s)$ respectively.

7.7.3 Tuple-Structured Type

For each tuple-structured attribute A in a class C, there are *two* control methods in C with signatures $A(\mathit{nil}) : C$ and $A(v : t) : \mathit{string}$, where t is the type of A. This can be realized by using method overloading. The task of the first form, $A(\mathit{nil}) : C$ consists only of maintaining attribute *updatePath* as in the case of atomic attributes; see steps [3]–[6] of Table 7.7. The semantics of the second form, $A(v : t) : \mathit{string}$ is the same as in the case of atomic attributes.

Introducing two forms of control methods, in the case of tuple structured attributes, is necessary to allow for updating the whole structure of the attribute A as well as any of its components. For instance, for the attribute *dateOfBirth* of class *Employee*, there are two control methods, $\mathit{dateOfBirth}(\mathit{nil}) : \mathit{Employee}$ and $\mathit{dateOfBirth}(\mathit{Date}) : \mathit{string}$. Thus we can

```
     A+(v : Type) : string
[1]  begin
[2]  updateTime : string, i : Constraint, PVCI : {Constraint};
/* constructing update path */
[3]  if self.updatePath ="  "
[4]  then self.updatePath ⟵ self.updatePath +" A"
[5]  else self.updatePath ⟵ self.updatePath +" .A"
[6]  endif;
[7]  self.updatePath ⟵" +" + self.updatePath;
/*  adding the value v to the set-valued attribute A  */
[8]  self.add(updatePath, v);
[9]  updateTime ⟵ currentTime();
 /*  finding potentially violated constraint instances  */
[10]  PVCI ⟵ {i | i ∈ self.hasConstraints ∧ updatePath ∈ i.shell.paths}
[11]  for each i ∈ PVCI do
          /*  checking immediate constaint instances  */
[12]      if i.mode = immediate then
[13]         if ¬i.check(self, t) then
[14]            undo();
[15]            return(error message);
[16]         endif
          /* Marking deferred constraint instances to unchecked  */
[17]      else i.setLastUpdate(t)
[18]      endif
[19]  endfor
[20]  self.updatePath ="  ";
[21]  end.
```

Table 7.7: The template for the control method $A+()$ of attribute A of a set type.

update the date of birth of an employee as a whole as in the statement (7.5) or through one of its components as in statements (7.6) – (7.8).

$$e.dateOfBirth(25.3.1967) \tag{7.5}$$
$$e.dateOfBirth().day(25) \tag{7.6}$$
$$e.dateOfBirth().month(3) \tag{7.7}$$
$$e.dateOfBirth().year(1967) \tag{7.8}$$

7.8 Control Methods of the *Transaction* class

For each persistent root r defined in the object schema there is one or more control method of *Transaction* class. The number and the signature of the control method depend upon the type of r :

(1) If r is an object of class C, then there is one control method in the *Transaction* class with signature $r(nil) : C$. For a given transaction T, the invocation of $r()$ on T $T.r()$ returns the persistent root r.

(2) If r is a set of objects of class C, then there are three control methods in the *Transaction* class with signatures $r{+}(C) : string$, $r{-}(C) : string$, and $r(C) : string$. Let o be an object of C. The semantics of $r{+}(o)$ consists of two parts. The first part is the same as the semantics of the $A{+}$ control method of set-structured attributes. The second part is that of enabling constraint instances of $o.hasConstraints$ which have a range that coincides with r.

(3) If r is of a tuple type then in this case r is treated in the same way as of tuple-structured attributes.

The template of the control method of $r{+}(v : t) : string$ is presented in Table 7.8.

Point (2) above needs some further explanation. We recall from Section 6.6 that each constraint instance of the *Constraints* extension is initiated such that all instances of explicit range, that is, $shell.range \neq nil$, are locally disabled. Let o be an object that has one of such instances, say i. Then the local status of $i.localStatus$ is *disabled* until the point of time at which o belongs to the range stored in $i.shell.range$. This will happen if o is attached to the persistent root described in $i.shell.range$. At that point, the constraint instance is locally enabled. For instance, let pr be an object of the *professor* class. If $i \in pr.hasConstraints$

represents a key constraint on the extension *Professors*, then we need not to maintain i as long as pr is not in *Professors*.

The semantics of the method $r-(o)$ is the dual of that of the method $r+(o)$. Thus the method $r-(o)$ is disabling every constraint i, $i \in o.hasConstraints$, such that the range of i is the persistent root r. Analogously, the semantics of $r(S)$, where S is a set of objects of class C, can be described.

7.9 Consistency Control: Application Example

Consider the transaction *SalaryRaise* shown in Figure 7.9. The semantics of *SalaryRaise* is to raise the salary of each employee whose salary is less than 1000 by a certain percentage. In steps [2] and [3], variables *vEmp* of type *Employee* and *newSalary* of type *float* are declared. Step [5] is an iterator, that is, for each object of type *Employee* satisfying the given condition, steps [6] and [7] are executed.

The transaction *SalaryRaise* as given in Figure 7.9 is safe in the following sense. Each object satisfing the condition of the iterator of step [5] is already stored in the database and hence it was already associated with constraint instances at its time of creation. In step [7], the *salary* attribute is modified by the *salary*() control method and not by the update statement, $vEmp.salary \longleftarrow newSal$. The salary is modified and potentially violated constraint instances, **PVCI**, in $o.hasConstraints$ are handled according to the semantics of control methods of attributes of atomic types:

- immediate constraint instances of **PVCI** are checked and the transaction is rolled back if one of them is violated, and
- the state of deferred constraint instances of **PVCI** is modified to *unchecked*.

During the iteration every modified object is added implicitly to $T.updatedObjects$. In step [9], the consistency of objects of $T.updatedObjects$ is checked. If the invocation of the *check*() method on transaction *SalaryRaise* returns an empty set then, the transaction does not violate consistency of any of the modified objects. Otherwise, the transaction *SalaryRaise.check*() returns a set of tuples of the same type as the one used in the declaration of variable *inc* in step [4]. In this set each tuple consists of an inconsistent object and the set of violated constraint instances. In this example we do not follow the traditional way of rolling-back the *whole* transaction but we undo only the modifications of inconsistent objects. This is carried out by letting *inc* run on the output of *SalaryRaise.check*() and undoing the modifications to *inc.object*.

```
     r+(o : C) : string
[1]  begin
[2]  updateTime : string, i : Constraint, PVCI : {Constraint};
/*   constructing update path  */
[3]  self.updatePath ⟵" +" +" r";
/*   adding the object o to the persistent root r */
[4]  self.add(updatePath, v);
[5]  updateTime ⟵ currentTime();
/*   finding potentially violated constraint instances */
[6]  PVCI ⟵ {i | i ∈ self.hasConstraints ∧ updatePath ∈ i.shell.paths};
[7]  for each i ∈ PVCI do
         /*  Marking constraint instances that have range r to enabled  */
[8]      if i.shell.range =" r" then i.setEnabled(o);
         /*  checking immediate constaint instances  */
[9]      if i.mode = immediate then
[10]         if ¬i.check(o, t) then
[11]             undo();
[12]             return(error message);
[13]         endif
         /*  Marking deferred constraint instances to unchecked */
[14]     else i.setLastUpdate(t);
[15]     endif
[16] endfor
[17] self.updatePath =" ";
[18] end.
```

Table 7.8: The template for control method $r+()$ of persistent root r of a set type.

```
[1]  begin Transaction SalaryRaise(per : float)
[2]  vEmp : Employee;
[3]  newSal : float;
[4]  inc : [object : Any, violatedInstances : {Constraint}];
[5]  for each vEmp ∈ {o | o ∈ Employees ∧ o.salary < 1000} do
[6]       newSal ⟵ vEmp.salary * per + vEmp.salary;
[7]       vEmp.salary(newSal);
[8]  endfor
[9]  for each inc ∈ SalaryRaise.check() do
[10]      inc.object.undo();
[11]  endfor
[12]  end SalaryRaise
```

Table 7.9: An example of safe transaction.

7.10 Constraint Manipulation

In our approach, the decoupling of the specification of integrity constraints and update transactions is achieved as follows. On the one hand, the constraint catalog is used as central repository for integrity constraints. On the other hand, control methods are used instead of primitive modify operations in update transactions. Thus, in our approach users are capable to change constraint specifications without changing application programs and update transactions. In the rest of this section we present how adding or deleting a constraint can be done without any modification to application programs or update transactions.

7.10.1 Adding a Constraint to the Constraint Catalog

Adding a new constraint to constraint catalog can be described as follows.

- Transform the user-defined constraint into the canonical form W as we have described in Section 4.4.
- Compile canonical constraint W into an IC object, $IC(W)$ and adding $IC(W)$ to the extension ICs. This step is described in detail in Chapter 4.
- Transform the constraint W into boolean form as we have described in Section 5.2.

$$
\begin{aligned}
deletedShell = [\ & constraint : nil, \\
& form : \text{'' ''}, \\
& class : \{\}, \\
& paths\&roots : \{\}, \\
& range : \text{'' ''}, \\
& objects : \text{'' ''}, \\
& shell : nil]
\end{aligned}
$$

Figure 7.4: Named object *deletedShell*.

- Obtain simplified forms of the constraint W by using the method of Section 5.3. Let the simplified forms of W be $(W|1), \ldots, (W|n)$.
- Compile each $(W|i)$ into a shell $S(W|i)$ and add $S(W|i)$ to the extension Shells. This step is described in detail in Section 5.5.
- Link the IC object $IC(W)$ with each of the shells $S(W|i)$, $i \in [1, n]$ such that

$$
IC(W).shells = \{(W|1), \ldots, (W|n)\} \text{ and}
$$
$$
S(W|i).constraint = IC(W).
$$

- Create an instance *inst* of class *Constraint* for each object $S(W|i)$; initiate attributes of *inst* as described in Section 6.7, and add the instance *inst* to the *Constraints* extension.

7.10.2 Removing a Constraint from the Constraint Catalog

Removing a constraint from the constraint catalog should imply that all instances of that constraint that are associated with individual objects are to be removed too. Removing these instances directly from objects they associated are with will be either incomplete or hard to achieve; this is because there is no single criterion that can characterize these objects. To overcome this problem, we define a named object of type Shell called *deletedShell*. The values of attributes of *deletedShell* are the default values of their types as shown in Figure in 7.4. When a constraint W is removed from the constraint catalog we make each of its shells refer to the *deletedShell* object. This will be propagated to all instances of W. Then all instances that have *deletedShell* will not be maintained until they are deleted from objects they are associated with at transactions commit time.

Removing a constraint W from the constraint catalog can be described as follows.

- Identify the IC object $IC(W)$ in the ICs extension such that $IC(W).name =$ W.
- Assign the named object *deletedShell* to each shell S in $IC(W).shells$. Then remove the shell S from the Shells extension.
- Remove the IC object $IC(W)$ from the ICs extension.
- Remove each constraint instance i from the *Constraints* extension such that

$$i.shell = deletedShell.$$

Then each constraint i associated with an object o is removed from $o.hasConstraint$ at transactions commit time. These instances can be characterized by the fact that their shells refer to the named object *deletedShell*.

Chapter 8

Conclusions

In this chapter we first briefly summarize the contributions of this thesis. Then we discuss the possible extensions to the work and topics we are currently investigating for future work.

8.1 Contributions

In this thesis we present a novel approach for consistency management for object databases. In this approach, integrity constraints are treated as "first class citizens". Thus integrity constraints can in principle be manipulated in the same way as all other objects in an object database. This enables to transfer the explicit manipulation of constraints as known from relational DBMS to object databases.

The approach described in this thesis improves on related approaches by having all of the following features in one framework.

Object-orientation. The constraint catalog is an object database. Also the structure of constraints is designed by using the basic concepts of object data models which are supported by most object database systems. In addition, tasks of the consistency management are encapsulated in individual objects. Thus, on the one hand, the constraint catalog can be integrated with object database management systems seamlessly. On the other hand, constraint objects can be manipulated in the same way as the individual objects of an object base. Moreover, the constraint catalog is designed such that constraints imposed on a superclass are inherited to its subclasses. Similarly, simplified forms of constraints imposed on a superclass are inherited to its subclasses.

Specification. For understandability and to facilitate their management, integrity constraints should be specified declaratively. In this thesis, a logic-based constraint specification language is presented. The language has a clear syntax and semantics. This helps to provide a simple yet a complete formal treatment to path expressions, primitive modification operations and update transactions. This in turn helps in adapting the definition of range restricted constraints of relational databases to cope with concepts of object databases and the style of object persistency taken in this thesis, i.e., persistence via reachability.

Interrelated Objects. The constraint specification language is defined according the type system of the underlying object database management system. Thus, as long as the type system allows to define objects that have complex structure, the constraints imposed on these objects can be specified by the language. Also the optimization method as well as the compilation of constraints into objects are based on syntax and not on semantics grounds. Thus, our approach can maintain constraints imposed on interrelated objects with complex structures.

Transactions. By using the property that constraints are first class citizens, the constraints imposed on an object are associated with that object throughout its life time. The consistency of an object are handled by integrating the consistency control methods with the dynamic part of that object. Tasks of the consistency control are achieved by wrapping the primitive modification operations with the control methods. Thus, tasks of the consistency control are orthogonal to the type of transactions supported by the underlying object database system.

Efficiency. The efficiency of constraint checking is supported as follows. First, we check simplified forms of constraints rather than the original specified constraints. The optimization method proposed in this thesis handles cases which cannot be handled by the conventional optimization techniques. Second, by considering the constraint structure as the aggregation of the kernel and the shell a serious kind of redundancy is avoided. In our approach, by using information stored in kernels, no duplication may arise due to the evaluation of the same simplified or a part of it. Third, we present a set of observations that a user should take into account when he/she derives simplified forms for key constraints and boolean forms for constraints. Fourth, in our approach inverse relationships are maintained by the modification operations. Thus, the efficiency of maintaining inverse relationships is guaranteed. Moreover, there is no need to maintain referential constraints explicitly.

Integrity Independence. To change constraint specifications without changing of application programs we have to decouple them. This is achieved as follows. On the one hand, the constraint catalog is used as central repository of integrity constraints and simplified forms. On the other hand, control methods are used instead of the primitive modification operations in update transactions. We have described in detail how a constraint can be added or deleted from the constraint catalog. Moreover, as constraints are first class citizen, this can be done in the same way as of adding or deleting an object.

Persistency Style. Essentially, there are two approaches for an object to be persistent: explicit persistence and persistence by reachability. The later style is more general than the former. Explicit persistence can be implemented by using persistence by reachability. In this thesis, we used the most general approach for object persistence, that is persistence by reachability. Thus, our approach can be applied to the other styles of object persistence.

Inconsistency. Traditional approaches for consistency management follow the principle of "all or nothing", they do not permit for inconsistent objects to persist. They cannot detect inconsistent objects in an object database unless all integrity constraints are checked against the whole object base. Thus, if the traditional approaches permitted inconsistency, then the space of inconsistent objects would be the whole object base. In our approach, the evaluation of a constraint can be determined from values stored in the attributes *lastCheck* and *lastUpdate* of its kernel. Every constraint either is in the *checked* or *unchecked* state. According to the semantics given to these states, the unchecked state can be seen as a *disjunction* of two states, namely the *unknown* state and the *invalid* state. Thus, in our approach we reduce the space of inconsistent objects to only those object that are in unchecked state.

Update Granularity. In this thesis, object consistency is considered at the lowest level of update granularity. This is achieved through the semantics of control methods of individual classes. For every class, there is a set of control methods that wrap updates to attributes of that class with tasks of consistency control. The semantics of these control methods depend on whether attribute is of an atomic or tuple or set type. Thus, our approach considers database consistency at different levels of update granularity for objects such as updating a simple attribute or a complex attribute or the whole state of an object. This is in addition to adding (resp. deleting) an object(s) to (resp. from) a set-valued persistent root.

Disabling and Enabling of Constraints. Our approach provides the tools for enabling and disabling constraints at various levels of abstraction like the whole database, or a class or a specific object. This is achieved by managing the global status and the local status of constraints by different categories of control methods. If the global status of a constraint is enabled, then this means that the constraint is applied to all objects of the classes restricted by the constraint, except objects where the local status of the constraint w.r.t. each of them is disabled. If the global status of a constraint is disabled, then the constraint is not applied to any of objects of the classes restricted by the constraint, regardless of the local status of the constraint w.r.t. any of them.

In the following section, we present possible extensions to the work and topics we are currently exploring for future work.

8.2 Future Work

Future work will concentrate on (1) the possible extensions to the work, (2) a prototype implementation, (3) transferring the results of this thesis to other data models and language and (4) schema evolution.

Extensions to the Work

Several lines of extensions of the work presented in this thesis are possible to follow. First, class methods are not considered in the constraint specification language presented in this thesis. It is possible that the salary of an employee is a derived attribute. This means that the salary will be calculated by using values of other attributes of the employee object such as the age and the rank of the employee. In this example, the salary will be modeled as method and not as an attribute. Thus, specification of a constraint on the salary attribute should consider salary as a method and not as an attribute. This example shows the need to include class methods of observer sort in the language. Second, considering observer methods in path expression will require a thorough revision of all results proved in Chapter 4 and Chapter 5 concerning path expressions, modifications operations and the optimization method. Third, we want to incorporate an important class of integrity constraints to our approach, namely aggregation constraints. This entails an adaptation to the constraint specification language to cope with this sort of constraints, and the investigation of methods for improving their checking. Fourth, although that non linear path expression [LV97, KM94] will not enhance the expressive power of the specification language presented in this thesis, we

think, considering them will be an interesting extension to the work. Finally, the optimization method proposed in this thesis does not handle existential constraints. Thus, we want to investigate how other optimization techniques that handle this class of constraints, e.g., optimization by redundancy, can be incorporated in our approach.

A Prototype Implementation

A prototype implementation is needed to investigate the additional costs and performance decrease resulting from the overhead of constraint checking. A certain overhead is of course unavoidable because we are adding new functionality to an OODBMS. One possible approach for prototyping is to use a precompiler to extend e.g. ODMG schemata with facilities for constraint checking. We have to closely investigate implementation variants, their storage overhead and their effects on runtime performance. This helps in answering the question of what design principles has a user to follow to make use of our approach.

Transfer to other Data Models and Languages

The results of this thesis have to be transferred to existing OODBMS data models and languages. Current developments in object model standardization (e.g., the object constraint language (OCL) of the unified modeling language (UML) [Bur95, Mul99]) have increased the interest in declarative constraint languages for object models. We have to develop a suitable constraint language which is as expressive as possible while still allowing automatic simplification as described in this thesis (this could be a suitable subset of OCL).

To meet the market demands, object-relational databases have evolved by extending relational DBMS technology with various object-oriented concepts. In [Oak95, OS99] we have addressed the problems of consistency management for relational databases. There we follow the same technique we presented here but by considering relational databases rather than object databases. Moreover, our approach is based on basic concepts of object-oriented data models. These basic concepts are also supported by object-relational databases [SM96]. Thus, we believe that it will be an interesting field for future work to do the following: First, to integrate the approach of [Oak95, OS99] with the one we presented here and then transferring the results of this integration to object-relational databases.

Schema Evolution

The aim of this thesis is to constitute a conceptual framework for integrity checking for *ad hoc* transactions in OODBMS. In contrast to so-called *a-priori* or *certified* transactions, the DBMS has to check integrity at runtime because transaction code cannot be assumed as safe. For such a framework, it is necessary to have constraints materialized in the database as metadata and to have a runtime constraint checking. As a by-product, such a framework will allow the evolution of constraints in a database as part of schema evolution. This evolutionary aspect should be investigated in future work, too.

Appendix A

Syntax for Class Definition

The syntax for class definition is as follows:

< classDecl >:= *class* < className > : < classType >
| *class* < className > : < superClasses >
| *class* < className > : < superClasses > < classType >

< classType >:= ”[” < attrDecl > ”]”

< attrDecl >:= < attrName > : < type >
| < attrName > : < type > , < attrDecl >

< type > := < atomic > | < tuple > | < set >

< atomic > := *integer* | *float* | *string* | *boolean* | < className >

< tuple > := ”[” < type > ”]”

< set > := ”{” < type > ”}”

< superClasses > := < className > | < className > , < superClasses >

< persistentRoot > := *name* < persistentRootName >: < type >

< method > := *method* < methodSignature > < methodBody >

< methodSignature >=:< methodName > "(" < paramDecl > ")" :< outputType >

< paramDecl > := < paramName > :< type >
| < paramName > :< type > , < paramDecl >

< outputType >:=< type >

< methodBody > := begin < code > end.

Appendix B

Syntax for Method Implementation

The syntax that will be presented in this appendix is a slight modification to the ones given in [AHV95] (Page 564) and in [BS98].

Let $M(t_0, \ldots, t_n) : t$ be a signature of a method, where M is the name of the method; t_0 is the class name to which the method is defined; $t_0, \ldots, t_n, t$ and t are types of input and output of the method, respectively. The implementation in the language **PL** for the method of signature $M(t_0, \ldots, t_n) : t$ is defined as follows:

```
M(p1, ... , pn)
begin
      var
      instruction
      return(p)
end.
```

where $p_1 \ldots p_n, p$ are parameters of types $t_1 \ldots, t_n, t$ respectively; and *self* is considered as the first parameter of the method. Here *self* refers to an object of the class t_0, that is, the one to which the method M is defined.

There are three parts of method implementations in the language **PL**. The first part, *var*, in which variables to be used in the *instruction* part are declared. The second part, *instruction*, in which the semantics of a method is coded into instructions of the language **PL**. The final part, `return`(p) which returns a value of type of t, this value may be *nil* if the type t is *nil*.

The syntax of the first two parts, *var* and *instruction*, in the language **PL** are given

below.

The syntax of *var*:

$var := x : t; \mid y : t', var$

The syntax of *instruction*:

$$
\begin{aligned}
instruction := {} & x \leftarrow y; \\
& \mid x.a \leftarrow y; \\
& \mid x \leftarrow y.a; \\
& \mid x \leftarrow m'(y_1, \ldots, y_m); \\
& \mid control; \\
& \mid iterator;
\end{aligned}
$$

control := <u>`return`</u>(*instruction*)
  | <u>`if`</u> *condition* <u>`then`</u>
    instruction
  <u>`endif`</u>
  | <u>`if`</u> *condition* <u>`then`</u>
    instruction
  <u>`else`</u>
    instruction
  <u>`endif`</u>

iterator := <u>`for`</u> <u>`each`</u> $x \in setValuedExpression$ [<u>`where`</u> *condition*] <u>`do`</u>
    instruction
  <u>`endfor`</u>

Bibliography

[AB95] Serge Abiteboul and Catriel Beeri. The Power of Languages for the Manipulation of Complex Values. *VLDB Journal*, 4(4):727–794, 1995.

[ABD+89] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The Object-Oriented Database System Manifesto. In W. Kim, J.-M. Nicolas, and S. Nishio, editors, *Proc. of the 1st Int. Conf. on Deductive and Object Oriented Databases, Kyoto, Japan*, pages 40–57, Amsterdam, December 1989. North-Holland.

[AHV95] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, Reading, Mass., 1995.

[BCB97] E. Bertino, B. Catania, and S. Bressan. Integrity Constraint Checking in Chimera. In V. Gaede, A. Brodsky, O. Günther, D. Srivastava, V. Vianu, and M. Wallace, editors, *Constraint Databases and Applications, Proc. of the 2nd Int. Conf. on Constraint Database Systems, CDB'97, Delphi, Greece, January 1997*, volume 1191 of *Lecture Notes in Computer Science*, pages 160–186, Berlin, 1997. Springer-Verlag.

[BD95] V. Benzaken and A. Doucet. Thémis: A Database Programming Language Handling Integrity Constraints. *VLDB Journal*, 4(3):493–517, July 1995.

[BdBZ93] H. Balsters, R.A. de By, and R. Zicari. Typed Sets as a Basis for Object-Oriented Database Schemas. In O. Nierstrasz, editor, *Proc. of the 7th European Conference on Object-Oriented Programming (ECOOP'93), July 26-30, 1993, Kaiserslautern, Germany*, volume 707 of *Lecture Notes in Computer Science*, pages 161–184. Springer-Verlag, 1993.

[BDK92] F. Bancilhon, C. Delobel, and P. Kanellakis, editors. *Building an Object-Oriented Database System – The Story of O_2*. Morgan Kaufmann Publishers, San Mateo, CA, 1992.

[BEST98] F. Bry, N. Eisinger, H. Schütz, and S. Torge. SIC: Satisfiability Checking for Integrity Constraints. In P. Fraternali, U. Geske, C.Ruiz, and D.Seipel, editors, *Proceedings of the 6th International Workshop on Deductive Databases and Logic Programming (DDLP'98)*, number 22 in GMD Report, pages 25 –36, 1998.

[BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, MA, 1987.

[BLR92] V. Benzaken, C. Lécluse, and P. Richard. Enforcing Integrity Constraints in Database Programming Languages. In A. Albano and R. Morrison, editors, *Proc. of the 5th International Workshop on Persistent Object Systems, 1–4 September 1992, San Miniato (Pisa), Italy*, Workshops in Computing, pages 282–299. Springer-Verlag, 1992.

[BM86] F. Bry and R. Manthey. Checking Consistency of DB Constraints: a Logical Basis. In Wesley W. Chu, Georges Gardarin, Setsuo Ohsuga, and Yahiko Kambayashi, editors, *Proc. of the 12th International Conference on Very Large Databases, August 25-28, Kyoto,Japan*, pages 13–20. Morgan Kaufmann, 1986.

[BM88] E. Bertino and D. Musto. Correctness of Semantic Integrity Checking in Database Management Systems. *Acta Informatica*, 26:25–57, 1988.

[BM91] M. Bouzeghoub and E. Métais. Semantic Modeling of Object Oriented Databases. In Guy M. Lohman, Amílcar Sernadas, and Rafael Camps, editors, *17th International Conference on Very Large Data Bases, September 3-6, 1991, Barcelona, Catalonia, Spain, Proceedings*, pages 3–14. Morgan Kaufmann, 1991.

[BMP91] C. Bauzer-Medeiros and P. Pfeffer. Object Integrity Using Rules. In P. America, editor, *Proc. of the 5th European Conf. on Object Oriented Programming (ECOOP'91), july 15-19, 1991, Geneva, Switzerland*, volume 512, pages 219–230. Springer-Verlag, 1991.

[BS96] V. Benzaken and X. Schaefer. Ensuring Efficiently the Integrity of a Persistent Object Store via Abstract Interpretation. In R. Connor and S. Nettle, editors,

Proc. of the 7th International Workshop on Persistent Object Systems, Cape-May, USA, pages 72–87. Morgan Kaufmann, May 1996.

[BS97] V. Benzaken and X. Schaefer. Static Integrity Constraint Management in Object-Oriented Database Programming Languages via Predicate Transformers. In M. Aksit and S. Matsuoka, editors, *Proc. of the 11th European Conference on Object-Oriented Programming, (ECOOP'97), June 9-13, 1997, Jyväskylä, Finland*, volume 1241 of *Lecture Notes in Computer Science*, pages 60–84, Berlin, 1997. Springer-Verlag.

[BS98] V. Benzaken and X. Schaefer. Static management of integrity in object-oriented databases: Design and implementation. In H. Schek, F. Saltor, I. Ramos, and G. Alonso, editors, *Advances in Database Technolog - Proc. of the 6rd Int. Conf. on Extending Database Technology (EDBT'98), March 23-27, 1998,Valencia, Spain*, volume 1377 of *Lecture Notes in Computer Science*, pages 311–325, 1998.

[Bur95] R. Burkhardt. *UML – Unified Modeling Language.* Addison-Wesley, Bonn, 1995.

[BV94] N. Bassiliadesand and I. P. Vlahavas. Modelling Constraints with Exceptions in Object-Oriented Databases. In P. Loucopoulos, editor, *Entity-Relationship Approach - ER'94, Business Modelling and Re-Engineering, 13th International Conference on the Entity-Relationship Approach, Manchester, U.K., December 13-16, 1994*, volume 881 of *Lecture Notes in Computer Science*, pages 205–222, Berlin, 1994. Springer.

[Car88] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76(2):138–164, 1988.

[Cat91] R. G. G. Cattell. *Object Data Management – Object-Oriented and Extended Relational Database Systems.* Addison-Wesley, Reading, MA, 1991.

[CB97] R. G. G. Cattell and D. K. Barry, editors. *The Object Database Standard: ODMG-93, Release 2.0.* Morgan Kaufmann Publishers, San Francisco, CA, 1997.

[CB00] R. G. G. Cattell and D. K. Barry, editors. *The Object Database Standard: ODMG 3.0.* Morgan Kaufmann, 2000.

[CF97] S. Ceri and P. Fraternali. *Designing Database Applications with Objects and Rules — The IDEA Methodology.* Addison-Wesley, Reading, MA, 1997.

[CFMS95] S. Castano, M. G. Fugini, G. Martella, and P. Samarati. *Database Security.* ACM Press, Addison-Wesley, Reading, MA, 1995.

[CFPT92] S. Ceri, P. Fraternali, S. Paraboschi, and L. Tanca. Constraint Enforcement through Production Rules: Putting Active Databases to Work. *Bulletin of the IEEE Technical Committee on Data Engineering*, 15(1–4):10–14, December 1992.

[CFPT94] S. Ceri, P. Fraternali, S. Paraboschi, and L. Tanca. Automatic Generation of Procedure Rules for Integrity Maintenance. *ACM Transactions on Database Systems*, 19(3):367–422, September 1994.

[Che76] P. P. Chen. The Entity-Relationship Model – Towards a Unified View of Data. *ACM Transactions on Database Systems*, 1(1):9–36, March 1976.

[Cho92] J. Chomicki. History-less Checking of Dynamic Integrity Constraints. In F. Golshani, editor, *Proc. of the 8th IEEE Int. Conf. on Data Engineering, ICDE'92, Tempe, Arizona, USA, February 2–3, 1992*, pages 557–564, Los Alamitos, CA, 1992. IEEE Computer Society Press.

[Cho94] J. Chomicki. Temporal Integrity Constraints in Relational Databases. *Bulletin of the IEEE Technical Committee on Data Engineering*, 17(2):33–37, June 1994.

[CKS] S. Conrad, H.-J. Klein, and K.-D. Schewe, editors. *Integrity in Databases – Proc. of the 7th Int. Workshop on Foundations of Models and Languages for Data and Object, Schloss Dagstuhl, Sept. 16-20, 1996.* Preprint 4, Institut für Technische Informationssysteme, Universität Magdeburg, 1996.

[Cod90] E. F. Codd. *The Relational Model for Database Management, Version 2.* Addison-Wesley, Reading, MA, 1990.

[Coo97] R. Cooper. *Object Databases: An ODMG Approach.* International Thomson Computer Press, Boston, 1997.

[CW90] S. Ceri and J. Widom. Deriving Production Rules for Constraint Maintenance. In D. McLeod, R.Sacks-Davis, and H.-J. Schek, editors, *Proc. of The 16th Inter-*

national Conference on Very Large Data Bases, August 13-16, 1990, Brisbane, Queensland, Australia, pages 566–577. Morgan Kaufmann, 1990.

[Dat90] C. J. Date. *An Introduction to Database Systems*, volume 1. Addison-Wesley, Reading, MA, 5 edition, 1990.

[Dem92] R. Demolombe. Syntactical Characterization of a Subset of Domain-Independent Formulas. *Journal of ACM*, 39(1):71–94, January 1992.

[Deß93] S. Deßloch. *Semantic Integrity in Advanced Database Management Systems.* PhD thesis, Universität Kaiserslautern, Fachbereich Informatik, September 1993.

[Deu91] O. Deux. The Story of O_2. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):91–108, March 1991.

[Día92] O. Díaz. Deriving Active Rules for Constraint Maintenance in an Object-Oriented Database. In A. M. Tjoa and I. Ramos, editors, *Proc. of the 3rd Int. Conf. on Database and Expert System Applications, DEXA'92, Valencia, Spain*, pages 332–337. Springer-Verlag, 1992.

[EGB93] S. M. Embury, P. M. D. Gray, and N. Bassiliades. Constraint Maintenance using Generated Methods in the P/FDM Object-Oriented Database. In N. W. Paton and M. H. Williams, editors, *Proc. of the 1st Int. Workshop on Rules in Database Systems, RIDS'93, 30 August - 1 September 1993, Edinburgh, Scotland*, Workshops in Computing, pages 364–381, Berlin, 1993. Springer-Verlag.

[EJK93] R. Elmasri, S. James, and V. Kouramajian. Automatic Class and Method Generation for Object-Oriented Databases. In S. Ceri, K. Tanaka, and S. Tsur, editors, *Deductive and Object Oriented Databases, Proc. of the 3rd Int. Conf., DOOD'93, Phoenix, Arizona, USA, December 1993*, volume 760 of *Lecture Notes in Computer Science*, pages 395–414, Berlin, 1993. Springer-Verlag.

[EL90] J. L. Encarnação and P. C. Lockemann, editors. *Engineering Databases — Connecting Islands of Automation through Databases.* Springer-Verlag, Berlin, 1990.

[EN94] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems.* Benjamin/Cummings, Redwood City, 2 edition, 1994.

[EN00] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems.* Addison-Wesley, Reading, Mass., 3 edition, 2000.

[End72] H. B. Enderton. *A Mathematical Introduction to Logic.* Academic Press, New York, 1972.

[FG95] C. Fahrner and G.Vossen. A Survey of Database Design Transformations Based on the Entity-Relationship Model. *Data & Knowledge Engineering*, 15:213–250, 1995.

[FMP97] C. Fahrner, T. Marx, and S. Philippi. DICE: Declarative Integrity Constraint Embedding Into the Object Database Standard ODMG-93. *Data & Knowledge Engineering*, 23(1):119–145, 1997.

[GA93] P. W. P. J. Grefen and P. M. G. Apers. Integrity Control in Relational Database Systems — An Overview. *Data & Knowledge Engineering*, 10:187–223, 1993.

[GD94] A. Geppert and K.R. Dittrich. Specification and Implementation of Consistency Constraints in Object-Oriented Database Systems: Applying Programming-by-Contract. ifi-94.14 1, Department of Computer Science, University of Zurich, October 1994.

[Ger96a] M. Gertz. An Extensible Framework for Repairing Constraint Violations. In S. Conrad, H.-J. Klein, and K.-D. Schewe, editors, *Integrity in Databases – Proc. of the 7th Int. Workshop on Foundations of Models and Languages for Data and Object, Schloss Dagstuhl, Sept. 16-20, 1996*, number 4, pages 41–56. Institut für Technische Informationssysteme, Universität Magdeburg, 1996.

[Ger96b] M. Gertz. *Diagnosis and Repair of Constraint Violations in Database Systems*, volume 19 of *Dissertationen zu Datenbanken und Informationssystemen.* infix-Verlag, Sankt Augustin, 1996.

[GGGM98] P. Godfrey, J. Grant, J. Gryz, and J. Minker. Integrity Constraints: Semantics and Applications. In J. Chomicki and G. Saake, editors, *Logics for Databases and Information Systems*, chapter 9, pages 265–306. Kluwer Academic Publishers, Boston, 1998.

[GGJ93] C. A. Goble, A. Glowinski, and K. G. Jeffrey. Semantic Constraints in a Medical Information System. In M. Worboys and A. F. Grundy, editors, *Advances in*

Databases, Proc. of the 11th British National Conf. on Databases, (BNCOD 11), July, 1993, Keele, UK, volume 696 of *Lecture Notes in Computer Science*, pages 40–57. Springer-Verlag, 1993.

[GJ82] J. Grant and E. B. Jacobs. On the Family of Generalized Dependency Constraints. *Journal of ACM*, 29(4):986–997, October 1982.

[GJ91] N. H. Gehani and H. V. Jagadish. Ode as an Active Database: Constraints and Triggers. In G. M. Lohman, A. Sernadas, and R. Camps, editors, *Proce. of the 17th International Conference on Very Large Data Bases, September 3-6, 1991, Barcelona, Catalonia, Spain*, pages 327–336. Morgan Kaufmann, September 1991.

[GL93] M. Gertz and U. W. Lipeck. Deriving Integrity Maintaining Triggers from Transition Graphs. In A. Elmagarmid and E. Neuhold, editors, *Proc. of the 9th IEEE Int. Conf. on Data Engineering, ICDE'93, Vienna, Austria, 19–23 April 1993*, pages 22–29, Los Alamitos, CA, 1993. IEEE Computer Society Press.

[GL95] M. Gertz and U. W. Lipeck. "Temporal" Integrity Constraints in Temporal Databases. In J. Clifford and A. Tuzhilin, editors, *Recent Advances in Temporal Databases – Proc. of the Int. Workshop on Temporal Databases*, Workshops in Computing, pages 77–92, Berlin, September 1995. Springer-Verlag.

[GMN84] H. Gallaire, J. Minker, and J. M. Nicolas. Logic and Databases: A Deductive Approach. *ACM Computing Surveys*, 16(2):151–184, June 1984.

[Gog94] M. Gogolla. *An Extended Entity-Relationship Model — Fundamentals and Pragmatics*, volume 767 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1994.

[Gre93] P. W. P. J. Grefen. Combining Theory and Practice in Integrity Control: A Declarative Approach to the Specification of a Transaction Modification Subsystem. In Rakesh Agrawal, Sean Baker, and David Bell, editors, *Proc. of the 19th International Conference on Very Large Data Bases, (VLDB'93), August 24–27, 1993, Dublin, Ireland.* Morgan Kaufmann, 1993.

[HS97] A. Heuer and G. Saake. *Datenbanken — Konzepte und Sprachen, 1. korrigierter Nachdruck.* International Thomson Publishing, Bonn, 1997.

[JJ91] M. A. Jeusfeld and M. Jarke. From Relational to Object-Oriented Integrity Simplification. In C. Delobel, M. Kifer, and Y. Masunaga, editors, *Proc. of the 2nd Int. Conf. on Deductive and Object-Oriented Databases (DOOD'91), December 1991, Munich, Germany*, volume 566 of *Lecture Notes in Computer Science*, pages 460–477. Springer-Verlag, 1991.

[JQ92] H. V. Jagadish and X. Qian. Integrity Maintenance in an Object-Oriented Database. In L.-Y. Yuan, editor, *Proc. of the 18th International Conference on Very Large Data Bases, August 23-27, 1992, Vancouver, Canada*, pages 469–480, San Mateo, Ca., USA, 1992. Morgan Kaufmann.

[Kim95] W. Kim, editor. *Modern Database Systems.* ACM Press, New York, NJ, 1995.

[KL89] M. Kifer and G. Lausen. F-logic: A Higher-Order Language for Reasoning about Objects, Inheritance, and Scheme. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 18(2):134–146, June 1989.

[KM94] A. Kemper and G. Moerkotte. *Object-oriented Database Management: Applications in Engineering and Computer Science.* Prentice-Hall, Inc., 1994.

[KMP92] A. Kemper, G. Moerkotte, and K. Peithner. Object-Orientation Axiomatised by Dynamic Logic. Technical Report 92-30, Technical University of Aachen (RWTH Aachen), 1992.

[KSS87] R. Kowalski, F. Sadri, and P. Soper. Integrity Checking in Deductive Databases. In P. M. Stocker, W. Kent, and P. Hammersley, editors, *Proc. of the 13th International Conference on Very Large Data Bases, September 1-4, 1987, Brighton, England*, pages 61–70. Morgan Kaufmann, September 1987.

[Laf82] G. M. E. Lafue. Semantic Integrity Dependencies and Delayed Integrity Checking. In *Eighth International Conference on Very Large Data Bases, September 8-10, 1982, Mexico City, Mexico, Proceedings*, pages 292–299. Morgan Kaufmann, 1982.

[LGS94] U. W. Lipeck, M. Gertz, and G. Saake. Transitional Monitoring of Dynamic Integrity Contraints. *Bulletin of the IEEE Technical Committee on Data Engineering*, 17(2):38–42, June 1994.

[LLOW91] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The ObjectStore Database System. *Communications of the ACM*, 34(10):50–63, October 1991.

[LV97] G. Lausen and G. Vossen. *Models and Languages of Object-Oriented Databases.* Addison-Wesley, Harlow, UK, 1997.

[Man90] R. Manthey. Satisfiability of Integrity Constraints: Reflections on a Neglected Problem. In Jutta Goeers and Andreas Heuer, editors, *Integrity in Databases – Proc. of the 2th Int. Workshop on Foundations of Models and Languages for Data and Object, Aigen(Austria), Sept. 24-28, 1990*, number 90/3 in Preprint, pages 169–179. Institut für Informatik, Technische Universität Clausthal, 1990.

[MH89] W. W. McCune and L. J. Henschen. Maintaining State Constraints in Relational Databases: A Proof Theoretic Basis. *Journal of ACM*, 36(1):69–91, January 1989.

[ML91] G. Moerkotte and Peter C. Lockemann. Reactive Consistency Control In Deductive Databases. *TODS*, 16(4):670–702, 1991.

[Mot89] A. Motro. Integrity = Validity + Completeness. *ACM Transactions on Database Systems*, 14(4):480–502, December 1989.

[Mul99] R. J. Muller. *Database Design for Smarties — Using UML for Data Modeling.* Morgan Kaufmann Publishers, San Francisco, CA, 1999.

[Nic78] J.-M. Nicolas. First Order Logic Formalization for Functional, Multivalued and Mutual Dependencies. In *ACM SIGMOD International Conference on Management Of Data*, pages 40–46, New York, 1978. ACM.

[Nic82] J.-M. Nicolas. Logic For Improving Integrity Checking in Relational Data Bases. *Acta Informatica*, 18(3):227–253, 1982.

[NNJ94] B. Narasimhan, S. B. Navathe, and S. Jayaraman. On Mapping ER and Relational Models into OO Schemas. In R. A. Elmasri, V. Kourajian, and B. Thalheim, editors, *Entity-Relationship Approach — ER'93, Proc. of the 12th Int. Conf. on the Entity-Relationship Approach, Arlington, Texas, December 1993*, volume 823 of *Lecture Notes in Computer Science*, pages 403–413, Berlin, 1994. Springer-Verlag.

[Oak95] H. Oakasha. Enforcing State Constraints in Relational Databases without Their Specification in Update Transactions. Master thesis, Cairo University, 1995.

[OCS99a] H. Oakasha, S. Conrad, and G. Saake. Consistency Control in Object-Oriented Databases. Preprint 22, Fakultät für Informatik, Universität Magdeburg, September 1999.

[OCS99b] H. Oakasha, S. Conrad, and G. Saake. Consistency Management in Object-Oriented Databases. In A. de Miguel, E. Ferrari, G. Kappel, G. Guerrini, and I. Merlo, editors, *Proc. of the 1st ECOOP Workshop on Object-Oriented Databases, Lisbon, Portugal, June 15th, 1999*, pages 97–108, 1999.

[OCS00] H. Oakasha, S. Conrad, and G. Saake. Consistency Management in Object-Oriented Databases. *Journal of Concurrency: Practice and Experience*, 2000. Accepted.

[Orm98] L. V. Orman. Differential Relational Calculus for Integrity Maintenance. *IEEE Transactions on Knowledge and Data Engineering*, 10(2):328–341, March/April 1998.

[OS98a] H. Oakasha and G. Saake. Compiling State Constraints. Preprint 11, Fakultät für Informatik, Universität Magdeburg, May 1998.

[OS98b] H. Oakasha and G. Saake. Integrity Independence in Object-Oriented Database Systems. In M. H. Scholl, H. Riedel, T. Grust, and D. Gluche, editors, *Proc. of the 10th Workshop Grundlagen von Datenbanken, Konstanz, Germany*, number 63 in Konstanzer Schriften in Mathematik und Informatik, pages 94–98. University of Konstanz, May 1998.

[OS99] H. Oakasha and G. Saake. Foundations for Integrity Independence in Relational Databases. In T. Polle, T. Ripke, and K.-D. Schewe, editors, *Fundamentals of Information Systems*, chapter 10, pages 143–165. Kluwer Academic Publishers, Boston, 1999.

[ÖV91] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, Eaglewood Cliffs, NJ, 1991.

[QS87] X. Qian and D. Smith. Integrity Constraint Reformulation for Efficient Validation. In P. M. Stocker, W. Kent, and P. Hammersley, editors, *Proc. of the*

3th International Conference on Very Large Data Bases, September 1-4, 1987, Brighton, England, pages 417–426. Morgan Kaufmann, September 1987.

[QW86] X. Qian and G. Wiederhold. Knowledge-Based Integrity Constraint Validation. In W. W. Chu and G. Gardarin and S. Ohsuga and Y. Kambayashi, editor, *Proc. of the 12th International Conference on Very Large Data Bases, August 25-28, 1986, Kyoto, Japan*, pages 3–12. Morgan Kaufmann, August 1986.

[Rei80] R. Reiter. Equality and Domain Closure in First-Order Databases. *Journal of ACM*, 27(2):235–249, 1980.

[RSS96] K. A. Ross, D. Srivastava, and S. Sudarshan. Materialized View Maintenance and Integrity Constraint Checking: Trading Space for Time. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 25(2):447–458, June 1996.

[RSSS98] K. A. Ross, D. Srivastava, P. J. Stuckey, and S. Sudarshan. Foundations of Aggregation Constraints. *Theoretical Computer Science*, 193(1-2):149–179, Februrary 1998.

[Saa91] G. Saake. Descriptive Specification of Database Object Behaviour. *Data & Knowledge Engineering*, 6(1):47–74, 1991.

[SB92] S. B. Sparg and S. Bergman:. Semantic Constructs for a Persistent Programming Language. In A. Albano and R. Morrison, editors, *Proc. of the 5th Int. Workshop on Persistent Object Systems, 1–4 September, 1992, San Miniato (Pisa), Italy*, Workshops in Computing, pages 300–316, London, 1992. Springer-Verlag.

[SB97] D. Spelt and H. Balsters. Automatic Verification of Transactions on an Object-oriented Database. In S. Cluet and R. Hull, editors, *Database Programming Languages (DBPL-6), Proc. of the 6th Int. Workshop on Database Programming Languages, Colorado, USA*, volume 1369 of *Lecture Notes in Computer Science*, pages 396–412, Berlin, 1997. Springer-Verlag.

[SH99] G. Saake and A. Heuer. *Datenbanken — Implementierungstechniken.* MITP-Verlag, Bonn, 1999.

[SL88] G. Saake and U. W. Lipeck. Foundations of Temporal Integrity Monitoring. In C. Rolland et al., editors, *Proc. of the IFIP Working Conf. on Temporal Aspects in Information Systems*, pages 235–249, Amsterdam, 1988. North-Holland.

[SM96] M. Stonebraker and D. Moore. *Object-Relational DBMSs — The Next Great Wave.* Morgan Kaufmann Publishers, San Francisco, CA, 1996.

[SS89] T. Sheard and D. Stemple. Automatic Verification of Database Transaction Safety. *ACM Transactions on Database Systems*, 14(3):322–368, September 1989.

[Tar92] Z. Tari. On the Design of Object Oriented Databases. In G. Pernul and A. M. Tjoa, editors, *Entity-Relationship Approach — ER'92, Proc. of the 11th Int. Conf. on the Entity-Relationship Approach, Karlsruhe, Germany, October 1992*, volume 645 of *Lecture Notes in Computer Science*, pages 389–405, Berlin, 1992. Springer-Verlag.

[TC98] P.L. Tarr and L.A. Clarke. Consistency Management for Complex Applications. In *Proc. of the 1998 Int. Conf. on Software Engineering*, pages 240–249. IEEE Computer Society, 1998.

[Tha91] B. Thalheim. *Dependencies in Relational Databases*, volume 126 of *Teubner-Texte zur Mathematik.* Teubner-Verlag, Stuttgart, Leipzig, 1991.

[TSS97] Z. Tari, J. Stokes, and S. Spaccapietra. Object Normal Forms and Dependency Constraints for Object-Oriented Schemata. *ACM Transactions on Database Systems*, 22(4):513–569, December 1997.

[Tür99] C. Türker. *Semantic Integrity Constraints in Federated Database Schemata.* Dissertation, University of Magdeburg, Germany, 1999.

[UKN92] S. D. Urban, A. P. Karadimce, and Ravi B. Nannapaneni. The Implementation and Evaluation of Integrity Maintenance Rules in an Object-Oriented Database. In Forouzan Golshani, editor, *Proceedings of the Eighth International Conference on Data Engineering, February 3-7, 1992, Tempe, Arizona*, pages 565–572. IEEE Computer Society, 1992.

[Ull88] J. D. Ullman. *Database and Knowledge-Base Systems, Volume I.* Computer Science Press, 1988.

[Var81] M.Y. Vardi. The Decision Problem for Database Dependencies. *Inf. Process.Lett.*, 12(5):251–254, 1981.

[WC96] J. Widom and S. Ceri, editors. *Active Database Systems: Triggers and Rules For Advanced Database Processing*. Morgan Kaufmann, 1996.

Symbol Index

$(W|i)$, 82
$I \models W$, 39
$\xleftarrow{+}$, 42
$\xleftarrow{+}_\tau$, 42
$\longleftarrow$, 42
$\longleftarrow_\tau$, 42
CPR, 72
F, 38
PE, 39
$\mathbf{PE}_\tau$, 41
PVCI, 135
$\mathbf{P}_\tau$, 41
RC, 71
SPVC, 101
SVC, 93, 94
T, 38
VC, 73
$\mathbf{V}_\tau$, 41
att, 19
class, 19
dom, 19
oid, 19
type(**C**), 21
val(**I**), 19
val, 20
Any, 21
$IC(W)$, 68
$K(W|o)$, 107
$Pcn(W)$, 72
$S(W|i)$, 90
$allCore(o, i)$, 124
$allInner(o, i)$, 123
$dom(\tau)$, 24
$unchkCore(o, i)$, 124
ICs, 68
IC, 67
Shells, 90
Shell, 90
Φ, 63
$\xleftarrow{-}$, 42
$\xleftarrow{-}_\tau$, 42
Υ, 45
$\leq$, 22
π, 23
π^*, 23
$\prec$, 21
$\psi[x/v]$, 39
$\rho(W)$, 59
σ, 21
$\sqsubseteq$, 74
nil, 19
$\mathcal{A}$, 35

Index

A

aggregation, 10

B

binary relationship, 26

C

cardinality ratio, 32

checking

  flat instance, 122

  nested instance, 124

class, 13

class *Constraint*, 111

  Constraints, 115

  check(), 126, 127

  checked(), 132

  disabled(), 135

  enabled(), 135

  localStatus, 111

  setDisabled(), 134

  setEnabled(), 134, 136

  setLastUpdate(), 130, 131

  unchecked(), 132

  structure, 119

class *Kernel*, 129

  lastCheck, 107, 130

  lastUpdate, 107, 130

class *Transaction*, 26

  check(), 129

  setDisbled(), 134

  setEnabled(), 134

  updatedObjects, 26

  control methods, 143

class IC, 67

  classes, 71

  mode, 70

  name, 68

  paths&roots, 72

  shells, 74

  status, 70

  wff, 70

  structure, 119

class Shell, 90

  class, 93

  constraint, 91

  form, 92

  objects, 95

  paths&roots, 93

  range, 94

  shell, 95

  structure, 119

class hierarchy, 14, 21

  semantics, 23

consistency checking, 127

consistency maintenance, 100, 135

constrained classes, 57

constrained paths, 58

constrained persistent roots, 58

constraint

add, 146
boolean form, 80
canonical form, 56, 70
deferred mode, 71
dependency, 33
domain, 52
dynamic, 30
existential, 87
explicit, 32, 51
global status, 134
immediate mode, 70
implicit, 32, 51
inclusion, 53
inherent, 51
inter-object, 53
intra-object, 53
key, 11, 52, 88
mode, 70
non null, 52
range restricted, 50
referential, 52
remove, 148
shells, 74, 90
simplified form, 82
state, 29
status, 70
transition, 29
well-formed, 49

constraint catalog, 55
structure, 118

constraint instance, 112
checked, 112
unchecked, 112
core, 124
flat, 113, 122
initiating, 116
inner, 123
kernel, 107
life cycle, 113
local status, 134
nested, 113
odd, 115
state, 129
structure, 109

constraint manipulation, 146

control methods, 137
atomic attribute, 138
set-structured attribute, 141
tuple-structured attribute, 141

D

database instance, 25
database schema, 25
dynamic binding, 14

E

encapsulation, 14
explicit range, 116
extension *Constraints*, 115
extension ICs, 68
extension Shells, 91

G

generalization, 10
generalized dependency, 33

I

IC object, 68
inheritance, 11
multiple, 14
inverse attribute, 47
maintenance, 49

inverse relationship, 26, 44, 47
  maintenance, 49
ISA hierarchy, 14
ISA relationship, 14

L

literal, 37

M

maintaining kernels, 129
method, 13, 24
  primitive, 13
  mutator, 13
  observer, 13
method overloading, 14
modification operations, 41
  add, 42
  assign, 42
  remove, 42

N

named object *deletedShell*, 119, 147
normalization
  object-oriented schemata, 34
  relational Schemata, 33

O

object, 20
  persistence, 15
  identity, 13
  role, 86
  state, 13
object sharing, 13
oid assignment, 23

P

part-of relationship, 10
participation constraints, 32
path dependency, 33
path expression, 39
  flat, 56
  inter, 58
  intra, 43
  linear, 39
  non-flat, 62
  prefix, 40
  suffix, 40
persistence
  by reachability, 15
  explicit, 15
persistent roots, 15, 25

R

range literals, 51
relational model
  impedance mismatch, 12
  limitations, 11
relevant constraints, 71
root class *Any*, 14, 21
  add(), 137
  assign(), 137
  check(), 128
  conIni(), 116, 117
  disabled(), 134
  enabled(), 134
  formEvaluation(), 138
  hasConstraints, 112
  new(), 13
  queryEvaluation(), 138
  remove(), 137
  updatePath, 137
  control methods, 137

structure, 119

S

semantic integrity, 1
shell, 102
core, 122
flat, 96
inner, 122
nested, 96, 104
signature, 24
simplified form, 78, 84
non outermost quantification, 84
outermost quantification, 83
range, 94
the method, 82
specialization, 10
subtyping relationship, 10, 22

T

type, 20
atomic, 13
complex, 13
semantics, 24

U

update units, 42

V

value
atomic, 13, 19
complex, 13, 19