Modeling Spatial and Temporal Data in an Object-Oriented Constraint Database Framework

vorgelegt von
Diplom-Informatikerin
Annalisa Di Deo

von der Fakultät IV-Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktorin der Ingenieurwissenschaften
- Dr.-Ing -

genehmigte Dissertation

Promotionsausschu"s:

Vorsitzender: Prof. Dr. K.Obermayer
Berichter: Prof. Dr. S.Jähnichen
Berichter: Prof. Dr. F.Turini

Tag der wissenschaftlichen Aussprache: 17. December 2001

Berlin 2002
D 83

Abstract

Constraints provide a flexible and uniform way to represent diverse data capturing spatio-temporal behavior, complex modeling requirements, partial and incomplete information etc, and have been used in a wide variety of application domains. Constraint databases have recently emerged to deeply integrate data captured by constraints in databases.

This thesis is aimed at defining a constraint data model, able to represent spatial and temporal information and, to exploit existing spatial and temporal tools, like the one of Oracle8, preserving the declarativeness of the first-order languages.

In detail, we present an object-oriented model to represent constraint databases. Constraints are basic data types (classes) of the model, defined by exploiting algebraic structures as monoids and semirings. Instances of the constraint classes are linear constraints, spatial and temporal predicates. The focal point of our work is achieving the right balance between expressiveness and complexity, making the model as general as possible and suitable to exploit features of existing tools.

We investigate the usefulness of our framework for two possible applications. The first application consists of two parts, showing how the framework models two different representations of spatial data. In the first part, we employ our framework on top of Spatial Cartridge of Oracle8. In the second part, we describe a mapping on linear constraints. The second application shows how the model is able to represent temporal and spatio-temporal information. Firstly, temporal data are represented as time intervals. Secondly, we extend the schema of our model, in order to represent and manage spatio-temporal objects, whose spatial part is represented by Oracle spatial objects. In this way we extend the features of the Spatial Cartridge, which so far handles exclusively with spatial data.

To my big uncle Nunziato

Acknowledgments

I would like to thank all the persons, who helped me during these years. First of all, I am very grateful to my advisors Stefan Jähnichen, Fosca Giannotti and Luca Simoncini for supporting me with their high competence.

Many thanks to Prof. Ulrich Geske, who has always stimulated my work and, to Dmitri Boulanger, with whom I have shared very useful discussions, he has been a good friend of mine and together we have had the key idea, that has brought to the accomplishment of this work. Then I express my gratitude to all my colleagues and friends at the GMD. A particular mention goes to my roommate Kathleen Steinhöfel, with whom I have shared many days and beautiful moments.

A very important experience has been to participate at the DeduGIS Working Group. I thank the members of this group, and in particular Alessandra Raffaeta', who has contributed to the development of this thesis and has reviewed it, giving me useful suggestions.

I would like to thank all my friends in Berlin and in particular my "sweet" roommate Judith Wahrheit, the "crazy" Silvia Schmehlik, the "indefatigable" Carlos Garaycochea, the "nostalgic" Ivan De Pablo and my berliner best friend Michael Fuhs. Without them my experience in Berlin would not have been so pleasant and exciting, as it has been.

Finally, I want to express my gratitude to my family, who has always helped me in the difficult moments. The most important thank goes to my boyfriend Pasquale, who with his love and support has made everything easier and surmountable.

Contents

List of Figures

Chapter 1

Introduction

With the increasing complexity of applications that store and manage spatial and temporal data, database research community is facing new challenges to efficiently represent and query such data, preserving the declarative and user-friendly features of the query languages [EJS98, Par95a, TCG+93].

The amount of spatial information is growing very rapidly. To satisfy the need of applications, different systems have been developed for various subareas related to multidimensional data. Among them, geographic information systems (GIS) constitute a large field of interest [Wor95, EH95]. Basic issues concern the representation of geometric information at different levels of abstraction, the description of spatial relationships and data types, the definition of user-friendly query languages, and other topics. Most systems devoted to the manipulation of geographic data are based on an architecture coupled with a relational DBMS, but dedicated to geographic information with special purpose tools for manipulating spatial data. This is the case of the most prominent systems for geographic information, namely ARC/INFO [Mor89]. Although these systems have been widely used for more than a decade, it is clear that the lack of integration between spatial data processing and standard data management is rather not desirable. There has been a large number of proposals in the literature for integrating spatial features with other traditional thematic alphanumeric features of geographic objects in a single framework [Güt94]. The first and most recent example of commercial spatial database is to assign to Oracle, which has developed a spatial cartridge offering predefined spatial data types and operators; in particular, Egenhofer topologic relationships ([EF91, Ege91]) have been implemented on top of a relational and an object-relational data model [Ora]. However, because of the complexity of spatial data and the variety of GIS applications, there has not been a common agreement on a satisfactory model for representing spatial data. Among various limitations there is no high level query facility.

On the other side, Temporal Database Management Systems (TDBMS) came into existence based on extensions to the relational, entity-relationship or object-oriented paradigms. This is due to the increased demand for functionality to handle

both static and dynamic time related information in databases. Traditional systems retain only the latest state of the modeled system (or the state at a specific point of time), presenting an up-to-date, but static view of the environment. Proposals for temporal data models are reviewed in [Sno92, TCG+93]. Basic issues concern the representation of time, the selection of appropriate temporal granularity, the level at which temporality should be introduced, support for temporal reasoning, and other topics. Allen' s temporal algebra hold parallels with spatial topological and order related relationships, and can serve as framework for adding temporal operators to the database. Most of these concerns have been addressed; the creation of TSQL2 [Sno95b] points to a steady progress toward establishing a TDBMS standard.

However, recently the problem of dealing with correlated spatial and temporal data has become an important topic to which a large effort has been devoted [AR99, BJS99], due to the need of many applications in which space and time are closely interconnected: much information which refers to space refers to time as well. Spatiotemporal systems would provide benefits in areas such as environmental monitoring, administration, real-time navigational systems, transportation scheduling, cadastral systems etc.

Traditional databases are not able to handle these complex data at a high level of abstraction. And, in most cases [GS95], the models consist in extending relational DBMSs with abstract data types encapsulating structures and operations. Due to the extensive range of applications which handle spatial and temporal data, there has not been so far a convergence towards a commonly accepted data model with well-defined data types and operations.

In this context, we place our project, whose aim is *the definition of a data model, that provides data types and operations to handle spatial, temporal and spatiotemporal information in a uniform way.*

To achieve our purposes, we choose the *constraint database* as framework. The recent field of constraint databases, initiated at the beginning of the decade [KKR90, KKR95], has lead to sound data models and query languages for multi-dimensional data [GK96, GRS98b, PdBG94]. The big challenge of constraint databases is to introduce constraints as basic data type in databases. The main principles are based on a simple and fundamental duality: a constraint can be seen as a relation, or, vice versa, that a relation can be interpreted as a simple form of constraint. This enables us to manipulate relations of the underlying DMBS in a symbolic way enjoying expressive power of first-order logic, i.e. the syntax consists of constraints, i.e. symbolic expressions; the semantics are the corresponding relations.

The use of constraints offers many advantages. At the data level, constraints are able to represent possibly infinite sets in a finite way. In particular, they are shown to be a powerful mechanism for modeling spatial and temporal concepts [BBC97a, CR97, Kou94, PdBG94]. For instance, the constraint $0 \leq x \leq 3 \wedge 0 \leq y \leq 3$ represents the square area with corners $(0,0)$, $(0,3)$, $(3,0)$ and $(3,3)$ and

$1993 \leq t \leq 1997$ represents the time interval between 1993 and 1997. From this data modeling perspective, constraints are used as a unifying data type for the conceptual representation of different sorts of multi-dimensional data; by varying the constraint theory, one can accommodate a variety of different data models.

The research trend on this field has focused on definition of constraints as *first-class object* of object-oriented models. In fact, the existing prototypes, implementing spatio-temporal constraint databases, namely [GRS98b, BSCE97], define constraints as C++ classes. Both rely on *linear constraints*, first appeared in [GST94, GRSS97]. They allow to manipulate relations of arbitrary dimension between points in a symbolic manner. In spite of their expressiveness, the existing constraint databases are not always efficient enough for real problems where the size and the complexity of data are high. In particular, spatial data might be so complex, e.g. non-convex geometries with multiple holes, that constraint database models cannot handle them efficiently and need to refer to optimized structures of the objects and operations possibly implemented in an underlying DBMS.

It is on the analysis of features and drawbacks of these prototypes, that we base the general principles of our project. In this context, the general objective is the design of a *constraint data model for representing spatial, temporal and spatio-temporal information* which fulfills the following requirements:

- In terms of constraint domains and operators, a careful balance between expressiveness and complexity must be achieved.
- The language and the model should be object-oriented, since many object-oriented features are important for the target applications.
- The model should be extensible with respect to (constraints and other) data types, operators and predicates.
- The model should interact with different data models.
- The model should allow an easy interaction with underlying existing commercial tools in order to exploit their optimization features.
- Provided the previous principles are met, the language should be high-level and easy to use.

1.1 Main Contributions of the thesis

The main achievement of the thesis is the definition of a constraint data model, able to represent spatial and temporal information and, to exploit existing spatial and temporal tools, like the one of Oracle8, preserving the declarativeness of the first-order languages.

The design of the model is based on two principles. Firstly, constraints must be treated as first-class objects, whose methods are typical logical operations, such as conjunction, disjunction and existential quantification. Secondly, an underlying object-based representation of data must be wrapped as constraints.

For this purpose, we define a formal background for a component-based programming tool, which enables to build constraints as instances of basic data types in object-relational/oriented DBMSs.

The general framework is an extended *monoid comprehension* model [FM95], in which constraint objects serve as special type. The data model of extended monoid comprehension is based on the notion of the algebraic structures *monoids* and *semirings* [BL97, HJA71], that can be used as conceptual data types, capturing uniformly aggregations and collections, and other types on which one can iterate. This approach offers the ability to treat constraints as collections. Constraints are defined as elements of a domain; the usual operations of conjunction and disjunction are operationally formalized as binary operations on and to elements of the domain.

The ability to treat disjunctive and conjunctive constraints uniformly as collections is a very important feature of our framework: it allows to implement many constraint operations through nested monoid comprehensions, i.e. in the same language as hosting queries. For example, the satisfiability test of a disjunction of conjunctions of constraints is expressed as a monoid comprehension query that iterates over the disjuncts (each being a conjunction), and tests the satisfiability of every conjunction.

The data model is adapted from *flat constraints* [GSG94], i.e. atomic formulas of first-order logic defined on a vocabulary of just predicates and constants. New constraint objects are constructed using logical connectives, existential quantifiers and variable renaming, within a multi-typed constraint algebra.

Flat constraints possess great modeling power and as such can serve as a uniform data type for conceptual representation of heterogeneous data, including spatial and temporal behavior. Because of their indefiniteness, flat constraints can be mapped either to linear constraints or to sets of spatial and temporal objects. To the best of our knowledge, this is the first constraint data model, whose constraints express relations between objects, whose implementation and representation is hidden.

To highlight the usefulness of the framework we investigate two possible applications.

The first application consists of two parts, showing how the framework models two different representations of spatial data.

In the first part, we employ our framework on top of Spatial Cartridge of Oracle8 [Ora]. The Spatial Cartridge implements efficiently Egenhofer' s spatial relationships ([EF91, Ege91]) on objects of high complexity, e.g. polygons with multiple holes, arcs, composed polygons etc.; these relationships can be wrapped as flat constraints. By means of this "wrapping step", Spatial Cartridge is provided with a high-level layer, i.e. first-order based language, to express relations between objects.

This provides a more friendly interface for spatial analysis. In fact, a problem of Oracle8 is that querying requires the knowledge of composition of the spatial objects and of filtering mechanisms; this is not that simple. With a higher-level querying facility, the user will not front these problems, nevertheless he can exploit their advantages.

In the second part, we show how the model works when flat constraints are mapped to linear constraints. Linear constraints have been shown to fit the need of spatial data in the vector model; the vector model is a way to represent spatial data in the database, and it is used in most of applications. The fundamental difference between linear constraint data model and vector model relies on the explicit definition of the spatial objects in the data model (e.g. a polygon is explicitly the infinite set of points it contains versus the implicit definition by the sequence of border points). It is possible to manipulate spatial objects through standard set operations. Furthermore, efficient algorithms have been developed, that implement operations on linear constraints, e.g. simplex algorithm to implement conjunction of linear constraints.

The second application shows how the model is able to represent temporal and spatio-temporal information. Firstly, temporal data are represented as Oracle objects (*intervals*) and temporal relationships are Allen' s temporal relationships [All83, All84]. Secondly, we extend the schema of our model, in order to represent and manage spatio-temporal objects, whose spatial part is represented by Oracle spatial objects. In this way we extend the features of the Spatial Cartridge, which so far handles exclusively with spatial data.

1.2 Plan of the thesis

Chapter 2 gives first some theoretical backgrounds on first-order logic and model theory which we largely use in the thesis, then it presents the intuition on which constraint database is based, its data model and it describes some approaches in literature; above all C^3, the approach of Brodsky et al. [BSCE97], on which our framework is based.

Chapter 3 briefly recalls the basic concepts of modeling temporal and spatial data, focusing particularly on those using constraints, and it describes some approaches in literature for handling temporal, spatial and spatio-temporal information.

Chapter 4 presents our general framework. The resulting model is obtained extending and applying the C^3 model to flat generic constraints. It defines flat constraints and the corresponding data types, the flat constraint monoid comprehension and the schema of classes, with relative operations.

Chapter 5 describes the first application. It shows how the model can represent Oracle spatial data and linear constraints as flat constraints, defining the respective

vocabulary and interpretation structure and presenting some query examples.

Chapter 6 describes the second application. It shows first how we can model temporal information using our framework. Furthermore, it gives an extension of the schema to model spatiotemporal information. At the end, it presents some query examples.

Finally, Chapter 7 contains some concluding remarks and future work.

Chapter 2

Overview on Constraint Databases

In this chapter we introduce the *constraint database (CDB)* framework.

The basic idea in constraint databases is the introduction of constraint formulas as a basic data type in databases. The notion of constraint data relies on a simple and fundamental duality: a constraint (formula) φ in free variables $x_1, \ldots, x_n$ is interpreted as a set of tuples $(a_1, \ldots, a_n)$ over the schema $x_1, \ldots, x_n$ that satisfy φ; and conversely, a finitely representable object in n-dimensional space can be viewed as a constraint. That is, the syntax is constraint, i.e. symbolic expressions; the semantics are the corresponding, possibly infinite, relations.

Constraints are usually represented with a sub-family of first-order logic and a family of atomic constraint, such as real polynomial, linear, or dense order constraints. In general constraints possess great modeling power and as such can serve as *uniform data type* for conceptual representation of heterogeneous data, including spatial and temporal behavior, complex design requirements and partial incomplete information.

Querying constraint databases is made by means of constraint query languages, i.e. database query language augmented by constraints. They are *declarative* and very *expressive*. The evaluation of queries can be optimized, defining a procedural language layer between the constraint query languages and low-level access methods. This intermediate language is the so-called *constraint algebra*. Corresponding to different classes of constraints, several algebras were developed, defining the operation on constraints, such as intersection, union, containment etc.

The first constraint database proposals are based on the relational model, but since the last five years several prototypes extend the object-oriented model and represent constraints as data types. The resulting prototypes combine the features of object-oriented databases with those of constraint databases. This combination opens to the possibility of using Java and efficient tools for the implementation of constraint databases, that makes them competitive in the commercial world.

The chapter is organized as follows: Section 2.1 introduces some theoretical backgrounds on first-order logic and on constraints, and fixes the notation used in the whole thesis; Section 2.2 describes the data model of CDBs and presents some exam-

ples of modeling with constraints; Section 2.3 defines queries on CDB and presents some well-known constraint query languages and algebras; finally, Section 2.4 describes the existing prototypes, that extend the object-oriented model.

2.1 Theoretical background

In this section we recall some basic definitions, which introduce first-order languages and their interpretation by means of models. Furtherly we introduce the notion of quantifier elimination, explaining its fundamental role in logical theories. These notions are necessary to understand the basic theory on which constraint database model lies. Most of the definitions has been taken from [dBar] and [CK90b].

2.1.1 First-order logic and models

We set up the first-order logic with identity, specifying a list of symbols and then giving precise rules by which sentences can be built up from the symbols.
A language is a sequence of symbols, defined by a vocabulary.

Definition 2.1 (Vocabulary) A vocabulary, or signature Ω consists of three sets: a set $\mathcal{F}$ of function symbols, a set $\mathcal{P}$ of predicate symbols, and a set $\mathcal{C}$ of constant symbols, together with an arity function that associates a natural number to each element of $\mathcal{F}$ and $\mathcal{P}$.

For instance, $\Omega = (+, \cdot, 0, 1, <)$ is a vocabulary with two function symbols of arity two $(+, \cdot)$, one predicate symbol of arity two $<$, and two constant symbols (0 and 1).
We denote function symbols with $f_1, \ldots, f_n$ and $g_1, \ldots, g_n$, predicate symbols with $p_1, \ldots, p_m$ and $q_1, \ldots, q_m$ and constant symbols with $c_1, \ldots, c_k$.
To formalize a language Ω, we need the following *logical symbols*:

variables $v_1, \ldots, v_n, \ldots$,

connectives $\wedge$, $\neg$,

quantifiers $\forall$,

one binary relation symbol $\equiv$ (identity).

We assume, of course, that no symbol in Ω occurs in the above list. We can now define the fundamental elements of the first-order language, namely, *terms*, denoted with $t_1, \ldots, t_n$, *atomic formulas* and *formulas*, denoted with $\varphi_1, \ldots, \varphi_n$, $\psi_1, \ldots, \psi_m$.

Definition 2.2 (First-order logic over Ω) *Terms* are defined inductively as either a variable, or a constant from $\mathcal{C}$, or $f(t_1, \ldots, t_n)$, where $f \in \mathcal{F}$ is of arity n, and

$t_1, \ldots, t_n$ are terms.

An *atomic formula* is a formula of the form $t \equiv t'$ or $p(t_1, \ldots, t_n)$, where $t, t', t_1, \ldots, t_n$ are terms, and $p \in \mathcal{P}$ is an n-ary predicate.

Formulas are either an atomic formula, or if φ and ψ are formulas, then $(\varphi \wedge \psi)$ and $(\neg\varphi)$ are formulas. If v is a variable and φ is a formula, then $\forall_v \varphi$ is a formula. A sequence of symbols is a formula only if it can be shown to be a formula by a finite number of applications of the rules above. □

Further symbols are used to compose formulas in first-order logic, making them more readable. The symbols $\vee$, $\rightarrow$, $\leftrightarrow$ and $\exists$ are abbreviations defined as follows:

$(\varphi \vee \psi)$ for $(\neg((\neg\varphi) \wedge (\neg\psi)))$,

$(\varphi \rightarrow \psi)$ for $((\neg\varphi) \vee \psi)$,

$(\varphi \leftrightarrow \psi)$ for $((\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi))$,

$\exists_v \varphi$ for $\neg\forall_v(\neg\varphi)$.

Formulas might contain *free* variables, i.e. variables that do not occur in the scope of any quantifier. A *sentence* is a formula without free variables.
After setting up the language, we need a model to specify the truth values of the formulas. The *truth definition* is the bridge connecting the formal language with its interpretation by means of models. If the truth value "true" goes with the sentence φ and a model $\mathcal{M}$, we say that φ is *true in* $\mathcal{M}$ and also that $\mathcal{M}$ is a *model of* φ.
A model for the language Ω consists, first of all, of a universe $\mathcal{A}$, i.e. a non empty-set, where

an n-ary predicate symbols p corresponds to an n-relation $r \subseteq \mathcal{A}^n$,

an m-ary function symbol f corresponds to an m-ary function $g : \mathcal{A}^m \rightarrow \mathcal{A}$,

a constant symbol c corresponds to a constant $a \in \mathcal{A}$.

The correspondence is given by an interpretation function $\mathcal{J}$ mapping symbols of Ω to relations, functions and constants in $\mathcal{A}$. To define them formally, we need an algebraic structure providing a domain and a set of operations on and to the domain.

Definition 2.3 (Model for Ω or Ω-structure). Given a non-empty set $\mathcal{D}$ and a signature Ω, a model $\mathcal{M}$ for Ω (or an Ω-structure) is defined assigning to each $f \in \mathcal{F}$ of arity n a function $f^{\mathcal{M}} : \mathcal{D}^n \rightarrow \mathcal{D}$, to each $p \in \mathcal{P}$ of arity n an n-ary predicate on $\mathcal{D}$, that is, a set $\mathcal{P}^{\mathcal{M}} \subseteq \mathcal{D}^n$, and to each $c \in \mathcal{C}$ an element $c^{\mathcal{M}} \in \mathcal{D}$. $\mathcal{D}$ is called the

domain[1] of the structure.

When it will not cause confusion, we will use the two words $\Omega - structure$ and *model for* Ω interchangeably. We introduce some conventions as regards the functions and constants. When

$$\Omega = \{p_0, \ldots, p_n, f_0, \ldots, f_m, c_0, \ldots, c_q\}$$

we write the models for Ω in displayed form as

$$\mathcal{M} = \{\mathcal{A}, r_0, \ldots, r_n, g_0, \ldots, g_m, a_0, \ldots, a_q\}$$

For example, a model for $\Omega = (+, \cdot, 0, 1, <)$ is $\mathcal{M} = \{\mathbf{R}, +^{\mathbf{R}}, .^{\mathbf{R}}, <^{\mathbf{R}}\}$, where $\mathbf{R}$ is the set of real numbers, and $+^{\mathbf{R}}, .^{\mathbf{R}}, <^{\mathbf{R}}$ the usual addition, multiplication, and the ordering on $\mathbf{R}$, and by interpreting $0^{\mathbf{R}}$ and $1^{\mathbf{R}}$ as $0, 1 \in \mathbf{R}$.
Note that we always omit the superscript, since the interpretation of the signature symbols is typically understood from the context.

Before introducing the notions of satisfaction of formulas, we give an important convention of notation. We use $t(v_0, \ldots, v_n)$ to denote a term t whose variables form a subset of $\{v_0, \ldots, v_n\}$. Similarly, we use $\varphi(v_0, \ldots, v_n)$ to denote a formula φ whose free variables form a subset of $\{v_0, \ldots, v_n\}$.

The notion of satisfaction is related to the possibility to answer to the question: "Given a formula $\varphi(v_0, \ldots, v_p)$ and a sequence $a_0, \ldots, a_p$ of elements of the universe $\mathcal{A}$, is φ true in $\mathcal{M}$ if the variables $v_0, \ldots, v_p$ are taken to be $a_0, \ldots, a_p$?"

This is making examining every sub-formula $\psi(v_0, \ldots, v_p)$ of φ and all elements $a_0, \ldots, a_p$. It was proven in [CK90b] that the answer depends only on the elements $a_0, \ldots, a_q$, $q \leq p$ corresponding to free variables of φ. We can now give the formal definition.

Definition 2.4 ($a_0, \ldots, a_q$ satisfies φ in $\mathcal{M}$)[CK90b]. Let $\mathcal{M}$ be a fixed model for Ω.
The *value of a term* $t(v_0, \ldots, v_q)$ at $a_0, \ldots, a_q$, denoted by $t[a_0, \ldots, a_q]$, is defined as follows:

i. If $t = v_i$, then $t[a_0, \ldots, a_q] = a_i$.

ii. If t is a constant symbol c, then $t[a_0, \ldots, a_q]$ is the interpretation of c in $\mathcal{M}$.

iii. If $t = f(t_1, \ldots, t_m)$, where f is an m-ary function symbol, then

$$t[a_0, \ldots, a_q] = g(t_1[a_0, \ldots, a_q] \ldots t_m[a_0, \ldots, a_q]),$$

where g is the interpretation of f in $\mathcal{M}$.

[1] In the literature can be found other terms: universe, carrier, etc.

The statement $a_0, \ldots, a_q$ satisfies φ in $\mathcal{M}$, denoted by $\mathcal{M} \models \varphi[a_0, \ldots, a_q]$ is defined as follows:

i. If $\varphi(v_0, \ldots, v_q)$ is the atomic formula $t_1 \equiv t_2$, where $t_1(v_0, \ldots, v_q)$ and $t_2(v_0, \ldots, v_q)$ are terms, then

$$\mathcal{M} \models (t_1 \equiv t_2)[a_0, \ldots, a_q] \text{ if and only if } t_1[a_0, \ldots, a_q] = t_2[a_0, \ldots, a_q].$$

ii. If $\varphi(v_0, \ldots, v_q)$ is the atomic formula $p(t_1, \ldots, t_n)$, where p is n-placed relation symbol and $t_1(v_0, \ldots, v_q), \ldots, t_n(v_0, \ldots, v_q)$ are terms, then

$$\mathcal{M} \models p(t_1, \ldots, t_n)[a_0, \ldots, a_q] \text{ if and only if } r(t_1[a_0, \ldots, a_q], \ldots, t_n[a_0, \ldots, a_q]),$$

where r is the interpretation of p in $\mathcal{M}$.

The third and last case is that $\varphi(v_0, \ldots, v_q)$ is a formula of the language Ω.

i. If φ is $\theta_1 \wedge \theta_2$, then $\mathcal{M} \models \varphi[a_0, \ldots, a_q]$ if and only if both $\mathcal{M} \models \theta[a_0, \ldots, a_q]$ and $\mathcal{M} \models \theta[a_0, \ldots, a_q]$.

ii. If φ is $\neg\theta$, then $\mathcal{M} \models \varphi[a_0, \ldots, a_q]$ if and only if not $\mathcal{M} \models \theta[a_0, \ldots, a_q]$.

iii. If φ is $\forall v_i \psi$, where $i \leq q$, then $\mathcal{M} \models \varphi[a_0, \ldots, a_q]$ if and only if for every $a \in \mathcal{A}$, $\mathcal{M} \models \psi[a_1, \ldots, a_{i-1} a a_{i+1} \ldots a_q]$.

iv. If φ is $\exists v_i \psi$, where $i \leq q$, then $\mathcal{M} \models \varphi[a_0, \ldots, a_q]$ if and only if there exists $a \in \mathcal{A}$ such that $\mathcal{M} \models \psi[a_1, \ldots, a_{i-1} a a_{i+1} \ldots a_q]$.

□

Logical Theory

In this paragraph we give some notions about logical theories. A *theory* is simply defined as a set of sentences. We denote theories with Σ, Γ, etc. New sentences can be deduced from a theory as *logical consequences*. Given a theory Σ, we say that $\mathcal{M}$ is a model for Σ iff $\mathcal{M}$ is a model for each σ in Σ. A sentence that holds in every model for Ω is called *valid*. A sentence φ is a *consequence of* Σ, in symbols $\Sigma \models \varphi$, iff every model of Σ is also a model of φ.

Theories can be *closed* iff they are closed under the $\models$ relation, and they are *complete* iff they contains all their consequences. In constraint databases only decidable logical theories are considered.

Definition 2.5 (Decidable logical theory) A language with an interpretation structure forms a *decidable logical theory* if there exists an algorithm which determines for each sentence whether the sentence is a consequence of the theory.

We will see in the next paragraph that quantifier elimination is a crucial property of decidable theories.

A theory Γ can be described by a *set of axioms*, i.e. a set of sentences with the same consequences as Γ. The next example shows well known closed theories.

Example 2.1 Let Ω consist of the single two-place relation symbol $\leq$. The theory of *partial order* has three axioms:

1. $(\forall xyz)(x \leq y \wedge y \leq z \rightarrow x \leq z)$,
2. $(\forall xy)(x \leq y \wedge y \leq x \rightarrow x \equiv y)$,
3. $(\forall x)(x \leq x)$.

They are respectively the transitive, antisymmetric, and reflexive properties of partial orders. Any model $\langle \mathcal{A}, \leq \rangle$ of this theory consists of a non-empty set $\mathcal{A}$ and a partial order relation $\leq$. If we add the comparability axiom

4. $(\forall xy)(x \leq y \vee y \leq x)$

we obtain the theory of *linear order*. Adding two more axioms

5. $(\forall xy)(x \leq y \wedge x \not\equiv y \rightarrow (\exists z)(x \leq z \wedge z \not\equiv x \wedge z \leq y \wedge z \not\equiv y))$,
6. $(\exists xy)(x \not\equiv y)$.

we have the theory of dense order. The rationals with the usual $\leq$ is an example of a model of this theory. □

Quantifier Elimination

Quantifier elimination can be defined as a property of very special theories. Before describing it, we need some more notation. In the previous paragraph we have introduced the notion of a sentence φ being a logical consequence of a set Σ of sentences. What's happen if φ is a formula? A formula $\varphi(v_0, \ldots, v_n)$ is a consequence of Σ, symbolically $\Sigma \models \varphi$, iff for every model $\mathcal{M}$ of Σ and every sequence $a_0, \ldots, a_n \in \mathcal{A}$, $a_0, \ldots, a_n$ satisfies φ. It follows that a *formula* $\varphi(v_0, \ldots, v_n)$ is a consequence of Σ if and only if the *sentence* $(\forall v_0, \ldots, v_n)\varphi(v_0, \ldots, v_n)$ is a consequence of Σ. Two formulas φ, ψ are $\Sigma - equivalent$ iff $\Sigma \models \varphi \leftrightarrow \psi$.

Definition 2.6 A theory Γ is said to *admit quantifier elimination* if for every sentence φ of Γ there exists a quantifier-free formula ψ over Ω such that

$$\Gamma \models (\varphi \leftrightarrow \psi)$$

If such a ψ can be effectively computed, given φ, we say that the quantifier elimination is effective. □

Only in the context of theories admitting quantifier elimination that we will be able, in principle, to implement constraint database systems with first-order logic query languages. Popular theories, that satisfy this property, include the real field and its restrictions that exclude multiplication, or include only order. That the real field admits quantifier elimination was essentially provided by Tarski' s famous decision method for the theory of the reals. A well known illustration of quantifier elimination over the reals is provided by high-school mathematics, the formula

$$\exists_x(ax^2 + bx + c = 0)$$

is equivalent to the quantifier-free formula

$$b^2 - 4ac \geq 0.$$

2.1.2 Constraints

Before introducing the framework of constraint databases, we need to define constraints formally.

Definition 2.7 (Constraints) Let Ω be a vocabulary. A *constraint* over Ω is an atomic first-order formula over Ω, or the negation of an atomic formula.

The representational power of constraints is characterized by their *extension* defined in the following definition.

Definition 2.8 (Extension of constraints) Given a vocabulary Ω and a structure $\mathcal{H} = (\mathcal{D}, \Omega)$, a set $X \subseteq \mathcal{D}^n$ is called the *extension* of the Ω-constraint φ if it can be obtained as a finite boolean combination of sets of the form

$$\{(a_1, \ldots, a_n) \in \mathcal{D}^n \,|\, \mathcal{H} \models \varphi(a_1, \ldots, a_n)\}$$

Given $\mathcal{H} = (\mathcal{D}, \Omega)$ and $\mathcal{H}' = (\mathcal{D}, \Omega')$, we said that Ω- and Ω'-constraints are equivalent over $\mathcal{D}$ if they have exactly the same extensions. □

The following are class of constraints that we will often use in the rest of the chapter:

- *Polynomial inequality constraints* are constraints over $\Omega = (+, \cdot, 0, 1, <)$. Such constraints correspond to conditions of the form $p > 0$ or $p \leq 0$, where p is a polynomial with integer coefficients. These constraints are usually interpreted over the structures $\mathbf{R}$=($\mathbf{R}, \Omega$), $\mathbf{Q}$=($\mathbf{Q}, \Omega$), $\mathbf{Z}$=($\mathbf{Z}, \Omega$), and the like.
- *Linear constraints* are constraints over $\Omega = (+, <, 0, 1)$. Such constraints correspond to conditions of the form $p > 0$ and $p \leq 0$, where p is a linear

function $a_1x_1 + \cdots + a_jx_j + a_0$ with integer coefficients. These constraints are usually interpreted over the structures $\mathbf{R}_{lin} = (\mathbf{R}, \Omega)$, $\mathbf{Q}_{lin} = (\mathbf{Q}, \Omega)$, $\mathbf{Z}_{lin} = (\mathbf{Z}, \Omega)$, and the like.

- *Dense order constraints over rationals* are constraints over $\Omega = (<, (c)_{c\in\mathbf{Q}})$, interpreted over the structure $(\mathbf{Q}, \Omega)$, that is rational numbers with order and constants for every $c \in \mathbf{Q}$. Such constraints are conditions of the form $x < y$, $x \geq y$, $x < c$ or $x \geq c$, where x and y are variables and c is a constant. Similarly, one can define dense order constraints over the reals, or for the matter, over any set on which a dense order is available.

- *Equality constraints over an arbitrary infinite domain* $\mathcal{U}$ are constraints over the signature $((c)_{c\in\mathcal{U}})$. Such constraints are conditions of the form $x = y$, $x \neq y$, $x = c$, $x \neq c$, where x and y are variables, and c is a constant.

2.2 Constraint databases

The fundamental idea, on which the design principles of constraint databases are based, is that *a conjunction of constraints is the correct generalization of a ground fact* [KKR90]. Tuples and relations of relational models can be mapped to quantifier-free first-order formulas. This enables us to manipulate information in a symbolic way. Let's show it with the following example.

Example 2.2 We suppose to have a database consisting of a set of rectangles in the plane, as shown in Figure 2.1.

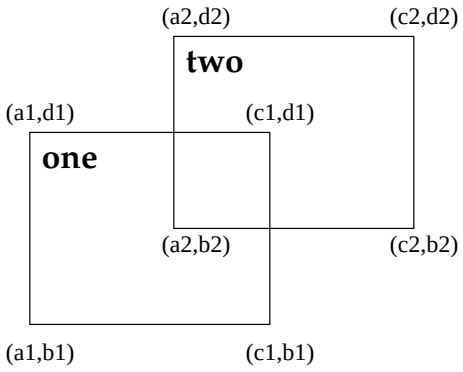


Figure 2.1: Database with rectangles

In the classical relational model, one possibility is to store the data in a 5-ary relation named R. This relation will contain tuples of the form (n, a, b, c, d), where n is the name of the rectangle with corners at (a, b), (a, d), (c, b) and (c, d).
In a constraint database model, the set of rectangles can be modeled by a ternary relation $R(z, x, y)$, that can be interpreted to mean that (x, y) is a point in the rectangle with

name z. The rectangle that was stored above by (n, a, b, c, d), would now be stored as the generalized tuple $(z = n) \wedge (a \leq x \leq c) \wedge (b \leq y \leq d)$. □

In the pioneer work of Kanellakis, Kuper and Revesz, [KKR90], a first constraint database model was designed extending the relational model. Tuples and relations of the relational model were "re-defined" as conjunctions and disjunctions of quantifier-free first-order formulas, respectively. The new model, called *generalized relational model*, see a schema in Figure 2.2, introduces constraints as "first-data type" in data models.

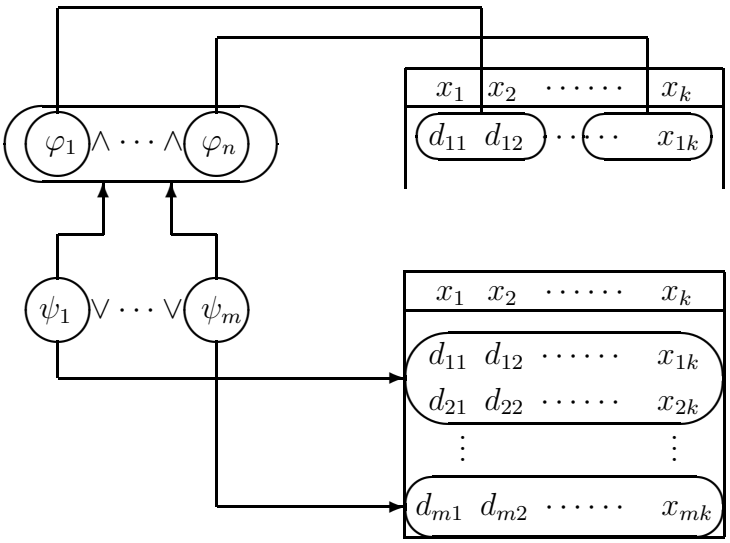


Figure 2.2: Schema of the *Generalized Relational Model* of [KKR90]

After the proposal of Kanellakis et al., a lot of constraint data models, which extend the object-oriented model, have been proposed. We present some of them in detail in Section 2.4.

2.2.1 The constraint database model

We are now ready to define formally a constraint database model. The definition is taken from [KKR90, KKR95].

Definition 2.9 (Constraint database model) Fix a vocabulary Ω.

i. A *constraint k-tuple*, in variables $x_1, \ldots, x_k$, over Ω, is a finite conjunction $\varphi_1 \wedge \cdots \wedge \varphi_N$, where each φ_i, for $1 \leq i \leq N$, is an Ω-constraint. The variables in each φ_i are all among $x_1, \ldots, x_k$.

ii. A *constraint relation of arity* k, over Ω, is a finite set $r = \{\psi_1, \ldots, \psi_M\}$, where each ψ_i, for $1 \leq i \leq M$, is a constraint k-tuple in the same variables $x_1, \ldots, x_k$. The corresponding formula is quantifier-free and it is the disjunction $\psi_1 \vee \cdots \vee \psi_M$.

iii. A *constraint database* is a finite collections of constraint relations.

□

In the database theory, a k-ary relation r is assumed to be a *finite* set of k-tuples, or points in a k-dimensional space, but normal relations are not able to represent infinite sets of points. The constraint relations of Definition 2.9 provide a finite representation of possibly infinite sets of points in a k-dimensional space.

The possibly infinite sets of points in a k-dimensional space constitute the interpretation of constraint relations. We denote them as *extended relations*. This concept is better exposed by the following definition.

Definition 2.10 (Interpretation of Constraint Relations) Let Ω be a vocabulary, and $\mathcal{H} = (\mathcal{D}, \Omega)$ an Ω-structure. Let r be a constraint relation of arity k over Ω, and let $\varphi_r(x_1, \ldots, x_k)$ be the formula corresponding to r. Then r represents all points $(a_1, \ldots, a_k)$ such that $\varphi_r(a_1, \ldots, a_k)$ is true in $\mathcal{H}$. That is, it represents the set

$$\{(a_1, \ldots, a_k) \in \mathcal{D}^k \mid \mathcal{H} \models \varphi_r(a_1, \ldots, a_k)\}^2. \qquad \square$$

Observe that interpretations of constraint relations over $\mathcal{H}$ are precisely the set, we called *extensions of* Ω*-constraints* defined in Definition 2.8.

A very important concept of CDB, that concerns their representational power, is the *definability with constraints*, [dBar].

Definition 2.11 (Definability with constraints). Given a vocabulary Ω and a structure $\mathcal{M} = \langle \mathcal{U}, \Omega \rangle$, a set $X \subseteq \mathcal{U}^n$ is called definable on $\mathcal{M}$ with Ω-constraints if it can be obtained as a finite Boolean combination of sets of the form

$$\{(a_1, \ldots, a_n) \in \mathcal{U}^n | \mathcal{M} \models \varphi(a_1, \ldots, a_n)\},$$

where φ is an Ω-constraint. □

In database theory, a k-ary relation r is assumed to be a finite set of k-tuples (or points in a k-dimensional space). We shall use the term *unrestricted relation* for arbitrary finite or infinite sets of points in a k-dimensional space. One can develop query languages using such unrestricted relations; however, in order to be able to

[2]The set of the definition is also called *extended relation*. An *extended database* is defined as union of extended relations.

do something useful with them, we need a finite representation that can be manipulated. This is exactly what the constraint tuples provide.

Definition 2.12 (Definable databases). Any finite collection of unrestricted relations is called an *unrestricted database*. An unrestricted database is called *definable* if all its relations are definable in the sense of Definition 2.11. A constraint database D represents a definable database D' if D consists precisely of one representation for each unrestricted relation in D'.

2.2.2 Modeling with constraints relations

We have seen in Section 2.1.2, that varying the vocabulary Ω, and the interpretation structure $\mathcal{H}$, it is possible to define different classes of constraints. We will show now, how the constraint relations, corresponding to those classes of constraints, are naturally suitable to model different data sets.

Polynomial Inequality Constraints

Let the constraint relation r consists of the two constraint tuples

$$(2y \geq 1) \wedge (-3x + 4y \leq 8) \wedge (3x + 4y \leq 8) \text{ and } (x^2 + y^2 = 1) \wedge (2y \leq 1)$$

The corresponding DNF formula is:

$$\varphi_r(x, y) = ((2y \geq 1) \wedge (-3x + 4y \leq 8) \wedge (3x + 4y \leq 8)) \vee ((x^2 + y^2 = 1) \wedge (2y \leq 1))$$

The formula φ_r describes the *infinite* set of points in 2-dimensional space shown in Figure 2.3: namely the triangle formed by the intersection of the three half planes described by the first constraint tuples, and the part of circle resulting from the intersection between $x^2 + y^2 = 1$ and the half-plane $2y \leq 1$.

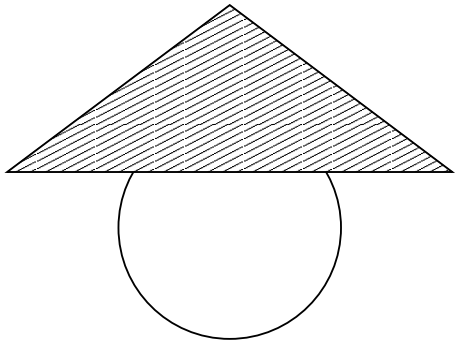


Figure 2.3: Figure in the real plane represented by the formula φ_r

Linear Constraints

We now illustrate the framework using linear constraints. Let the constraint relation s consist of the two constraint tuples

$$(y = 2x \wedge \neg(x = y)) \text{ and } (1 \leq x + y).$$

Corresponding to s is the DNF formula

$$\varphi_s(x, y) = (y = 2x \wedge \neg(x = y)) \vee (1 \leq x + y).$$

φ_s describes an *infinite* set of points in 2-dimensional space: namely the half plane $x + y \leq 1$ and the line $y = 2x$ without the point $x = y = 0$, see Figure 2.4.

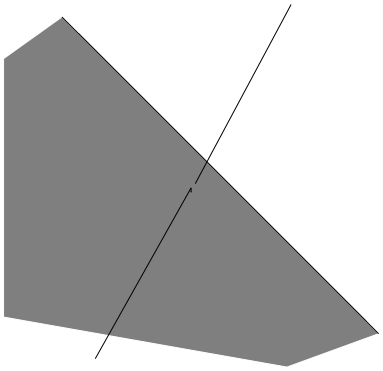


Figure 2.4: Figure in the real plane represented by the linear constraint formula φ_s

Equality constraints

Let' s consider a relation q, that consists of the tuples $(1, 2)$ and $(3, 4)$. These tuples are equivalent to the constraint 2-tuples $x = 1 \wedge y = 2$ and $x = 3 \wedge y = 4$. Therefore, q corresponds to the formula $\varphi_q \equiv (x = 1 \wedge y = 2) \vee (x = 3 \wedge y = 4)$.
In general, every finite k-ary relation r on a set $\mathcal{D}$ with n tuples $a_1^i, \ldots, a_k^i$, $i = 1, \ldots, n$, can be represented as a constraint relation using just equality constraints. The corresponding formula is

$$\varphi_r = \bigvee_{i=1}^{n}((x_1 = a_1^i) \wedge \cdots \wedge (x_k = a_k^i)).$$

Dense Order Constraints over Rationals

We consider the constraint relation p consisting of two constraint tuples

$$x = 1 \text{ and } 2 \leq x \wedge x \leq y \wedge y \leq 6$$

The corresponding formula is

$$\varphi_p = (x = 1) \vee (2 \leq x \wedge x \leq y \wedge y \leq 6)$$

In 1-dimensional space φ_p represents the point 5 and the infinite set of subintervals of the interval $(2, 6)$.

2.3 Formalisms to query constraint databases

In this section, we formally define queries on constraint databases, obtained combining constraints with the fundamental notion of database query and we introduce firstly the relational calculus (augmented with constraints) as a basic constraint query languages and, secondly the family of constraint algebras proposed in the literature.

2.3.1 Constraint queries

Recall Definition 2.9 and Definition 2.10. A constraint database is defined as a finite collection of constraint relations. It is useful to give these relations a name, when querying them. Therefore, we define a *database schema* as a finite non-empty set SC of relation names, each of which has a given arity. A *constraint database with schema SC* then becomes a mapping, associating to each relation name $R \in SC$ a constraint relation of the arity given for R. An *extended database with schema SC* is defined analogously.

We discuss now the definition of a query on constraint databases. A query on a relational database can be considered as a mapping, associating to each database an answer relation. In the constraint setting, it is more complicated. Conceptually, constraint databases are symbolic representations of extended databases. So we can think of a query in two ways: on the conceptual level, as a mapping on extended databases (*extended queries*); or on the representation level, as a mapping on constraint databases (*constraint queries*). A query on the representation level is consistent only if it corresponds to a query on the conceptual level, as in the diagram in Figure 2.5:

Consistency lets to avoid that a constraint query maps two different constraint representations of a same extended database to constraint relations representing different extended relations. Note that the extended query represented by a consistent constraint query is uniquely defined.

Another important property of constraint queries is the *closure property*, i.e. the output of a constraint query applied to any constraint relation must be also a constraint relation.

Example 2.3 illustrates the above definitions in the concrete contest of polynomial inequality constraints over the real field $\mathbf{R}$ (recall Section 2.2.2).

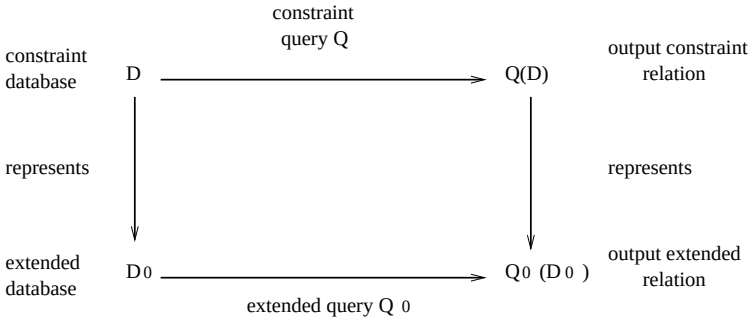


Figure 2.5: Consistency of a constraint query

Example 2.3 Fix SC to consist of a single relation name T of arity 3. So the extended database consists of a set of tuples in $\mathbf{R}^3$, which can be obtained as a finite union of extended representation sets of polynomial inequality constraints.

Define the following 2-ary constraint query Q: given a constraint database D with $D(T) = \{\psi_1, \ldots, \psi_M\}$, consider the formula

$$\chi := \exists_{x_3}(\psi_1 \vee \cdots \vee \psi_M).$$

Since $\mathbf{R}$ admits effective quantifier elimination, we can compute from χ a quantifier-free formula φ, in the variables x_1 and x_2, equivalent to χ over $\mathbf{R}$. We then transform χ into disjunctive normal form:

$$\chi \equiv \gamma_1 \vee \cdots \vee \gamma_l.$$

Then define $Q(D) := \{\gamma_1 \vee \cdots \vee \gamma_l\}$.

This constraint query is consistent; it represents the 2-ary extended query mapping each semi-algebraic set in $\mathbf{R}^3$ to its projection on the x_1x_2-plane. □

Finally we give an example of a constraint query that is not consistent.

Example 2.4 Consider a query that maps each constraint database D to the singleton set consisting of that constraint tuple in $D(T)$ that has the largest sum of all constants appearing in it. Consider the constraint databases D_1 and D_2 with $D_1(T) = \{0 < x_1 \wedge x_1 < 3, 1 < x_1 \wedge x_1 < 4\}$ and $D_2(T) = \{0 < x_1 \wedge x_1 < 3, 2 < x_1 \wedge x_1 < 4\}$. Both represent exactly the same extended relation, namely the interval $(0, 4)$. However, on D_1 the output of the query is $\{1 < x_1 \wedge x_1 < 4\}$, while on D_2 it is $\{2 < x_1 \wedge x_1 < 4\}$. These two singleton constraint relations represent different extended relations. Hence, this constraint query cannot possibly represent some extended query, as the latter, being a mapping, cannot have two different results on the same input. □

2.3.2 Constraint query languages

The relational calculus (first-order logic), with a given class of constraints, constitutes a very basic constraint language, which we next introduce. The extension with

constraints was first made by Kanellakis et al. in [KKR90, KKR95] for developing a constraint query language.

Let' s fix some database schema SC and a vocabulary Ω. A relational calculus formula over Ω is a first-order logic formula over the extended vocabulary (Ω, SC), obtained extending Ω with the predicate symbols belonging to the schema SC. We denote with $FO(\Omega, SC)$ this set of formulas.

Example 2.5 shows some well-known classes of formulas, obtained extending the calculus with linear constraints and polynomial inequality constraints, respectively. Recall from Section 2.2.2 the vocabularies of linear constraint and polynomial inequality constraints be $(+, <, 0, 1)$ and $(+, \cdot, 0, 1, <)$, respectively.

Example 2.5 Let R denote a relation name of arity 2, the following is a formula of relational calculus + linear constraints, e.g. over $\Omega = (+, <, 0, 1)$

$$R(x_1, x_2) \vee \exists_y (R(x_1, y) \wedge R(y, x_2) \wedge (x_1 + x_2 < y) \wedge (x_2 - y < 0)):$$

and the following is a formula of relational calculus + polynomial linear constraints, e.g. over $\Omega = (+, \cdot, 0, 1, <)$:

$$R(x_1, x_2) \vee \exists_y (R(x_1, y) \wedge R(y, x_2) \wedge (x_1 \cdot x_2 < y) \wedge (x_2 - y^3 < 0)). \qquad \square$$

These are pure syntactical examples; the semantics of relational calculus formulas is defined next.

Given an Ω-structure $\mathcal{M}$, relational calculus formulas over Ω express total (i.e. everywhere defined) extended queries over $\mathcal{M}$ in the obvious manner:

Definition 2.13 (Interpretation of $FO(\Omega, CS)$ formulas) Fix an Ω-structure $\mathcal{M} = (\mathcal{U}, \Omega)$. A $FO(\Omega, SC)$ formula $\varphi(x_1, \ldots, x_k)$ expresses the k-ary extended query Q over $\mathcal{M}$ defined as follows. Given an unrestricted database D with schema SC over $\mathcal{M}$, we can expand $\mathcal{M}$ to a structure $\langle \mathcal{M}, D \rangle = \langle \mathcal{U}, \Omega, D \rangle$ over the expanded vocabulary (Ω, SC) by adding the interpretation $D(T)$, for each relation name $T \in SC$ in D, to $\mathcal{M}$. Then

$$Q(D) := \{(a_1, \ldots, a_k) \in \mathcal{U}^k | \, \langle \mathcal{M}, D \rangle \models \varphi(a_1, \ldots, a_k)\}. \qquad \square$$

The semantics of a relational constraint query is depicted in Figure 2.6.

In Figure 2.6, $\phi = \varphi(x_1, \ldots, x_m)$ is a constraint query with free variables $x_1, \ldots, x_m$. Let predicate symbols $R_1, \ldots, R_n$ in ϕ be the name of the input constraint relations and let $r_1, \ldots, r_n$ be the corresponding constraint relations. Let $\phi[r_1/R_1, \ldots, r_n/R_n]$ be the formula of the logical theory obtained by replacing in ϕ each database atom $R_i(z_1, \ldots, z_k)$ by the DNF formula for input generalized relation r_4, with its variables appropriately renamed to $z_1, \ldots, z_k$. Let D be the domain, then the constraint query ϕ applied to the constraint database containing the constraint relations $r_1, \ldots, r_n$ is a formula of the logical theory of constraints used, i.e., $\phi[r_1/R_1, \ldots, r_n/R_n]$. The

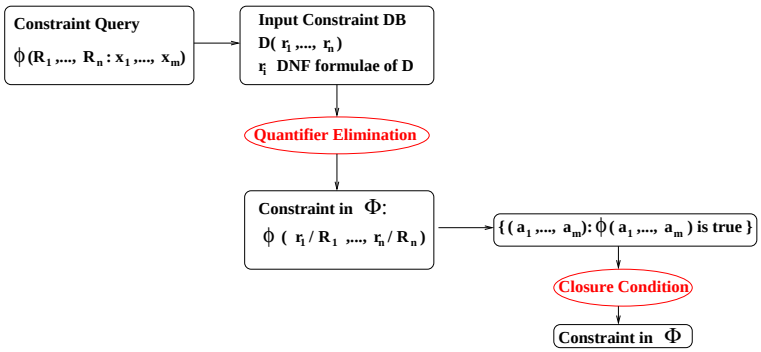


Figure 2.6: Evaluation of Constraint Queries

output is the possibly infinite set of points in m-dimensional space D^m, that satisfy the formula. This brings to the following fundamental theorem ([dBar]):

Proposition 2.14 $\mathcal{M}$ admits effective quantifier elimination if, and only if, every relational calculus formula φ expresses a consistent, effectively computable, total constraint query, that represents the extended query expressed by φ.

The proof of the proposition can be found in [dBar].

Example 2.6 We illustrate the above proposition using relational calculus + polynomial inequality constraint formulas over $\mathbf{R}$. Assume that the schema consists of one binary relation name R, and consider the simple formula $\varphi(y) \equiv \exists_x R(x, y)$. Let D be the constraint database in which $D(R)$ equals the singleton $\{y = x^2\}$. To evaluate φ on D, we replace the occurrence of R in φ by its defining formula to get

$$\chi(y) \equiv \exists_x (y = x^2).$$

This formula is equivalent, in $\mathbf{R}$, to the quantifier-free formula

$$\varphi(y) \equiv y = 0 \vee y > 0,$$

which is already in disjunctive normal form. Hence, we can define the result of the constraint query Q_φ defined by φ applied to D as $Q_\varphi(D) := \{y = 0, y > 0\}$.

Note that this example would fail if we would omit the $<$ predicate, i.e., if we work in the constraint structure $\langle \mathbf{R}, +, \cdot, 0, 1 \rangle$. In this context the extended query expressed by φ would *not* be closed. The motivation is that the structure $\langle \mathbf{R}, +, \cdot, 0, 1 \rangle$ does not admit quantifier elimination either. □

2.3.3 Constraint algebras

Just like relational algebras, constraint algebras are used for program optimization.

Two families of algebras were defined for manipulating constraint relations. First, algebras that correspond exactly to the traditional relational algebra, and whose semantics is defined with respect to the effect of the usual algebraic operators on infinite relations [GST94, PdBG94]. Second, algebras that apply on special encoding for relations and constraint tuples [KG94]. Furthermore, we distinguish a third group of proposals, aiming to design algebras modeling complex objects [BBC98, GRS98a].

Before introducing the two families, we recall some useful definitions from [dBar]. A set X consisting of tuples of domain elements is *definable (with constraints)* if it can be obtained as a finite union of extensional representation sets of constraints.

Definition 2.15 (Real Semi-Algebraic and Semi-Linear Sets) A set X consisting of tuples of domain elements is called *semi-algebraic* if it is definable with polynomial inequality constraints over the reals. X is called *semi-linear* if it is definable with linear constraints over the reals.

Thus, the constraint relations definable with polynomial inequality constraints over the reals are precisely the semi-algebraic sets, and the constraint relations definable with linear constraints over the reals are precisely the semi-linear sets.

Relational Algebra for Constraint Relations

The algebras, which belong to this family, differ from each other on the constraint theory, on which they are defined. Algebras have been defined to manipulate dense order constraints [GK96], polynomial constraints [PdBG94] and, linear constraints [GST94]. The last two ones have been strongly exploited in spatial databases. Both are classical relational algebras with the standard operators on relations, the difference is that in this case the operators are defined on constraint tuples. We present in the following the algebra for polynomial constraints, and we postpone a description of the linear algebra in Section 3.2.2.

Algebra for Real Polynomial Inequality Constraints [PdBG94] The underlying constraint data model consists of:

- A database scheme that is a finite set of *relation names*. Each relation name R has a type $\tau(R)$, which is a pair of natural numbers $[n, m]$.
- An *intensional tuple* (or *i-tuple*) of type $[n, m]$ is a tuple $(a_1, \cdots, a_n; \phi)$ with $a_1, \cdots, a_n \in \mathbf{U}$, which has been assumed to be a countably infinite domain of atomic values, and ϕ is a quantifier-free real formula of arity m.
- An *instance* I of a scheme S is a mapping on S, assigning to each relation name R of S a finite set of i-tuples of type $\tau(R)$, that is called *i-relation*.
  Intensional tuples and relations are finite representations of possibly infinite subsets of $\mathbf{U}^n \times \mathbf{R}^m$ which are called *extensional relations* or *e-relations*.

- For an i-tuple $t = (a_1, \cdots, a_n; \phi)$, let S be the semi-algebraic set defined by ϕ. t represents the e-relation $\{(a_1, \cdots, a_n)\} \times S$, denoted by $ext(t)$. An i-relation r represents the e-relation $ext(r) := \bigcup_{t \in r} ext(t)$.

In the model, a relation type of the form $[n, 0]$ is called *flat*. *Flat relations* are the standard value-based relations of classical relation databases, in which there is no difference between intensional and extensional level.

On the other hand, relation types of the form $[0, m]$ are called *purely spatial* and are exactly those considered in [KKR90].

The operators of the algebra have been defined. Given an i-tuple $t = (a_1, \cdots, a_n; \phi)$, let $val(t)$ denote the value part $(a_1, \cdots, a_n)$ and $spat(t)$ the spatial part ϕ; r, r_1 and r_2 be i-relations of type $[n, m]$:

- The *union* $r_1 \cup r_2$ is the standard set-theoretic union.
- Let $veq_i(t), i = 1, 2$ denote the set $\{t' \in r_i | val(t') = val(t)\}$. The *difference* $r_1 - r_2 = \{(val(t); \bigvee_{t' \in veq_1(t)} spat(t') \wedge \neg \bigvee_{t' \in veq_2(t)} spat(t')) | t \in r_1\}$
- The *Cartesian product* $r \times r' = \{(val(t), val(t'); spat(t) \wedge spat(t')) | t \in r, t' \in r'\}$, of type $[n + n', m + m']$
- The *value selection* $\sigma_{i=j}(r) = \{t \in r | val(t)(i) = val(t)(j)\}$.

  Let ϕ be a quantifier-free real formula on the variables $x_1, \cdots, x_m$. The *spatial selection* $\sigma_\phi(r) = \{(val(t); spat(t) \wedge \phi) | t \in r\}$.
- For $1 \leq i_1, \cdots, i_p \leq n$, the *value projection* $\pi_{i_1, \cdots, i_p} = \{(val(t)(i_1), \cdots, val(t)(i_p); spat(t)) | t \in r\}$, of type $[p, m]$.

  Let $1 \leq i_1, \cdots, i_p \leq m$ and let $y_1, \cdots, y_p$ be real variables different from each x_i. For a real formula ϕ with free variables $x_1, \cdots, x_m$, define $\phi_{x_{i_1}, \cdots x_{i_p}}(\phi)$ as the quantifier-free formula equivalent of the real formula

$$(\exists x_1) \cdots (\exists x_m)(\phi \wedge \bigwedge_{l=1}^{p} y_l = x_{i_l})$$

  Then the *spatial projection* $\pi_{i_1, \cdots, i_p} = \{(val(t); \pi_{i_1, \cdots, i_p}(spat(t))) | t \in r\}$, of type $[n, p]$.
- The algebra includes the constant $\mathbf{R}^k$ for each k, standing for the "full" relation of type $[0, k]$.

Paradeans et al. in [PdBG94] have proved, that the calculus and the algebra are equivalent and that the spatial calculus (algebra) is a conservative extension of the standard relational calculus (algebra). Consider an example. The query

"Give the pairs of persons living at least 10 miles apart"

is

$$\pi_{1,2}\sigma_{(x_3-x_1)^2+(x_4-x_2)^2\geq 100}(Lives \times Lives)$$

The disadvantage of this approach is that it may be too general in an implementational perspective. For this reason, the set of spatial objects has been restricted in various ways. The basic idea is to focus on "linear" data types. This choice is motivated by two factors: this class of data types includes most of the common used spatial data, and operations on typical linear data, such as lines, and polygons, can be implemented by efficient algorithms.

Algebras for Encoded Constraint Relations

The algebras defined by Kanellakis and Goldin in [KG94, GK96] belong to this second family . Their approach defines a first-order algebra for dense order constraints interpreted on a countably infinite set D, which is equivalent to the calculus defined in [KKR90], and it's based on the assumption that the generalized tuples have a specific form, beyond being conjunctions of constraints. In their algebra they use generalized tuples that are "tightest" or *canonical*. Kanellakis et al. proved that this algebra has the desiderable *closure* property. The usual operators of the relational Algebra are defined on a special encoding of generalized tuple.

Algebras for Nested Constraint Models

The algebras, we consider in this paragraph, have been proposed in [BBC98]. Their expressions are defined on terms that are constraint relations, interpreted under the *nested relational model*. In this model, a constraint relation is represented as a finite set of (possibly) infinite sets, each representing the extension of a single constraint tuple, contained in the considered relation.

The use of different semantic reference model for constraint databases leads to the definition of new languages. New connectives, and not only conjunction as proposed in [KKR95], can be used. For example, to model concave sets inside a constraint tuple, disjunction must be used.

External functions are also inserted in the constraint language, to handle specific procedures and implementations needed by different applications.

The model is the one proposed by [KKR90]. In that case, the constraint tuples were interpreted on Φ (a decidable logical theory) and a signature $\Omega = \{\wedge\}$. In the nested constraint model, the constraint tuples are interpreted on Φ and $\Omega = \{\wedge, \vee\}$ and represent all sets that can be characterized in first-order logic without quantifiers and are called *disjunctive constraint tuples* or *d-constraint tuples*. Other operators have been defined, considering the extension of each constraint tuple as a single object.
The algebras dealing with the new model, are divided in two classes:

- *R-based algebras:* extend the usual relational algebra to constraint databases.

- *N-based algebras:* are based on the nested relational algebras, defined for complex objects, [AHV95].

The algebra contains two classes of operators: one class handling constraint relations as an infinite set of relational tuples (*tuple operators*), and one class handling each constraint relation as a finite set of sets (*set operators*).

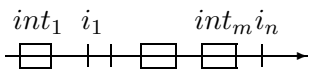
Analytical and graphical representation	**Representation using d-gen. tuples in Φ_D**
NON-CONTIGUOUS INTERVAL (int_D) $(i_1 \cup \ldots \cup i_n) \cup (int_1 \cup \ldots \cup int_m)$ 	$C(int_D) \equiv$ $(C_{instant}(i_1) \vee \ldots \vee C_{instant}(i_n)) \vee$ $(C_{interval}(int_1) \vee \ldots \vee C_{interval}(int_m))$

Table 2.1: Representation of non-contiguous subsets of the time axis in the dense-order constraint theory

The Table 2.1 shows how non-contiguous intervals can be represented using disjunctive constraint tuples. In such representation, each disjunct represents an instant or an interval belonging to the representation of the noncontiguous interval.

These algebras offer more facilities to express queries than [GK96], mostly because of its set-operators and of the non presence of the tuple-identifier, but it is equivalent to it when identifiers are introduced in input databases. The study of the data complexity proved the extended algebra is in class C or in NC.
In this case, the constraint tuples are not represented in any canonical form like [GK96].

2.4 Constraints in the object-oriented framework

Before describing how constraints have been modeled in object-oriented frameworks, we briefly recall the main concepts of the object-oriented model:

- The basic modeling entities are the **object** and the **literal**. Each object has a unique identifier. A literal has no identifier.

- Objects and literals can be categorized by their **types** (in Java settings, **classes**). All elements of a given type have a common range of states (i.e. the same set of properties) and common behavior (i.e. the same set of defined operations). An object is sometimes referred to as an instance of its type.

- The state of an object is determined by the values of its set of properties. These properties can be **attributes** of the object itself or **relationships** between

the objects and one or more other objects. Typically the value of object's properties can change over time.

- The behavior of an object is defined by the set of operations, **methods**, that can be executed on or by the object. Operations may have a list of input and output parameters, each with a specified type. Each operation may also return a typed result.

- There are two kinds of relationships between objects: **inheritance**, that is a hierarchy relationship between classes, in which the methods of a class are inherited by its sub-classes; and **composition**, that is reference relationship, with which an object can refer to other objects. By means of these relationships it is possible to navigate through the population of objects of a databases.

- A database stores objects, enabling them to be shared by multiple users and applications. A database is based on a schema that is defined in ODL (Object Definition Language) and contains instances of the types defined by its schema. Querying object is performed by means of a OQL (Object Query Language).

On this framework, constraints are usually defined as instances of particular predefined classes. In the next section, we will see some example of existing prototypes, which implement constraints as objects.

Example 2.7 The following is an example of an SQL query on the structure depicted in Figure 2.7:

select	c.address
from	Person p, p.chidren c
where	p.address.street.name = "Main Street"
and	count(p.children) $\geq$ 2
and	c.address.city $\neq$ p.address.city

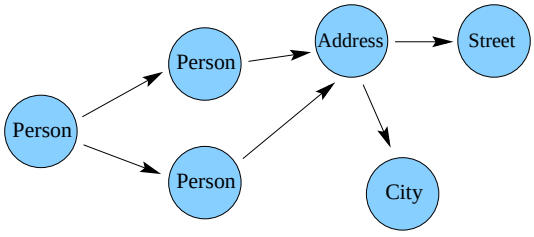


Figure 2.7: Navigating an object data structure with OQL

The "dot" notation used in the query traverses the data structure as shown in Figure 2.7. The query inspects all children of all "Persons" to find people who live on Main Street with at least two children. It returns only those addresses of children who do not

live in the same city as their parents. It navigates from the *Person* class using the child reference to another instance of the *Person* class and then to the *Address* and *City* classes. □

2.4.1 Object-oriented constraint databases

In the object-oriented data model, constraints must be objects, i.e instances of a predefined class. Object query languages, like OQL or SQL, have no logic foundation, so that the representation of constraint objects implies an ad-hoc construction of quantifier-free formulas, like in [GRSS97], where a class Relation is defined, whose instances are generalized relations. We show in the following an example, taken from [GRSS97].

Example 2.8 The example describes the schema for the geographic object *GO* (standing for *Ground Occupancy*), that is represented by some descriptive attributes followed by a geometric attribute of type *Relation* (a generalized relation) representing the set of its generalized tuples: each generalized tuple has two attributes, x and y coordinates of the plane. The generalized relation describes the spatial extent of the object.

```
class   GO (
        No: integer,
        ground − type : string,
        owner : string,
        geometry : Relation
);
```

Generalized relations are defined according to the following schema:

```
class   Relation (
        tuples : set(Tuple)
);
```

```
class   Tuple (
        constraints : list(Constraint),
        geo − type : GEO-TYPE
);
```

where GEO-TYPE = {*Point*, *Segment*, *Polygon*}.
The type *Constraint* defines linear constraints, i.e. constraint of the form

$$a_1x_1 + a_2x_2 \ldots + a_kx_k \, op \, b$$

where k is the arity of the relation, x_i is the variable associated with the A_i attribute, each a_i and b are rational and op belongs to OP = $\{=, <, \leq, \geq, >\}$. Therefore a constraint is represented as

```
class   Constraint (
        constants : list(rational),
        op : OP
);
```

where the vector of constraints *constants* is a list of $k + 1$ rationals. □

Anyway, also if the proposal of Grumbach et al. is interesting and represents one of the first prototypes, the way of representing constraints is made ad-hoc and it is fixed, i.e. it works only with quantifier-free formulas, whose atomic formulas are linear constraints. This implies the application of an algorithm of quantifier elimination whenever a quantifier occurs and it is well-known that such an algorithm is exponential with the number of variables to eliminate.

More expressive formalisms than the mentioned one were defined to represent and query objects. Kim et al. in [KKS92] propose the language XSQL, that is built around the idea of extended path expressions and on an adaptation of first-order formalization of object-oriented languages.

Furthermore, Fegaras and Maier in [FM95] propose an algebraic formalism, called *monoid calculus* and prove its effectiveness for object-oriented query language, such as OQL. The monoid calculus readily captures such features as multiple collection types, aggregation, arbitrary composition of type constructors and nested query expression. It is based on the representation of data types as monoids and queries on them. As we based our proposal on this framework, we will not go deeply in its explanation and postpone it to Chapter 4.

Both formalisms were extended with linear constraints, to define constraint object query languages. The formalism of Kim et al. [KKS92] was extended in [BK95], to define $\mathcal{L}_{iri}C$ language, and the formalism of Fegaras and Maier in [BSCE97, BSCE99] to define the CCUBE language. CCUBE is a constraint object-oriented system, that implements linear constraints. Constraints in this framework have canonical forms, and as such they can be represented as monoids. The resulting *constraint monoid calculus* is a very expressive constraint query language.

Below is an example of a definition of my_desk of the class DESK in CCUBE, taken from [BSCE97]:

```
Desk my_desk =  desk(
                catalog_no = 22354,
                name = "one_drawer_desk",
                color = "red",
                extent = (−4 ≤ w ≤ 4 ∧ −2 ≤ q ≤ −1),
                drawer = new drawer(
                        color = "blue",
                        center = (p = −2 ∧ −3 ≤ q ≤ −1),
                        extent = (−1 ≤ w1 ≤ 1 ∧ −1 ≤ z1 ≤ 1),
                        translation = (w = w1 + p ∧ z = z1 + q)
                )
);
```

In the above example, the attribute *extent* is of type CST, which represents the linear constraint class. The CST class is defined as sub-class of the monoid class, providing the methods to perform the operations $\wedge$ and $\vee$. Variables are syntactic

objects, i.e. strings.

The next query finds, for each desk in all_desks, its extent in the rooms coordinates, assuming the center of the desk is located at the point $(6, 4)$ and its translation equation (with respect to the room' s coordinate) is $u = w + x \wedge v = z + y$:

```
select   new CST( (u,v) | (dsk→extent ∧
         u = w + x ∧ v = z + y∧
         x = 6 ∧ y = 4 ))
into     {bag⟨CST⟩} result
from     all_desks as {desk} dsk
```

Note that {bag⟨CST⟩} in the into clause indicates the type of the result, and CST is the constraint type, that can be used as the other data types. In this case {bag⟨CST⟩} denote a bag (set) of constraint objects and in the select there is the creation of a new CST object, where (u,v)| denotes the projection on the variable u,v.

Chapter 3

Spatial and Temporal Data Models

In this chapter, we briefly recall the basic concepts of the modeling of temporal and spatial data focusing particularly on those using constraints, and we describe some approaches in literature for handling temporal, spatial and spatio-temporal information. The chapter is structured as follows: Section 3.1 presents temporal data models and distinguishes two strands: temporal algebras 3.1.1 and temporal databases 3.1.2, in 3.1.4 we describe some main approaches, using constraint data models.

3.1 Temporal data models and algebras

Interest in research concerning the handling of temporal information has been growing steadily over the past two decades. On the one hand, much effort has been spent in developing formalisms modeling temporal relationships, among them there temporal logics (see, e.g., [OM94]) and algebras. On the other hand, in the field of databases, many approaches have been proposed to extend existing data models, such as the *relational*, the *object-oriented* and the *deductive* models, to cope with temporal data (see, e.g., the books [TCG+93, EJS98] and references therein). It is clear that a close connection exists between these two strands of research, since temporal formalism can provide solid theoretical foundations for temporal databases, and powerful knowledge representation and query languages for them [Cho94, GM91, Org96].

We skip the description of temporal logic as they are not concerning the topic of the thesis, concentrating ourself on algebras reasoning with time. Before presenting some well-known temporal algebras and giving an overview on temporal databases, we introduce some concepts of general interest in temporal knowledge representation. In particular we list the alternatives choices for the structure of time and temporal relations. We refer the reader to *The Consensus Glossary of Temporal Database* [EJS98] for a complete collection of definitions of concepts related to time.

Structure of time

If we want to deal with time, then each fact, assertion, or predicate must be associated with a *temporal entity* that specifies when it is true. There are several alternatives for the definition of such temporal entities, as listed below.

- **Instants** vs. **time intervals**. An instant is a time point on an underlying time axis, while a time interval is the time between two instants. Correspondingly we can distinguish two view points according to the granularity of time: point-based and interval-based. In the database context the point-based view is predominant (e.g., see [TCG+93, Cho94]), on the other hand, there is a lot of Artificial Intelligence research (e.g., the Allen' s seminal work [All83, All84]) that takes the interval-based view.

- **Discrete** vs. **dense** vs. **continuous**. The instants in a *discrete* model of time are isomorphic to the natural numbers, i.e., there is the notion that every instant has a unique successor. Instants in the *dense* model of time are isomorphic to (either) the real or rational numbers: between any two instants there is always another one. Time is *continuous* if it is isomorphic to real numbers.

- **Bounded** vs. **unbounded**. This concerns the question of whether time is considered infinite in either or both directions.

- **Linear** vs. **branching** vs. **cyclic**. On time instants is imposed an *order*. Time is *linear* if it is provided with a *total ordering* relation. A partial order that satisfies *left-linearity* is used to model *branching time* [Eme90], i.e., time is linear from the past to the present and branching from the present to the future in order to represent the possible evolutions of the world. Moreover, a linear transitive order which is not irreflexive or anti-symmetric is used to model *cyclic* time.

Temporal relations

Time is said to be *absolute* if it is independent of, e.g., the time of another fact and of the current time, *now*, for instance it is expressed in terms of dates. Time is said to be *relative* if it is related to some other time, e.g., the time of another fact or the current time, *now*. Relationships between times can be *qualitative*, such as before, after, as well as *quantitative* if they involve some kind of metrics, such as "interval I lasts 2 hours" or "3 days before".

3.1.1 Temporal algebras

The family of algebras of temporal relationships can be divided in three sub-families, each of them modeling a different information:

- algebras modeling ordering information,
- algebras modeling metric information,
- algebras modeling ordering and metric information.

Each category includes point-based-as well as interval-based formalisms.

Modeling Ordering Information The most commonly used interval-based formalism is *Allen' s interval algebra*[All83]. It models the relationship between any two intervals as a suitable subset of a set of 13 basic relations, namely, *equal, before, meets, overlaps, starts, during* and *finishes*, showed in Tab. 3.1 where t and s are two time intervals and t^- and t^+ indicate respectively the lesser and the greater end-point of t, together with their inverses:

Interval relation	Equivalent relations on endpoints
$t\, equal\, s$	$(t^- = s^-) \wedge (t^+ = s^+)$
$t\, before\, s$	$t^+ < s^-$
$t\, meets\, s$	$t^+ = s^-$
$t\, overlaps\, s$	$(t^- < s^-) \wedge (t^+ > s^-) \wedge (t^+ < s^+)$
$t\, starts\, s$	$(t^- = s^-) \wedge (t^+ < s^+)$
$t\, during\, s$	$(t^- > s^-) \wedge (t^+ < s^+)$
$t\, finishes\, s$	$(t^- > s^-) \wedge (t^+ = s^+)$

Table 3.1: Allen' s interval relations defined by endpoints

Allen describes then a method of representing the relationships between temporal intervals in a hierarchical manner using constraint propagation techniques. Temporal information is maintained in a network and when a new relation is entered, all consequences are computes. This is done by computing the transitive closure of the temporal relations. For instance, if the fact that i is *during* j is added and j is $before\ k$, then it is inferred that i must be $before\ k$. The Tab. 3.2 shows a part of the Allen' s transitivity table, the complete table can be found in [All83].

$(A\, r_1\, B) \wedge (B\, r_2\, C)$	$<$	$>$	d	di
$before\ (<)$	$<$	all	$<$, o, m, d, s	$<$
$after\ (>)$	all	$>$	$>$, oi, mi, d, f	$>$
during (d)	$<$	$>$	d	all
contains (di)	$<$, o, m, di, fi	$>$,oi, di, mi, si	o, oi, d, di, $=$	di

Table 3.2: Part of the transitivity table for the temporal relations.

In 3.2, the symbols m, o, s, f stand respectively for *meets*, *overlaps*, *starts* and $finishes$, and mi, oi, si, fi, for their respective inverses.

Point algebras were also proposed, (see [MP93] for an overview), that model the relationship between any two points as a subset (disjunction) of a set of three basic relations ($<$, $>$, $=$). Relations between intervals are expressed as compositions (conjunctions) of relations between their endpoints. As an example, the interval relation $I_1\,(before \vee meets \vee overlaps)\,I_2$ can be expressed as the conjunction of the point relations:

$$I_1^- < I_2^- \wedge I_1^+ < I_2^+$$

Modeling Metric Information Interval and point algebras deal only with ordering temporal relations. A formalism that permits the representation and processing of metric temporal information is Dechter, Meiri and Pearl's *distance algebra*, which models distances between time points and duration of intervals [DMP91]. According to a point-based approach, it consider variables over continuous domains, whose value can be constrained using both unary and binary constraints. Let $X_1, \ldots, X_n$ be a set of variables. A unary constraint C_i on X_i restricts its domain to a finite set of intervals $I_1, \ldots, I_n$ ($domain\,constraints$) and it is represented by the disjunction

$$I_1^- \leq X_i \leq I_1^+ \vee \ldots \vee I_n^- \leq X_i \leq I_n^+$$

A binary constraint $C_{i,j}$ on variables X_i and X_j constraints the admissible values for the distance $X_j - X_i$ to belong to a finite range $R_1, \ldots, R_n$. Each range R_k is an ordered pair $\langle min_k, max_k \rangle$, where min_k and max_k are an upper and lower bound on the distance separating X_i and X_j in time, respectively. A constraint $C_{i,j} = \{R_1, \ldots, R_n\}$ is represented by the disjunction

$$min_1 \leq X_j - X_i \leq max_1 \vee \ldots \vee min_n \leq X_j - X_i \leq max_n$$

Point algebra can be easily encoded within distance algebra on the basis of a suitable set of equivalences, on the other hand, interval algebra cannot be encoded within distance algebra, i.e. they have orthogonal expressive power. Properties like disjointedness cannot be represented by a binary constraint, but require a 4-ary constraint.

Modeling Ordering and Metric Information Formalisms that integrate interval and distance algebras to deal with both ordering and metric information in a uniform framework have been developed by [KL91, Mei91].

[KL91] define a simple logical formalism for expressing both kinds of temporal information. They assume a linear model of time, where time points are identified with the rationals under the usual ordering relation $<$ and time intervals are represented as pairs of points $\langle n, m \rangle$, with $n < m$. They also define a subtraction function that return the rational number corresponding to the difference between two time points. They formalize such a model of time using a typed predicate calculus with equality and providing a suitable set of axioms.

Meiri [Mei91] defines a model of time capable of dealing with disjunctive constraints, involving both metric and ordering. It consists of a set of variables $\{X_1, \ldots, X_n\}$, which may represent points as well as intervals so that one does not have to commit to a single type of objects, and a set of unary and binary constraints. The domains of point and interval variables are the set of real numbers and the set of ordered pairs of real numbers, respectively. Ordering constraints are disjunctions of the form

$$O_1(r_1)O_2 \vee \ldots \vee O_1(r_n)O_2$$

where O_1, O_2 are points or intervals, and each one of the r_i's is an admissible relation between O_1 and O_2, belonging to set of basic interval/interval, point/point/, point/interval and interval/point relations. Metric constraints are borrowed from distance algebra.

The relevant feature of these models is that they extend the expressive power of point algebras, preserving their expressive power.

3.1.2 Temporal databases

The design of a Temporal Database consists of three fundamental phases:

- definition of a **temporal domain**;
- definition of a **temporal data model**;
- definition of a **query language**;

We skip the third point, and describe the first two ones.

Temporal Domains The definition of a temporal domain implies, first, the choice of the *temporal ontology*, i.e., we establish the structure of time choosing among the different alternatives listed above. Second, a *mathematical structure* must be defined. The most common structure imposed on time instants is *order*: partial or total (linear). In the existing temporal databases the orders are usually linear, but there exist exceptions: for instance, a linear transitive order that is not irreflexive or asymmetric is used to model *cyclic time* or a partial order that satisfies *left-linearity* is used to model *branching time* [Eme90].

The standard temporal domains in the linear context are: natural numbers $\mathbf{N}=(\mathcal{N}, 0, <)$, integers $\mathbf{Z} =(\mathcal{Z}\ , 0, <)$, rationals $\mathbf{Q}=(\ \mathcal{Q}\ , 0, <)$ and reals $\mathbf{R} =(\ \mathcal{R}\ , 0, <)$. Equality is usually assumed to be available in the temporal domain (it is violated, for example, by TSQL2, a temporal extension of SQL2 [ea94b, ea94a]). In most temporal database applications, it is assumed that time is discrete and isomorphic to natural numbers [Sno92]. In the AI view [All83, DMP91], however, time is usually assumed to be dense. Moreover, continuous time has turned out to be extremely valuable in mathematics and physics, and is now the standard in the area of *hybrid systems* [GNRR93]. The notion of *constraint formula* [KKR90] makes possible to finitely represent dense sets in computer storage.

Temporal Data Models Chomicki in [Cho94] splits the conceptual level of a temporal database in *abstract* temporal database and *concrete* one. An *abstract database* captures the formal, representation-independence meaning of a temporal database, while a concrete one provides a specific, finite representation for it in terms of a specific temporal data model. Moreover, he defines three different, but equivalent, ways to view an abstract temporal database. The *model-theoretic* view, which is the most basic, treats an abstract temporal database as a first-order structure. The *snapshot* view treats it as a function that maps every instant to a set of tuples. Finally, the *timestamp* view treats it as a mapping associating a set of instants with every tuple. The notion of abstract temporal database is not sufficient to deal with temporal database applications, because they may be infinite, while only finite objects can be explicitly represented in computer storage; it is necessary to design concrete data models that are specific finite representations of abstract temporal databases.

The essential aim of the definition of a temporal data model is to associate temporal characteristics to data and to provide temporal data management and retrieval functionalities. [MP93] introduces the notion of *basic temporal entities* as any elementary entity in a database, for instance, an attribute in a relational database or an instance of an object, with which temporal information is associated with. In general, a basic temporal entity has values varying with the time; for instance, the salary of an employee is increased at a given date or the air temperature of a room is sampled at regular intervals. The values of basic temporal attributes have been classified according to their persistence in time as follows:

- Stable values.
- Uniformly changing values.
- Values changing according to a given function of time.
- Irregular values.
- Discrete values.

The association of time-related characteristics is not limited to time sequences for a single attribute of an entity or a relation. In general, reasoning on time-dependent data requires modeling all time-related characteristics of an entity and of other entities related to it.
The first thing to consider is the identification of parts of a schema that present homogeneous characteristics in terms of their time-related aspects. Several proposals in the literature in this direction are based on the relational model of data. Most of the data models that have attempted to extend the relational model to incorporate the time dimension have generally taken one of the following approaches. In the first approach (e.g. [Sno87, ea94b]) each relation scheme is extended to include some time attributes (see Table 1(a)). Values of these time attributes come from some

(a) A temporal relation from [Sno87]

Name	Rank	From	To
Jane	Assistant	9.71	12.76
Jane	Associate	12.76	11.80
Jane	Full	11.80	∞

(b) A temporal relation from [Gad88]

Name	Salary	Dept.
[11,61)John	[11,50)15K	[11,45)Toys
	[50,55)20K	[45,61)Shoes
	[55,61)25K	

Table 1

fixed temporal domain. Given a tuple, the values of these time attributes indicate when the information (i.e. the remaining values of the tuple) is valid. Unlike the first approach that stays within first-normal form relations, the second approach (e.g. [CT85, Gad88]) deals with non first-normal form relations. Under the second approach, a timestamp is associated with each attribute value of a tuple (see Table 1(b)). The intuition is that the actual value of a particular attribute at a particular time point is the value whose attached timestamp contains that time point.

[Sno87] proposes a distinction based on characteristics of durations of temporal properties: *event relations* concerns attributes representing events that take place at an instant of time in the real world, while in *interval relations* the valid time is an interval of time, and a value of the attribute is considered to apply to the duration of the interval.

Multidimensionality of time Multiple temporal dimensions are necessary to model intervals (as pairs of points) or multiple kinds of time, e.g., *valid time* (the time when a fact is true in the modeled reality) and *transaction time* (the time when a fact is stored in the database) [SA86]. Based on this classification of the time, four models of data are defined, depending on which time they support:

- Snapshot databases do not offer any temporal support (Figure 3.1a).
- Rollback databases support transaction time (Figure 3.1b).
- Historical databases support valid time (Figure 3.1c).
- Bitemporal databases support both valid and transaction time (Figure 3.1d).

3.1.3 Time in object-oriented models

Adding the time dimension to object-oriented systems aims at modeling how the entities and the relationships the objects denote may change over time. Often an entity or a relationship is created at a given time and it relevant to a system for only a limited period of time. It may also happen that an entity could be killed and reincarnated many times before being definitively destroyed. Object-oriented temporal systems must then support mechanisms to create, to kill, to reincarnate,

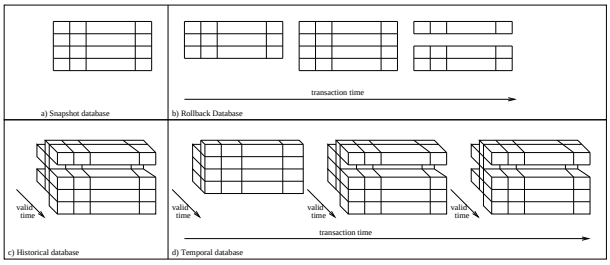


Figure 3.1: Ahn and Snodgrass Classification

and to definitively remove objects [MP93]. Furthermore, during their existence objects may change the values of their properties or their operations. Object-oriented temporal systems must provide mechanisms to update the structure and behavior of objects to model their temporal evolution. The introduction of the time dimension also result in the definition of new object constructor, for example, constructors that allow one to compose elementary events into processes, and to assert temporal integrity constraints.

Modeling Temporal Objects

The first refinement to the object structure allowed by the introduction of time, concerns the value of object attributes. Systems, without time dimension, associate to each single-valued (multiple-valued) attribute of objects only a single value or a set of values, usually the current one(s). If and when an attribute's value is update, its previous value is lost. On the contrary, object-oriented temporal systems provide each attribute of an object with a value for each time instant. The sequence of values in time of a given attribute is called its *history*.

Research on temporal object-oriented databases is still in its early stage. Although various object-oriented temporal models have been proposed (see for an overview [Sno95a]), there is no amount of theoretical work comparable to the work reported for the relational model. Some approaches associate a timestamp with the whole object state, whereas other associate a timestamp separately with each object attribute. Among the approaches associating timestamps with single attribute values, the majority regard the value of a temporal attribute as a function from a temporal domain to the set of legal values for the attribute. Another important characteristic, along which the existing approaches can be classified, is whether temporal, immutable and non-temporal attributes are supported. An *immutable* attribute is an attribute whose value cannot be modified during the object lifetime. For instance, in T_CHIMERA [BFG96], a set *temporal types* is defined, that type variable for which the history of changes over time is recorded and variables for which only the current value is kept. Classes in T_CHIMERA can be *static* or *historical*. A class is static if all its attributes are static, i.e. they do not have as

On_Service			
Veh	City	Time	Con
V_1	Athens	i_1	$9 \leq i1_L, i1_R \leq 15$
V_2	Patra	i_2	$10 \leq i2_L, i2_R \leq 14$

Figure 3.2: A generalized temporal relation

domain a temporal type, it is historical otherwise. Moreover a *lifespan* is associated with each class, representing the time interval during which the class has existed.

3.1.4 Time in constraint databases

An interesting application of constraint databases to the modeling of time is the approach by Koubarakis [Kou94]. The aim of Koubarakis' proposal is to unify into a single framework the representation of *definite*, *indefinite*, *finite*, and *infinite* temporal information about *valid time* of tuples by introducing a hierarchy of temporal data models: *temporal relations*, *generalized temporal relations*, and *temporal tables*. The temporal relations model is simply a theoretical tool for providing semantics for the other two models. The model of generalized temporal relations offers the ability to represent infinite temporal information in a finite way. The model of temporal tables extends the model of generalized temporal relations by allowing indefinite (i.e., imprecise and/or qualitative) temporal information.

Time is assumed to be linear, dense and unbounded. There is a single time line which is isomorphic to the set of rational numbers $\mathbb{Q}$. The time entities are points and intervals, defined as pairs of points.

In order to show how infinite and indefinite information is represented in this approach we present two examples taken from [Kou94]. In Figure 3.2 the generalized temporal relation represents the availability of some vehicles: vehicle V_1 is on service from 9am to 3pm in Athens. The constraint $9 \leq i1_L, i1_R \leq 15$ allows one to express that the vehicle is on service for each subinterval in the interval $[9am, 3pm]$, thus a generalized relational tuple is a *finite representation* of an infinite set of relational tuples. Generalized temporal relations extend temporal relations in the following way:

- *variables* are allowed as values for temporal attributes, and
- every tuple is tagged with a *local temporal constraint* on its variables.

Variables in a generalized relational tuple are essentially universally quantified.

On the other hand, Figure 3.3 shows a temporal table which allows the representation of indefinite information by using *existentially quantified variables* called e-variables. E-variables represent values that *exist* but are not known precisely. The *global temporal constraint* asserts all the known constraints on such variables. In

On_Service			
Veh	City	Time	Con
V_1	Athens	i_1	$w1_L \leq i1_L, i1_R \leq w1_R$
V_2	Patra	i_2	$w2_L \leq i2_L, i2_R \leq w2_R$

$$G : w1_L = 9, 14 \leq w1_R \leq 16, 10 \leq w2_L \leq 10.5, 2 \leq w2_R - w2_L \leq 4$$

Figure 3.3: A temporal table and the global constraint G

particular in the example, $w1_L$, $w1_R$, $w2_L$ and $w2_R$ are e-variables and the global constraint G states that (i) interval $w1$ starts at time 9 and ends at some time between 14 and 16, (ii) interval $w2$ starts at some time between 10 and 10.5 and its duration is 2 to 4 units of time.

Temporal tables are syntactic devices for the representation of indefinite (finite or infinite) temporal data. Each table is semantically equivalent to a set of possible worlds, which capture all the possible ways the domain of application could have been according to the indefinite information. Each possible world can be represented by a generalized temporal relation. One can determine all the possible worlds represented by a table as follows:

- Find an assignment to the e-variables which satisfies the global constraint.
- Substitute these values for the e-variables in the table.

3.2 Spatial models and formalisms

Following Güting [Güt94], a spatial database is a database system that offers spatial data types in its data model and query language and supports such data types in its implementation, providing at least spatial indexing and efficient algorithms for spatial join.

Like in the case of temporal information, also in this field we can distinguish two different strand of research, the one aiming to develop efficient representation of spatial data and the other designing formalism to reason with spatial entities.

3.2.1 Spatial data models

The spatial data models proposed in the literature have been classified as follows [Par95b]:

- the Raster Model [GS93, Sam90]. An object is given by a finite number of its raster points. Raster points are equally distributed following an easy geometric pattern, which is normally a square.

- the Spaghetti Model or Vector Model [LT92]. An object is intensionally deduced from its contour, which is approximated by segments and points, and it is usually represented by a list of points.

- The Peano Model [LT92] also uses a finite number of object-points, but here these points are distributed non-uniformally, according to the form of the object. This distribution is based on the Peano curve.

- Polynomial Model [KKR90, PVV94]. The spatial information is stored in relations. Each relation contains at most one spatial attribute for representing spatial objects described by a semi-algebraic set of the form

$$\{(x_1, \ldots, x_n) \mid x_1, \ldots, x_n \in \mathbf{R} \ \wedge \ \phi(x_1, \ldots, x_n)\}$$

  where $\phi(x_1, \ldots, x_n)$ is a semi-algebraic formula that contains boolean operators, existential and universal quantifiers and whose terms are comparisons, using $<$, $\leq$, $>$, $\geq$, $=$, $\neq$, between polynomial whose coefficients are rational numbers. $x_1, \ldots, x_n$ are free variables of ϕ.

- PLA-Model [LT92]. Only some kind of topological information is handled without dealing with the exact position and form of the spatial objects.

Applicative Data Models (Storage oriented Data Models)

The Raster model [GS93, Sam90] and the Vector model [LT92] or Spaghetti model are the basic models for the representation of spatial data in Geographical Information Systems (GIS). The focus is more on practical issues such as implementation efficiency and less on theoretical concerns such as genericity [PK98]. Let us give a more detailed description of such models.

Raster model The space is regularly divided into cells. Thus raster data are structured as an array or grid of cells 3.4. Each cell in a raster is addressed by its position in the array (row and column). Rasters are able to represent a large range of computable spatial objects. Among the advantages of rasters are the fact that they require simple data structures (many commonly used programming languages support array handling and operations), and they are well adapted to set intersection operations, such as map overlay. On the other hand, among the limits of rasters are their size (a raster when stored in a raw state with no compression can be inefficient in terms of usage of computer storage), and the cost of computing topological relations, for instance to find all neighbor cells with same function value.

Vector model or 2-Spaghetti Model The vector model allows one to represent only spatial objects that are composed of a finite set of closed polygons. Actually, each object can be decomposed into a set of triangles (some are degenerate triangles

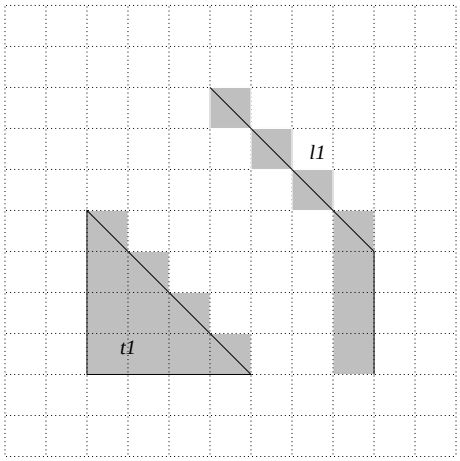


Figure 3.4: Raster model.

like line segments or points) where each triangle is represented by its three corners in a single relational database table. Worboys [Wor92] uses the notion of *simplex* to capture this basic component of the object.

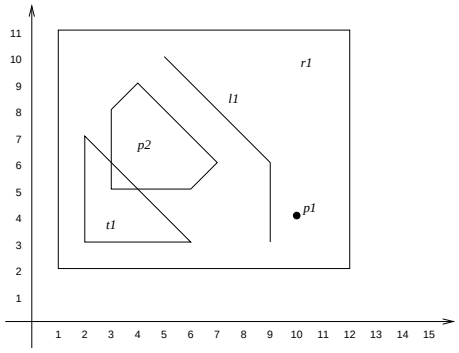


Figure 3.5: Some spatial objects.

Example 3.1 Let us consider Figure 3.5. In the 2-spaghetti data model this figure is represented by the following relation.

ID	x	y	x'	y'	x''	y''
p_1	10	4	10	4	10	4
l_1	5	10	9	6	9	6
l_1	9	6	9	6	9	3
t_1	2	3	2	7	6	3
r_1	1	2	1	11	12	11
r_1	12	11	12	2	1	2
p_2	3	8	5	3	6	5
p_2	3	8	6	5	4	9
p_2	4	9	6	5	7	6

Each tuple specifies the name of the object it belongs to, the vertex coordinates of the simplex. For instance, notice that a rectangle is represented by two triangles. □

The abstract semantics of a 2-spaghetti data model is for each object the set of points that belong to the area of the plane that is within any of the triangles associated with that object. Thus, it is point-based. Usually, the query languages hide the internal representation of spatial objects, referring explicitly to the objects themselves. They provide operators that work on polygons, e.g. overlap, union, intersection, however the semantics of those languages is also point-based.

This model is popular because there exist very efficient algorithms for triangulating polygons [Sam90] and for detecting properties, such as whether two polygons overlap, whether a point lies in a polygon and so on.

Polynomial model

Among the models which are primarily geared to handle theoretical issues we describe the polynomial model, interested mainly in representing the exact position of objects, as the representative one of the class of theoretical models.

The Polynomial Model is more general than the Spaghetti Model since it allows to represent all geometrical figures definable in elementary geometry, i.e. first-order logic over the reals. As a representative of this class, we describe the approach of Paredaens et al. [PVV94] which extends constraint databases and offers a calculus for specifying spatial queries.

A spatial database scheme, $\mathcal{S}$, is a finite set of *relation names*. Each relation name, R, has a type $\tau(R)$ which is a pair of natural numbers $[n, m]$. Here, n denotes the number of non-spatial columns and m the dimension of the single spatial column of R. Consider a relation type $[n, m]$. An *intensional tuple* (or *ituple*) of type $[n, m]$ has the form $(a_1, \ldots, a_n; \phi(x_1, \ldots, x_m))$, with $a_1, \ldots, a_n$ non-spatial values of some domain, $\mathbf{B}$, and with $\phi(x_1, \ldots, x_m)$ a quantifier-free real formula of arity m. An *i-relation* of type $[n, m]$ is a finite set of ituples of type $[n, m]$. An *i-instance* of a scheme $\mathcal{S}$ is a mapping on $\mathcal{S}$, assigning to each relation name R of $\mathcal{S}$ an i-relation of type $\tau(R)$.

This model differs from constraint databases because it allows more general formula in the spatial component ($\wedge$ and $\vee$ of constraints) and it distinguishes a spatial and a value part (ordinary attribute) of an ituple. This distinction is needed to define the notion of *genericity* (consistency criterion) for spatial queries.

Example 3.2 In the following table, we show how Figure 3.5 is modeled in the Polynomial model of Paredaens et al. [PVV94].

ID	x, y
p_1	$x = 10 \wedge y = 4$
l_1	$(5 \leq x \wedge x \leq 9 \wedge y = -x + 15) \vee (x = 9, 3 \leq y, y \leq 6)$
t_1	$2 \leq x, x \leq 6, 3 \leq y, y \leq 7, y \leq -x + 9$
r_1	$1 \leq x, x \leq 12, 2 \leq y, y \leq 11$
p_2	$3 \leq x, 5 \leq y, x - 1 \leq y, y \leq -x + 13, x + 5 \geq y$

The spatial component is modeled by its analytic representation. □

The semantics of an ituple $t = (a_1, \ldots, a_n; \phi(x_1, \ldots, x_m))$ of type $[n, m]$ is the possibly infinite subset of $\mathbf{B}^n \times \mathbf{R}^m$ called *e-relation*, denoted by $ext(t)$ and defined as the Cartesian product $\{(a_1, \ldots, a_n)\} \times S$, in which $S \subseteq \mathbf{R}^m$ is the semi-algebraic set defined by $\{(x_1, \ldots, x_m) \mid \phi(x_1, \ldots, x_m)\}$. The semantics of an i-relation r is the e-relation denoted as $ext(r)$ and defined as $\bigcup_{t \in r} ext(t)$. The semantics of an i-instance I is defined in the obvious way in term of the e-relations of $I(R)$, where R is a relation name of the database.

The provided query language gives the usual set of operations of the relational algebra (union, difference, Cartesian product, selection, projection) suitably modified in order to deal with the spatial component of the ituples. For instance the Cartesian product between two i-relations r_1 of type $[n, m]$ and r_2 of type $[n', m']$ is defined as follows:

$$r_1 \times r_2 = \{(val(t), val(t'); spat(t) \wedge spat(t')) \mid t \in r_1, t' \in r_2\}, \text{ of type } [n+n', m+m']$$

where $val(t)$ and $spat(t)$ respectively denote the value and spatial part of the ituple t. The ituples belonging to the Cartesian product have as value part the value parts of an ituple t in r_1 and an ituple t' in r_2 and as spatial part the conjunction ($\wedge$) of the spatial parts of t and t'.

The disadvantage of this approach is that it may be too general in an implementational perspective. For this reason, the set of spatial objects has been restricted in various ways [VGV95, BBC97b, Gru97]. The basic idea is to focus on "linear" data types, that is object represented by formulas built from linear polynomials. This choice is motivated by two factors: this class of data types includes most of the common used spatial data, and operations on typical linear data, such as lines, and polygons, have efficient algorithms. In Section 3.2.3 we give further hints on the linear constraint model.

OracleSpatial database

In this section we present the first commercial spatial database, released by Oracle.

Oracle8i Spatial, referred to as Spatial, provides a standard SQL schema and functions that facilitate the storage, retrieval, update, and query of collections of spatial feature in a Oracle8i database. It consists of four components:

1. a schema that describe the storage, syntax, and semantics of supported geometric data types,
2. a spatial indexing mechanism,
3. a set of operators and functions for performing area-of-interest and spatial join queries,
4. administrative utilities.

The spatial attribute of a spatial feature is the geometric description of its shape in some coordinate space. This is referred to as **geometry**.

This release of Spatial supports two mechanisms of representing geometry. The first, an object-relational scheme, and the second a relational scheme. The first corresponds to a "SQL with Geometry Types" implementation of spatial feature tables, and the second an implementation of spatial feature tables using numeric SQL types for geometric storage. We consider the first one.

Data model The Spatial data model is a hierarchical structure consisting of elements, geometries, and layers, which correspond to representations of spatial data. Layers are composed of geometries which in turn are made up of elements.

An *element* is the basic building block of a geometry. the supported spatial elements are those illustrated in Figure 3.6 and Figure 3.7. For example, elements might model star constellations (point clusters), roads (line strings), and country boundaries (polygons). Each coordinate in an elements is stored as an (x, y) pair. The exterior ring and the interior ring of a polygon with holes are considered as two distinct elements that together make up a complex polygon.

Spatial supports three geometric primitive types:

- 2-D point and point cluster,
- 2-D line strings,
- 2-D n-point polygons.

2-D points are elements composed of two ordinates, x and y, often corresponding to longitude and latitude. Line strings are composed of one or more pairs of points that define line segments. Polygons are composed of connected line strings that form a closed ring and the interior of the polygon is implied. Figure 3.6 illustrates the supported geometric primitive types.

Self-crossing polygons are not supported although self-crossing line strings are. If a line string crosses itself, it does not become a polygon.

Together with primitive types, Spatial supports six further geometric types, shown in Figure 3.7:

- 2-D arc line strings (all arcs are generated as circular arcs),

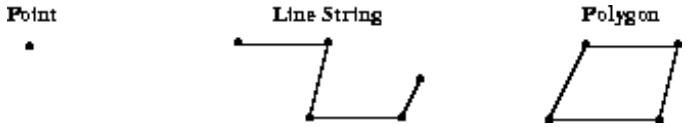


Figure 3.6: Geometric primitive types

- 2-D arc polygons,
- 2-D compound polygons,
- 2-D compound line strings,
- 2-D circles,
- 2-D optimized rectangles.

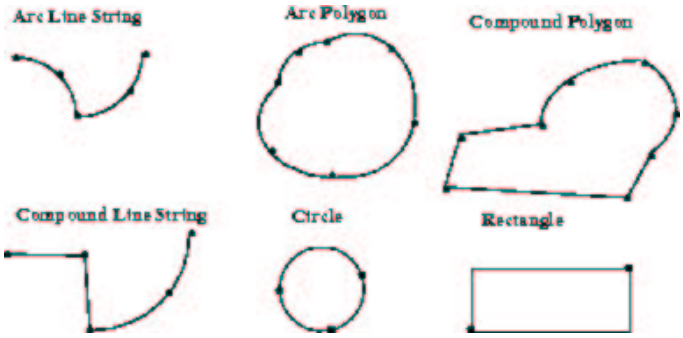


Figure 3.7: Other geometric types supported by Spatial

A *geometry* is the representation of a user's spatial feature, modeled as an ordered set of primitive elements. A geometry can consist of a single element, which is an instance of the supported primitive types, or a homogeneous or heterogeneous collection of elements. A multi-polygon, such as one used to represent as set of islands is a homogeneous collection. A heterogeneous collection is one in which the elements are of different types. Spatial supports the following geometry types:

- polygons with multiple holes,
- collection, that a heterogeneous collection of elements, where the polygons must be disjoint,
- multipoint,
- multistring,
- multipolygon, composed of multiple, disjoint polygons.

A *layer* is a heterogeneous collection of geometries having the same attribute set. For example, one layer in GIS might include topographical features, while another describes population density, and a third describes the network of roads and bridges in the area (lines and points). Each layer's geometries and their associated spatial index are stored in the database in standard tables.

Example 3.3 The example shows how to store a polygon with hole in the database. The polygon is illustrated in Figure 3.8. The coordinate values for element 1 and 2 are:

Element 1= [p1(6,15), p2(10,10), p3(20,10), p4(25,15), p5(25,35), p6(19,40), p7(11,40), p8(6,25), p1(6,15)]

Element 2= [h1(12,15), h2(15,24)]

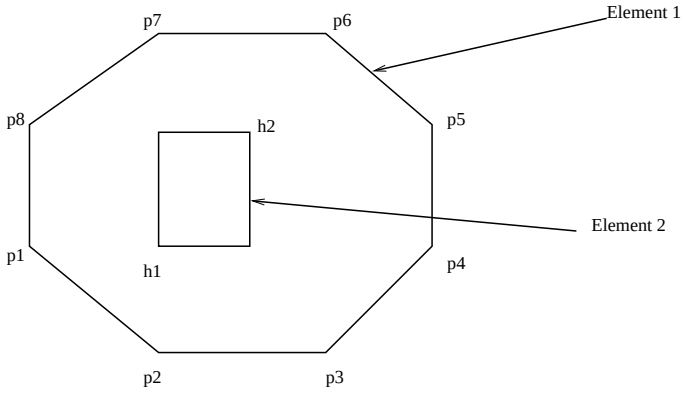


Figure 3.8: Example Geometry Obj_1

We assume that a table Parks was created as follows:

```
create table Parks   (name, varchar(32),
                      shape mdsys.sdo_geometry);
```

The SQL statement for inserting the data for geometry Obj_1 is:

```
insert into Parks
        values   ('Obj_1', mdsys.sdo_geometry(3, null, null,
                 mdsys.sdo_elem_info_array(1,3,1, 19,3,3)
                 mdsys.sdo_ordinate_array (6,15, 10,10, 20,10, 25,15, 25,35,
                 19,40, 11,40, 6,25, 6,15, 12,15, 15,24)));
```

The geometry Obj_1 is stored using the predefined Spatial type sdo_geometry, in which the number 3 specifies the type of the geometry (in this case polygon with hole), the array mdsys.sdo_elem_info_array(1,3,1, 19,3,3) specifies how to interpret the ordinates stored in the sdo_ordinate attribute. It has two triplets (1,3,1, 19,3,3), because there are two elements. The first number indicates the offset with the sdo_ordinate array where the first ordinate for this element is stored, and the other two numbers indicate the element type. □

Spatial relations and filtering Spatial uses *filter methods* to determine the spatial relationship between entities in the database. The spatial relation is based on geometry locations. The most common are based on topology and distance. For example, the boundary of an area consists of a set of curves that separate the area from the rest of the coordinate space. The interior of an area consists of all points in the area that are not on its boundary. Given this, two areas are adjacent if they share part of the boundary but not points in their interior. Next, the distance between two spatial objects is the minimum distance between any points in them. Two objects are said to be in a given distance of one another if their distance is less than the given distance.

Spatial has two filter methods. One method evaluates topological criteria and a second method determines if two spatial objects are within an Euclidean distance of each other.

The filter evaluating topological criteria is called relate and implements the 9-intersection model for categorizing binary topological relations between points, lines, and polygons. The 9-intersection model has been described in Section 3.2.2. The other filter is called within_distance.

3.2.2 Formalisms for spatial reasoning

Among the numerous formalisms designed to query and reason with spatial data, we can distinguish between the ones interested to describe the exact position of objects, and that allowed then to perform *quantitative reasoning* and the ones oriented to represent only certain kind of information, to perform *qualitative reasoning*. Formalism belonging to the first group are all the constraint algebras presented in 2.3.3. Among those belonging to the second group, we recall the *topological* ones [EF91, KPV95, KPV98, PSV96] which focus on topological information, and those ones which are interested in *directional (or orientation) relations*, such as [GE99, IC98, ZF96]. In the following we present the fundamental work by Egenhofer on topological relations [EF91, Ege91, Ege95].

9-intersection model

Egehnofer [EF91, Ege91] focuses on the class of *topological relationships* between spatial objects. A topological relation is a property invariant under homeomorphisms, for instance it is preserved if the objects are translated or scaled or rotated. We restrict our attention to a space with only two dimensions and we present the 9-intersection model. Such a model is based on the intersection of the interior (A°,B°), the complement (A^-,B^-) and the boundary (δA,δB) of two 2-dimensional connected objects A and B. Therefore a relation between A and B is represented by $R(A, B)$

as follows:

$$R(A,B) \quad = \quad \begin{pmatrix} \delta A \cap \delta B & \delta A \cap B^{\circ} & \delta A \cap B^{-} \\ A^{\circ} \cap \delta B & A^{\circ} \cap B^{\circ} & A^{\circ} \cap B^{-} \\ A^{-} \cap \delta B & A^{-} \cap B^{\circ} & A^{-} \cap B^{-} \end{pmatrix}$$

Each of these intersections can be empty or not empty. Of the 2^9 possible different topological relations, only eight of these relations can be realized between two 2-dimensional objects and they are illustrated in Figure 3.9.

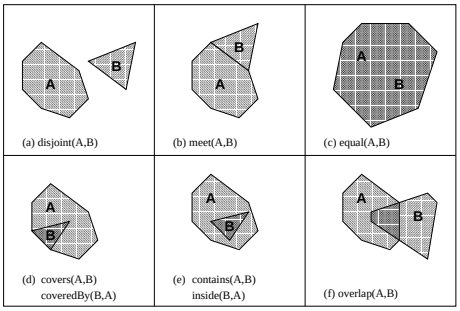


Figure 3.9: Topological relations between two 2-dimensional objects

As observed in [PSV96], these relations are not sufficient to fully characterize the topological relationships between objects. For instance, if three regions A, B and C pairwise overlap, the intersection of the three together might be or might not empty but this cannot be expressed by the previous relations. [PSV96] defines languages which are *complete* for topological properties.

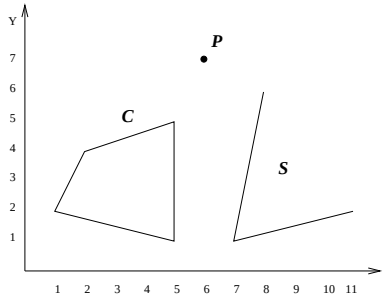
3.2.3 Space in constraint databases

The constraint databases found their key intuition on finitely representing possibly infinite sets, their data models offer a promising paradigm for the representation of many sorts of data in a unified framework. These motivations lead to their suitability for modeling spatial objects. In this way, a spatial object seen as an infinite set of points, can be dealt with as a first class citizen with an explicit representation. For instance a convex polygon, which is the intersection of a set of half-planes, is defined by the conjunction of the inequalities defining each half-plane. A non convex polygon by the union (logical disjunction) of a set of convex polygons.

To model spatial information in Constraint Databases, the *linear constraints over rationals* or *real polynomial constraints* is often used as logical theory of constraints. The trend, initiated by [PdBG94] to develop first-order based algebraic query languages using as underlying theory the real polynomial constraint theory, is moving to the definition of languages based on the linear constraint theory because of its efficient algorithms. Linear constraints have been shown to fit the need of spatial data in the vector mode. The fundamental difference between the two approaches

relies on the explicit definition of the spatial objects in the data model (e.g. a polygon is explicitly the infinite set of points it contains versus the implicit definition by the sequence of border points). It is possible to manipulate spatial objects through standard set operations (e.g. those of relational algebra).

Example 3.4 Here is an example of modeling with linear constraints:



ID	x	y	x'	y'	x"	y"
P	6	7	6	7	6	7
S	8	6	7	1	7	1
S	7	1	11	2	11	2
C	1	2	2	4	5	1
C	2	4	5	1	5	5

2-spaghetti model

$object(P, x, y) : - x = 6, y = 7.$
$object(S, x, y) : - y \leq 6, 1 \leq y, 5x - y - 34 = 0.$
$object(S, x, y) : - 7 \leq x, x \leq 11, x - 4y - 3 = 0.$
$object(C, x, y) : - y \leq 2x, x + 4y - 9 \leq 0, x \leq 5, x - 3y + 10 \leq 0.$

Linear constraint database model □

A lot of research has developed, starting from Kanellakis et al. generalized relational model [KKR90], new models that can represent better spatial objects. Most of the existing proposals define new generalized relational models using particular classes of constraints to represent the tuples, and algebras that correspond exactly to the traditional relational algebra, and whose semantics is defined with respect to the effect of the usual algebraic operators on infinite relations. This is the case of the *real polynomial model* of [PdBG94] (described in the previous section) and the *linear constraint model* of Grumbach et al. [GST94]. Other proposals redefine logically the generalized relational model to model complex objects as the *nested model* of Bertino et al. [BBC98] and design algebras that apply on special generalized tuples. In the following we describe the linear constraint model and the nested model.

Linear Constraint Model Linear constraint models consider linear constraints in the first-order language $\mathcal{L} = \{\leq, +\} \cup \mathbb{Q}$ over the structure $\mathcal{Q} = \langle \mathbb{Q}, \leq, +, (q)_{q \in \mathbb{Q}} \rangle$ of the linearly ordered set of the rational numbers with rational constants and addition. Constraints are in this case linear equations or inequalities of the form $\sum_{i=1}^{p} a_i x_i \Theta a_0$, where Θ is a predicate among $=$ or $\leq$, the x_i's denote variables and the a_i's are integer constants.
Let $\sigma = \{R_1, \ldots, R_n\}$ be a database schema such that $\mathcal{L} \cap \sigma = \emptyset$, where $R_1, \ldots, R_n$ are relation symbols. We distinguish between *logical predicates* (e.g., $=, \leq$) in $\mathcal{L}$ and *relations* in σ. The following defines *finitely representable structures* ([GST94]).

Definition 3.1 (Finitely representable structures) Let $S \subseteq D^k$ be some k-ary relation. The relation S is *finitely representable in* $\mathcal{L}$ *over* $\mathbf{Q}$ ($\mathcal{L}$*-representable* for short) if there exists a quantifier-free formula $\phi(x_1, \ldots, x_k)$ in $\mathcal{L}$ with k distinct free variables $x_1, \ldots, x_k$ such that:

$$\mathbf{Q} \models \forall x_1 \cdots x_k (S(x_1, \ldots, x_k) \leftrightarrow \phi(x_1, \ldots, x_k)).$$

Let $\mathcal{A}$ be an expansion of $\mathbf{Q}$ to σ. The structure $\mathcal{A}$ if finitely representable (over $\mathbf{Q}$) if for every relation symbol R in σ, $R^{\mathcal{A}}$ is $\mathcal{L}$-representable (over $\mathbf{Q}$).

A (*database*) *instance* (*of* σ) is a mapping which associates with each k-ary relation symbol R in σ a quantifier-free formula in DNF with k distinct variables. The algebra of [GST94] called $\mathrm{ALG}_{\mathcal{L}}$ can be viewed as a simplified sub-language of the algebra previously described. In this case a n-ary relation R is represented by a quantifier-free formula ϕ, of the form:

$$\phi \equiv \bigvee_{i=1}^{k} \bigwedge_{j=1}^{l_i} \phi_{i,j}$$

where the $\phi_{i,j}$ are atomic formulas. ϕ can also be represented as a collection of constraint tuples t_i in the set notation:

$$\left\{ t_i | 1 \leq i \leq k, t_i = \bigwedge_{j=1}^{l_i} \phi_{i,j} \right\}$$

Two restrictions are imposed on the class of all infinite relations: the first one is to represent relations by quantifier free formulae and the second one enforces the formulae to be in disjunctive normal form (DNF), as in the generalized relational model of [KKR90]. Furthermore, they extend the previous classical concept of linear constraint relation and also consider relations that combine the uninterpreted domain $\mathcal{D}$ with the interpreted one $\mathbf{Q}$. The concept extends easily to such relations, with representation formulas in $\mathcal{L} \cup \mathcal{D}$, with two sorts of constraints, (i) equality constraints over objects of cD, and linear constraints over objects of $\mathbf{Q}$.

In terms of 2-dimensional applications, the restriction to DNF formulas implies that polygons must be decomposed into convex component polygons (convexified). Grumbach et al. in [GRS98a] call this restriction the *convex normal form* and generalize it, motivated both by its logical simplicity and its algorithmic efficiency, to a representation of objects with "holes" (see Figure 3.10):

Definition 3.2 (Convex with holes normal form). A formula is in the convex with holes normal form (CHNF) if it is either quantifier free in DNF, or it is of the form $F - F'$, where F and F' are two formulae in CHNF of the same arity, and $-$ is the set difference (i.e $\wedge, \neg$). □

The Figure 3.10 shows how a spatial object can be represented by a formula in DNF or in CHNF. In DNF it is seen as the union of the three polygons p_1, p_2, p_3 while in CHNF it is seen as the outside polygon p_1 minus the inner one p_2.

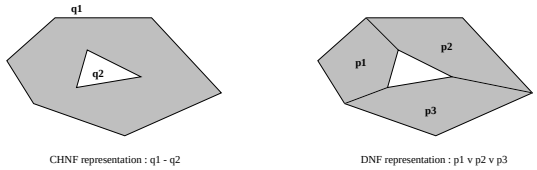


Figure 3.10: Representation of a polygon with a hole

The symbolic algebra for the CHNF representation contains the traditional operators of relational algebra. Let $F_1 = \alpha - \alpha'$ and $F_2 = \beta - \beta'$ be two formulas in CHNF, where α, α', β and β' are CHNF.

- $F_1 - F_2$ is defined recursively by $(\alpha \cap \beta) - (\alpha' \cup \beta')$.
- $F_1 \times F_2$ is defined recursively by $(\alpha \times \beta) - ((\alpha' \times \beta) \cup (\beta' \times \alpha))$.
- $\sigma_F(F_1)$ is defined recursively by $\sigma_F(\alpha) - \alpha'$.
- $F_1 \cup F_2$ is defined recursively by $(\alpha \cup \beta) - [(\alpha' \cup \beta') - ((\alpha' \cap \beta) \cup (\alpha \cap \beta') - \alpha' \cap \beta')]$.
- $F_1 - F_2$ is defined recursively by $(\alpha - \alpha') - (\beta - \beta')$.
- $\pi_{\bar{x}} F_1$ is defined recursively using an algorithm slightly more complex than the Fourier-Motzkin one but still quadratic when eliminating one variable. The output is $(\pi_{\bar{x}} \alpha) - \gamma$ where γ is obtained by computing the intersection points of α and β, and extracting those which are part of the boundary of γ.

Grumbach et al. in [GRS98a] compute the complexity of the new algebra, comparing it with linear algebra for the convex normal form. They showed that union, intersection, Cartesian product, and projection are quadratic, while selection is linear and set difference is constant time. Under the DNF representation, the complexity of union and set difference are inverted, that is union is constant time and set difference is quadratic.

Other languages have been proposed to model complex objects in the generalized relational model [BK95, GS95]. For example, $\mathcal{L}yriC$ is a language introducing constraints in an object-oriented framework, thus allowing any kind of nesting [BK95].

3.3 Spatio-temporal data models

Among the temporal and spatial approaches we presented in the previous sections only constraint databases can provide a uniform data model for both time and space. In fact, if we consider as class of constraints linear polynomials, we can model both temporal and spatial data with efficient implementations.

For instance, Grumbach et al. have extended their approach for handling spatial information to include also the treatment of time, in [GRS98b].

Recall Definition 3.1 in Section 3.2.3. Linear constraint relations of Grumbach et al. represent two-dimensional spatial objects, therefore are defined on $\mathbf{Q}^2$.

Moving objects, i.e. spatial objects changing with time, are represented as a (spatio-temporal) linear constraint relation in $\mathbf{Q}^3$, adding a third dimension for time. Figure 3.11 illustrates the trajectory of a moving object. There are 4 tuples that associate space and time. Tuple A for instance has a geometry (a convex polygon in the plane) which gives the valid position of the object during the interval $[t_1, t_2]$. It can be finitely represented as a conjunction of one constraint on t and 5 constraints on x and y (one for each of the 5 half-planes). By adding disjunction, one can easily represent the evolution of the geometry (shape, position or both) with respect to time: tuple B is for instance a segment in the plane associated with the time interval $[t_2, t_3]$. Note that for each tuple T the following holds: $T \equiv \pi_{space}(T) \times \pi_{time}(T)$.

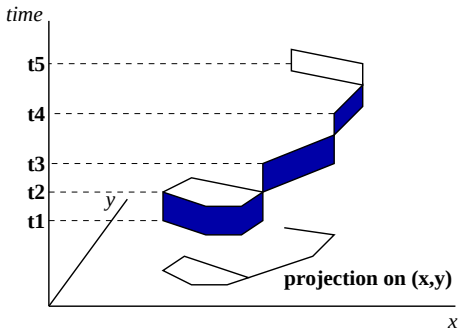


Figure 3.11: A spatio-temporal object

Another unified model for spatial and temporal information has been introduced by Worboys [Wor92]. The basic idea is to attach to each elemental spatial object a bitemporal element which is a finite set of pairs of intervals, representing respectively the database and the event times of the object. The elemental spatial object is a *simplex* which is either a single point, finite straight line segment or triangular area. The pair $\langle$*simplex*, *set of bitemporal elements*$\rangle$ is called *ST-simplex*. A complex spatial object, temporally referenced, is called *ST-complex* and it is modeled by a finite set of ST-simplexes, subject to some constraints that we do not report.

Example 3.5 [Wor94] An example of *ST-complexes* is given in Figure 3.12. To each vertex, line, and to the area (i.e the simplexes) of the triangle a bitemporal element is associated, having on the horizontal the transaction time and on the vertical the valid time.

□

The query language provides operators to make the union, difference and intersection of two ST-complexes, and the selection on the temporal and spatial component. Moreover, it is also given a topological operator to return the boundary of an ST-complex.

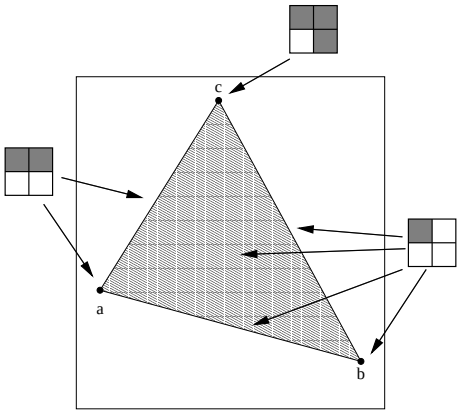


Figure 3.12: ST-complex

In the Worboys model, the temporal and the spatial dimensions are independent and this prevents from describing spatial relations which may be parametric with respect to time, i.e. with respect to their evolution, for instance the relationship that exists between time and the area covered by an incoming tide.

Finally, Chomicki and Revesz [CR97, CR99] have introduced a new model, called Parametric 2-spaghetti model, that generalizes the 2-spaghetti model by allowing an interaction between spatial and temporal attributes. Vertex coordinates can now be *linear functions of time*.

Example 3.6 [CR97] Let us suppose that we have a rectangular area on a shore and a tide is coming in. The front edge of the tide water is a linear function of time. The behavior of the tide is described in Figure 3.13: at 1:00 am the area flooded by the water will be a point, at 8:00 a triangle and so on. These data are represented in the following table.

ID	x	y	x'	y'	x''	y''	*From*	*To*
r_1	3	10	3	10	3	10	1	$+\infty$
r_1	3	10	3	$11-t$	$2+t$	10	1	8
r_1	3	10	3	3	10	10	8	$+\infty$
r_1	3	3	10	10	3	$11-t$	8	10
r_1	10	10	3	$11-t$	10	$18-t$	8	10
r_1	3	3	10	10	3	1	10	$+\infty$
r_1	10	10	3	1	10	8	10	$+\infty$
r_1	3	1	10	8	$t-7$	1	10	17
r_1	$t-7$	1	10	$18-t$	10	8	10	17
r_1	3	1	10	1	10	8	17	$+\infty$
r_1	10	1	10	1	10	1	17	$+\infty$

It is worth noting that this relation is not a standard one because its values are parametric. □

Query languages for the Parametric 2-spaghetti data model have not been defined yet. This model has been used in [CLR99] to animate spatio-temporal objects.

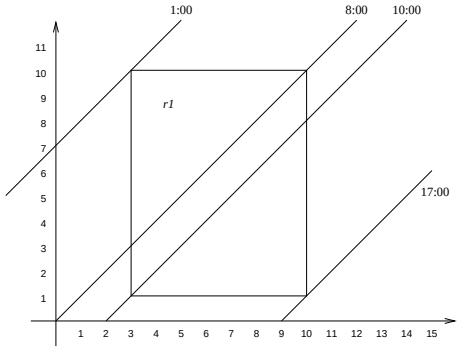


Figure 3.13: Parametric 2-spaghetti model.

Finally, we want to analyze the approach proposed by Erwig, Güting, Schneider and Vazirgiannis [EGSV99, GBE+98]. They define collections of *abstract data types* for spatial values changing over time. Essentially, they introduce data types for moving points and moving regions together with a set of operations on such entities.

The design of the model for spatio-temporal data is based on a type constructor τ which transforms any given atomic data type α into a type $\tau(\alpha) = \mathit{time} \rightarrow \alpha$ where $\mathit{time} = \mathbb{R}$, that is a continuous model of time is supported. In particular this constructor is applied to two spatial data types, *point* and *region*, allowing one to represent two fundamental abstractions *moving point* and *moving region*. A moving point is of type $\mathit{mpoint} = \mathit{time} \rightarrow \mathit{point}$ and it is used to represent objects for which only the position in space is relevant, whereas a moving region is of type $\mathit{mregion} = \mathit{time} \rightarrow \mathit{region}$ and both the position and extent are of interest, i.e., the object can change its position and/or grow or shrink.

Then a set of operations are defined on these types. For instance consider the following functions

$$\begin{array}{lll} \mathit{mdistance}: & \mathit{mpoint} \times \mathit{mpoint} & \rightarrow \mathit{mreal} \\ \mathit{trajectory}: & \mathit{mpoint} & \rightarrow \mathit{line} \\ \mathit{lenght}: & \mathit{line} & \rightarrow \mathit{real} \end{array}$$

where $\mathit{mreal} = \tau(\mathit{real})$ (a moving real) and *line* is a data type describing a curve in two-dimension space. The function *mdistance* returns the distance between two moving points at all times, and hence returns a time changing real number; *trajectory* is the projection of a moving point onto the plane and *length* returns the total length of a line value.

The presented data types can be embedded into any DBMS data model as attributes data types, and the operations be used in queries. Consider, for example, the relation flights defined as follows

flights(id: *string*, from: *string*, to: *string*, route: *mpoint*)

We can ask a query *Give me all flights from Düsseldorf that are longer than 5000 kms*:

SELECT id
FROM flights
WHERE from = "DUS" AND *length(trajectory(route))* > 5000

In [ES99] this approach has been extended in order to provide a framework which allows one to build more and more complex spatio-temporal predicates starting with a small set of elementary ones. These predicates are particularly suitable to characterize developments, for instance, a predicate for capturing the scenario of a point entering or crossing a region can be easily modeled.

Chapter 4

A Formal Background to build *Constraint Objects*

In this chapter, we describe the design of an object-oriented constraint database model, based on flat constraints.

Our approach exploits algebraic structures to represent a general class of constraints, namely flat constraints, in an object-oriented framework, in order to have a methodology to define in a correct way constraint classes, whose instances (*constraint objects*) are parameterized by their vocabulary and their domain of interpretation. Algebraic structures generate *constraint objects*, (CO), i.e. instances of a predefined *class* or *type* in the object-oriented model, that means that new CO are constructed using logical connectives and existential quantifiers within a multi-typed algebra. Hence, the data model is adapted from CO, conceptually represented by constraints, i.e. symbolic expressions of a restricted first-order logic.

Following the approach of Brodsky et al. [BSCE97, BSCE99], we adopt the *Constraint Comprehension Calculus*, as *Constraint Calculus* to manipulate the CO, that is an integration of a constraint language (first-order logic + atomic constraints) within *monoid comprehensions*, which were suggested by [FM95] as an optimization-level language for querying object-oriented databases.

The chapter is organized as follows: Section 4.1 describes the family of constraints used to represent data; Section 4.2 presents the data model, defining the constraint data types and operations; Section 4.3 introduces the formalism of monoid comprehensions and our extension to query and manipulate the constraint objects; in Section 4.4, we define the schema of classes, using the Java platform. Finally in Section 4.5 we describe the motivations of our choices.

4.1 Constraints families

We start the design of our constraint data model with the definition of the *constraint families* and the *algebra*.

A constraint family $\mathcal{F}$ is defined by the choice of:

- an atomic constraint domain,
- the structure of the logical formula allowed.
- the required canonical form (e.g. whether to eliminate existential quantifiers, to eliminate each redundant disjunct etc.)

The definition of a constraint algebra amounts to choosing the operations, by which formulas of the families are composed. For instance, in the frameworks of Grumbach et al. [GST94], the atomic constraint domain is represented by linear constraints, the structure of formulas are quantifier-free DNF formulas and constraint algebra is the linear constraint algebra described in Section 3.2.3.

We follow the approach of Brodski et al. [BSCE97], in the choice of the families of constraints, but we let the atomic constraint domain "undefined", i.e. we define abstract data types to represent atomic constraints, that can be instantiated with different classes of constraints. The constraint families of our framework consist of *flat constraints*, i.e. atomic formulas without function symbols. As we will see in the following sections, this choice is due to have more facilities in modeling constraints using algebraic structures.

4.1.1 Atomic formulas

Recall Section 2.1. Flat constraints are defined on a *flat vocabulary*.

Definition 4.1 (Flat vocabulary) A flat vocabulary Ω consists of two sets: a set $\mathcal{C}$ of constant symbols and a set $\mathcal{P}$ of predicate symbols, together with an arity function that associates a natural number to each element of $\mathcal{P}$.

Terms of the first-order language over Ω consist only of variables and constants. An $\Omega-$structure $\mathcal{M} = \langle \mathcal{D}, \Omega \rangle$, where Ω is flat is defined assigning to each predicate symbol $p \in \mathcal{P}$ with arity n an n-ary relation, i.e a set in $\mathcal{D}^n$ and to each constant an element of $\mathcal{D}$.

Therefore, atomic flat formulas are n-ary predicates, whose terms are constants and variables. We denote with $AC(\Omega)$ the set of atomic formulas defined over Ω. In the following, Ω will denote a flat vocabulary.

The usage of flat formulas does not limit the expressiveness of the language. The language of flat constraints is equivalent to the first-order language (see [CK90a]). We show as example how to model two-dimensional linear constraints with flat constraints.

Example 4.1 Two-dimensional linear constraints over reals, see Section 2.1.2, are atomic formulas in free variables x, y, $\varphi(x, y)$, such that

$$\varphi(x, y) \equiv a \cdot x + b \cdot y + c \leq 0.$$

where $+, \cdot$ are the functions $+(x, y)$ and $\cdot(x, y)$ that compute respectively the sum and multiplication between real numbers. Let $plus(x, y, z)$ be a predicate that is true iff $x + y = z$ and $times(x, y, z)$ that is true iff $x \cdot y = z$, then $\varphi(x, y)$ is equivalent to the following flat formula in free variables x, y:

$\varphi_{flat}(x, y) \equiv \exists_{w,x_1,x_2,y_1}(p(w) \wedge plus(x_1, x_2, w) \wedge times(a, x, x_1) \wedge plus(y_1, c, x_2) \wedge times(b, y, y_1))$

where $p(w)$ is true iff $w \leq 0$.
Note that while $\varphi(x, y)$ is interpreted over the structure $(\mathcal{R}, \Omega)$ where $\Omega = (+, <, 0, 1)$, $\varphi_{flat}(x, y)$ is interpreted on the structure $(\mathcal{R}, \Omega_{flat})$ where $\Omega_{flat} = (\{p, times, plus\}, \{1, 2, 3, \ldots\})$.

□

4.1.2 Structure of the logical formulas

We defined the structure of the logical formulas over Ω, allowed in our framework.

Definition 4.2 (Flat constraints over Ω) Let $\mathcal{V}$ be a countable set of variables. Flat constraints over Ω are defined recursively as follows:

i. An atom is an atomic formula of the form $p(t_1, \ldots, t_n)$, such that $t_1, \ldots, t_n$ are terms defined from $\mathcal{V}$ and $\mathcal{C}$ and $p \in \mathcal{P}$ is an n-ary predicate.

ii. If φ_1 and φ_2 are formulas, then $(\varphi_1 \vee \varphi_2)$, $(\varphi_1 \wedge \varphi_2)$ are formulas.

iii. If φ is a formula and $x \in \mathcal{V}$ is a free variable in φ, $\exists_x \varphi$ is a formula.

iv. Formulas are generated only by a finite number of applications of (i), (ii) and (iii).

□

General formulas are then built up from atomic formulas by using the connectives defined in Definition 4.2. Given a formula (or flat constraint) $\varphi(x_1, \ldots, x_n)$ and $a_1, \ldots, a_n \in \mathcal{D}$, the satisfaction $\mathcal{M} \models \varphi(a_1, \ldots a_n)$ is defined in the usual manner, see Section 2.1.1. The set of all flat formulas over a vocabulary Ω is denoted by $FC(\Omega)$.

Before characterizing $FC(\Omega)$ as an algebraic structure, we have to define which formulas are allowed in our framework. We distinguish four families of constraints. Let $\varphi_i, i = 1, \ldots, k, \ldots$ be a formula in $FC(\Omega)$ and $X \subseteq \mathcal{V}$ be a set of free variables in φ_i, then:

$\mathcal{F}_C(\Omega)$: *Conjunctive*, stands for constraints represented in the form $\wedge_{i=1}^{n}\varphi_i$. The family of conjunctive constraints contains conjunctions of quantifier-free atomic formulas. For example generalized tuples , see Sections 2.2 and 2.3.2, belong to this family.

$\mathcal{F}_{EC}(\Omega)$: *Existential Conjunctive*, corresponds to the form $\exists_X \wedge_{i=1}^{n}\varphi_i$. To this family belong formulas, that are quantified conjunctions of eventually quantified atomic formulas. Constraints of $\mathcal{F}_C(\Omega)$ are also constraints of $\mathcal{F}_{EC}(\Omega)$.

$\mathcal{F}_{DC}(\Omega)$: *Disjunction of Conjunctions*, corresponds to the form $\vee_{i=1}^{m} \wedge_{j=1}^{n} \varphi_{i,j}$. Like $\mathcal{F}_C(\Omega)$, this family has quantifier-free formulas, but in this case the formulas are disjunctions of conjunctions of atomic formulas. The generalized relations, see Section 2.3.2, belong to this family. Constraints of $\mathcal{F}_C(\Omega)$ are also constraints of $\mathcal{F}_{DC}(\Omega)$.

$\mathcal{F}_{DEC}(\Omega)$: *Disjunction of Existential Conjunctive*, corresponds to the form $\vee_{i=1}^{m}(\exists_X \wedge_{j=1}^{n} \varphi_{i,j})$, that is equivalent to $\exists_X \vee_{i=1}^{m} \wedge_{j=1}^{n} C_{i,j}$[1]. Constraints of $\mathcal{F}_{EC}(\Omega)$ and $\mathcal{F}_{DC}(\Omega)$ are also constraints of $\mathcal{F}_{DEC}(\Omega)$.

From the definition of the constraint families it follows that $\mathcal{F}_C(\Omega) \subseteq \mathcal{F}_{EC}(\Omega), \mathcal{F}_{DC}(\Omega)$ and $\mathcal{F}_{EC}(\Omega), \mathcal{F}_{DC}(\Omega) \subseteq \mathcal{F}_{DEC}(\Omega)$. Figure 4.1 shows the relationships between the families of CO, the arrow indicates $\subseteq$.

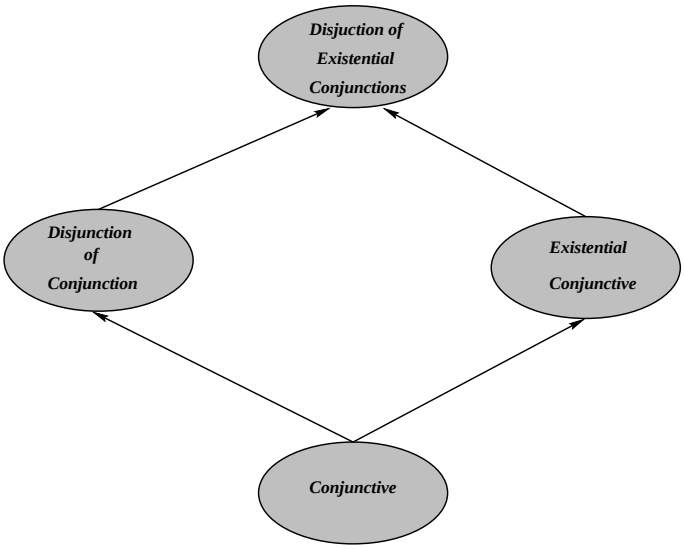


Figure 4.1: Families of CO objects

Definition 4.3 (Constraint object) A *Constraint Object*, CO, is any formula belonging to one of the constraint families $\mathcal{F}_C(\Omega)$, $\mathcal{F}_{EC}(\Omega)$, $\mathcal{F}_{DC}(\Omega)$, $\mathcal{F}_{DEC}(\Omega)$. Let $\mathcal{F}_{CO}(\Omega)$ denote the set of all possible CO. Thus $\mathcal{F}_{CO}(\Omega) \subseteq FC(\Omega)$.

[1]It follows from the axioms of first-order logic, see [CL73]

4.1.3 An Algebraic structure for $\mathcal{F}_{CO}(\Omega)$

As defined in [JJL87], the semantics of constraints are given in terms of an algebraic structure that interprets constraint formulas. In this section we introduce an incremental algebraic specification for constraint objects, because we are interested in the algebraic properties on which the semantic constructions are based. Constraints in $\mathcal{F}_{CO}(\Omega)$ are then viewed as elements of an algebraic structure, providing a uniform treatment of semantic domain (collections of constraints) and domain dependent operators, [DB99, DB00a].

We start with the definition of *monoid*, which is the simplest algebraic structure used having only one operation. Monoids are used to generate the constraint objects, belonging to the families $\mathcal{F}_C(\Omega)$ and $\mathcal{F}_{EC}(\Omega)$.

Then, monoids will be extended to *semirings*, which have two operations and can be used to generate constraints of the remaining families $\mathcal{F}_{DC}(\Omega)$ and $\mathcal{F}_{DEC}(\Omega)$.

Definition 4.4 (Monoid [BL97]) A monoid is an algebraic structure $(\mathcal{C}, \mathcal{Q})$, where $\mathcal{C}$ is the domain and $\mathcal{Q}$ is an associative binary function $\otimes$, i.e. $(a \otimes b) \otimes c = a \otimes (b \otimes c)$ for all a, b, $c \in \mathcal{C}$; and there is an identity element, say $\mathbf{1}$, such that $a \otimes \mathbf{1} = \mathbf{1} \otimes a = a$ for any $a \in \mathcal{C}$. A monoid $\mathcal{M} = (\mathcal{C}, \mathbf{1}, \oplus)$ is called *commutative* and *idempotent* if the function $\oplus$ is commutative[2] and idempotent[3].

Definition 4.5 (Semirings[4] [Eil74]) A *semiring* is an algebraic structure $(\mathcal{D}, \mathbf{1}, \mathbf{0}, \otimes, \oplus)$ satisfying the following properties:

R_1. $(\mathcal{D}, \mathbf{1}, \otimes)$ and $(\mathcal{D}, \mathbf{0}, \oplus)$ are commutative and idempotent monoids.

R_2. $\mathbf{0}$ is an *annihilator* for $\otimes$, i.e. for every $c \in \mathcal{D}$, $c \otimes \mathbf{0} = \mathbf{0} \otimes c = \mathbf{0}$.

R_3. for any possibly infinite family $\{a_i\}_{i \in I}$ of elements of $\mathcal{D}$: the sum $a_1 \oplus a_2 \oplus \cdots$, denoted $\sum_{i \in I} a_i$ exists and is unique, i.e. it is a well defined element in $\mathcal{D}$. Moreover associativity, commutativity and idempotence of $\oplus$ apply to infinite as well as to finite applications of $\oplus$.

R_4. $\otimes$ is $left-$ and $right-distributive$ over applications of $\oplus$, i.e. for any a, b, $c \in \mathcal{D}$, $a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$ and $(a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c)$.

□

Let' s denote one of the families $\mathcal{F}_C$ and $\mathcal{F}_{EC}$ with ${\mathcal{F}_{CO}}^{\wedge}$ and one of the families $\mathcal{F}_{DC}$ and $\mathcal{F}_{DEC}$ with ${\mathcal{F}_{CO}}^{\wedge,\vee}$ to highlight the connectives used in the respective families.

[2] $\forall\, a, b \in \mathcal{C} : a \otimes b = b \otimes a$

[3] $\forall\, a \in \mathcal{C} : a \otimes a = a$

[4] Actually, Definition 4.5 define *closed semirings*. We abuse with notation and we call them semiring, but referring to the closed ones.

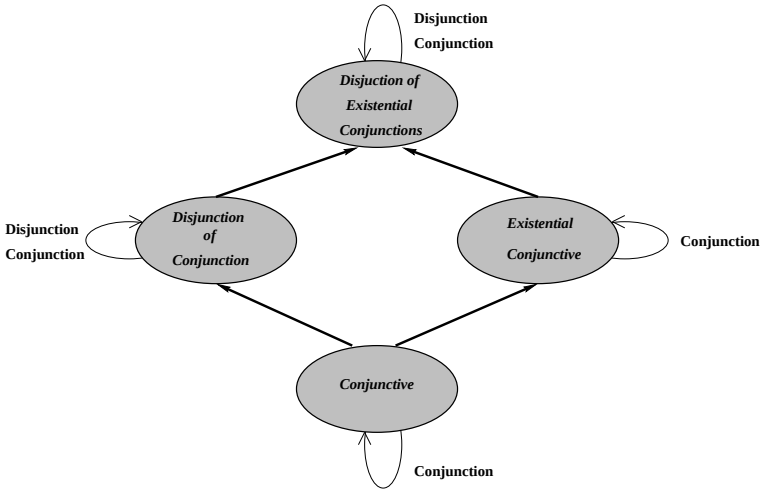


Figure 4.2: Families of CO objects with operations

Figure 4.2 shows the four families and the operations under which they are closed. The set ${\mathcal{F}_{CO}}^{\wedge}$ of conjunctive and existential conjunctive formulas is closed under the operation $\wedge$ and contains as element **true**, while ${\mathcal{F}_{CO}}^{\wedge,\vee}$ is closed under the operations $\wedge$ and $\vee$ and contains elements as **true** and **false**. Hence the structures

$\mathcal{M} = ({\mathcal{F}_{CO}}^{\wedge}, \mathbf{true}, \wedge)$
$\mathcal{S} = ({\mathcal{F}_{CO}}^{\wedge,\vee}, \mathbf{true}, \mathbf{false}, \wedge, \vee)$

are algebraic structures (in a wide sense of the word). At this point, no algebraic equation formulated in terms of fundamental operations is identically satisfied in these structures, for example the operations $\wedge$ and $\vee$ are neither commutative nor associative. The situation changes essentially when we introduce the notion of consequence and use it to construct from $\mathcal{M}$ and $\mathcal{S}$ some derivative structures. To this end we have first to explain how the formal language Ω can be interpreted, i.e. how definite meanings can be ascribed to symbols, formulas, and sentences.

As a base for interpretation we choose an arbitrary relational structure $\mathcal{R} = (\mathcal{D}, r_0, \ldots, r_\xi, \ldots)_{\xi<\beta}$, where $\mathcal{D}$ is an arbitrary non-empty set and $r_0, \ldots, r_\xi, \ldots$ are finitary relations among elements of $\mathcal{D}$. The sequence of relations r_ξ in $\mathcal{R}$ has the same length as the sequence of predicates p_ξ in Ω; each relations r_ξ is assumed to be of the same rank as the corresponding predicate p_ξ, so that, e.g. r_ξ is a binary relation if p_ξ is a two-place predicate. This implies that the logical connectives assumed the usual meaning.

The symbol **true** has the meaning of some particular sentence which is obviously true and the symbol **false** as the negation of **true**.

The interpretation of Ω in $\mathcal{R}$ extends in an obvious way from symbols to formulas and sentences, to which we can apply the equivalence relation, (see Section 2.1.1). We can now define a set of axioms for the sets of families. In the following φ_1, φ_2, ...

are arbitrary formulas belonging to ${\mathcal{F}_{CO}}^{\wedge}$, and ψ_1, ψ_2, ... are arbitrary formulas belonging to ${\mathcal{F}_{CO}}^{\wedge,\vee}$ and, the symbol $\forall$ bounds all free variables of the formulas to which it is applied.

Axioms for ${\mathcal{F}_{CO}}^{\wedge}$:

a_1: $\mathcal{R} \models \forall(\varphi_1 \wedge \varphi_2 \leftrightarrow \varphi_2 \wedge \varphi_1)$ (commutativity)

a_2: $\mathcal{R} \models \forall(\varphi_1 \wedge (\varphi_2 \wedge \varphi_3) \leftrightarrow (\varphi_1 \wedge \varphi_2) \wedge \varphi_3)$ (associativity)

a_3: $\mathcal{R} \models \forall(\varphi \wedge \varphi \leftrightarrow \varphi)$ (idempotence)

a_4: $\mathcal{R} \models \forall(\varphi \wedge \mathbf{true} \leftrightarrow \varphi)$ (identity element)

Axioms for ${\mathcal{F}_{CO}}^{\wedge,\vee}$:

b_1: The axioms $a_1 - a_4$,

b_2: $\mathcal{R} \models \forall(\psi_1 \vee \psi_2 \leftrightarrow \psi_2 \vee \psi_1)$ (commutativity)

b_3: $\mathcal{R} \models \forall(\psi_1 \vee (\psi_2 \vee \psi_3) \leftrightarrow (\psi_1 \vee \psi_2) \vee \psi_3)$ (associativity)

b_4: $\mathcal{R} \models \forall(\psi \vee \psi \leftrightarrow \psi)$ (idempotence)

b_5: $\mathcal{R} \models \forall(\psi \vee \mathbf{false} \leftrightarrow \psi)$ (identity element)

b_6: $\mathcal{R} \models \forall(\psi \wedge \mathbf{false} \leftrightarrow \mathbf{false})$ (annihilator)

b_7: $\mathcal{R} \models \forall(\psi_1 \wedge (\psi_2 \vee \psi_3) \leftrightarrow (\psi_1 \wedge \psi_2) \vee (\psi_1 \wedge \psi_3))$ and $\forall((\psi_1 \vee \psi_2) \wedge \psi_3 \leftrightarrow (\psi_1 \wedge \psi_3) \vee (\psi_2 \wedge \psi_3))$ (left- and right-distributivity)

The proof is obvious, as it comes directly from the axioms of the first-order logic, see [CL73].

The two sets of axioms establish criteriums to derive equivalent formulas, i.e. the equivalence relations is based on the above axioms. The equivalence relation partitions our sets in mutually exclusive equivalence classes, that we called respectively $\Phi^{\wedge}_{CO}$ and $\Phi^{\wedge,\vee}_{CO}$.
$a_1 - a_4$ and $b_1 - b_7$ prove the followings two theorems:

Theorem 4.6 The algebraic structure $\mathcal{M} = (\Phi^{\wedge}_{CO}, \mathbf{true}, \wedge)$ is a monoid.

Theorem 4.7 The algebraic structure $\mathcal{S} = (\Phi^{\wedge,\vee}_{CO}, \mathbf{true}, \mathbf{false}, \wedge, \vee)$ is a semiring.

Remark 4.8 Monoids and closed semirings summarize in an algebraic framework, all aspects of dealing with composition of constraints, such as unification and set union. Idempotence, associativity and commutativity are necessary to allow the operator $\vee$ to model in a general way the merging together of information via set

union. The operator $\wedge$ corresponds to conjunction of constraints and plays the important role of collecting information during computation. Distributivity allows the representation of constraints as possibly infinite disjunctions of conjunctions (also called in [GSG94] *simple constraints*).
Monoids and closed semirings are thus an appropriate algebraic generalization to model constraint construction.

Syntactic Equality and Projection

As explained in Rem. 4.8, monoids and semirings tailor all aspects of conjunctions and disjunctions of constraints. Other two important operations are *projection* and *variable renaming/dependencies*. Therefore, we need a further structure, which allows us to define axioms for them.

Cylindric algebras, formed by enhancing semirings with a family of unary operations called *cylindrifications*, provide a suitable framework for this [HJA71]. The intuition is that given a formula φ, the cylindrification operation $\exists_X \varphi$ yields the constraint obtained by projecting out from φ all information about the variables in X. To handle variable dependencies between constraints, cylindric algebras provide the *diagonal elements*. Such elements, denoted by $d_{x,y}$, model atomic formulas of the form $x = y$ for arbitrary variables x and y. This provides an algebraic characterization for syntactic equality.

Given a set of variables $\mathcal{V}$, let $\mathcal{C} = (\Phi_{CO}^{\wedge,\vee}, \wedge, \vee, \mathbf{true}, \mathbf{false}, \exists_X, d_{x,x'})_{X \subseteq \mathcal{V}; x,x' \in \mathcal{V}}$ be an algebraic structure where

- $\mathbf{true}$, $\mathbf{false}$, $d_{x,x'}$ are distinct atomic elements of $\Phi_{CO}^{\wedge,\vee}$,
- the structure $(\Phi_{CO}^{\wedge,\vee}, \wedge, \vee, \mathbf{true}, \mathbf{false})$ is a semiring,

such that the following postulates are satisfied for any constraints $\varphi, \varphi' \in \Phi_{CO}^{\wedge,\vee}$, variables $\{x\}, X, Y \subseteq \mathcal{V}$ and $x, x'x" \in \mathcal{V}$:

C_1. $\exists_X \mathbf{false} = \mathbf{false}$;

C_2. $\varphi \vee \exists_X \varphi = \exists_X \varphi$;

C_3. $\exists_X(\varphi \wedge \exists_X \varphi') = \exists_X(\exists_X \varphi \wedge \varphi') = \exists_X \varphi \wedge \exists_X \varphi'$;

C_4. $\exists_X \exists_Y \varphi = \exists_{(X \cup Y)} \varphi$;

C_5. $\exists_X(\varphi \vee \varphi') = \exists_X \varphi \vee \exists_X \varphi'$;

D_1. $d_{x,x} = \mathbf{true}$

D_2. $d_{x,x'} = d_{x',x}$

D_3. $d_{x',x"} = \exists_{\{x\}}(d_{x',x} \wedge d_{x,x"})$ where $x \neq x', x"$;

D_4. $\exists_{\{x\}}(d_{x,x'} \wedge (\varphi \wedge \varphi')) = \exists_{\{x\}}(d_{x,x'} \wedge \varphi) \wedge \exists_{\{x\}}(d_{x,x'} \wedge \varphi')$.

Theorem 4.9 The structure $\mathcal{C}$ is a cylindric algebra.

The proof comes from the definition of cylindric algebra in [HJA71].

The meaning of existential quantification is given by the axioms $C_1 - C_5$, while equality are specified by the axioms $D_1 - D_4$. Notice that Axioms D_3 and D_4 relate the notion of renaming of variables with equality.
It is possible to derive a family of operations δ_x^y, defined on $\Phi_{CO}^{\wedge,\vee}$ as follows, for $x, y \in \mathcal{V}$ such that $x \neq y$:

$$\delta_x^y \varphi = \exists_{\{x\}}(d_{x,y} \wedge \varphi) \tag{4.1}$$

These operations capture semantics of *variable renaming*, since intuitively they allow to change variables names. In particular, notions of "independence" and "occurrence" of variables apply in the obvious way to constraints in $\Phi_{CO}^{\wedge,\vee}$. Let φ be a constraint in $\Phi_{CO}^{\wedge,\vee}$, given a variable x, we define:

$$x \, ind \, \varphi \Leftrightarrow \delta_x^y \varphi = \varphi \tag{4.2}$$

for any $y \in \mathcal{V}$ such that $x \neq y$. A renaming of φ with respect to x is a constraint $\delta_x^y \varphi$ such that $x \neq y$ and $x \, ind \, \varphi$.

From the above axioms, it follows that for any constraints $\varphi, \varphi' \in \Phi_{CO}^{\wedge,\vee}$ and variables $x \in \mathcal{V}$, $X \subseteq \mathcal{V}$ and $y, y', y'' \in \mathcal{V}$ such that $x \neq y$, the following properties hold:

P1: $\exists_X \exists_X \varphi = \exists_X \varphi$;

P2: $\exists_{\{x\}} \varphi = \varphi$ if $x \; ind \; \varphi$ (in particular $\exists_{\{x\}} d_{y',y''} = d_{y',y''}$ when $x \neq y', y''$);

P3: $\partial_x^y \exists_{\{x\}} \varphi = \exists_{\{x\}} \varphi$;

P4: $\exists_X \mathbf{true} = \mathbf{true}$; $\exists_X \varphi = \mathbf{false}$ iff $\varphi = \mathbf{false}$;

P5: $\exists_{\{x\}} d_{x,y} = \mathbf{1}$;

P6: $(d_{y,y'} \wedge d_{y',y''}) \wedge d_{y,y''} = d_{y,y''}$ (transitivity).

In particular, from P3 if x is bound in φ then $x \; ind \; \varphi$. Therefore, if φ is a renaming apart of φ' with respect to x, then $x \; ind \; \varphi$. The property P2 extends to conjunctions of constraints and describes the interaction of projection with (hidden) variables: for any constraints φ and φ' and X a set of variables such that $x \; ind \; \varphi$ for every $x \in X$, holds:

$$\exists_X (\varphi \wedge \varphi') = \varphi \wedge \exists_X (\varphi').$$

There is an important relation between existential quantification (hiding variables) and renaming apart of constraints with "fresh" variables. Namely, given constraints φ and φ', holds:

$$\varphi \wedge \exists_{\{x\}}\varphi' = \exists_{\{y\}}(\varphi \wedge \partial_x^y \varphi'), \text{ for } y \; ind \; \varphi, \varphi' \text{ and } y \neq x$$

Example 4.2 This example describes the case of *linear constraints*, already introduced in Section 2.3.3. In this case the number of variables are restricted *a priori* to some fixed value n. In the following $\vec{x} = (x_1, \ldots, x_n)$ is a point in $\mathbf{R}^n$ and x_i is its i-th element.
The atomic constraints, in this case, are the *hyperplanes*, i.e. the sets of points $\vec{x} \in \mathbf{R}^n$ satisfying the linear equation of the form:

$$a_1 x_1 + \cdots a_n x_n = b$$

and the corresponding *halfspaces*:

$$a_1 x_1 + \cdots a_n x_n < b$$
$$a_1 x_1 + \cdots a_n x_n > b$$

The *conjunctive constraints* are the *convex polyhedra*, i.e. the (possibly unbounded) set of points constituting the intersection of a finite number of half-spaces.
Let $\mathcal{V}_n = \{x_1, \ldots, x_n\}$ be a set of n variables and $\mathcal{P}$ the set of all space regions in $\mathbf{R}^n$ defined as possibly infinite unions of convex polyhedra. For any finite n, the constraint system of n-dimensional linear constraints is

$$(\mathcal{P}, \cap, \cup, \mathbf{R}^n, \emptyset, \exists_X, [x_i, x_j])_{\{x_i, x_j\}, X \in \mathcal{V}}$$

Each constraint $c \in \mathcal{P}$ can be represented as a set of conjunctions of the above linear equations and disequations on the variables $\mathcal{V}_n = \{x_1, \ldots, x_n\}$.
The variable restriction operation $\exists$ is performed by *cylindrification parallel to an axis*[HJA71]. For any constraint $c \in \mathcal{P}$ and $i \leq n$, we define:

$$\exists_{x_i} c = \{\vec{y} \in \mathbf{R}^n | y_j = x_j \; for, \vec{x} \; and \; j \neq i\}$$

$\exists_{x_i} c$ is the cylinder generated by moving the point set c parallel to the x_i axis (see Figure 4.3).

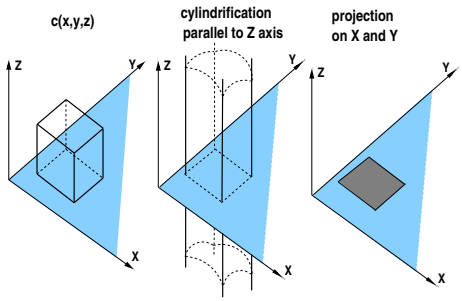


Figure 4.3: Linear constraint

□

4.2 The data model

Recall Section 2.4. A database consists of a set of *object classes* of different types. Each object class has an associated set of *objects*; each object has a number of attributes with values drawn from certain *domains* or *atomic data types*. There may be additional features, such has object valued attributes, methods, object class hierarchies, etc. But the essential features are the ones mentioned above.

In our model, we want to treat constraints as a normal data type, to do that we exploit the properties of constraint families described in Section 4.1.3, that allow to treat constraints as *collection data type*. In fact, monoids and semiring are particularly suitable in data models to formally define such types.

In this section, we define first the abstract data types to represent constraints. Then, we describe how to use monoids and semirings to represent uniformly types in the data model, and in particular way the type constraint.

4.2.1 Constraint types

Beside objects, we are interested in attributes describing "constraints". Hence, objects can refer to the "constraint", representing particular features. In Figure 4.4, the object "House" has among its attributes, two references to constraints, which describe the geometry and life time of the house, respectively.

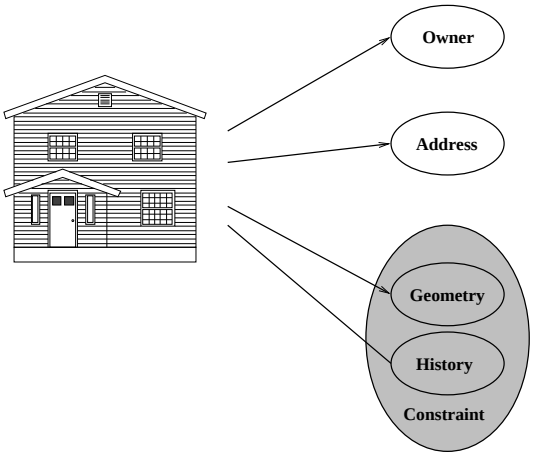


Figure 4.4: The object "House"

Therefore, we would like to define data types, or in fact *many-sorted algebras* containing several related types and their operations, for constraints.

We distinguish between *primitive (or atomic) types* and *constructed types*, depicted in Table 4.1.

Types, generated by their signature are e.g. int, $pred$, set, dec, etc. The set type constructor is applicable to all types in the kind $BASE$, hence all types that can be constructed by it are $set(int)$, $set(real)$, $set(string)$ and $set(bool)$; in the same way

Type constructor	Signature	
int, *real*, *string*, *bool*		$\longrightarrow BASE$
list	$BASE$	$\longrightarrow LIST$
pred		$\longrightarrow PREDICATE$
atom	$PREDICATE$	$\longrightarrow ATOM$
c	$ATOM$	$\longrightarrow C$
ec	C	$\longrightarrow EC$
dc	C	$\longrightarrow DC$
dec	$EC \cup DC$	$\longrightarrow DEC$
set	$BASE \cup DEC$	$\longrightarrow SET$

Table 4.1: Signature describing the type system

the type constructors *c*, *ec*, *dc* and *dec* construct the types $c(atom)$, $dec(ec)$, $dec(dc)$, etc.

So far we have just introduced some names for types. In the sequel, we describe their semantics by defining their *carrier* sets, i.e. the domain on which they will be interpreted.

We start with the constant (or primitive) types, i.e. type constructors without arguments, and then discuss proper type constructors.

Base types The base types are *int*, *real*, *string* and *bool*. All base types have the usual interpretation, except that each domain is extended by the value $\perp$ (undefined).

Definition 4.10 (Base types) The carrier sets for the types *int*, *real*, *string* and *bool*, are defined as:

$$A_{int} = \mathbf{Z} \cup \{\perp\},$$

$$A_{real} = \mathbf{R} \cup \{\perp\},$$

$$A_{string} = V^* \cup \{\perp\}, \text{ where } V \text{ is a finite alphabet,}$$

$$A_{bool} = \{true, false\} \cup \{\perp\}.$$

Constraint types The types defined in this paragraph describe constraints syntactically and thus instances of constraint types are logical formulas.

We defined first the type for predicate arguments and names, coming from the flat vocabulary Ω of Definition 4.1. Arguments consist of variables and constants.

Definition 4.11 (Variables and constants) The carrier set for the types *var* and *const* are defined as:

$$A_{var} = \mathcal{V} \cup \{\bot\},$$

$$A_{const} = \mathcal{C} \cup \{\bot\}.$$

Definition 4.12 (Predicates arguments) The carrier set of the types of predicates arguments is defined as:

$$A_{arg} = A_{var} \cup A_{const}.$$

Predicates are defined together with its rank.

Definition 4.13 (Predicates) The carrier sets of the types predicate name is defined as:

$$A_{pred\_name} = \mathcal{P} \cup \{\bot\}.$$

$$A_{pred} = A_{pred\_name} \times A_{int},$$

Then, we define the type *atom*; instances of this type are predicates, which represent the atomic formulas. Atomic formulas are set of lists $(pred, arg_1, \ldots, arg_{rank(pred)})$. The type constructor *atom* serves at this purpose. To do this, we need to define the function *rank* that given a predicate, returns its rank.

Definition 4.14 (Predicate's rank) The rank of a predicate is defined as follows:

$$rank : A_{pred} \rightarrow A_{int}$$

$$rank((p, i)) = i.$$

From the definition of *rank*, a class of sub-types of predicates can be defined, namely predicates with the specific rank n, for instance to these sub-types belong unary, binary predicates. We define a partition of the carrier set A_{pred}, denoted with $A_{pred}(i)$, such that:

$$A_{pred}(i) = \{p \in A_{pred} : rank(p) = i\}$$

Definition 4.15 (Atomic constraints' carriers) The carrier set for the type *atom* is defined as:

$$A_{atom} = \bigcup_{i=0}^{n} (A_{pred}(i) \times \underbrace{A_{arg} \times \cdots \times A_{arg}}_{i\,times}).$$

In order to control the computational complexity, we design a more flexible first-order logic structure by constructing four interrelated constraint classes, corresponding to the constraint families depicted in Figure 4.1. The containment relationship between the families corresponds to the type (class) hierarchy, in which the type *dec* is the super-type of the classes *dc* and *ec* and, *dc* and *ec* are super-type of the class *c*, meaning that a constraint object of type C may be used as an argument wherever its super-types are allowed. Informally, constraints types are described in the following:

- $c(atom)$ is the type corresponding to the family of conjunctive constraints.
- $ec(atom)$ is the type corresponding to the family of existential conjunctive.
- $dc(atom)$ corresponds to the family of disjunction of existential conjunctive.
- $dec(atom)$ corresponds to the family of disjunction of existential conjunctive.

The type constructor c yields for any subset of objects of given type $atom$, the conjunction of the object.

Definition 4.16 (c's carrier) The carrier set of type c is defined as follows:

$$A_c = \bigcup_{i=1}^{n} \{ p_1 \wedge p_2 \wedge \cdots \wedge p_i \,|\, \forall j,\, 1 \leq j \leq i,\, p_j \in A_{atom} \}.$$

The constraint types dc and ec are derived from the type c, and they yield for any subset of objects of given type c, the disjunction and the projection of the objects, respectively.

Definition 4.17 (dc's carrier) The carrier set of the type dc is defined as follows:

$$A_{dc} = A_c \cup \bigcup_{i=1}^{n} \{ c_1 \vee c_2 \vee \cdots \vee c_i \,|\, \forall j,\, 1 \leq j \leq i,\, c_j \in A_c \}.$$

Before introducing, the carrier set of the type ec, and thus the particular type of quantified constraints, we need a function $variable$, that given a conjunction of constraints returns the set of their variables.

Definition 4.18 (Function $variable$) Let $A^*_{var} = \{X \subseteq A_{var}\}$ and $c \in A_c, c = p_1 \wedge \cdots \wedge p_n$ and $\forall i, 1 \leq i \leq n, p_i = (q_i,\, m_i,\, arg_1, \ldots,\, arg_{m_i})$, where $m_i = rank(q_i)$. In fact each p_i is of type A_{atom}. The function $variable$ is defined as follows:

$variable : A_c \rightarrow A^*_{var}$

$$variable(c) = \begin{cases} \bigcup_{i=1}^{n} (variable(p_i)) & : \quad c = p_1, \ldots, p_n \\ \{x | x \in X \wedge \exists_j, 1 \leq j \leq m_i : x = arg_j\} & : \quad c = p_i \end{cases}$$

Definition 4.19 (ec' carrier) The carrier set of the type ec is defined as follows:

$$A_{ec} = \{\exists_{X_1} c_1 \wedge \cdots \wedge \exists_{X_n} c_n \,|\, \forall i,\, 1 \leq i \leq n : (c_i \in A_c,\, X_i \subseteq variable(c_i) \cup \{\emptyset\} \vee c_i \in A_{ec},\, X_i \subseteq freevar(c_i) \cup \{\emptyset\})\} \cup A_c$$

The function $freevar : A_{ec} \rightarrow A^*_{var}$ is defined as follows:

if $c \in A_c$ then $freevar(c) = variable(c)$

if $c = \exists_X s$, $s \in A_c$ then $freevar(c) = variable(s)/X$

if $c = s_1 \wedge \cdots \wedge s_j \wedge \exists_{Y_1} s_{j+1} \wedge \cdots \wedge \exists_{Y_{n-j}} s_n$ then

$$freevar(c) = \bigcup_{i=1}^{j} freevar(s_i) \cup \bigcup_{\substack{k=j+1 \\ l=j+1}}^{\substack{n-j \\ n}} freevar(\exists_{Y_k} s_l)$$

if $c = \exists_X(s_1 \wedge \cdots \wedge s_j \wedge \exists_{Y_1} s_{j+1} \wedge \cdots \wedge \exists_{Y_{n-j}} s_n)$ then

$$freevar(c) = (\bigcup_{i=1}^{j} freevar(s_i) \cup \bigcup_{\substack{k=j+1 \\ l=j+1}}^{\substack{n-j \\ n}} freevar(\exists_{Y_k} s_l))/X \qquad \square$$

The last constraint type is *dec*. Instances of this type are disjunctions of eventually quantified conjunctive constraints.

Definition 4.20 (*dec*'s carrier) The carrier set of the type *dec* is defined as follows:

$$A_{dec} = A_{ec} \cup A_{dc} \cup \{c_1 \vee \cdots \vee c_n : \forall i,\, 1 \leq i \leq n,\, c_i \in A_{ec}\}$$

4.2.2 Constraint objects and algebra

Constraint objects, i.e. instances of the types defined in the previous subsection, are manipulated by means of a *constraint algebra*, whose operators are expressed using connectives of the first-order logic, renaming of variables, and atomic (i.e. flat) constraints. For example, if φ and ψ are constraint objects in $x_1, \ldots, x_n$, their intersection can be represented by $\varphi \wedge \psi$; their union by $\varphi \vee \psi$ and the projection of φ on $x_1, \ldots, x_i$, $1 \leq i \leq n$, by $(x_1, \ldots, x_i)|\varphi$. Let's denote with $A_{constraint}$ the union of all the carriers sets defined for constraints, i.e. $A_{constraint} = A_c \cup A_{ec} \cup A_{dc} \cup A_{dec}$, then

$$\vee, \wedge : A_{constraint} \times A_{constraint} \rightarrow A_{constraint},$$

$$\exists_X : A_{constraint} \times A^*_{var} \rightarrow A_{constraint},$$

$\vee$ returns a constraint resulting from applying the union operation to the arguments. $\wedge$ returns the intersection of two constraints and $\exists_X$ performs the projection. All operations are defined on all constraints types, but they have a different definition depending on the type of constraints to which they are applied. Table 4.2 describes, for each family, the allowed operations and the type of the result, which may belong to a different family.

From Table 4.2, it follows that operators can be overloaded: for example $\wedge$ in $c(atom)$ is different from $\wedge$ in $dc(atom)$; they will be implemented differently and returns the results of different types (with different representations). The

	$\vee$	$\wedge$	(any)
$dec(ec, dc)$	$dec(ec, dc)$	$dec(ec, dc)$	$dec(ec, dc)$
$dc(c)$	$dc(c)$	$dc(c)$	$dec(ec, dc)$
$ec(c)$	$dec(ec, dc)$	$ec(c)$	$ec(c)$
$c(atom)$	$dc(c)$	$c(atom)$	$ec(c)$

Table 4.2: Operations on constraint types.

actual operator applied depends on the types of its arguments. As an example, $\wedge(c(atom), c(atom))$ will use the $c(atom)$ operator; whereas, $\wedge(dc(c), dc(c))$, as well as $\wedge(c(atom), dc(c))$ will use the operator from $dc(c)$. For the application of $c\_op(arg_1, arg_2)$ we will use the lattice structure of the type hierarchy, where

$$sub - type \preceq super - type$$

that, following the Figure 4.1, is:

$$c(atom) \preceq ec(c), dc(c), \;\; ec(c), dc(c) \preceq dec(ec, dc) \tag{4.3}$$

The actual *c_op* chosen is the one of the constraint type that is the least upper bound of type (arg_1) and type (arg_2) on which *c_op* is defined. For every *c_op* used, the least upper bound, if exists, is unique; hence there is no ambiguity. If no such bound exists, *c_op* is not allowed on arg_1 and arg_2.

In addition to the logical algebraic operators, all families have the following operators. Let φ be a constraint object with variables $x_1, \ldots, x_n$:

- $rename(\varphi, x_1/e_1, \ldots, x_n/e_n)$ where $x_1, \ldots, x_n$ are replaced with the variables $e_1, \ldots, e_n$.
- $satisfy(\varphi)$ to check satisfiability of the argument, i.e. whether there exists an assignment of $\mathcal{D}$ into its free variables that makes it true.
- $truth\_value(var\_assign, \varphi)$ returns the truth value of the constraint under the assignment *var_assign* of constants into the φ free variables.

Furthermore, constraint algebras operate on a family of canonical representations of constraint objects. For constraint objects $C_1, \ldots, C_n$ a first-order logic formula $\phi(C_1, \ldots, C_n)$, such as $\delta_{x_i}^{y_i} C_i \equiv \exists_{x_i}(x_i = y_i \wedge C)$, $1 \leq i \leq n$ denotes the variable replacement, defines the following constraint algebra operator *op*:

$$op : \mathcal{F}_{CO} \times \cdots \times \mathcal{F}_{CO} \longrightarrow \mathcal{F}_{CO}$$

such that:

1. replaces each C_i by the corresponding constraint expression,

2. does all variable replacement,

3. transforms the resulting constraint expression into the required (equivalent) representation in one of the families $\mathcal{F}_\Omega$.

The operator *op* has an interpretation $\mathcal{I}(op)$, which maps n relations to one. Given $\mathcal{I}(C_1), \ldots, \mathcal{I}(C_n)$, where $\mathcal{I}(C_i)$, $1 \leq i \leq n$, is the relational interpretation of C_i, introduced in Section 4.1 and $x_1, \ldots, x_m$, $\mathcal{I}(op)$ computes the following relation:

$$\{(x_1, \ldots, x_m) \mid \phi(C_1, \ldots, C_n)\} \tag{4.4}$$

Clearly, the duality between constraints and sets carries over the constraint algebra, that is, the following commutative property holds:

$$\mathcal{I}(op(C_1, \ldots, C_n)) = \mathcal{I}(op)(\mathcal{I}(C_1), \ldots, \mathcal{I}(C_n)) \tag{4.5}$$

4.2.3 Monoids and semirings as abstract data types

Monoids are particularly suitable in data models, to represent abstract data types [FM95], and in particular way collection data types, i.e. lists, sets, bags etc.

Example 4.3 Consider lists and sets. One way of constructing sets is to union together a number of singleton set elements, e.g. $\{1\} \cup \{2\} \cup \{3\}$ construct the set {1, 2, 3}. Similarly, one way of constructing lists is to append singleton list elements, e.g. [1] $\pm$ [2] $\pm$ [3] constructs the list [1, 2, 3] (where $\pm$ is the list append function). Both $\cup$ and $\pm$ are associative operations. The empty set {} is the identity of $\cup$, while the empty list [] is the identity of $\pm$. Therefore we say both $(set, \{\}, \cup)$ and $(list, [], \pm)$ are monoids. □

Primitive types, such as integers and booleans, can be represented as monoids too. For example, both $(int, 0, +)$ and $(int, 1, *)$ are integers monoids and both $(bool, false, \vee)$ and $(bool, true, \wedge)$ are boolean monoids. We take the denotation of [FM95], in which the monoids for collection types are called *collection monoids* and the monoids for primitive types *primitive monoids*.

Collection monoids have also a function that takes an elements of some type as input and constructs a singleton value of the collection type. It follows the formal definition of a collection monoid, taken from [FM95].

Definition 4.21 (Collection monoid) Let $T(\alpha)$ be a type determined by the type parameter α and $\mathcal{M} = (T(\alpha), zero, \oplus)$ is a monoid. The quadruple

$$(T(\alpha),\ f^{T(\alpha)},\ zero,\ \oplus),$$

where $f^{T(\alpha)}$ is a type constructor, is a collection monoid.

We can consider semiring as "structured nested monoids", i.e. commutative and idempotent monoids, where the constructed type $T(\alpha)$ is a monoid, that has a different function and a different identity elements, and the function of the internal monoid is right and left distributive over applications of the external operation. The domain of the structure semiring consists in this case of instances of a collection monoid. A formal definition of *collection semiring* follows.

Definition 4.22 (Collection semiring) Let $T(\alpha)$ be a type determined by the type parameter α, $\mathcal{M} = (T(\alpha), f^{T(\alpha)}, \mathbf{1}, \otimes)$ is a collection monoid. Let $T(\mathcal{M})$ be a type determined by the collection monoid $\mathcal{M}$ and $S = (T(\mathcal{M}), \mathbf{1}, \mathbf{0}, \otimes, \oplus)$ is a semiring, then the structure

$$\mathcal{S} = (T(\mathcal{M}), f^{T(\mathcal{M})}, \mathbf{0}, \oplus)$$

is a collection semiring.

The family $\mathcal{F}_C(\Omega)$, as simple conjunctions of conjuncts, is generated by a finite number of applications of the operator $\wedge$ on elements created by the type constructor c, defined in the last paragraph and thus are characterized by the monoids

$\mathcal{M}_C = (C, \mathit{true}, c, \wedge)$

Let p be of type *atom*, the type constructor performs a simple casting, such that

$c : A_{atom} \rightarrow A_c$
$c(p) = p_c$, where $p_c \in A_c$

The family $\mathcal{F}_{EC}(\Omega)$ contains existentially quantified conjuncts, that are generated by elements of the class $\mathcal{F}_C(\Omega)$,

$\mathcal{M}_{EC} = (EC, \mathit{true}, ec, \wedge)$

Let p be of type c, the type constructor is as follows

$ec : A_c \rightarrow A_{ec}$
$ec(p) = \exists_X p$, where $X \subseteq freevar(p)$

The families $\mathcal{F}_{DC}(\Omega)$ and $\mathcal{F}_{DEC}(\Omega)$, as proved in Section 4.1.3, can be represented as semirings, where the operations are $\wedge$ and $\vee$; this implies that the corresponding types can be defined as collection semirings, following the Definition 4.22.

Elements of the family $\mathcal{F}_{DC}(\Omega)$, being disjunctions of unquantified conjuncts, are simply generated by the monoid $\mathcal{M}_C$. From Definition 4.17 and the definition

of $\mathcal{M}_C$, it follows that:

$$\mathcal{S}_{DC} = (DC(\mathcal{M}_C), \textbf{false}, dc, \vee)$$

The type constructor performs also a casting:

$$dc : A_c \rightarrow A_{dc}$$
$$dc(\varphi) = \varphi_{dc}, \text{ where } \varphi_{dc} \in A_{dc}$$

where φ is an element of type C.

Elements of the family $\mathcal{F}_{DEC}(\Omega)$ are disjunctions of "eventually" quantified conjuncts, therefore they are generated by both monoid types $\mathcal{M}_{EC}$ and $\mathcal{M}_{DC}$. Let $\uplus$ be the type union, then

$$\mathcal{S}_{DEC} = (DEC(\mathcal{M}_{EC} \uplus \mathcal{M}_{DC}), \textbf{false}, dec, \vee)$$

In this case the type constructor performs the following casting:

$$dec : A_{ec} \cup A_{dc} \rightarrow A_{dec}$$

$$dec(\varphi) = \varphi_{dec}, \text{ where } \varphi_{dec} \in A_{dec}$$

Remark 4.23 $\mathcal{M}_{EC}$ is a nested monoid, because its domain is constructed from the monoid $\mathcal{M}_C$, but it is not a semiring, as the inner monoid has the same operation $\wedge$, while a property of semiring is to have two different binary operations.

Remark 4.24 The definition of constraint monoids and semirings follows the structures of constraint families depicted in Figure 4.1. For instance, the family $\mathcal{F}_C(\Omega)$ is a sub-family of the families $\mathcal{F}_{EC}(\Omega)$ and $\mathcal{F}_{DC}(\Omega)$, i.e. the type C is a subtype of the types EC and DC, that implies the mechanism of generation defined above, i.e. elements of structures $\mathcal{M}_{DC}$ and $\mathcal{M}_{EC}$ are constructed from constraints in $\mathcal{M}_C$.

4.3 Constraint calculus

In this section we introduce the formalism to query the constraint object-oriented model. Querying object-oriented databases implies dealing with collection types, nesting of type constructors, methods, etc. We have defined in Section 4.2.3 constraint data types as collection types, so we have a uniform construction of data types in the model. At this point we need a query language, suitable to query the defined structures. For this task, we exploit the well-known *monoid comprehension calculus*, see [FM95]. This calculus is based on monoids, that can capture most collection and aggregate operators currently in use for object-oriented databases.

Monoid comprehensions give a uniform way to express queries that simultaneously deal with more than one collection type.

4.3.1 The monoid/semiring comprehension calculus

We have shown in Section 4.2.3 that types can be represented as monoids, and in particular way constraints types. We have shown also that particular classes of constraints are representable as semiring that can be defined as nested monoids.

Since all types are represented as monoids, a query in our framework is a mapping from some monoids to a particular monoid. These mappings are called in [FM95] *monoid homomorphisms*. Recall Example 4.2.3, a monoid homomorphism from lists to sets is captured by an operation of the form

$$hom^{list \to set}(f)A$$

where A is a list and f is a function that takes an element x of A and returns the set $f(x)$. In other words, $hom^{list \to set}(f)A$ replaces [] in A by $\{\}$, $++$ by $\cup$, and the singleton $[x]$ by $f(x)$. That is, if A is the list $[a_1, \ldots, a_n]$, which is generated by $[a_1] + + \cdots + + [a_n]$, then the result is the set $f(a_1) \cup \cdots \cup f(a_n)$.

Not all monoid homomorphisms are allowed. For instance, sets cannot be converted into lists as this would introduce nondeterminism.

There exists an order relation between monoids that establish the properties of homomorphisms.

Recall Definition 4.4. A monoid $(T, \mathbf{1}, \oplus)$ is called commutative (idempotent, resp.) if the operation $\oplus$ is commutative (idempotent, resp.). For example, the monoid $set^{\alpha} = (set(\alpha), \{\}, f', \cup)$, where $f'(i) = \{i\}$ for each instance i of type α, is a commutative and idempotent monoid, and $bag^{\beta} = (bag(\beta), \{\!\{\}\!\}, f", \hat{\cup})$, where $f"(i) = \{\!\{i\}\!\}$ for each instance i of type β and $\hat{\cup}$ is the additive bag union, is a commutative monoid. Intuitively, less properties correspond to more structure. For example, the monoid *bag* has more structure that the monoid *set* because repetitions in a *bag* do matter (since it is not idempotent), whereas in a *set* do not (since it is idempotent. Similarly, the monoid *list*, being not commutative, has more structure then the monoid *bag*, which is commutative.

Fegaras and Mayer, in[FM95], have defined a mapping g from monoids to the set $\{C, I\}$ as: $C \in g(\mathcal{M})$ iff $\mathcal{M}$ is commutative and $I \in g(\mathcal{M})$ iff $\mathcal{M}$ is idempotent. The partial order between monoid names $\preceq$ is defined as :

$$\mathcal{N} \preceq \mathcal{M} \equiv g(\mathcal{N}) \subseteq g(\mathcal{M}) \tag{4.6}$$

This corresponds to the intuitive notion that $\mathcal{N}$ has more structure than $\mathcal{M}$. From 4.6, it follows that $bag^{\beta} \preceq set^{\alpha}$.

If $\mathcal{N} \preceq \mathcal{M}$, then an instance of type $\mathcal{N}$ can be translated deterministically, by using the merge function of the monoid $\mathcal{M}$, into an instance of the type $\mathcal{M}$, but not vice versa.

The monoid homomorphism is the only form of bulk manipulation of collection types supported in the calculus of [FM95]. Nevertheless monoid comprehension is very expressive. They go also beyond that capturing operations over multiple collection types, such as the join of a list with a bag that returns a set, plus predicates and aggregates. For example, an existential predicate over a set is a monoid homomorphism from the set to the monoid $(bool, false, \vee)$, while an aggregation, such as summing all elements of a list, is a monoid homomorphism from the list monoid to the monoid $(int, 0, +)$.

Definition 4.25 (Monoid homomorphism [FM95])
A homomorphism $hom^{\mathcal{M}\to\mathcal{N}}(f)A$ from the collection monoid $\mathcal{M} = (T(\alpha), \mathbf{1}, unit^{\mathcal{M}}, \otimes)$ to any monoid $\mathcal{N} = (S, \mathbf{0}, \oplus)$, where $\mathcal{M} \preceq \mathcal{N}$, is defined by the following inductive equations:

$$hom^{\mathcal{M}\to\mathcal{N}}(f)(\mathbf{1}) \;=\; \mathbf{0}$$

$$hom^{\mathcal{M}\to\mathcal{N}}(f)(unit^{\mathcal{M}}(a)) \;=\; f(a)$$

$$hom^{\mathcal{M}\to\mathcal{N}}(f)(x \otimes y) \;=\; hom^{\mathcal{M}\to\mathcal{N}}(f)x \oplus hom^{\mathcal{M}\to\mathcal{N}}(f)y$$

4.3.2 Queries on monoids and semirings

Queries on monoids are expressed as monoid comprehensions [FM95]. Informally a monoid comprehension over the monoid $\mathcal{M}$ takes the form $\mathcal{M}\{e|r_1, \ldots, r_n\}$, where e is an expression called the *head* of the comprehension, and $r_1, \ldots, r_n$ is a list of $qualifiers$, each of which is either

- an *iterator* of the form $v \leftarrow e'$, where v is a variable, and e' is an expression that evaluates to an instance of a collection monoid of type $\mathcal{M}$ or
- a $selection - predicate$, which is an expression that evaluates to *true* or *false*.

The expressions in turn can include monoid comprehensions. An important condition for the monoid comprehension is that for each $1 \leq i \leq n$, each free variable (i.e. free variables in the expressions and predicates) appearing in $r_i, \ldots, r_n$ must appear as variables of an iterator among $r_1, \ldots, r_{i-1}$, and each free variable in e must appear as the variable of an iterator among $r_1, \ldots, r_n$. Formally, monoid comprehensions are defined in terms of monoids homomorphisms. In particular all qualifiers in the comprehension are eliminated from the left to right until no qualifier is left:

- $\mathcal{M}\{e \,|\, \} \;=\; f^{\mathcal{M}}(e)$
- $\mathcal{M}\{e \,|\, x \leftarrow u, \bar{q}\} \;=\; hom^{\mathcal{N}\to\mathcal{M}}(\lambda x.\, \mathcal{M}\{e \,|\, \bar{q}\})u$.[5]

[5] $\lambda x.e$ is the function f such that $f(x) = e$.

- $\mathcal{M}\{e \,|\, pred, \bar{q}\} = \textbf{if}\, pred\, \textbf{then}\, \mathcal{M}\{e \,|\, \bar{q}\}\, \textbf{else}\, \mathbf{0}^{\mathcal{M}}$

As semirings can be considered nested semirings, we can use monoid comprehensions to query semiring structures. In our case, monoids comprehensions have the fixed form

$$\mathcal{S}\{e|m_1, \ldots, m_n\}$$

where e is the head of the comprehension, and $m_1, \ldots, m_n$ is a list of iterators of the form $v \leftarrow \mathcal{M}\{e|r_1, \ldots, r_k\}$, and $\mathcal{M}\{e|r_1, \ldots, r_k\}$ is a monoid comprehension.

We assume that each instance of a semiring is represented as an expression involving $\vee$, $\wedge$, **false**, $f^{\mathcal{M}}$ and **true**. For example the formula $(p_1 \wedge p_2) \vee p_3$ can be represented as

$$\vee\ (\ \vee\ (\ \mathbf{false}, \wedge\ (\ \wedge\ (\ \mathbf{true}, f^{\mathcal{S}_{dc}}(p_1)), f^{\mathcal{S}_{dc}}(p_2))), f^{\mathcal{S}_{dc}}(p_3)).$$

We will assume that every semiring instance is conceptually represented in this way, and thus, the notation $\mathcal{S}\{\mathcal{M}_1, \ldots, \mathcal{M}_n\}$ will denote the expression

$$\vee(\ldots \vee(\vee(\mathbf{false}, \mathcal{M}_1), \mathcal{M}_2), \ldots, \mathcal{M}_m)$$

where $\mathcal{M}_i, 1 \leq i \leq n$ are instances of type monoid and $\mathcal{M}_i\{a_1, \ldots, a_m\}$ will denote the expression

$$\wedge(\ldots \wedge(\wedge(\mathbf{true}, f^{\mathcal{M}_i}(a_1)), f^{\mathcal{M}_i}(a_2)), \ldots, f^{\mathcal{M}_i}(a_m))$$

Furthermore, $\mathcal{S}\{\}$ and $n = 0$ will both denote *false*, and if $n = 1$ then $\mathcal{S}\{\mathcal{M}\} = \mathcal{M}$.

Let M be a monoid comprehension over a monoid $\mathcal{M} = (T, zero, unit, merge)$, we can define instances of type T by first initializing result with $zero$, and then invoking the procedure insert_MC(result, M) (developed by Brodski et al. in [BSCE97]) defined recursively by the following rules:

(r1) insert_MC(result, $\mathcal{M}\{e|\}$)
$\rightarrow$ result := $merge$(result,$unit$(e))

(r2) insert_MC(result, $\mathcal{M}\{e|false, \vec{r}\}$)
$\rightarrow$ nil

(r3) insert_MC(result, $\mathcal{M}\{e|true, \vec{r}\}$)
$\rightarrow$ insert_MC(result,$\mathcal{M}\{e|\vec{r}\}$)

(r4) insert_MC(result, $\mathcal{M}\{e|x \leftarrow \mathcal{N}\{a_1, \ldots, a_n\}, \vec{r}\}$)
$\rightarrow$**for** $i = 1$ **to** n **do** insert_MC(result,$\mathcal{M}\{e|\vec{r}\}[x/a_i]$)

4.3.3 Syntax and semantics of queries

The syntax of comprehensions is of the form:

```
select   expr
into     [{monoid_type}] [result]
from     iterators
where    conditions
```

expr in the select clause is an arbitrary expression that evaluates to the type of result's element. expr may involve variables instantiated in the from clause and may also contain nested monoid comprehensions. The first parameter in the into clause specifies the type of result. Any number of iterators, separated by commas can appear in the from clause; further, any number of predicates (conditions) may appear in the where clause. The semantics of monoid comprehensions is defined by the corresponding formal monoid comprehension. The basic evaluation is made by the nested loop algorithm, described in Section 4.3.2. We show a query with the next example.

Example 4.4 We want to represent parcels and buildings. We define two classes representing them:

```
class            parcel {          class   building {
string           owner             int     no_of_rooms
int              area              int     no_of_floors
list(building)   constructed}       date    year_of_construction}
```

Let all_parcels be of type set(parcel). The query *find a set of parcels, whose area is bigger that 100mq with a building having no more than two floors* can be written using the following query:

```
select   pc
into     {set(parcel)} result
from     all_parcel as {parcel*} pc          % iterator: pc iterates over set
where    pc.area > 100
from     pc.constructed as {building*} bu    % iterator: bu iterates over list
where    bu.no_of_floors < 2
```

The above query corresponds to the following monoid comprehension:

$$\begin{array}{ll} \mathsf{set}\ \{\ pc\ \mid & pc \longleftarrow parcel, \\ & pc.area > 100, \\ & bu \longleftarrow pc.constructed, \\ & bu.no\_of\_floors < 2\ \} \end{array}$$

The semantics of the query is given by the following nested loop program:

```
result = empty_set;
FOREACH pc IN parcel DO
    IF pc.area > 100 THEN
        FOREACH bu IN {list} pc.constructed DO
            IF bu.no_of_floors < 2 THEN
                INSERT pc INTO result
```

□

4.4 Constraint classes

In this section, we define and create constraints as classes in an object-oriented model, using the Java platform. This automatically enables us to deploy them on a database server and then use them as data types extending data types of SQL.
We follow the following basic principles:

- constraints are organized in classes as the other objects,
- constraints have methods, e.g. conjunction, disjunction etc.
- constraints can occur in attributes of any object, e.g. geometry of a region in a GIS.

We start defining the classes Monoid and Semiring that constitute the bricks, on which all the constraint classes, and not only, will be built. Then, it follows the definition of classes implementing the types described in Section 4.2.

4.4.1 The classes Monoid and Semiring

To represent the minimum requirements for primitive and collection monoids, consider the recursive rules defining the result of a monoid comprehension. For the monoid $\mathcal{M}$ to appear in the result of the monoid comprehension, we only need to (1) use **false**/**true** and (2) know how to perform

$$\mathsf{result} := merge(\mathsf{result}, unit(\mathsf{e}))$$

which is, in fact the insert_MC(result, $\mathcal{M}\{e|\}$) operation (the operator $merge$ of a monoid). In order for the collection monoid $\mathcal{N}$ to appear inside the comprehension, we only need to be able to *iterate* over $\mathcal{N}\{a_1, \ldots, a_n\}$, i.e. to perform the for loop.

The representation of monoids and semirings in our framework is based on two Java (abstract) classes, and a Java interface, parameterized with the type A of collection elements.

```
interface CollectionMonoidI {
    void Merge(Monoid m);
```

```
        Iterator CreateIterator();
    }

    abstract class CollectionMonoid implements CollectionMonoidI{
        Collection A;              % class implementing the domain of the monoid
        CollectionMonoid();              % constructor used as 0
        void Merge(CollectionElement A);
        Iterator CreateIterator()= return A.Iterator();
    }
```

The class CollectionMonoid reflects the minimum requirements: it has **0**, implemented as class constructor, and Merge and CreateIterator as methods. An Iterator object, created by CreateIterator, has First, More and Next methods which can be directly used in the for loop. The class is an abstract class, because the method Merge cannot be defined a priori, as it is different for each monoid.

Semirings are monoids as well, therefore objects of type *semiring* have to be "casted" to monoid types. Java allows it, by using $interfaces$.

"A Java interface defines a set of methods but does not implement them. A class that implements the interface agrees to implement all of the methods defined in the interface, thereby agreeing to certain behavior"([Jav]). Defining a new interface means in essence defining a new reference data type. One can use interface names anywhere any other data type name can be used.

We define a semiring class as the following:

```
    abstract class Semiring implements MonoidI {
        CollectionMonoid M1;
        Semiring();
        void Merge(CollectionMonoid M2);
        Iterator CreateIterator() return M1.CreateIterator();
    }
```

The Subclasses C, EC, DC and DEC

The classes, representing the four constraint families, are subclasses of the classes CollectionMonoid and Semiring. In these classes the collection elements are instantiated as set of predicates, for C, a set of objects of type C, for EC and DC, and, a set of objects of type EC and DC, for DEC. The casting operation between classes is made possible by the relations between the corresponding interfaces, see Figure 4.5.

Moreover, each class has methods to implement the operations described in Section 4.2.2, and in particular way they implement the lattice structure expressed by the formula 4.3

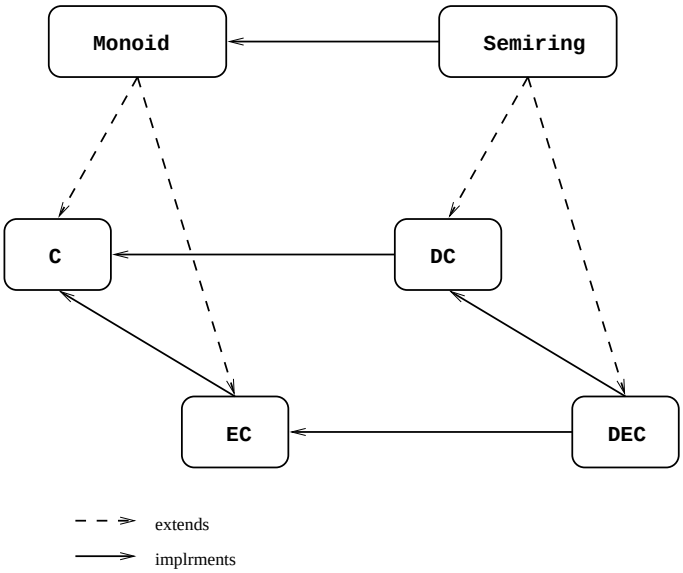


Figure 4.5: The schema of the classes

Constraint classes are abstract as well, since some of the operations, such as projection, depend on the type of the atomic formulas. We will study this case in the next chapters, where we will map atomic formulas on temporal and spatial predicates and on linear constraints.

```
abstract    class C extends CollectionMonoid {
            C() { return CollectionMonoid()}
            C(set(Predicate) atoms) { A = atoms };
            void Merge(Collection A){
            %implement *conjunction*}
%methods:   rename, satisfy, op
};
abstract    class DC extends CollectionSemiring implements C{
            DC() { return CollectionSemiring()}:
            DC(C conj) { A = conj };
            void Merge(CollectionMonoid A){
            %implement *disjunction*}
%methods:   rename, satisfy, op
};
```

The class Predicate represents the atomic formulas and it is an abstract class, which will be specialized by the predicates belonging to $\mathcal{P}$ of the vocabulary Ω, on which the constraints are defined. The class C initializes the global variable A with the set of atomic formulas, composing the conjunctive constraints. In the next chapter, we will define the class Predicate as a set of classes, each one implementing

a predicate of $\mathcal{P}$.

The constructors C(), DC() are those of the parent classes, while the constructor DC(C conj) initializes the variable A with the conjunctive constraints, (which are of type CollectionMonoid), that constitutes the domain of the semiring structure, represented by the disjunction of conjunction constraints.

Methods in the subclasses implement the operations *conjunction* and *disjunction* and they are different depending on which constraint family they operate. On the conjunctive families, the $\wedge$ operator simply combines its arguments and it is constant time.

On the disjunctive families, the $\vee$ operator is constant time, while $D1 \wedge D2$, where $D1 = \vee_{i=1,\ldots,n} C1_i$ and$D2 = \vee_{j=1,\ldots,m} C2_j$ is more involved:

$$D1 \wedge D2 = \bigvee_{i=1,\ldots,n} C1_i \wedge \bigvee_{j=1,\ldots,m} C2_j = \bigvee_{i=1,\ldots,n,j=1,\ldots,m} (C1_i \wedge C2_j) \qquad (4.7)$$

that is, the result consists of all combinations of $C1_i$ and $C2_j$ that are mutually consistent (i.e. their conjunction is satisfiable). Since the constraint families are monoids, $D1 \wedge D2$ can be implemented as the following query:

select	$c_1 \wedge c_2$
into	$\{DEC\}$ conj_D1_and_D2
from	$D1$ as $\{EC\}\, c_1$
	$D2$ as $\{EC\}\, c_2$
where	satisfy(c_1, c_2)

The classes EC and DEC are similarly implemented to the classes C and DC, respectively. They have a method more, which implements projection. The method is abstract, because it will be specialized in the sub-classes.

Remark 4.26 Java gives the possibilities of defining interfaces, with which one can simulate the inheritance (and multiple inheritance) relationships between classes. Defining the four constraint classes as interfaces allows us to use the "casting" between the constraint classes. In Section 4.2.1, we said that C is a sub-type of DC and EC, i.e. a constraint of type C is also a constraint of type EC and DC and operations defined on the types EC and DC have to apply to constraints of type C as well. To model this relationship, we can declare C as interface, with the method conjunction, then we force the classes/interfaces EC and DC to "implement" the interface C, meaning that 1) the operation of conjunction must be defined on objects of the classes EC and DC and, 2) objects (constraints) of the classes DC and EC can be casted to objects (constraints) of the class C.

4.5 Considerations

In this section, we make some considerations about the usefulness of our choices, described in the previous sections.

In Section 4.1 we described the constraint families, chosen to represent data. The challenge here is the development of constraint families and algebras, that strike a careful balance between

(1) *expressiveness*,

(2) *computational* complexity,

(3) *representation usefulness*

An example of a very expressive, but having high (exponential) time data complexity is the DISCO (Datalog with Integer and Set order Constraint) query language [BR95].

On the other hand, Kanellakis, et al. in [KKR90] represent data by using a fairly restricted sub-family of first-order logic, i.e. disjunction of unquantified conjunctions of atomic constraints and the algebra allows quantifier elimination. Still, for some atomic constraint families, such as linear inequalities over reals, this framework may be computationally unmanageable: the quantifier elimination may result in a constraint exponential in the size of the original conjunction, although for many sub-families more efficient algorithms were developed.

Thirdly, the framework of [BSCE97] implements with low complexity linear constraints, and it is for our opinion a good compromise between the three requirements described above. Anyway we think that the approaches, that choose a priori a particular class of atomic constraints limit in general the expressiveness and force the employment of ad-hoc algorithms, to perform operations between constraints.

In Section 4.2 we defined types and operations for the four families. The four types are carefully constructed with the complexity consideration in mind as follows. First, all operations allowed on the families have polynomial data complexity. This is the reason, for example, that $c(atom)$ *is not closed under general projection*: transforming the result into $c(atom)$ will require quantifier elimination and thus the size of the result (and, of course, time complexity) may be exponential in the number of variables eliminated. Whereas, $ec(atom)$ is closed under the general projection since the general projection is *lazy*: $ec(atom)$ allows quantifiers in the internal representation and hence no physical quantifier elimination is performed.

Chapter 5

Modeling Spatial Information

In this chapter, we present an extension of the model described in Chapter 4, defining spatial types and operators, to represent spatial information. The chapter is divided in two main sections, proposing two different mappings: Section 5.1 presents a mapping of flat constraints to predicates, interpreted as sets of spatial objects, of the Spatial Cartridge of Oracle8; in Section 5.2 flat constraints are mapped to linear constraints.

5.1 Mapping flat constraints to OracleSpatial predicates

In this section we design a spatial constraint database, whose atomic constraints consist of the OracleSpatial predicates, see Section 3.2.1. The challenge here is the definition of the duality between spatial objects and constraints. Let' s show this concept with the following example.

Example 5.1 We suppose to have a database that stores polygons. As shown in Figure 5.1, the database contains seven polygons P_i, each of them belongs to a class-type *polygon* and it is modeled by the constraint C_i

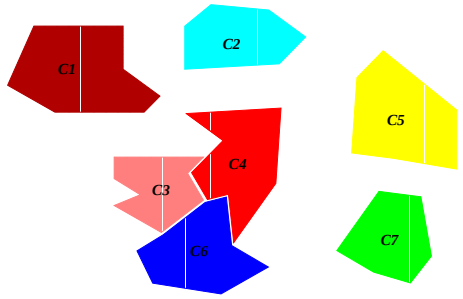


Figure 5.1: A database of polygons

Suppose that we want to query for adjacent objects, and to store the result like a new polygon, as shown in Figure 5.2, in which the objects C_3, C_4 and C_6 are adjacent and their relation can be expressed by the constraint $C_3\, touch\, C_4\, touch\, C_6$, that can be restored as the object C_8.

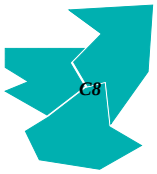


Figure 5.2: The new object $C_8 = C_3\, touch\, C_4\, touch\, C_6$

□

Example 5.1 shows the particular nature of the constraint $C_3\, touch\, C_4\, touch\, C_6$, that is to express, in this case, a particular object in the space by means of relations between other objects in the database. This duality is very important for defining consistent and closed queries, (recall Section 2.3).

At this purpose, we map flat constraints to OracleSpatial predicates. In Chapter 4 we have defined a data model based on constraints, whose vocabulary Ω and interpretation structure $\mathcal{M} = \langle \mathcal{D}, \Omega \rangle$ have been let unspecified. The process of mapping defines first the vocabulary and the interpretation domain, then it defines the spatial types and operations.

5.1.1 Spatial constraints

As described in Section 4.1, a flat vocabulary Ω is a pair $\langle \mathcal{P}, \mathcal{C} \rangle$, where $\mathcal{P}$ is the set of predicates and $\mathcal{C}$ the set of constants, which will be used to define atomic constraints.

Definition 5.1 (Spatial vocabulary) The spatial vocabulary Ω_S consists of the set $\mathcal{P}_S = \mathcal{SP}_{bin} \cup \mathcal{SP}_{una}$, where $\mathcal{SP}_{bin}$ contains the following two-place predicate symbols: *disjoint, equal, touch, contains, inside, covers, covered_by, overlap* and $\mathcal{SP}_{una} = \{point, line, polygon\}$.
$\mathcal{C}_S = A_{string}$.

The two-place predicate symbols correspond to the spatial predicates of OracleSpatial (see Section 3.2.1), that implements the spatial relationships between 2-dimensional spatial objects, defined by Egenhofer and described in this thesis in Section 3.2.2. A_{string} is the set of strings defined in Definition 4.10. Therefore, spatial constraints are made up from atomic formulas, defined on Ω_S. We denote the set of spatial formulas with $\mathcal{F}_S$.

Recall Definition 2.3. To define a model $\mathcal{M}_S$ for the spatial flat vocabulary Ω_S, first we have to define the spatial domain $\mathcal{D}_S$ and then an interpretation of spatial predicates on elements of the domain.

We define $\mathcal{D}_S$ as the set of *geometries* (i.e. spatial objects, supported by OracleSpatial). This implies that $\mathcal{D}_S = \perp \cup A_{point} \cup A_{line} \cup A_{polygon} \cup \mathcal{D}_{multiobjects}$, where

$$A_{point} = \{(x, y) | \, x, y \in \mathbf{R}\},$$

$$A_{line} = \{(p_1, \ldots, p_n) | \, \forall i, 1 \leq i \leq n, p_i \in A_{point}\},$$

$$A_{polygon} = \{(p_1, \ldots, p_n, p_1) | \, \forall i, 1 \leq i \leq n, p_i \in A_{point}\}$$

The above three sets are the carrier sets of the OracleSpatial primitive types and they show the way how simple spatial objects are represented by OracleSpatial, i.e. a sequence of points. Among the objects of $\mathcal{D}_S$ are included lines and polygons, whose vertices are connected by straight line segments and circular arcs; Figure 3.6 and Figure 3.7 illustrate all kind of objects of $\mathcal{D}_S$. Arcs are represented in OracleSpatial by three vertices. We won' t go into detail in defining them, and we assumed that the above sets contain lines and polygons made up from arcs, as well.

Furthermore, OracleSpatial supports five types of multi-objects, namely *polygons with holes*, *point cluster*, (Figure 5.3(a)), *multilinestring*, (Figure 5.3(b)), *multipolygon*, (Figure 5.3(c)), and *collections*, (Figure 5.3(d)).

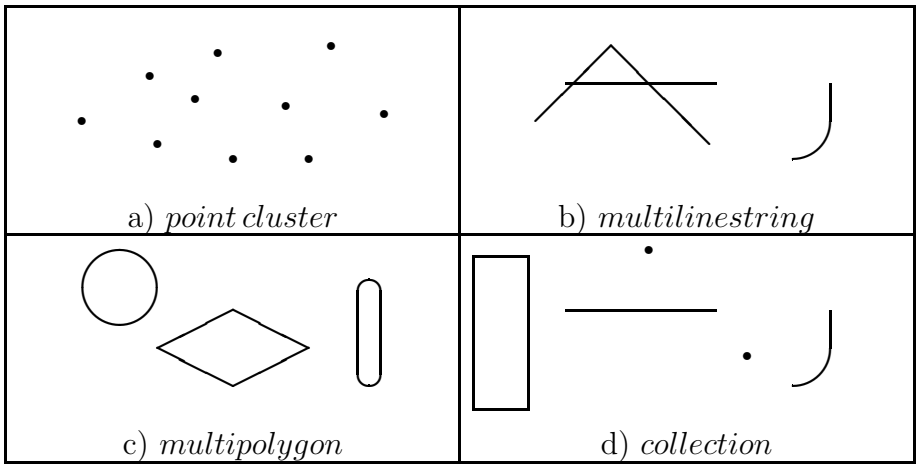


Figure 5.3: Representation of *multiobjects*

Oracle requires that polygons in *collection* and *multipolygon* must be disjoint. Thus, $\mathcal{D}_{multiobjects} = A_{polygon\,with\,holes} \cup A_{point\,cluster} \cup A_{multilinestring} \cup A_{multipolygon} \cup A_{collections}$.

Hence, the interpretation of Ω_S-constraints is given on the spatial domain $\mathcal{D}_S$, assigning to each constraint a binary relation between elements of $\mathcal{D}_S$, that corresponds to one of the spatial geometries described above. At this purpose, we define

an interpretation function. Let $\mathcal{F}_S(n)$ be the set $\{\varphi \in \mathcal{F}_S \,|\, freevar(\varphi) = n\}$, $\mathcal{D}^*$ denote $\{X \subseteq \mathcal{D}\}$, then we define a class of functions:

$$\mathcal{I}_n \,:\, \mathcal{F}_S(n) \longrightarrow (\underbrace{\mathcal{D}_S \times \cdots \times \mathcal{D}_S}_{n\,times})$$

The following definition builds the interpretation functions, defined on atomic formulas.

Definition 5.2 (Interpretation of atomic constraints). Let $\mathcal{P}_S(i)$ be the set of i-place predicates of $\mathcal{P}_S$, $p \in \mathcal{P}_S$ and $o, o_1, \ldots, o_n$ elements of $\mathcal{D}_S$, then the interpretation function $\mathcal{I}$ on atomic formulas is:

$$\mathcal{I}_{n\in\{1,2\}} \,:\, \mathcal{P}_S(n) \longrightarrow \mathcal{D}_S^n$$

where $\mathcal{I}$ is defined for each predicate of $\mathcal{P}_S$ as follows:

$$\mathcal{I}_{n\in\{1,2\}}(p(x_1, x_2)) = \{(o_1, o_2) \,|\, rel_p(o_1, o_2)\}$$

where $rel_p(o_1, o_2)$ is the OracleSpatial relation obtained by the following select:

select	o_1, o_2
from	$obj_1\, o_1,\ obj_2\, o_2$
where	$p(o_1, o_2)$='true'

□

Theorem 5.3 Each tuple of rel_p is representable as an OracleSpatial object.

Proof. To prove the theorem we define a family of functions

$$compose_{n\in\mathbf{N}} \,:\, \mathcal{D}_S^n \longrightarrow \mathcal{D}_S$$

such that

$$compose_2(disjoint(o_1, o_2)) = \begin{cases} o \in A_{point\,cluster} & : \ (o_1, o_2) \in A^2_{point} \\ o \in A_{multilinestring} & : \ (o_1, o_2) \in A^2_{line} \\ o \in A_{multipolygon} & : \ (o_1, o_2) \in A^2_{polygon} \\ o \in A_{collection} & : \ otherwise \end{cases}$$

$$compose_2(touch(o_1, o_2)) = \begin{cases} o \in \{o_1, o_2\} : & (o_1, o_2) \in A^2_{point} \\ o_i,\ i \in \{1, 2\} : & j \neq i,\ o_j \in A_{point} \\ o \in A_{line} : & (o_1, o_2) \in A^2_{line} \\ o \in A_{polygon} : & (o_1, o_2) \in A^2_{polygon} \\ o \in A_{collection} : & (o_1, o_2) \in (A_{line} \times A_{polygon}) \cup \\ & (A_{polygon} \times A_{line}) \\ \bot : & otherwise \end{cases}$$

$compose_2(equal(o_1, o_2)) = o \in \{o_1, o_2\}$

$compose_2(overlap(o_1, o_2)) = intersection(o_1, o_2)$

where $intersection(o_1, o_2)$ returns the spatial object resulting from the intersection between o_1 and o_2.
Figure 5.4 and Figure 5.5 show the new polygon, generated by the constraints $touch(p_1, p_2)$ and $overlap(p_1, p_2)$ respectively.

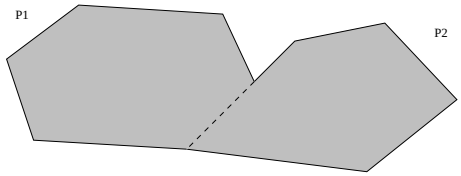


Figure 5.4: Interpretation of the atomic constraint $touch(p_1, p_2)$

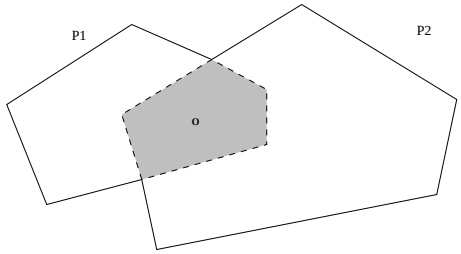


Figure 5.5: Interpretation of the atomic constraint $overlap(p_1, p_2)$

$$\left.\begin{array}{l} compose_2(covers(o_1, o_2)) \\ compose_2(contains(o_1, o_2)) \end{array}\right\} = o_2$$

$$\left.\begin{array}{l} compose_2(covered\_by(o_1, o_2)) \\ compose_2(inside(o_1, o_2)) \end{array}\right\} = o_1$$

□

The spatial object o, resulting from the constraints *disjoint*, *equal* and *touch*, is obtained by the following procedure:

1. In the first step, the OracleSpatial predicate sdo_relate(obj_1, obj_2, rel), where rel $\in Mask =$ {DISJOINT, TOUCH, EQUAL}, is applied to x_1, x_2 to find the pair (o_1, o_2) of objects in the database, which satisfies the relation specified in rel.

2. In the second step the object o, is created from the objects resulting from the first step and inserted in the appropriated table. This is made using the Oracle construct insert.

Example 5.2 We show the two steps, taking as example the $touch(x_1, x_2)$ constraint. We create a view where we insert the object, that results from composition of the pairs of spatial objects, which touch each other. We assume that the spatial objects are stored in tables $obj_i(id_i, geo)$.

create view	$touching\_obj$ as
select	$o_1.id_1$, $o_2.id_2$, sdo_geom.sdo_poly_union($o_1.geo, o_2.geo$)
from	$obj_1\, o_1$, $obj_2\, o_2$
where	mdsys.sdo_relate(o_1, o_2, 'mask=touch, querytype=join')='true'

□

The objects $intersection(o_1, o_2)$, resulting from the constraint $overlap(x_1, x_2)$, is the intersection of the objects, which satisfy the OracleSpatial predicate sdo_relate(obj_1, obj_2, overlap) and they are obtained by the following command:

create view	$overlap\_obj$ as
select	$o_1.id_1$, $o_2.id_2$, sdo_geom.sdo_poly_intersection($o_1.geo, o_2.geo$)
from	$obj_1\, o_1$, $obj_2\, o_2$
where	mdsys.sdo_relate(o_1, o_2, 'mask=overlap, querytype=join')='true'

Finally, the objects resulting from the interpretation of the constraints *covers*, *covered_by*, *contains* and *inside* are obtained by the following commands, where $rel1 \in \{$COVERS, CONTAINS$\}$ and $rel2 \in \{$COVERED_BY, INSIDE$\}$:

create view	$rel1\_obj$ as
select	$o_1.id_1$, $o_2.id_2$, $o_2.geo$,
from	$obj_1\, o_1$, $obj_2\, o_2$
where	mdsys.sdo_relate(o_1, o_2, 'mask=rel1, querytype=join')='true'
create view	$rel2\_obj$ as
select	$o_1.id_1$, $o_2.id_2$, $o_1.geo$,
from	$obj_1\, o_1$, $obj_2\, o_2$
where	mdsys.sdo_relate(o_1, o_2, 'mask=rel2, querytype=join')='true'

The process of interpretation of atomic constraints assigns to each atomic formula an OracleSpatial relation, i.e. a set of elements or pairs of elements of the spatial domain, which are OracleSpatial objects themselves. For constraint formulas the process is more complicated. In this case, we have to assign to the logical connectives $\wedge$, $\vee$ and $\exists$, operations on OracleSpatial relations. As we said in Section 4.2.2, these operations operate on canonical forms of constraints, performed by the operator *op*. In the case of constraints on Ω_S, *op* is reduced to the renaming operator δ_x^y, defined by Eq. 4.1. The renaming of variables may change the type of the constraint, to

which it is applied; from Eq. 4.2 it follows, that the operation $\delta_x^y\varphi$ introduces an equality constraint and a projection, if x is not independent in φ. This implies that, for instance, if φ is of type C, the renaming would transform it in a constraint of type EC.

Example 5.3 We consider the following C constraint:

$$disjoint(x_1, x_2) \wedge touch(x_2, x_3)$$

In the above formula, there is a variable dependency, due to the condivision of x_2, thus the operator *op* transforms it in the following EC constraint:

$$\exists_y(disjoint(x_1, x_2) \wedge equal(x_2, y) \wedge touch(y, x_3)) \tag{5.1}$$

□

This transformation allows us to apply the equality in Eq. 4.5. Then, the interpretation function $\mathcal{I}$ can be defined inductively, first on simple conjunction of constraints, then on existentially quantified conjunctives and, finally on disjunctions of existentially quantified conjunctives.

Definition 5.4 (Interpretation of simple conjunctives). Let $\bigwedge_{\substack{1\le i\le n\\ j\in\{1,2\}}} p_i^j$ be a simple conjunctive, where $p_i^{j_i}$, $1 \le i \le n$ are atomic formulas with rank j_i, then

$$\mathcal{I}_n(\bigwedge_{\substack{1\le i\le n\\ j_i\in\{1,2\}}} p_i^{j_i}) = \mathcal{I}_n(\mathcal{I}_{j_i}(p_1^{j_1}), \ldots, \mathcal{I}_j(p_n^{j_n}))$$

where $\mathcal{I}_{j_i}(p_i^{j_i}), 1 \le i \le n$ is given by Definition 5.2. □

We apply the function $compose_n$ to generate the spatial objects corresponding to each tuple of the relation corresponding to the simple conjunctive constraint $\bigwedge_{\substack{1\le i\le n\\ j\in\{1,2\}}} p_i^j$, with free variables $x_1, \ldots, x_k$, $k \le 2 * n$, as follows:

$$compose_n(\mathcal{I}_n(\bigwedge_{\substack{1\le i\le n\\ j\in\{1,2\}}} p_i^j)[o_1, \ldots, o_k]) =$$
$$compose_n(compose_j(\mathcal{I}_j(p_1^j))[o_1, o_2], \ldots, compose_j(\mathcal{I}_j(p_n^j))[o_{k-1}, o_k])$$

The function *compose*, as the operator *op* in Section 4.2.2, is commutative as well. Its application to a conjunction of atomic constraints can be translated as the interpretation of the tuple whose each element is the result of the application of *compose* to each atomic formula. Therefore, we need to define *compose*, when applied to n spatial objects.

$$compose_n(o_1, \ldots, o_n) =$$

$$\begin{cases} o \in A_{point\_cluster}: & \forall i, 1 \leq i \leq n,\ o_i \in A_{point} \cup A_{point\_cluster} \\ o \in A_{multilinestring}: & \forall i, 1 \leq i \leq n,\ o_i \in A_{line} \cup A_{multilinestring} \\ o \in A_{multipolygon}: & \forall i, 1 \leq i \leq n,\ o_i \in A_{polygon} \cup A_{multipolygon} \\ o \in A_{collection}: & otherwise \end{cases} \tag{5.2}$$

Definition 5.5 (Interpretation of existentially quantified conjunctives). Let $n \in \mathbf{Z}, n \geq 1$, an existentially quantified conjunctive is the constraint $\varphi = \exists_X(c_1 \wedge \cdots \wedge c_n)$, where $X \subseteq var(c_1 \wedge \cdots \wedge c_n)$. Let $var(\varphi) = \{x_1, \ldots, x_m, m \leq 2*n\}$, we suppose, without loss of generality, that $Y = var(\varphi)/X = \{x_1, \ldots, x_j \,|\, j < m\}$, then $\exists_X(\varphi)$ is the projection of φ on Y, $(Y)|\varphi$, that is the following OracleSpatial relation:

$$\mathcal{I}_j((Y)|\varphi) = \{(o_1, \ldots, o_j) \in D_S^j \,|\, rel_\varphi(o_1, \ldots, o_j)\} \equiv$$

select $x_1.geo, \ldots, x_j.geo$
from $obj_1\, x_1, \ldots, obj_m\, x_m$
where mdsys.sdo_relate($x_1.geo, x_2.geo$, 'mask=p_1 querytype=join')='true' and
. . . . . .
mdsys.sdo_relate($x_{m-1}.geo, x_m.geo$, 'mask=p_n querytype=join')='true'

□

It follows that

$$compose_j(\exists_X(c_1 \wedge \cdots \wedge c_n))[o_1, \ldots, o_j] = compose_j(o_1, \ldots, o_j)$$

defined by Eq. 5.2.

Example 5.4 Recall Example 5.3. The process of interpretation of the constraint 5.1 generates first the table:

x_1	x_2	x_3
p_1	p_2	p_3

. . . . . .

as result of the SQL query

select $x_1.geo, x_2.geo, x_3.geo$
from $obj_1\, x_1$, $obj_2\, x_2$, $obj_2 y$, $obj_3\, x_3$
where mdsys.sdo_relate($x_1.geo, x_2.geo$, 'mask=$disjoint$ querytype=join')='true' and
mdsys.sdo_relate($x_2.geo, y.geo$, 'mask=$equal$ querytype=join')='true' and
mdsys.sdo_relate($y.geo, x_3.geo$, 'mask=$touch$ querytype=join')='true'

Depending on the spatial relationship existing between the two objects p_1 and p_3, the resulting spatial object is one of the four objects shown in Figure 5.6.

□

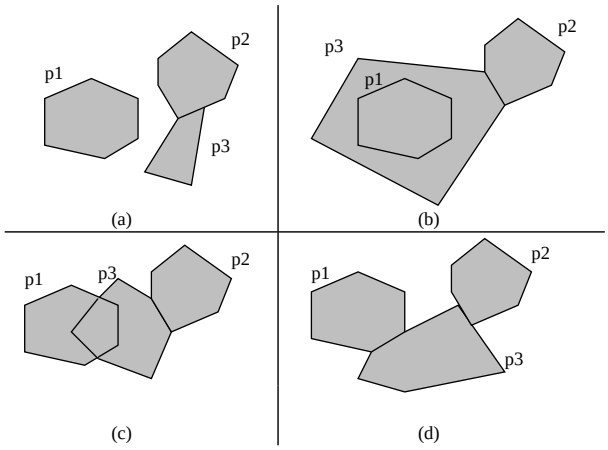


Figure 5.6: Interpretation of the constraint $disjoint(x_1, x_2) \wedge touch(x_2, x_3)$

Definition 5.6 (Interpretation of disjunction of existentially quantified conjunctives). Let $n \in \mathbf{Z}, n \geq 1$, a disjunction of existentially quantified conjunctives is the constraint $\varphi = (c_1 \vee \cdots \vee c_n)$, where $\forall i, 1 \leq i \leq n, c_i$ is either a conjunctive or an existential conjunctive constraint with x_{j_i} free variables. We define the interpretation function as follows:

$$\mathcal{I}_j(c_1 \vee \cdots \vee c_n) = \bigcup_{i=1}^{n} \mathcal{I}_{j_i}(c_i)$$

□

Therefore, the interpretation structure (or model) is $\mathcal{M}_s = \langle \mathcal{D}_S, \mathcal{I}_{n \in \mathbf{N}} \rangle$.

In order to have the spatial object/constraint duality, we have to prove that the OracleSpatial unrestricted relations are definable on $\mathcal{M}_S$ with Ω_S-constraints[1].

Definition 5.7 (Oracle unrestricted relations). An Oracle unrestricted tuple consists of a spatial object, belonging to $\mathcal{D}_S$. An unrestricted relation is a set, that is a union, of spatial object, i.e. a subset of $\mathcal{D}_S$. Any collection of Oracle unrestricted relations defines an Oracle unrestricted database.

Theorem 5.8 The Oracle unrestricted database is definable on $\mathcal{M}_S$ with Ω_S-constraints.

Proof: To prove the theorem, we just have to show how to represent each spatial object in $\mathcal{D}_S$ using Ω_S-constraints. At this purpose, we define the function *decompose*, which transforms an OracleSpatial object in a Ω_S-constraint.
The function is defined for each object of $\mathcal{D}_S$ and returns a constraint, that belongs to one of the four families. First, we define the function on unrestricted tuples, i.e.

[1]Unrestricted relations and definability have been described in Section 2.2.1.

elements of $\mathcal{D}_S$, and then on unrestricted relations, unions of tuples, i.e. subsets of $\mathcal{D}_S$.
The definition of *decompose* proves the theorem, as it translates unrestricted tuples and relations in Ω_S-constraints, and therefore proves that the Oracle unrestricted database is definable on $\mathcal{M}_S$ with Ω_S-constraints:

$$decompose : \mathcal{D}_S \longrightarrow \mathcal{F}_S$$

$$decompose(x) =$$

$$\begin{cases} point(x) & : \ x \in A_{point} \\ line(x) & : \ x \in A_{linestring} \\ polygon(x) & : \ x \in A_{polygon} \\ (\bigwedge_{i=1}^{n} polygon(p_i)) \wedge (\bigwedge_{1=1}^{n-1} contains(p_i, p_{i+1})) & : \ x = (p_1, \ldots, p_n) \in A_{polygon} \\ \bigwedge_{i=1}^{n} point(p_i) & : \ x = (p_1, \ldots, p_n) \in A_{point\_cluster} \\ \bigwedge_{i=1}^{n} line(l_i) & : \ x = (l_1, \ldots, l_n) \in A_{multilinestring} \\ \bigwedge_{i=1}^{n} polygon(p_i) \wedge \bigwedge_{\substack{i,j=1 \\ i \neq j}}^{n} disjoint(p_i, p_j) & : \ x = (p_1, \ldots, p_n) \in A_{multipolygon} \\ \bigwedge_{i=1}^{n} point(p_i) \bigvee \bigwedge_{j=1}^{m} line(l_j) \bigvee & \\ (\bigwedge_{h=1}^{k} polygon(s_h) \bigwedge_{\substack{i,j=1 \\ i \neq j}}^{k} disjoint(s_i, s_j)) & : \ x = (p_1, .., p_n, l_1, .., l_m, s_1, .., s_k) \in A_{collection} \end{cases}$$

An OracleSpatial unrestricted relation $\{t_1, \ldots, t_n | t \in \mathcal{D}_S\}$ is a set of unrestricted tuples, thus it follows that

$$\{t_1, \ldots, t_n | t \in \mathcal{D}_S\} \text{ is definable by the constraint } \bigvee_{i=1}^{n} decompose(t_i)$$

□

Theorem 5.8 states that an arbitrary OracleSpatial relation is representable as an Ω_S-constraint, which proves that the mapping of Ω_S-constraints to OracleSpatial objects is complete.

5.1.2 Extension of the data model with OracleSp types

In this subsection, we define the spatial constraint types and extend the data model defined in Section 4.2 with them. In that section, we drew Table 4.1, which gives a scheme of the "abstract" constraint types, thus constraints types have been defined

on a generic carrier set. In the following we specialize those types, instantiating them with OracleSpatial constraints. Such instantiation is made defining the carrier set of the type constructors $PREDICATE\_ORA\_SP$. Table 5.1 shows the spatial constraint types. The types $name\_ORA\_SP$ are subtypes of the corresponding constraint types *name* defined in Section 4.2.

Type constructor	Signature	
$disjoint, touch, equal,$ $overlap, inside, contains,$ $covers, covered\_by,$ $point, line, polygon$		$\longrightarrow ORA\_SP\_PRED$
$atom\_oraSp$	ORA_SP_PRED	$\longrightarrow ORA\_SP\_ATOM$
c_oraSp	ORA_SP_ATOM	$\longrightarrow C\_ORA\_SP$
ec_oraSp	C_ORA_SP	$\longrightarrow EC\_ORA\_SP$
dc_oraSp	C_ORA_SP	$\longrightarrow DC\_ORA\_SP$
dec_oraSp	$EC\_ORA\_SP \cup DC\_ORA\_SP$	$\longrightarrow DEC\_ORA\_SP$

Table 5.1: Signature describing the spatial constraint types

The type ORA_SP_PRED has several type constructors, each of them corresponding to a different OracleSpatial predicate. This is made, because then we can parameterize the type ORA_SP_ATOM with the predicates, since atomic constraints have different interpretations, according to the relation, they are expressing, and for this they represent different spatial objects. Therefore, the carrier sets of these type constructors are pairs, composed of a singleton $\{p\} \in \mathcal{P}_S$, where p is the corresponding spatial predicate and the rank of p, e.g. the carrier set of $disjoint$ is $A_{disjoint} = \{(disjoint, 2)\}$. The carrier sets of the other types remain the same as in Section 4.2.

The schema of the classes extends the one in Figure 4.5. Figure 5.7 shows a simplified schema, where constraint classes have been "extended" with the OracleSp class.

Actually, four OracleSpatial classes have to be defined, corresponding to the four families of constraints, namely OracleSpC, OracleSpEC, OracleSpDC, OracleSp (DEC), each of them has methods to implement renaming, intersection, union, projection and the function $satisfy$, which is actually a SQL query. Each class has the methods *compose* and *decompose*: the first one is used by the constructor, to create a constraint from an OracleSpatial object and, the second one is used to create from a constraint a OracleSpatial object. Moreover, all classes need methods to implement the Oracle functions area, length, within_distance. They exist also as Java "interfaces", as in the abstract case and, all of them refer to all classes corresponding to the spatial predicates. The more detailed schema is illustrated by Figure 5.8. The classes $disjoint, \ldots, touch$ have the method *query*, performing the

query to find tuples of objects, satisfying the atomic constraints, represented by the class, and, the methods *compose* and *decompose* to create the spatial object from the retrieved tuple.

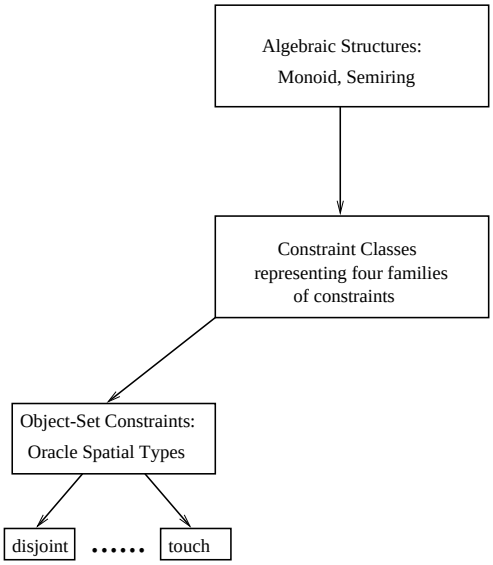


Figure 5.7: Simplified schema of the constraint classes

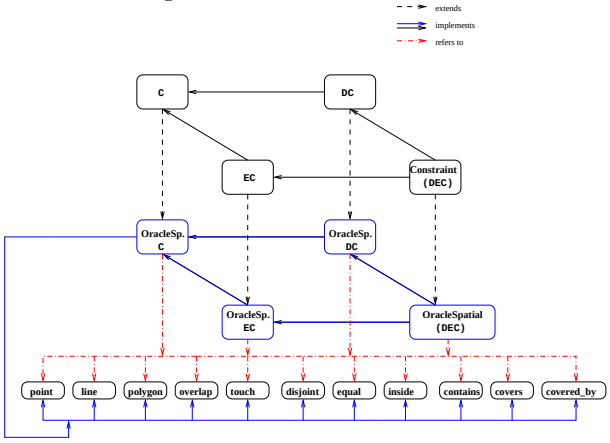


Figure 5.8: Complete schema of the spatial constraint model

Except for the atomic formulas, each constraint will be created using the command new OracleSp(arg1,..., argn), that uses the constructors of the spatial class OracleSP, which is the most general one. The constructors, then, will call the constructor of the class, corresponding to the type of the constraint. The concept is better understandable through Example 5.5.

Example 5.5 This example shows the mechanism, with which constraints are created. As there doesn' t exist an implementation yet, but only a design, some parts are resumed and written in normal language.
Let's suppose that we want to create the constraint object:

$$disjoint(x_1, x_2) \wedge disjoint(x_3, x_4)$$

Let's assume that x_1, x_2, x_3, x_4 have been stored in the vector objs, as the array [(x1,x2),(x3,x4)]. To create the corresponding constraint we use the command:

new OracleSp("disjoint", objs)

The constructor of the class checks if the constraint has dependent variable, and if yes the constraint has to be solved as a unique block as described in Definition 5.5 and therefore one of the constructor of the class OracleSpEC will be called, otherwise it calls the constructor of the class OracleSpC.

```
OraclSp   (String pred, OracleSp[] objs) {
          if (dependentVariables(objs)== null) new OracleSpC(pred, objs)
          else new OracleSpEC(pred, objs)
}
```

Below it is the constructor of the class OracleSpC:

```
OracleSpC   (String pred, OracleSp[] objs
            for i=1 to objs.length() atoms[i]= new pred(obj[i]);
            A= Set(atoms);
```

The constructor initializes the global variable A with the set of atomic formulas, composing the conjunctive constraint, i.e. the domain A of the monoid is instantiated. new pred(objs) is the constructor of atomic formulas, i.e. in this case disjoint, and creates pairs of disjoint objects.

□

5.1.3 Example

We give an example of spatial modeling using OracleSp constraints. The example describes parts of subway and city-train nets in a city, depicted in Figure 5.9.

The sets of stations, each of them representing stations of a particular line, can be modeled in OracleSpatial by multi-polygons. We define first a class Station,

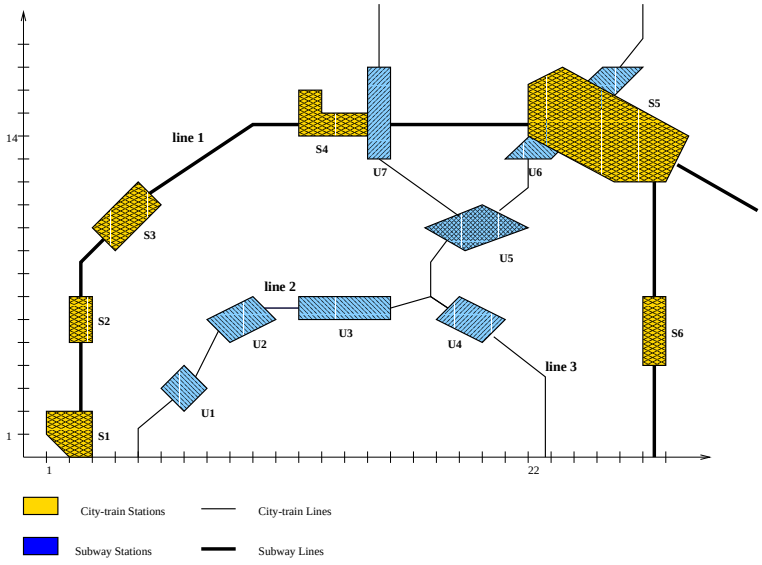


Figure 5.9: Map of subway and city-train lines

represented by its name and the extent, describing its shape and location, as follows:

```
class          Station (
             String name;
        OracleSP extent;
                Station   (String n, Polygon ext) {
                          name = n;
                          extent =(OracleSP)ext;
                        }
);
```

The assignment extent =(OracleSP)ext contains a casting operation, which transforms an object of type polygon, in an object of type OracleSp, i.e. a (atomic) constraint. The possibility of casting is due to the relationships existing between the two classes, see Figure 5.8. This means that the spatial extent of class Station is represented by an OracleSp constraint.

Below is an example of creation of the object S5, (see Figure 5.9), as instance of the class Station:

```
Station S5 = new Station (   "Ost_bahnhof",
                             new Polygon(
                             mdsys.sdo_geom(null,null, 3,
                             mdsys.sdo_array_elem_info(1,3,3),
                             mdsys.sdo_coordinates ((22,14),(22,16),
                             (24,17),(29,14),(18,12),(25,12),(22,14)))
                             )
);
```

The extent of the object S5 is created using the spatial data type of OracleSpatial mdsys.sdo_geom, that with appropriate parameters defines the polygon S5 depicted in Figure 5.9. The arguments null, null, 3 state that the type of geometry is polygon, the array mdsys.sdo_array_elem_info(1,3,3) serves to read the coordinates array, and says that the geometry is composed of one element (as it has only one triplet), whose type is polygon (specified by 3,3) and, that the first vertices can be read at position 1 of coordinates array. Finally, the mdsys.sdo_coordinates specifies the vertices of the polygon.

We would like to model a map of the stations in a city. We assume that the city has a subway line and a city-train line, each of them having a set of stations. Each set of stations is representable as a collection of disjoint polygons. To model a set of stations, we define the following Station_maps class.

```
class   Stations_map          (
        String name;
        OracleSp[] station=   new OracleSp[50];
        OracleSp extent;
        Stations_map          (String n, OracleSp[] s) {
                              name = n;
                              for i=0 to s.length() stations[i] = s[i];
                              extent = new OracleSp ("disjoint", s)
        };
        Stations_map          (String n, OracleSp ext) {
                              name = n;
                              extent = ext;
                              station = ext.decompose();
        };
);
```

The OracleSp constructor creates the constraint $\bigwedge_{i,j=1}^{s.length()} disjoint(s[i], s[j])$, that describes a multi-polygon object. Also in this case, the spatial extent is represented by a constraint, that is an OracleSpC constraint. Instances of the Stations_map are easily created by:

```
Stations_map city-train_st =new Stations_map (   "City-train_st",
                                                 [S1.extent,S2.extent,S3.extent,
                                                 S4.extent,S5.extent,S6.extent]
Stations_map subway2_st =new Stations_map(   "Subway2_st",
                                             [U1.extent,U2.extent,U5.extent,
                                             U6.extent]
Stations_map subway3_st =new Stations_map(   "Subway3_st",
                                             [U4.extent,U5.extent,U7.extent]
```

In the same way, we can represent train lines. We suppose, that each line is composed of a set of segment lines. As each segment represents the line that joins two stations together, the line belonging to one set are disjoint with each other. The class to represent them is similar to Stations_map.

```
class   One_line          (
        int number;
        line[] segments=  new line[50];
        OracleSp extent;
        One_line          (int no, line[] l) {
                          number = no;
                          for i=0 to l.length() line[i] = l[i];
                          extent = new OracleSp ("disjoint", l)
        };
);
```

Instances of One_line are created giving the set of line segment. In Figure 5.9, three lines are illustrated, e.g. line 1, line 2 and line 3.
- The following query creates a railway line, composed of stations and railway segments:

```
select   new touch(x.extent, y.extent)
into     Set(OracleSp) railway_line
from     Stations_map x, One_line y
```

The new touch(x1.extent, x2.extent) has been defined in Example 5.2 and creates the set of touching objects. Let $\varphi_1(x_1, \ldots, x_n)$ and $\varphi_2(y_1, \ldots, y_m)$, $n \leq m$ be the constraint formulas, which represent x1.extent and x2.extent, respectively. The result of the query is the set of geometries, satisfying the constraint:

$$\varphi_1(x_1, \ldots, x_n) \wedge \varphi_2(y_1, \ldots, y_m) \wedge touch(x_1, y_1) \wedge touch(y_1, x_2) \wedge \cdots \wedge touch(y_{m-1}, x_n) \wedge touch(x_n, y_m)$$

- Let's suppose, we want to know in which station of the line 2 we can change to the line 3. The query finds the stations that overlap.

```
select   new Stations_map("change_st", new overlap(x1, x2))
into     Set(Stations_map) result
from     Stations_map x1, x2
where    x1.name = 'Subway2_st'
and      x2.name = 'Subway3_st'
```

The above query creates a new object of the class Stations_map, (using the second constructor specified in the declaration of the class), giving the name "change_st" and the extent resulting from overlapping the extents of the objects x1 and x2.

- Display the part of lines which are inside the rectangle *@rectangle* drawn on the screen.

```
select   new overlap(@rectangle, x)
into     Set(OracleSp) result
from     railway_line x
```

The above query exploits a feature of OracleSpatial, that uses the so called *query_windows*. new overlap(@rectangle, x) can be defined as a method of the class overlap, which implements query_windows in the following way:

```
select   x.extent
from     railway_line x
where    mdsys.sdo_relate(x.extent,   mdsys.sdo_geometry(3,NULL,NULL,
                                      mdsys.sdo_elem_info(1,3,3),
                                      mdsys.sdo_ordinates(x1,y1, x2,y2)),
                                      'mask=overlap querytype=window') = 'TRUE')
```

5.2 Mapping flat constraints to linear constraints

In this section, we briefly show how the flat constraint model can be mapped to two-dimensional linear constraints. The description will be short, as a lot of work has been already done, for developing linear constraint databases and promising prototypes already exist. Indeed, our model is based on the CCUBE model ([BSCE97]), a constraint database, whose data are represented by linear constraints. The challenge in this section is to show the versatility of our framework, in modeling constraints, based on different theories.

Linear constraints have been already largely introduced in this thesis, for recalling see Section 2.1.2 and Section 3.2.3. The vocabulary on which linear constraints are defined is

$$\Omega_{lin} = \langle +, <, >, 0, 1 \rangle$$

and atomic constraints are the formulas

$$\varphi(x_1, x_2) = ax_1 + bx_2 + c\,\Theta\,0 \tag{5.3}$$

where a, b, c are constants and $\Theta = \{<, >\}$. For a better readability, we denote atomic formulas using the formula 5.3, also if it is not a flat constraint, but the equivalence between linear constraint formulas and flat constraints has been already explained by Example 4.1 on page 58. The Example 4.2 on page 4.2 explains how linear constraint formulas are interpreted. Of course, it has to be assumed that an underlying data model, like vector model (see Section 3.2.1) exists, to make data representable as linear constraints.

5.2.1 Extending the data model with the LinearConstraint class

After the previous assumptions, we can extend the schema, adding a LinearConstraint class and C_lin, EC_lin, DC_lin, DEC_lin subclasses, representing the four constraint families, as we have done for OracleSpatial constraints. Figure 5.10 shows a simplified extension.

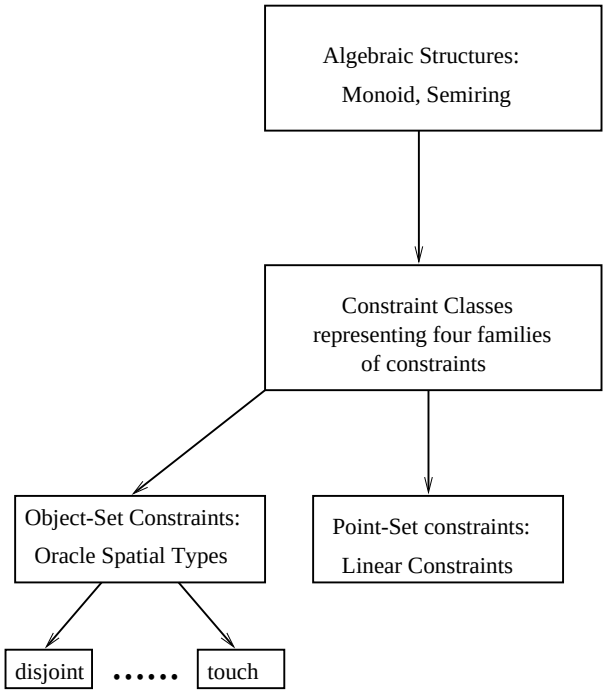


Figure 5.10: Extension of the schema with the LinearConstraint class

The class LinearConstraint has a different method *rename*, such that for constraint objects $c_1, \ldots, c_n$, the renaming replaces the variables as follows:

$$\exists y(c_1[u_1/y, v_2/z] \wedge \cdots \wedge c_n[u_n/y, v_n/z]).$$

The method $satisfy$ implements the simplex algorithm, to verify the satisfaction of formulas and conjunction of formulas.

5.2.2 Examples

This example has been taken from [BSCE97]. Consider a two dimensional desk "my-desk" depicted in Figure 5.11 as the larger rectangle. The smaller square is the desk's drawer, which may be open, i.e. move relatively to the desk. Similarly, the desk may be moved in the room. There are three systems of coordinates used in Figure 5.11: U, V, the room's system: W, Z, the desk's coordinate system used to describe the desk's shape independently of the location of the desk in the room: and $W1$, $Z1$, the drawer' s coordinate system. The pair of variables (x, y) describes the center of the desk in the room' s coordinates; similarly, (p, q) describe the center of the drawer in the desk's coordinates.

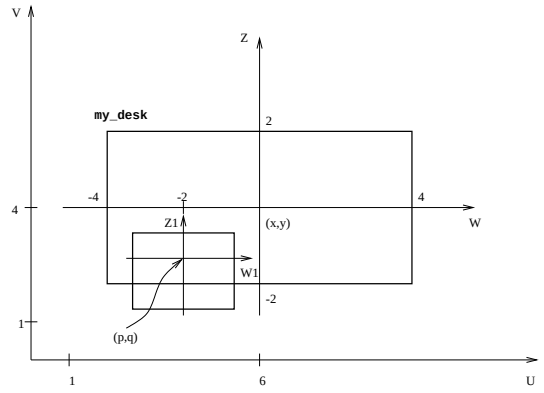


Figure 5.11: An instance of a desk with drawer in the room

The constraint formula $(-4 \leq w \leq 4) \wedge (-2 \leq z \leq 2)$ describes the extent of desk in the desk's coordinates. Similarly, the extent of the desk's drawer in the drawer' s coordinates can be described by the constraint $(-1 \leq w1 \leq 1) \wedge (-1 \leq z1 \leq 1)$. The possible location (p, q) of the drawer' s center in the desk's coordinates can be described by $p = -2 \wedge -3 \leq q \leq -1$.

The translation between the desk's W, Z and the room' s U, V systems of coordinates can be captured by the constraint $u = x + w \wedge v = y + z$. Suppose that the desk's center has room' s coordinates $(6, 4)$, i.e. $x = 6 \wedge y = 4$. Then, for example, we can find the extent of the desk in the room' s coordinate by simplifying the following formula:

$$\exists x, y, w, z(((-4 \leq w \leq 4) \wedge (-2 \leq z \leq 2)) \wedge (u = x + w \wedge v = y + z) \wedge (x = 6 \wedge y = 4))$$

where the first component is the desk's extent in the local coordinates, the second is the translation between the coordinate systems and the third is the position of the desk's center.

Consider now an architectural design example schema depicted in Figure 5.12. We assume that the database keeps a set all_office_objects. Similarly, all_desks and all_file_cabinets are kept.

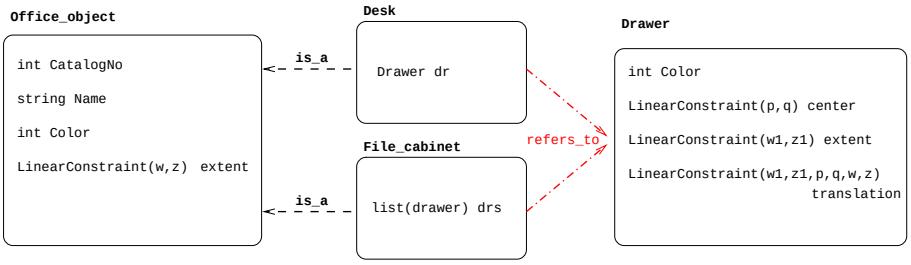


Figure 5.12: A schema for the desk's example

Objects of the class Office_object have standard attributes such as Catalog, Name and Color, and also the extent attribute, describing the object's shape. The extent attribute is declared as LinearConstraint(w,z), to indicate that it must be represented as a constraint with free variables w and z.

The classes Desk and File_cabinet are subclasses of Office_objects, and, therefore, inherit all its attributes. In addition, the classes Desk and File_cabinet have an attribute dr, which is declared as reference to an object of the class Drawer.

Each drawer, in turn, is characterized by possible locations of its center, declared as LinearConstraint(p,q); its extent as LinearConstraint(w1,z1); and translation, declared as LinearConstraint(w1,z1,p,q,w,z), which would hold an equation describing the interconnection between $(w1, z1)$, a point in the drawer' s coordinates and (w, z), the same point in the desk's coordinates, provided that the center of the drawer is at (p, q) in the desk's coordinates.
Below is an example of instantiation of my_desk of the class Desk:

Desk my_desk	= Desk (	
	22354,	% Catalog
	"one_drawer_desk",	% Name
	"red",	% Color
	$(-4 \leq w \leq 4 \wedge -2 \leq z \leq 2)$,	% extent
new	Drawer(	% dr
	"blue",	% Color
	$(p = 2 \wedge -3 \leq q \leq -1)$,	% center
	$(-1 \leq w1 \leq 1 \wedge -1 \leq z1 \leq 1)$,	% extent
	$(w = w1 + p \wedge z = z1 + q)$	% translation
	)	
);		

The next query finds, for each desk in all_desk, its extent in the rooms coordinates, assuming the center of the desk is located at the point $(6, 4)$ and the desk orientation is aligned with the room's walls, i.e. the translation equation is $u = w+x \wedge v = z+y$.

```
select   new LinearConstraint(     (u,v)| (dsk.extent ∧
                                   u = w+x ∧ v = z+y ∧
                                   x=6 ∧ y=4))
into     {Set(LinearConstraint)}   result
from     all_desks                 as Desk dsk
```

The next query finds all pairs (dsk, dsk.extent) for all desks that, if centered at $(6, 4)$, would intersect the area $(3 \leq u \leq 4 \wedge 8 \leq v \leq 10)$ in the room's coordinates.

```
select   new pair(dsk, dsk.extent)
into     {Set(pair)} result
from     all_desks as Desk dsk
define   area as LinearConstraint (3 ≤ u ≤ 4 ∧ 8 ≤ v ≤ 10)
define   transl as LinearCostraint (u = x + w ∧ v = y + z)
where    satisfy(area ∧ dsk.extent ∧ transl ∧ x=6 ∧ y=4)
```

Here, the pair(dsk, dsk.extent) is a constructor of the class pair; expression define expr1 as expr2 causes the replacement expr1 by expr2 in the remainder of the comprehension; they are used simply as shortcuts. satisfy performs the satisfiability test of the constraint inside the parentheses which checks whether area intersects the desk's extent. Finally, the last query finds all desks whose drawer may intersect, if closed or partly or fully open, the desk area $(-1 \leq w \leq 1 \wedge -1 \leq z \leq 1)$.

```
select   dsk
into     {Set(Desk)} result
from     all_desks as Desk dsk
define   dr_ext as LinearConstraint dsk.drawer.extent
define   dr_loc as LinearConstraint dsk.drawer.center
define   dr_transl as LinearConstraint dsk.drawer.translation
define   dr_ext_in_dsk as LinearConstraint (w,z)|(dr_ext ∧ dr_loc ∧ dr_transl)
where    satisfy (dr_ext_in_dsk ∧ -1≤ w≤ 1 ∧ -1≤ z ≤ 1)
```

Example of integration of the two models

This example shows a possible integration between two different data models. We suppose that spatial objects are represented as linear constraints, and we will show how to change their representation from linear constraints to OracleSp constraints, in order to use particular OracleSpatial functions, as within_distance(geo1, geo2), which returns the distance between the two geometries geo1 and geo2.
We assume to have a database of rectangles, represented by the following class:

```
class   LinearRectangles            (
        String name;
        LinearConstraint(x,y) extent;
        LinearRectangles            (String n, LinearConstraint ext) {
                                    name = n;
                                    extent = ext;
        };
);
```

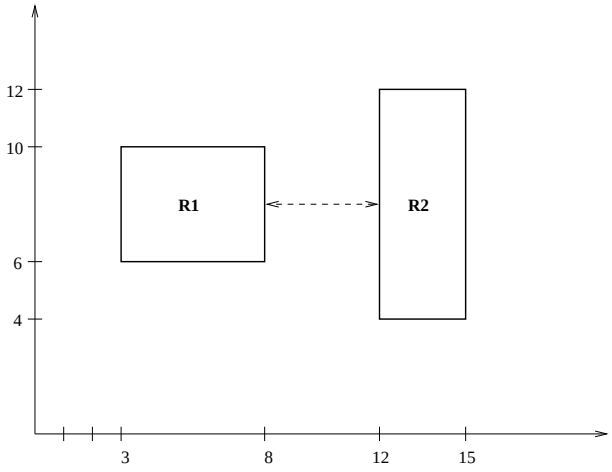


Figure 5.13: Distance between two rectangles

Hence, to create the two rectangles, depicted in Figure 5.13, we use the following commands:

```
LinearRectangles R1 = new LinearRectangles (   "r1",
                                               3 ≤ x ≤ 8 ∧ 6 ≤ y ≤ 10)
LinearRectangles R2 = new LinearRectangles (   "r2",
                                               12 ≤ x ≤ 15 ∧ 4 ≤ y ≤ 12)
```

The following query finds the distance between the two rectangles, using the OracleSpatial with_distance function.

```
select   sdo_geom.sdo_within_distance(geo1, geo2)
into     real result
define   OracleSp geo1 as new OracleSp(r1.extent.vertices()),
define   OracleSp geo2 as new OracleSp(r2.extent.vertices()),
from     LinearRectangles r1,r2
where    r1.name = 'r1'
and      r2.name = 'r2'
```

Note that vertices() is a method of the class Linearconstraint, which given a linear formula, representing a spatial object, returns its vertices.

Chapter 6

Modeling Temporal Information

In this chapter, we present first an extension of the model described in Chapter 4 to represent temporal information. In particular we define temporal data types and flat atomic formulas on the vocabulary, constituted by the Allen's predicates, introduced in Section 3.1.1. Secondly, we present an integration between the defined temporal model and the spatial model of Section 5.1, to model spatio-temporal information. The chapter consists of two main sections: Section 6.1 describes the temporal model and Section 6.2 presents the spatio-temporal model.

6.1 Mapping flat constraints to Allen' s predicates

In this section, we design a temporal constraint database, whose constraints are built up from atomic formulas, consisting of Allen' s interval relations, see Section 3.1.1. As in the spatial case, the challenge in designing the temporal constraint model is the demonstration of its completeness, i.e. temporal objects can be represented as constraints and vice versa.

Constraints are interpreted on a domain of *temporal objects*, that are time points, time intervals and particular combinations of them, for instance *meeting intervals*. Also in this case, we do use OracleSpatial as underlying model of data. Moreover, differently from the spatial case described in Section 5.1, in which spatial objects were already defined by OracleSpatial, in the temporal case, we will define temporal objects as special subsets of the spatial *point* and *line* objects, i.e. unidimensional points and lines. Then, the process of mapping consists of the definition of the vocabulary Ω_T and of the interpretation structure $\mathcal{M}_T = \langle \mathcal{D}_T, \Omega_T \rangle$.

6.1.1 The temporal domain

Let $\mathcal{D}_T$ be a set of well-defined time intervals. We assume that well-defined intervals are all closed intervals whose starting and ending points are finite and reals.

Time intervals can be represented as OracleSpatial objects, in particular as a subset of *linestring* objects. The data type *linestring* in Oracle is represented as a list of two-dimensional points, i.e. the vertices of the line-string.

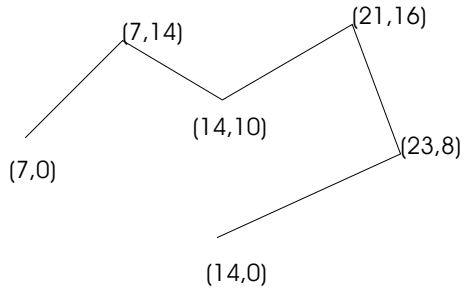


Figure 6.1: Representation of a *linestring* object in OracleSpatial

For instance, the object in Figure 6.1 will be stored as the sequence of points ((7, 0),(7, 14),(14, 10),(23, 8), (14, 0)); line strings are elements of the set A_{line}, defined in Section 5.1.1. Time intervals are 1-dimensional lines and to define them, we first define the set of 1-dimensional points, as subsets of A_{point}, (also defined in Section 5.1.1), as follows

$$A_{1-point} = \{(x, 0) | \, x \in \mathbf{R}\}.$$

It easily follows that $A_{1-point} \subseteq A_{point}$. Then the set of time intervals is defined as follows

$$A_{interval} = \{(p_1, p_2) | \, p_1, p_2 \in A_{1-point} \wedge p_1 \preceq p_2\},$$

where

$$p_1 \preceq p_2 \Leftrightarrow p_1 = (x_1, 0), p_2 = (x_2, 0) \wedge x_1 \leq x_2.$$

p_1 and p_2 represent the interval's start and the ending point, respectively. Intervals are particular cases of line-strings, i.e. $A_{interval} \subseteq A_{line}$, therefore they can be represented as OracleSpatial objects. Note that the particular case of intervals, with $p_1 = p_2$ represents time points.

In the same way, we can define the *multinterval* object, as a subset of $A_{multilinestring}$, as follows

$$A_{multinterval} = \{(l_1, \ldots, l_n) | \, \forall i, 1 \leq i \leq n, l_i = (p_i^1, p_i^2) \in A_{interval}$$

Figure 6.2 shows two particular elements of $A_{multinterval}$, they are particular sequences which we will often use in time modeling and belong to a subset of $A_{multinterval}$, that we call the set of not overlapping intervals $A_{\neg overlap}$. (a) illustrates a *multinterval*

object composed of two disjoint intervals, represented by the sequence $((q_1, q_2), (q_3, q_4))$; (b) shows three "meeting" intervals, i.e. $((p_1, p_2), (p_3, p_4), (p_5, p_6))$, which have collapsing ending points, i.e. $p_2 = p_3$ and $p_4 = p_5$.

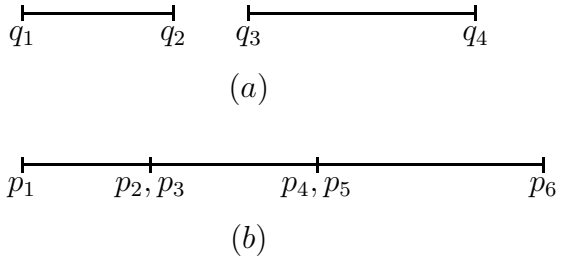


Figure 6.2: Representation of *multinterval* objects

We are now ready to define formally the temporal domain $\mathcal{D}_T$ as follows:

$$\mathcal{D}_T = \perp \cup A_{interval} \cup A_{multinterval} \tag{6.1}$$

In the follow, we denote with $i, i_1, i_2, \ldots, i_n$ elements of $\mathcal{D}_T$. We define the two functions sp and ep on $\mathcal{D}_T$, as follows:

$sp, ep : \mathcal{D}_T \longrightarrow A_{1-point}$, where

$$sp(i) = \begin{cases} p_1 & : \quad i \in A_{interval}, i = (p_1, p_2) \\ p_l & : \quad i \in A_{multinterval}, i = ((p_1, p_2), \ldots, (p_{n-1}, p_n)), 1 \leq l \leq n, \\ & \quad \wedge \forall j, 1 \leq j \leq n, p_l \preceq p_j \end{cases} \tag{6.2}$$

$$ep(i) = \begin{cases} p_2 & : \quad i \in A_{interval}, i = (p_1, p_2) \\ p_l & : \quad i \in A_{multinterval}, i = ((p_1, p_2), \ldots, (p_{n-1}, p_n)), 1 \leq l \leq n, \\ & \quad \wedge \forall j, 1 \leq j \leq n, p_j \preceq p_l \end{cases} \tag{6.3}$$

$sp(i)$ and $ep(i)$ return respectively the starting and the ending point of the interval i. Thus the set of not overlapping intervals can be defined as the following:

$A_{\neg overlap} = A_{meet} \cup A_{disj}$

where

$A_{meet} = \{(l_1, \ldots, l_n) \in A_{multinterval} : \forall i, 1 \leq i \leq n-1, ep(l_i) = sp(l_{i+1})\}$.

$A_{disj} = \{(l_1, \ldots, l_n) \in A_{multinterval} : \forall i, 1 \leq i \leq n-1, ep(l_i) \prec sp(l_{i+1})\}$.

Therefore, on $A_{\neg overlap}$, there is a linear order, while on the other multi-intervals, the order of the elements is not linear, nevertheless there exists a linear order on the ending points of the intervals. Hence, on $\mathcal{D}_T$, we can define a binary order relation $\subset$, as follows:

$$i_1 \subset i_2 \Leftrightarrow$$
$$sp(i_2) \prec sp(i_1) \text{ and } ep(i_1) \prec ep(i_2) \vee sp(i_1) = sp(i_2) \text{ and}$$
$$ep(i_1) \prec ep(i_2) \vee sp(i_2) \prec sp(i_1) \text{ and } ep(i_1) = ep(i_2) \text{ iff } i_1, i_2 \in A_{interval}$$

$$(i_1^1, \ldots, i_n^1) \subset (i_1^2, \ldots, i_m^2) \Leftrightarrow$$
$$n \leq m \wedge \forall h, 1 \leq h \leq n, \exists j, 1 \leq j \leq m : i_h^1 \subset i_j^2$$
$$\text{iff } (i_1^1, \ldots, i_n^1), (i_1^2, \ldots, i_m^2) \in A_{multinterval}$$

It is easy to proof that $\subset$ is *antisymmetric* and *transitive*.
The next example shows the use of intervals and multi-intervals in modeling different kinds of temporal information.

Example 6.1 We want to model the information about the places where John is during his working day, assuming that John is a creature of habit. We assume that the time granularity is one hour. We describe three places of John: *John at the bar*, *John at work* and *John at home*. Figure 6.3 illustrates the three temporal objects associated to the three places.
Supposed that John goes to the bar only in the night, the associated timestamp is represented by one interval $(21, 24)$, while in the other two cases the timestamps are supposed not to be continued, therefore they are represented by a sequence of disjoint intervals, i.e elements of $A_{multinterval}$. The time in which John is at work is represented by the multi-interval $((9, 13), (14, 18))$, while John is at home in the multi-interval $((0, 8), (19, 21))$.

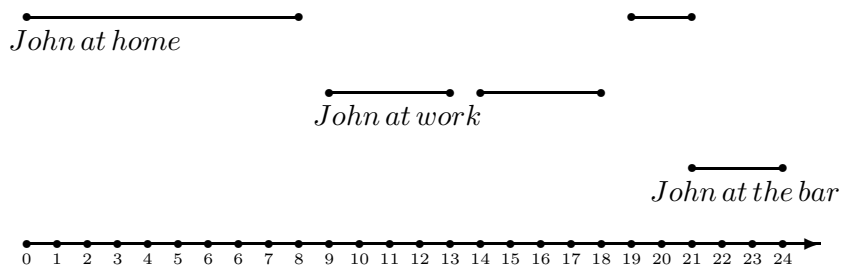


Figure 6.3: John's usual places during a day

□

6.1.2 Temporal constraints

Allen in [All83] defined a set of binary relations between time intervals, which cover all possible relations between two intervals. They are illustrated, together with their semantics by Table 3.1 on page 33. We define the temporal vocabulary, as the set of Allen' s two-place predicates, plus two other one-place predicates.

Definition 6.1 (Temporal vocabulary). The temporal vocabulary Ω_T consists of the set $\mathcal{TP} = \mathcal{TP}_{una} \cup \mathcal{TP}_{bin}$, where $\mathcal{TP}_{bin}$ contains the following two-place predicates: *equal*, $before$, $after$, *meets*, *overlap*, *starts*, *during*, $finishes$ and $\mathcal{TP}_{una}$ contains the 1-place predicates *time_point*, *time_interval*.
$\mathcal{C}_T = A_{string}$. □

Hence, temporal constraint are built up from atomic formulas, defined on Ω_T. We denote the set of temporal formulas with $\mathcal{F}_T$. The following formulas are examples of well-defined temporal formulas, i.e. elements of $\mathcal{F}_T$:

(1) $meets(x_1, x_2) \wedge overlap(x_2, x_3)$,

(2) $\exists x_1(before(x_1, x_2) \wedge during(x_1, x_3))$,

(3) $starts(x_1, x_2) \vee before(x_1, x_3)$

The interpretation of Ω_T-constraints is given on the temporal domain $\mathcal{D}_T$, defined in Section 6.1.1, assigning to each constraint of $\mathcal{F}_T$ a binary relation between elements of $\mathcal{D}_T$, that results either to an interval or to a multinterval. At this purpose, we define the interpretation function $\mathcal{I}^T_{i\in\mathbf{N}}$. Let $\mathcal{F}_T(n) = \{\varphi \in \mathcal{F}_T | freevar(\varphi) = n\}$, then we define the following class of functions:

$$\mathcal{I}^T_n : \mathcal{F}_T(n) \longrightarrow (\underbrace{\mathcal{D}_T \times \cdots \times \mathcal{D}_T}_{n\,times})$$

We present the interpretation function step by step, first defining it on atomic formulas with the following definition.

Definition 6.2 (Interpretation of atomic constraints). Let $\mathcal{P}_T(j)$ be the set of j-place predicates of $\mathcal{P}_T$, $p \in \mathcal{P}_T$ and $i, i_1, \ldots, i_k$ elements of $\mathcal{D}_T$, then the interpretation function $\mathcal{I}^T_n$ on atomic formulas is defined as follows:

$$\mathcal{I}^T_{n\in\{1,2\}} : \mathcal{P}_T(n) \longrightarrow \mathcal{D}^n_T$$

where $\mathcal{I}^T_{n\in\{1,2\}}$ is defined as follows:

$$\mathcal{I}^T_{n\in\{1,2\}}(p(x_1,x_2)) = \{(i_1,i_2)|\, rel_p(i_1,i_2)\}$$

rel_p is an OracleSpatial relation, obtained by the SQL select:

select i_1, i_2
from $obj_1\, i_1,\ obj_2\, i_2$
where $p(i_1,i_2) =$ "$true$"

While in the spatial case the relationship p can be automatically translated in the corresponding OracleSpatial operator, in the temporal case the translation is not direct, but anyway easy, as i_1, i_2 are OracleSpatial objects. In the following, we describe the relation p, in each of its case:

$$p(i_1,i_2) = \begin{cases} p(i_1,i_2) & : \quad p \in \{equal,\, overlap\} \\ disjoint(i_1,i_2) \wedge ep(i_1) \preceq sp(i_2) & : \quad p = before \\ disjoint(i_1,i_2) \wedge ep(i_2) \preceq sp(i_1) & : \quad p = after \\ touch(i_1,i_2) & : \quad p = meets \\ covered\_by(i_1,i_2) \wedge sp(i_1) = sp(i_2) & : \quad p = starts \\ inside(i_1,i_2) & : \quad p = during \\ covered\_by(i_1,i_2) \wedge ep(i_1) = ep(i_2) & : \quad p = finishes \end{cases} \tag{6.4}$$

$$p(i) = \begin{cases} sp(i) = ep(i) & : \quad p = time\_point \\ sp(i) \preceq ep(i) & : \quad p = time\_interval \end{cases} \tag{6.5}$$

□

The above definition shows that the definition of temporal objects as particular cases of OracleSpatial objects, makes the mapping on Oracle database possible.

The equations 6.4 and 6.5 hold for simple intervals, i.e. elements of $A_{interval}$. For multi-interval objects, the definition of some of them change. *overlap*, *during* (and *starts*, $finishes$ which can be considered particular cases of *during*) of multi-intervals may have different number of overlapping intervals.

Figure 6.4 shows some examples. In $during(i_1,i_2)$, the multi-interval i_1 is wholly contained in only one of the intervals of i_2, while in $during(i_3,i_4)$ each interval of i_3 is contained in exactly one interval of i_4. Let's assume that $i_1 = l^1_1, \ldots, l^1_m$ and $i_2 = l^2_1, \ldots, l^2_n$, it follows that

$$during(i_1,i_2) \Leftrightarrow \forall j, 1 \le j \le m, \exists k, 1 \le k \le n : during(l^1_j, l^2_k)$$

For *starts* and $finishes$, we add to the above condition, the ones for the starting and ending points, respectively.

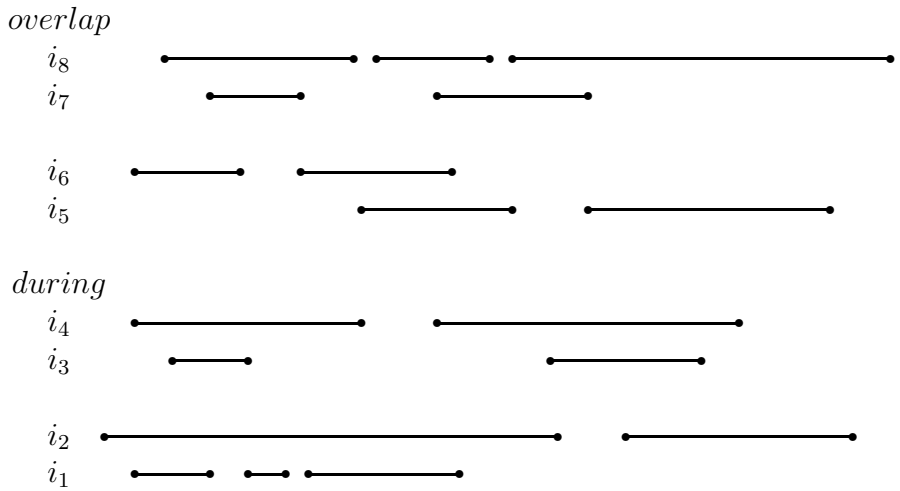


Figure 6.4: *overlap* and *during* of multi-intervals

$$starts(i_1, i_2) \Leftrightarrow during(i_1, i_2) \wedge \exists j, k, 1 \leq j \leq m, 1 \leq k \leq n : starts(l_j^1, l_k^2)$$
$$finishes(i_1, i_2) \Leftrightarrow during(i_1, i_2) \wedge \exists j, k, 1 \leq j \leq m, 1 \leq k \leq n : finishes(l_j^1, l_k^2)$$

In *overlap*, it is enough that at least two intervals overlap, for instance in $overlap(i_7, i_8)$, one interval of i_7 is contained in one interval of i_8, nevertheless as there exist two overlapping intervals, the relation is *overlap*. Therefore, it follows that

$$overlap(i_1, i_2) \Leftrightarrow \exists j, k, 1 \leq j \leq m, 1 \leq k \leq n : overlap(l_j^1, l_k^2)$$

As we have done in OracleSpatial case, we will prove that the tuples of the relation rel_p can be stored as Oracle temporal objects.

Theorem 6.3 $\forall p \in \mathcal{P}_T$, each tuple of rel_p can be represented as a temporal object.

Proof. We define a family of functions, which we will use to transform tuples of rel_p in intervals or multintervals:

$$t\_compose_{(n \in \mathbf{N})} : \mathcal{D}_T^n \longrightarrow \mathcal{D}_T$$

Each rel_p corresponds to the set of tuples $\{(i_1, i_2) | p(i_1, i_2)\}$, therefore we can denote each tuple with $p(i_1, i_2)$, $(i_1, i_2) \in \mathcal{D}_T^2$.

$$t\_compose(p(i_1, i_2)) = \begin{cases} (i_1, i_2) \in A_{multinterval} & : \quad p = before, meet \\ (i_2, i_1) \in A_{multinterval} & : \quad p = after \end{cases}$$

$t\_compose(p(i_1, i_2)) = i_2,\ p \in \{starts, during, finishes\}$
$t\_compose(equal(i_1, i_2)) = i_1\ t\_compose(overlap(i_1, i_2)) = i \in A_{multinterval}$
where $i =$sdo_geom.sdo_poly_intersection(i_1, i_2) □

Example 6.2 The example shows how to create a multi-interval object by means of OracleSpatial queries. In the next, the multi-interval $((9, 13), (14, 18))$, will be created as an instance of the OracleSpatial type mdsys.sdo_geom. The value 6 corresponds to the geometry type, that is *multilinestring*. The values 2,1, in the triplets of the mdsys.sdo_array_elem_info, state that type of each element of the multilinestring is a simple linestring. Note that all the vertices have 0 as ordinate:

```
new Multinterval((9,13),(14,18))≡  mdsys.sdo_geom(null,null,6,
                                   mdsys.sdo_array_elem_info(1,2,1, 5,2,1),
                                   mdsys.sdo_coordinate(19,0,13,0,14,0,18,0))
```

□

For the interpretation of the constraint formulas, i.e. simple conjunctives, existential conjunctive and disjunction of conjunctives formulas, we refer to the Definitions 5.4, 5.5, 5.6, in Section 5.1.1.

The interpretation structure (or model) is $\mathcal{M} = \langle \mathcal{D}_T, \mathcal{I}_{n \in N} \rangle$. In order to have the temporal object/constraint duality, we have to prove that the OracleSpatial unrestricted relations are definable on $\mathcal{M}_T$ with Ω_T-constraints.

Definition 6.4 (Temporal unrestricted relations). A temporal unrestricted tuple consists of a temporal object, i.e. an element of $\mathcal{D}_T$. A temporal unrestricted relation is a set of temporal objects, i.e. a subset of $\mathcal{D}_T$. Any collection of temporal unrestricted relation defines a temporal unrestricted database. □

Theorem 6.5 The temporal unrestricted database is definable on $\mathcal{M}_T$ with Ω_T-constraints.

Proof. To prove the theorem we have to show first that each temporal object, i.e. each element of $\mathcal{D}_T$ is definable as an Ω_T-constraint. At this purpose, we define the function $t\_decompose$:

$$t\_decompose : \mathcal{D}_T \longrightarrow \mathcal{F}_T,$$

such that time points are definable as

$$t\_decompose(x) = time\_point(x) \text{ iff } x \in A_{interval} \wedge x = (p, p)$$

simple time intervals are definable as

$$t\_decompose(x) = time\_interval(x) \text{ iff } x \in A_{interval} \wedge x = (p, q) \wedge p \preceq q$$

multi-interval objects, composed of disjoint intervals as

$$\begin{aligned} t\_decompose(x) = \; & before(i_1, i_2) \wedge \cdots \wedge before(i_{n-1}, i_n), \\ & iff x = (i_1, \ldots, i_n) \in A_{multinterval} \wedge i_1 \preceq i_2 \preceq \cdots \preceq i_n \end{aligned}$$

multi-interval objects, composed of meeting intervals, as

$$\begin{aligned} t\_decompose(x) = \; & meet(i_1, i_2) \wedge meet(i_2, i_3) \cdots \wedge meet(i_{n-1}, i_n) \\ & iff\, x = (i_1, \ldots, i_n) \in A_{multinterval} \\ & \wedge \forall j, 1 \leq j \leq n-1, meet(i_j, i_{j+1}) \end{aligned}$$

□

6.1.3 Temporal classes

In this subsection, we define the temporal constraint types and extend the data model defined in Section 4.2 with them. We specialize the abstract data types of the schema in Section 4.2, defining the appropriate carrier sets. The four fundamental temporal constraint types are called: $TempC, TempEC, TempDC, Temp(DEC)$ and they represent the four constraint families, whose atomic formulas are composed of temporal predicates. Table 6.1 shows these types.

Type constructor	Signature	
$before, after, equal,$ $overlap, meets, during,$ $starts, finishes,$		$\longrightarrow TEMP\_PRED$
$atom\_oraSp$	$TEMP\_PRED$	$\longrightarrow TEMP\_ATOM$
c_temp	$TEMP\_ATOM$	$\longrightarrow TempC$
ec_temp	$TempC$	$\longrightarrow TempEC$
dc_temp	$TempC$	$\longrightarrow TempDC$
dec_temp	$TempEC \cup TempDC$	$\longrightarrow Temp$

Table 6.1: Signature describing the temporal constraint types

The type $TEMP\_PRED$ has several type constructors, each of them corresponding to a different Allen' s temporal predicate. Therefore, the type $TEMP\_ATOM$ is parameterized by the predicates composing the atomic constraints. This is made, because atomic constraints have different interpretations according to the relation, which they express, and for this they represent different temporal objects.

The four temporal types are defined as sub-types of the corresponding generic constraint types, described in Section 4.2. To each of them corresponding a class and the schema is depicted in Figure 6.5.

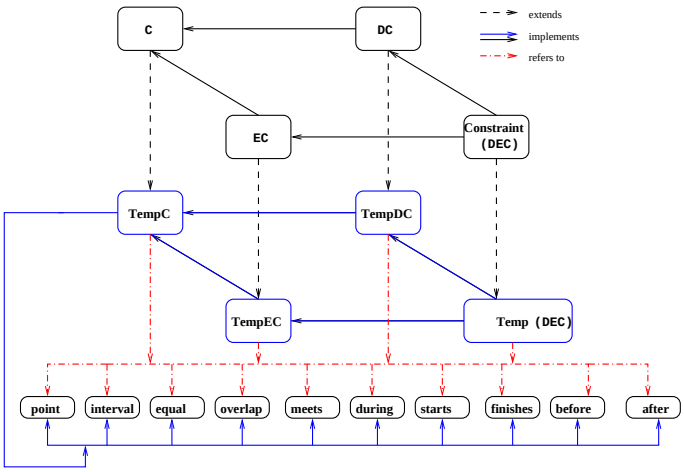


Figure 6.5: Schema of the temporal classes

Temporal classes are designed like their corresponding spatial ones and, methods are almost the same as the ones described in Section 5.1.2; therefore, they have methods, to implement intersection, union and difference and the functions *compose*, *decompose* and $satisfy$, they don' t have the method area, as it doesn' t have sense to compute the area of intervals, but the other methods within_distance, length, are also available in the temporal classes, and they enables the performing of metric queries, see 3.1.1.

6.1.4 Example

As example of temporal queries, let' s recall the Figure 6.3, and let' s suppose that we want to model the persons' activities. At this purpose, we define the class Activities, having as attributes the name of the person, his action, and the time interval(s), during which the person is usually doing his activity.

```
class   Activities {
        String name;
        String action;
        Temp time;
        Activities(String name, String a, Temp t) {
        name=n; action= a;
        time=t };
        Activities( String name, String a, Interval[] t) {
        name=n; action= a;
        time=new Temp("before", t) };
}
```

The second class constructor creates the temporal object of the class Temp, by the array of intervals t. The resulting object is a multi-interval, obtaining solving the constraint $before(t[1], t[2]) \wedge \cdots \wedge before(t[n-1], t[n])$.

In the following, we present some interesting queries, showing the features of our model.

- Query1: What does John do after work?

```
select   a2.action
into     Set(String) result
define   t1 as a1.time
define   t2 as a2.time
from     Activities a1, a2
where    a1.name = a2.name='John'
and      a1.name = 'work'
and      satisfy(after(t2,t1)∨meet(t1,t2))
```

Query1 shows the usefulness of the function satisfy, which can appear only in the where clause, and which finds an assignment of the variables t1, t2 that satisfy the constraint formula after(t2,t1)∨meet(t1,t2). The result of the query is a set of action that John does after working.

- Query2: How long do Jonh and Mary stay together at the bar?

```
select   sdo_geom.sdo_length(i)
into     Set(real) result
define   Temp i as new overlap(a1.time, a2.time)
from     Activities a1,a2
where    a1.name='john'
and      a2.name='mary'
and      a1.action=a2.action='bar'
```

Query2 creates a new temporal constraint and assigns it to the variable i. Therefore,

i is the constraint representing an interval or a multi-interval, resulting from the intersection of the temporal timestamps, in which John and Mary are at the bar. The possibility to create a new temporal object allows to use the OracleSpatial function length, which computes how many hours John and Mary have been together at the bar.

- Query3: Give me the average of the person' s working time.

```
select   sdo_geom.sdo_length(i)
into     avg(real) result
define   Temp i as a.time
from     all_activities as Activities a
where    a.action='work'
```

In Query3 we assumed that all_activities represents a set of activities, the query shows the feature of the monoid comprehension calculus to perform in a easy way aggregations. All intervals' length, returned from the query, will be add to the variable result, that is a collection monoid, whose merge operation is the average.

The next query shows how the constraint monoid calculus enriches Oracle query language, allowing in one query the creation of new temporal objects. We assume to have defined a global constant day=[0,24], i.e. an interval object storing a whole day.

- Query4: Give me the time intervals in which John is neither at work nor at the bar.

```
select   sdo_geom.sdo_difference(day, t)
into     Set(Temp) result
define   Temp t1 as a1.time
define   Temp t2 as a2.time
define   Temp t as new Temp(t1 ∧ t2)
from     Activities a1,a2
where    a1.name=a2.name='John'
and      a1.action='work'
and      a2.action='bar'
```

The clause define Temp t as new Temp(t1 ∧ t2) creates the new temporal object, resulting from the composition of the intervals t1 and t2, calling the function *compose* the returns a multi-interval (a tuple), that can be used as argument of the OracleSpatial function sdo_geom.sdo_difference.

6.2 Modeling spatio-temporal information

In this section, we show how to model spatio-temporal information, integrating the spatial and the temporal models, described in Sections 5.1, 6.1. Among the

several examples of spatiotemporal information, we dwell upon *moving objects*, see Section 3.3, i.e. spatial objects changing with the time. A highlighting example of moving object is the trajectory of a car, see Figure 6.6.

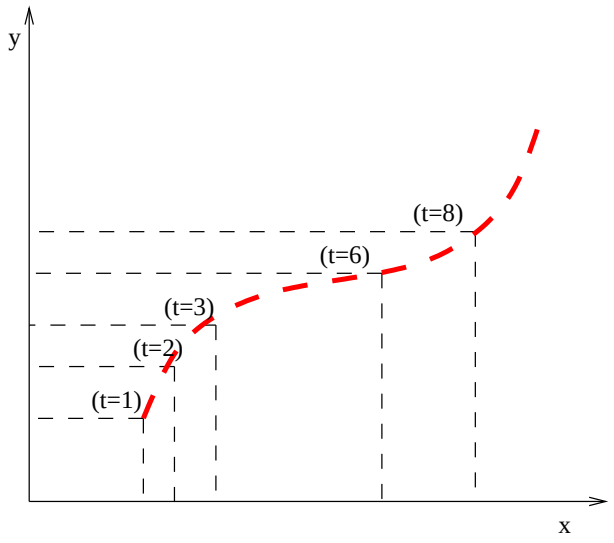


Figure 6.6: A possible trajectory of a car.

The dashed line represents the route of the car, modeled as a curve in the two-dimensional space, the time points (t=1), (t=2), etc. stating that in that particular time point the car was in a particular two-dimensional point.

With linear constraints, it is possible to model trajectories in the three dimensional space, where time represents the third dimension and spatiotemporal objects are expressed as functions in three free variables. Our model can model them, as it has been developed to generalize the spatiotemporal constraint database of [BSCE97].

Therefore, our aim in this section is to use OracleSpatial to model moving objects. At this purpose, we will follow the approach of [GRS98b] and we will not define spatio-temporal classes, with specific methods, but rather we define classes, referring to both space and time, whose class constructors satisfy particular integrity constraints. For instance, to model trajectories, we define a class having as attributes a spatial extent, that is the sequence of two-dimensional lines representing the route of car, or person etc. and a temporal extent giving the sequence of time intervals corresponding to these lines. For better understanding see Figure 6.7, which depicts in a "discrete" way the trajectory of a car.

In Figure 6.7, to each line segment corresponds a time interval, saying that in the interval of time t_i the car moves along the line l_i. This is our way to model trajectories. This implies that the number of intervals must be equal to the number of line segments. Recall that line segments in Oracle can be arcs as well.

Hence, to model trajectories we use lines and intervals. Thus, we define the

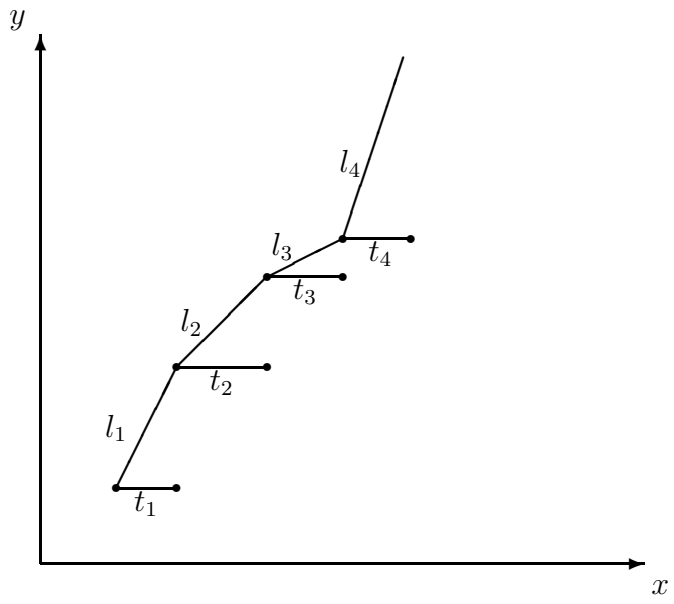


Figure 6.7: Trajectory of a car represented in OracleSpatial

classes touching_lines and meeting_intervals, which implements[1] the classes touch and meets, respectively and are subclasses of the classes OracleSpEC and TempEC respectively. Instances of these classes are (eventually) quantified conjunctives.
Instances of touching_lines are

$$touch(l_1, l_2) \wedge \cdots \wedge touch(l_{n-1}, l_n) \wedge \bigwedge_{i=1}^{n} line(l_i)$$

while instances of meeting_intervals are

$$meets(t_1, t_2) \wedge \cdots \wedge meets(t_{n-1}, t_n) \wedge \bigwedge_{i=1}^{n} time\_interval(t_i)$$

Particular methods of the above classes include $()_T|$, that implements projection on intervals. In the following we assume that $m = meet(m_1, m_2) \wedge meet(m_2, m_3) \wedge \cdots \wedge meet(m_{n-1}, m_n)$, $1 \leq i \leq j \leq n$ and $i \in A_{meet}$

$$()_T| : A_{meet} \times A_{meet} \longrightarrow A_{meet}$$

$$(i)_T|m = meet(m_i, m_{i+i}) \wedge \cdots \wedge meet(m_{j-1}, m_j) \wedge before(m_{i-1}, i) \wedge after(i, m_{j+1})$$

and $()_S|$, that implements projection on touching lines. Let' s assume that $l =$

[1] In this case "implements" has to be interpreted in the Java sense

$touch(l_1, l_2) \wedge touch(l_2, l_3) \wedge \cdots \wedge touch(l_{n-1}, l_n)$, $1 \leq i \leq j \leq n$ and $s \in A_{line}$, then

$$()_S| : A_{line} \times A_{touching\_lines} \longrightarrow A_{touching\_lines}$$

$$(s)_S|l = overlap(s, l) \wedge$$
$$(touch(l_i, l_{i+i}) \wedge \cdots \wedge touch(l_{j-1}, l_j) \wedge disjoint(m_{i-1}, i) \wedge disjoint(i, m_{j+1}))$$

The above functions return an approximation of the real projection, because the resulting object can be bigger than the real one, as in the example below. Figure 6.8 shows an example of projection on intervals. The interval resulting from the projection of i on m is given by the composition of the three intervals m_2, m_3, m_4, which overlap with i. Note that the projection $()_S|$ needs to check if the arguments overlap, while $()_T|$ don' t. This due to linear order of meeting intervals, for them it is enough to check that the interval i is contained in m just with the constraint $before(m_{i-1}, i) \wedge after(i, m_{j+1})$, while in the touching line the order, given by the Oracle storing method, described in Section 6.1.1, is not linear.

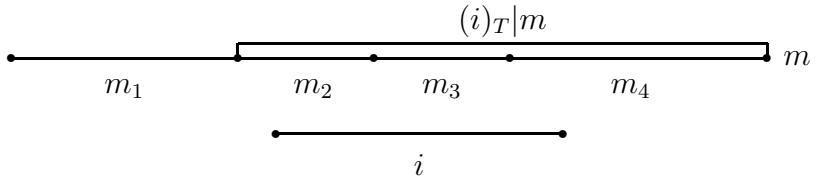


Figure 6.8: The operator $()_T|$

We are ready to model moving objects. Figure 6.9 shows a part of the classes schema: the class trajectory is composed of a spatial extent of type touching_lines and a timestamp of type meeting_intervals. Each moving object can then be modeled as a class referring to trajectory.

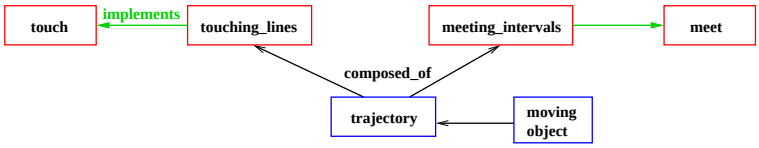


Figure 6.9: Extension of the schema to model trajectories

At this purpose, we need two important functions, to perform trajectory projections. Let' s suppose to have a trajectory of John, and we want to know where John was in a given interval i, we need a function that projects the interval i on the trajectory and returns the part of line, representing the route of John in the determinate

interval. Vice versa, we want to know the time interval, in which John has gone from work place to the bar, represented by the line s, for this we need a function, projecting s on the trajectory and returning the desired interval. The definition of these two functions uses the above projections $()_T|$ and $()_S|$. Let' s assume that $A_{traj} = \{((t_1, \ldots, t_n), (l_1, \ldots, l_n)) : (t_1, \ldots, t_n) \in A_{meet}, (l_1, \ldots, l_n) \in A_{touching\_lines}\}$ and $traj = ((t_1, \ldots, t_n), (l_1, \ldots, l_n))$, then we define the following two functions:

$$ProjOnSpace: A_{meet} \times A_{traj} \longrightarrow A_{touching\_lines}$$
$$ProjOnTime: A_{touching\_lines} \times A_{traj} \longrightarrow A_{meet}$$

$$ProjOnSpace(i, traj) = (l_i, \ldots, l_j),\ 1 \leq i \leq j \leq n \wedge (i)_T|(t_1, \ldots, t_n) = (t_i, \ldots, t_j)$$

$$ProjOnTime(s, traj) = (i_i, \ldots, i_j),\ 1 \leq i \leq j \leq n \wedge (s)_S|(l_1, \ldots, l_n) = (l_i, \ldots, l_j)$$

The above functions first find the indices i, j calling the projections $()_T|, ()_S|$ and the returned indices determine the lines or intervals of the result.

Furthermore, a new overlap operator has to be defined. Two trajectories overlap if and only if at the same time the spatial extents overlap. Let' s assume that $traj_1 = (i^1, l^1)$ and $traj_2 = (i^2, l^2)$, then

$$Soverlap: A_{traj} \times A_{traj} \longrightarrow A_{touching\_lines}$$
$$Toverlap: A_{traj} \times A_{traj} \longrightarrow A_{meet}$$

$$Soverlap(traj_1, traj_2) = overlap(ProjOnSpace(overlap(i^1, i^2)), ProjOnSpace(overlap(i^1, i^2)))$$

$$Toverlap(traj_1, traj_2) = overlap(ProjOnTime(overlap(l^1, l^2)), ProjOnTime(overlap(l^1, l^2)))$$

Below, there is the declaration of the class Trajectory. The class constructor checks if the number of intervals is equal to the number of line segments, which will create an instance of the class, in the case they are different, it returns error.

```
class Trajectory {
Touch_lines: space;
Meet_int: time;
Trajectory (Touch_lines s,Meet_int t) {
                                        if (#(s)==#(t)) space = s; time = t
                                        else Error }
Touch_lines ProjOnSpace(Meet_int i) {
                                        Meet_int I[i,j]= (i)_T| time;
                                        Touch_lines s = findPart(space, I[i,j]);
                                        return s };
Meet_int ProjOnTime(Touch_lines s) {
                                        Touch_lines S[i,j]= (s)_S| space;
                                        Meet_int i = findPart(time, S[i,j]);
                                        return i };
Touch_lines Soverlap( Traj tr1,tr2){
                                        Meet_int I= overlap(tr1.time, tr2.time);
                                        return overlap(tr1.ProjOnSpace(I), tr2.ProjOnSpace(I))};
Meet_int Toverlap( Traj tr1,tr2){
                                        Touch_Lines S= overlap(tr1.space, tr2.space);
                                        return overlap(tr1.ProjOnTime(S), tr2.ProjOnTime(S))};
} ;
```

The function findPart returns the segment of the multi-interval/multilinestring specified by the indices i, j.

In the following, we give some examples of queries. We model persons' trajectories and for this we define a class People, which refers to Trajectory, being people moving objects, and a class Places, which represents particular places whose attributes are name and a spatial extent, such as bar, terrace etc.

```
class   People {              class     Places {
        String: name;                   String: name;
        Trajectory: traj;     OracleSp  extent
};                            };
```

The first query shows the usage of $ProjOnSpace$ applied to a user-defined interval.
- Query1: Where is Martin between 12 and 14?

```
select   p.traj.ProjOnSpace([12,14])
into     Set(Touch_lines) result
from     people p
where    p.name = 'Martin'
```

The query returns the line, which Martin has covered in the given interval of time. The next query is more complicated, and implies the creation of a new spatial object. It shows the usage of the function $ProjOnTime$.

- Query2: When does Monica stay at the bar's terrace?

```
select   p.traj.ProjOnTime(atTheTerrace)
into     Set(Meet_int) result
define   Touch_lines s as new overlap(p.traj.space, terr.extent)
define   Touch_lines atTheTerrace as (s)_S|p.traj.space
from     people p, places terr
where    p.name = 'Monica'
and      terr.name= 'terrace'
```

The result of the query is a time interval (or a sequence of time intervals), in which Monica has been at the bar' s terrace. The resulting time interval corresponds to the line segment, resulting from the intersection of her trajectory with the spatial extent of the bar' s terrace.

The next query is similar to the above one, only a bit more complicated, as it needs some more computations.

- Query3: Where is John while Monica is at the bar's terrace?

```
select   j.traj.ProjOnSpace(time)
into     Set(Touch_lines) result
define   Meet_int time as ProjOnTime(atTheTerrace)
define   Touch_lines atTheTerrace as (s)_S|m.traj.space
define   Touch_lines s as new overlap(m.traj.space, terr.extent)
from     people j,m, places terr
where    j.name = 'John'
and      m.name= 'Monica'
and      terr.name= 'terrace'
```

The next queries show the usage of $Soverlap$ and $Toverlap$, to perform the intersection of two trajectories. In Query4 we are interested to know where two persons have met, and then we use $Soverlap$, while in Query5 we want to know when the two given persons have met and therefore we use $Toverlap$.

- Query4: Where did John and Monica meet?

```
select   new Soverlap(j.traj, m.traj)
into     Set(Touch_lines) result
from     people j,m
where    j.name = 'John'
and      m.name= 'Monica'
```

The query returns the line segment that Monica and John have covered together in

the same interval of time.

- Query5: When did John and Monica meet?

```
select   new Toverlap(j.traj, m.traj)
into     Set(Meet_int) result
from     people j,m
where    j.name = 'John'
and      m.name= 'Monica'
```

For the next query we recall the class Activities defined in the previous section. Activities is a temporal class modeling persons' activities. Here we made some reasoning using the three classes, just defined. The usage of the functions makes the query easy to write.

- Query6: Who ate in the skiing area?

```
select   p.name
into     Set(String) result
define   eat as act.time
from     People p, Activities act, Places pl
where    act.action='eat'
and      pl.name='skiing_area'
and      p.name=act.name
and      satisfy(overlap(p.traj.ProjOnSpace(eat),pl.extent))
```

For further examples we refer the reader to [DB00b].

Chapter 7

Conclusions

In this thesis we have proposed a framework based on constraints where temporal and spatial information can be represented and handled, and, at the same time, different information sources can be combined and integrated by means of generic data types and operations.

In particular, we have introduced a formal background for a component-based programming tool, which enables to use constraints as basic data types in an object-oriented database management system. We have used algebraic structures, as monoids, semirings and cylindric algebras, as a backbone since they allow to formalize the notion of constraint objects, preserving all the most important ingredients of constraint programming. Such structures are particular suitable in defining collection types; the representation of constraints as algebraic structures inserts them in the mechanism of type definition, where each type is represented as a domain, an identity element and, a "merge" operation. Therefore, constraints in this framework are treated as normal collection types, like sets, lists etc. The usual operations of conjunction and disjunction are handled as algebraic operators, with a well-defined operational semantics. Queries are solved as homomorphisms between the algebraic structures.

Moreover, the employment of monoids and semirings enables the definition of a model suitable to represent different classes of constraints, in particular a class of "black box" constraints, i.e. constraints with an undefined vocabulary and interpretation structure, but with a well-defined syntax. This particular construction allows an easy mapping to "specific constraints", like linear constraints, and particular classes of predicates, as spatial and temporal predicates and, constitutes a layer where these different classes of constraints can coexist and be combined.

The model gives the possibility for the development of a toolkit, which, given an object-oriented/relational database schema, enables its light-weight wrapping as a set of corresponding constraints. The "light-weight wrapping" means that the interpretation of the corresponding constraint-based data manipulation statements must be done directly by the underlying database management system.

We have shown how the framework can be used to represent different data models

in a uniform way. In particular, we have mapped the constraints of our model to linear constraints and to the spatial predicates of OracleSpatial; furthermore, we have defined temporal objects (time points and intervals) as special OracleSpatial objects and Allen' s temporal predicates, using OracleSpatial predicates applied to time intervals and, then we have mapped the black box constraints to them. The process of mapping has been made defining a vocabulary and an interpretation structure, corresponding to the class of constraints that we wanted to represent. We have proved that all the defined mappings and the corresponding interpretation models are complete.

Therefore, we mapped black box constraints to linear constraints, defining the vocabulary $\Omega = \langle <, +1, 0 \rangle$, and the interpretation structure $\mathcal{M} = \langle \mathbf{R}, \Omega \rangle$; the mapping on the OracleSpatial model has been made defining a vocabulary whose predicates correspond to the OracleSpatial relationships and the domain of interpretation has been defined as the set of spatial objects, supported by that model. The specific constraints are then solved by means of queries to the underlying model, that in the case of OracleSpatial allows to exploit its efficient features of spatial indexing and spatial join. A particular contribution of having defined spatial constraints as sub-classes of monoids and semirings is the facility to model normal and nested aggregation queries.

We have presented an example of integration between the linear constraint database and OracleSpatial, by means of type conversion functions, which transform an object represented by linear constraints to an OracleSpatial object and, vice versa. Such a conversion allows, for instance, to compute the distance between two spatial object and, therefore to overcome the lack of linear algebras in expressing metric relationships.

Finally, a second example has been presented, that shows the integration between OracleSpatial and the temporal model. This example shows the feature of our model to represent spatio-temporal information. In particular we have modeled trajectories and moving objects, extending the features of OracleSpatial, which so far handles exclusively with spatial data.

We believe that the features of our model fulfill the requirements, described in the introduction.

Ongoing and future work concerns the implementation of the model. We believe that a suitable Java-based implementation of the algebraic structures can be done as a collection of components to be used as abstract super class of all constraints in the system. This would allow to guide an implementation of particular constraints just exploiting the type checking facilities of the standard Java compiler. In other words, our ultimate goal is a component-based programming environment for developing different constraint databases. Basic elements of developing a particular constraint system starting from underlying objects has been illustrated using spatial and temporal objects.

Bibliography

[AHV95] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[All83] J.F. Allen. Maintaining knowledge about temporal intervals. In *Communication of the ACM*, volume 26 (11), pages 832–843. November 1983.

[All84] J.F. Allen. Towards a general theory of action and time. *Artificial Intelligence*, 23:123–154, 1984.

[AR99] T. Abraham and J.F. Roddick. Survey of spatio-temporal databases. *GeoInformatica*, 3(1):61–99, 1999.

[BBC97a] A. Belussi, E. Bertino, and B. Catania. Manipulating spatial data in constraint databases. In *Proc. of Advances in Spatial Databases*, volume 1262 of *Lecture Notes in Computer Science*, pages 115–141. Springer Verlag, 1997.

[BBC97b] A. Belussi, E. Bertino, and B. Catania. Manipulating Spatial Data in Constraint Databases. In *Advances in Spatial Databases, fifth International Symposium*, volume 1262 of *Lecture Notes in Computer Science*, pages 115–141, 1997.

[BBC98] A. Belussi, E. Bertino, and B. Catania. An extended algebra for constraint databases. *IEEE Transactions on Knowledge and Data Engineering*, 10(5):686–705, September/October 1998.

[BFG96] E. Bertino, E. Ferrari, and G. Guerrini. A formal temporal object-oriented data model. In *EDBT*, pages 342–356, 1996.

[BJS99] M.H. Böhlen, C.S. Jensen, and M.O. Scholl. *Spatio-Temporal Database Management*, volume 1678 of *Lecture Notes in Computer Science*. Springer Verlag, 1999.

[BK95] A. Brodsky and Y. Kornatzky. The l_{yric} language: Querying constraint objects. In *Int. Conf. on Management of Data*, ACM SIGMOD, 1995.

[BL97] G. Birkhoff and S.Mac Lane. *A Survey on Modern Algebra*. AKPeters, Ltd., 1997.

[BR95] J.-H. Byon and P. Revesz. Disco: a constraint database with sets. In *CONTESSA Workshop on Constraint Databases and Applications*, 1995.

[BSCE97] A. Brodsky, V.E. Segal, J. Chen, and P.A. Exarkhopoulo. The CCUBE constraint object-oriented database. *Constraint: An International Journal*, 2:245–277, 1997.

[BSCE99] A. Brodsky, V.E. Segal, J. Chen, and P.A. Exarkhopoulo. The CCUBE constraint object-oriented database. In *Proc. ACM SIGMOD Conf. on Management of Data*, pages 577–579, Philadelphia PA, 1999.

[Cho94] J. Chomicki. Temporal Query Languages: A Survey. In *Temporal Logic: Proceedings of the First International Conference, ICTL'94*, volume 827 of *Lecture Notes in Artificial Intelligence*, pages 506–534. Springer Verlag, 1994.

[CK90a] Chang and Keisler. *Model Theory*. 3. Ed., Amsterdam, Holland, 1990.

[CK90b] C.C. Chang and H. Jerome Keisler. *Model Theory*, volume 73 of *Studies in Logic and the Foundation of Mathematics*. Elsevier Science B.V., North Holland, 1990.

[CL73] C.-L Chang and R. Char-Tung Lee. *Symbolic Logic and Mechanical Theorem Proving*. Computer Science and Applied Mathematics. Academic Press, New York, 1973.

[CLR99] J. Chomicki, Y. Liu, and P.Z. Revesz. Animating Spatiotemporal Constraint Databases. In *Spatio-Temporal Database Management*, volume 1678 of *Lecture Notes in Computer Science*, pages 224–241. Springer Verlag, 1999.

[CR97] J. Chomicki and P. Z. Revesz. Constraint-Based Interoperability of Spatiotemporal Databases. In *Advances in Spatial Databases, fifth International Symposium*, volume 1262 of *Lecture Notes in Computer Science*, pages 142–161, 1997.

[CR99] J. Chomicki and P.Z. Revesz. Constraint-based interoperability of spatiotemporal database. *Geoinformatica*, 3(3), September 1999.

[CT85] J. Clifford and A.U. Tansel. On an algebra for historical relational databases: Two views. In S. Navathe, editor, *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 247–265, 1985.

[DB99] A. Di Deo and D. Boulanger. Using objects to build constraint databases. In *International Workshop on Logic Programming (WLP99)*, 1999.

[DB00a] A. Di Deo and D. Boulanger. A formal background to build constraint objects. In B.C. Desai, Y. Kiyoki, and M. Toyama, editors, *International Database Engineering and Application Symposium (IDEAS00)*, IEEE, pages 7–15, Yokohama, Japan, September 2000.

[DB00b] A. Di Deo and D. Boulanger. Using generic constraints to represent spatiotemporal data. In *accepted by GIScience 2000*, Savannah, Georgia, 2000.

[dBar] J. Van den Bussche. *Constraint Databases*, chapter 1. G. Kuper, L. Libkin, and J. Paredaens, To appear.

[DMP91] R. Dechter, I. Meiri, and J. Pearl. Temporal constraint networks. *Artificial Intelligence*, 49:61–95, 1991.

[ea94a] R. Snodgrass et al. Tsql2 language specification. *SIGMOD Record*, 23(1):65–86, March 1994.

[ea94b] R. Snodgrass et al. A tsql2 tutorial. *SIGMOD Record*, 23(3):27–34, September 1994.

[EF91] Max J. Egenhofer and Robert D. Franzosa. Point-set topological spatial relations. In *International Journal of Geographical Information Systems*, volume 5, pages 161–174. 1991.

[Ege91] M. J. Egenhofer. Reasoning about binary topological relations. *Lecture Notes in Computer Science*, 525:143–160, 1991.

[Ege95] M. Egenhofer. User interfaces. In R. Laurini T. Nyerges, D. Mark and M. Egenhofer, editors, *Cognitive Aspects of Human-Computer Interaction for Geographical Information Systems*, pages 1–8. Kluwer Academic Publishers, Dortrecht, The Netherlands, 1995.

[EGSV99] M. Erwig, R. H. Güting, M. Schneider, and M. Vazirgiannis. Spatiotemporal Data Types : An Approach to Modeling and Querying Moving Objects in Databases. *Geoinformatica*, 3(3), 1999. To appear.

[EH95] M.J. Egenhofer and J. Herring, editors. *Proc. Intl. Symp. on Large Spatial Databases (SSD)*, volume 951 of *Lecture Notes in Computer Science*. Springer Verlag, 1995.

[Eil74] S. Eilenberg. Automata, languages, and machines. In *Academic Press*, volume A. 1974.

[EJS98] O. Etzion, S. Jajodia, and S. Sripada, editors. *Temporal Databases: Research and Practice*, volume 1399 of *Lecture Notes in Computer Science*. Springer, 1998.

[Eme90] E.A. Emerson. Temporal and modal logic. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 16, pages 995–1072. Elsevier/MIT Press, 1990.

[ES99] M. Erwig and M. Schneider. The honeycomb model of spatio-temporal partitions. In *Workshop on Spatio-Temporal Database Management*, volume 1678 of *Lecture Notes in Computer Science*, pages 39–59, 1999.

[FM95] L. Fegaras and D. Maier. Towards an effective calculus for object query processing. In *Proc. ACM SIGMOD Conf. on Management of Data*, pages 47–58, 1995.

[Gad88] S.K. Gadia. A homogeneous relational model and query languages for temporal databases. *ACM Trans. on Database Systems*, 13(4):418–448, 1988.

[GBE+98] R.M. Güting, M.H. Böhlen, M. Erwig, C.S. Jensen, N. Lorentzos, M. Schneider, and M. Varzigiannis. A foundation for representing and querying moving objects. Technical report, Technical Report 238, FernUniversität Hagen, Informatik, september 1998.

[GE99] R.K. Goyal and M.J. Egenhofer. Cardinal directions between extended spatial objects. *IEEE Transactions on Knowledge and Data Engineering*, 1999. in press.

[GK96] D.Q. Goldin and P.C. Kanellakis. Constraint query algebras. *Constraint Journal, 1st issue*, pages 45–83, 1996.

[GM91] D.M. Gabbay and P. McBrien. Temporal Logic & Historical Databases. In *Proceedings of the Seventeenth International Conference on Very Large Databases*, pages 423–430, September 1991.

[GNRR93] R.L. Grossman, A. Nerode, A.P. Ravn, and H. Rischel, editors. *Hybrid Systems*. LNCS 736. Springer-Verlag, 1993.

[GRS98a] S. Grumbach, P. Rigaux, and L. Segoufin. The DEDALE system for complex spatial queries. In *Intl. Conf. on the Management of Data*, SIGMOD, 1998.

[GRS98b] S. Grumbach, P. Rigaux, and L. Segoufin. Spatio-temporal data handling with constraints. In Robert Laurini, Kia Makki, and Niki Pissinou, editors, *Proceedings of the 6th International Symposium on Advances in*

Geographic Information Systems (GIS-98), pages 106–111. ACM Press, 1998.

[GRSS97] S. Grumbach, P. Rigaux, M. Scholl, and L. Segoufin. DEDALE, a spatial constraint database. In *Proc. Int. Workshop on Database Programming Langages (DBPL'97)*, East Park, Colorado, USA, 1997.

[Gru97] S. Grumbach. Modeling and querying spatio-temporal data. In *Sistemi Evoluti per Basi di Dati*, pages 11–28, 1997.

[GS93] R. Güting and M. Schneider. Realms: A foundation for spatial data types in database systems. In D. Abel and B. C. Ooi, editors, *Third International Symposium on Large Spatial Databases, SSD '93*, volume 692 of *Lecture Notes in Computer Science*, pages 14–35. Springer, 1993.

[GS95] S. Grumbach and J. Su. Dense-order constraint databases. In *Proc. 14th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, pages 66–77, 1995.

[GSG94] R. Giacobazzi, S.K.Debray, and G.Levi. Generalized semantics and abstract interpretation for constraint logic programs. *Journal of Logic Programming*, 1994.

[GST94] S. Grumbach, J. Su, and C. Tollu. Linear constraint query languages: Expressive power and complexity. In D. Leivant, editor, *Logic abd Computational Complexity*, Indianapolis, 1994. Springer Verlag. LNCS 960.

[Güt94] R.H. Güting. An introduction to spatial database systems. *VLDB Journal*, 3(4):357–400, October 1994.

[HJA71] L. Henkin, J.D.Monk, and A.Tarski. Cylindric algebras. part i and ii. North-Holland, Amsterdam, 1971.

[IC98] A. Isli and A.G. Cohn. An Algebra for Cyclic Ordering of 2D Orientations. In *Proceedings of AAAI-98*, pages 643–649, 1998.

[Jav] http://java.sun.com/docs/books/tutorial/java/interpack/interfacedef.html.

[JJL87] J. Jaffar and J.-L.Lassez. Constraint logic programming. In *Proc. 14th Annual ACM Symp. on Principles of programming Languages*, ACM, pages 111–119, 1987.

[KG94] P.C. Kanellakis and D.Q. Goldin. Constraint programming and database query languages. In *Symposium on Theoretical Aspects of Computer Software*, volume 789 of *LCNS*, pages 96–120, Sendai Japan, April 1994.

[KKR90] P.C. Kanellakis, G.M. Kuper, and P.Z. Revesz. Constraint query languages. In *Proc. 9th ACM PODS*, pages 299–313, 1990.

[KKR95] P.C. Kanellakis, G.M. Kuper, and P.Z. Revesz. Constraint query languages. *Journal of Computer and System Sciences*, 51(1):26–52, 1995.

[KKS92] M. Kifer, , W. Kim, and Y. Sagiv. Querying object-oriented databases. In *Int. Conf. on Management of Data*, ACM SIGMOD, pages 393–402, 1992.

[KL91] H. Kautz and P. Ladkin. Integrating metric and qualitative temporal reasoning. In *Proc. of the 10th AAAI*, Anaheim, 1991.

[Kou94] M. Koubarakis. Foundations of indefinite constraint databases. In Alan Borning, editor, *Principles and Practice of Constraint Programming, 2nd International Workshop, PPCP94*, volume 874 of *Lecture Notes in Computer Science*, pages 266–280, Rosario, Orcas Island, Washington, USA, May 1994. Springer-Verlag.

[KPV95] B. Kuijpers, J. Paredaens, and J. Van den Bussche. Lossless representation of topological spatial data. In M. J. Egenhofer and J. R. Herring, editors, *Advances in Spatial Databases*, volume 951 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 1995.

[KPV98] B. Kuijpers, J. Paredaens, and L. Vandeurzen. Semantics in spatial databases. In B. Thalheim L. Libkin, editor, *Semantics in Databases*, volume 1358 of *Lecture Notes in Computer Science*, pages 114–135. Springer, 1998.

[LT92] Robert Laurini and Derek Thompson. *Fundamentals of Spatial Information Systems*. Number 37. Academic Press, New York, 1992.

[Mei91] I. Meiri. Combining qualitative and quantitative constraints in temporal reasoning. In *Proc of the 9th AAAI-91*, Anheim, CA, 1991.

[Mor89] S. Morehouse. The architecture of ARC/INFO. In *Proc. Intl. Symp. on Computer Assisted Cartography (Auto-Carto 9)*, pages 266–277, 1989.

[MP93] A. Montanari and B. Pernici. *Temporal Databases: Theory, Design and Implementation*, chapter Temporal Reasoning, pages 534–562. Benjamin/Cummings, 1993.

[OM94] M. A. Orgun and W. Ma. An Overview of Temporal and Modal Logic Programming. In *Lecture Notes in Artificial Intelligence*, volume 827, pages 445–479, 1994.

[Ora] http://technet.oracle.com.

[Org96] M. A. Orgun. On temporal deductive databases. *Computational Intelligence*, 12(2):235–259, May 1996.

[Par95a] J. Paradaens. Spatial databases, the final frontier. In *Proc. of the fifth International Conference on Database Theory*, volume 893 of *Lecture Notes in Computer Science*, pages 14–32. Springer Verlag, 1995.

[Par95b] Jan Paredaens. Spatial databases, the final frontier. In Georg Gottlob and Moshe Y. Vardi, editors, *Database Theory—ICDT'95, 5th International Conference*, volume 893 of *Lecture Notes in Computer Science*, pages 14–32. Springer, 1995.

[PdBG94] J. Paradaens, J. Van den Bussche, and D. Van Gucht. Towards a theory of spatial database queries. In *Proc. 13th ACM Symp. on Principles of Database Systems*, pages 279–288, 1994.

[PK98] J. Paredaens and B. Kuijpers. Data models and query languages for spatial databases. *Data & Knowledge Engineering*, 25:29–53, 1998.

[PSV96] C. H. Papadimitriou, D. Suciu, and V. Vianu. Topological queries in spatial databases. In ACM, editor, *Proceedings of the Fifteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS 1996, Montréal, Canada, June 3–5, 1996*, volume 15, pages 81–92. ACM Press, 1996.

[PVV94] J. Paredaens, J. Van den Bussche, and D. Van Gucht. Towards a theory of spatial database queries. In ACM, editor, *Symposium on Principles of database systems*, volume 13, pages 279–288, New York, NY 10036, USA, 1994. ACM Press.

[SA86] R. Snodgrass and I. Ahn. Temporal databases. *IEEE Computer*, 19(9), 1986.

[Sam90] Hanan Samet. *The design and analysis of spatial data structures*. 1990.

[Sno87] R. Snodgrass. The temporal query language tquel. *ACM Transactions on Database Systems*, 12(2):247–298, July 1987.

[Sno92] R.T. Snodgrass. Temporal databases. In *Proc. of the International Conference on GIS - from Space to Territory: Theories and Methods of Spatio-Temporal Reasoning in Geographic Space*, volume 639 of *Lecture Notes in Computer Science*, pages 22–64. Springer Verlag, 1992.

[Sno95a] R. Snodgrass. *Modern Database Design: The object Model, Interoperability and Beyond*, chapter Temporal Object-Oriented Databases: A Critical Comparison. Addison-Wesley/ACM Press, 1995.

[Sno95b] R.T. Snodgrass. *The TSQL2 Temporal Query Language*. Kluwer Academic Publishers, 1995.

[TCG+93] A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass editors. *Temporal Databases: Theory, Design, and Implementation.* Benjamin/Cummings, 1993.

[VGV95] L. Vandeurzen, M. Gyssens, and D. Van Gucht. On the desirability and limitations of linear spatial database models. *Lecture Notes in Computer Science*, 951:14–28, 1995.

[Wor92] Michael F. Worboys. A generic model for planar geographical objects. In *International Journal of Geographical Information Systems*, volume 6, pages 353–372. 1992.

[Wor94] M. F. Worboys. A unified model for spatial and temporal information. *The Computer Journal*, 37(1):26–34, 1994.

[Wor95] M.F. Worboys. *GIS - A Computing Perspective*. Taylor & Francis, 1995.

[ZF96] K. Zimmermann and C. Freksa. Qualitative Spatial Reasoning Using Orientation, Distance, and Path Knowledge. *Applied Intelligence*, 6:46–58, 1996.