Temporal reasoning in a logic programming language with modularity

Vitor Beires Nogueira

Orientador: Professor Doutor Salvador Pinto Abreu

Tese submetida à Universidade de Évora
para obtenção do grau de Doutor em Informática

Departamento de Informática
Universidade de Évora

Dezembro de 2008

Esta tese não inclui as críticas e sugestões feitas pelo júri

Temporal reasoning in a logic programming language with modularity

Vitor Beires Nogueira

Orientador: Professor Doutor Salvador Pinto Abreu

Departamento de Informática
Universidade de Évora

Dezembro de 2008

Esta tese não inclui as críticas e sugestões feitas pelo júri

To Miguel and Susana with love

Acknowledgments

First of all, I would like to thank my son Miguel and my wife Susana for their caring support throughout the years of this work. To my parents and sister also goes my endless gratitude.

I thank my supervisor, Prof. Salvador Pinto Abreu, without whom this work wouldn't have started, continued and most of all, finished. His participation on this thesis went far beyond the expected critical watching and directing. I am thankful for the many interesting discussions that we had, during which he guided me into how to do scientific research. I thank him also for the great working conditions and for carefully reviewing not only the contents but also the English, making this work not only understable but also readable.

I would also like to thank Prof. Gabriel Torcato David for his participation in this work. Besides providing me with great working conditions at the Faculdade de Engenharia of Universidade do Porto, he also supervised the beginning of this thesis.

Throughout this thesis, I also had the chance of working with members of Projet Contraintes at INRIA Rocquencourt and in particular with Prof. Daniel Diaz. Several important results came out of such cooperation. I acknowledge the INRIA/GRICES project "Extensions au Logiciel Libre GNU Prolog" for the financial support that made this collaboration possible.

I would like to acknowledge the Universidade de Évora and specifically the Departmento de Informática not only for the conditions given but also for the leave that allowed me to focus entirely on this work.

I also acknowledge the Centria (Centro de Inteligência Artificial) that by providing supervisor funds, allowed me to participate in summer schools, conferences, workshops, doctoral programs, etc.

Last, but definitely not least, I would like to dedicate this work to my friends, and in particular to Paulo Estudante who was there for me on many occasions.

Abstract

Current Organisational Information Systems (OIS) deal with more and more information that is time dependent. In this work we provide a framework to construct and maintain Temporal OIS. This framework builds upon a logical language called Temporal Contextual Logic Programming that deeply integrates modularity with temporal reasoning making the usage of a module time dependent. This language is an evolution of another one, also introduced in this thesis, that combines Contextual Logic Programming with Temporal Annotated Constraint Logic Programming where modularity and time are orthogonal features. Both languages are formally discussed and illustrated.

The main contributions of the work described in this thesis include:

- Optimisation of Contextual Logic Programming (CxLP) through abstract interpretation.
- Syntax and operational semantics for an independent combination of the temporal framework Temporal Annotated Constraint Logic Programming (TACLP) and CxLP. A compiler for this language is also provided.
- Language (syntax and semantics) that integrates in a innovative way modularity (CxLP) with temporal reasoning (TACLP). In this language the usage of a given module depends of the time of the context. An interpreter and a compiler for this language are described.
- Framework to construct and maintain Temporal Organisational Information Systems. It builds upon a revised specification of the language ISCO, adding temporal classes and temporal data manipulation. A compiler targeting the language presented in the previous item is also given.

Sumário

Actualmente os Sistemas de Informação Organizacionais (SIO) lidam cada vez mais com informação que tem dependências temporais. Neste trabalho concebemos um ambiente de trabalho para construir e manter SIO Temporais. Este ambiente assenta sobre um linguagem lógica denominada Temporal Contextual Logic Programming que integra modularidade com raciocínio temporal fazendo com que a utilização de um módulo dependa do tempo do contexto. Esta linguagem é a evolução de uma outra, também introduzida nesta tese, que combina Contextual Logic Programming com Temporal Annotated Constraint Logic Programming, na qual a modularidade e o tempo são características ortogonais. Ambas as linguagens são formalmente discutidas e exemplificadas.

As principais contribuições do trabalho descrito nesta tese incluem:

- Optimização de Contextual Logic Programming (CxLP) através de interpretação abstracta.
- Sintaxe e semântica operacional para uma linguagem que combina de um modo independente as linguagens Temporal Annotated Constraint Logic Programming (TACLP) e CxLP. É apresentado um compilador para esta linguagem.
- Linguagem (sintaxe e semântica) que integra de um modo inovador modularidade (CxLP) com raciocínio temporal (TACLP). Nesta linguagem a utilização de um dado módulo está dependente do tempo do contexto. É descrito um interpretador e um compilador para esta linguagem.
- Ambiente de trabalho para construir e fazer a manutenção de SIO Temporais. Assenta sobre uma especificação revista da linguagem ISCO, adicionando classes e manipulação de dados temporais. É fornecido um compilador em que a linguagem resultante é a descrita no item anterior.

Contents

List of Tables

List of Figures

List of Acronyms

ACL	Annotated Constraint Logic
BCDM	Bitemporal Conceptual Data Model
CLP	Constraint Logic Programming
CRUD	Create, Read, Update and Delete
CSP	Constraint Satisfaction Problem
CxLP	Contextual Logic Programming
EC	Event Calculus
IA	Interval Algebra
ISCO	Information System COnstruction language
ISTO	Information System Tools for Organisations
MuTACLP	Multi-theory Temporal Annotated Constraint Logic Programming
OIS	Organisational Information Systems
SC	Situation Calculus
STP	Simple Temporal Problems
TACLP	Temporal Annotated Constraint Logic Programming
TCSP	Temporal Constraint Satisfaction Problem
TCxLP	Temporal Contextual Logic Programming

Preface

This work is the synthesis of several years[1] of research. One of the most important lessons learned during that time was that research is by no means a straight line, one might even say that it is an (acyclic) graph. Not only did we follow paths that turned out to be dead-ends but also sometimes we had to abandon others which, although promising, would cause us to stray from our goals.

One must say, that due to the (at least) lack of excellency of the author in the English language, writing this thesis in such language was a bold act. We ask for the reader sympathy on this subject.

Structure of the Work

Chapter 1 briefly introduces the concepts of time and modularity together with a logic programming perspective of Organisational Information Systems (OIS). This chapter also motivates for the integration of temporal reasoning with modularity in order to obtain a language to construct and maintain OIS.

For self containment reasons, we survey several approaches to temporal reasoning in Artificial Intelligence, temporal databases and modularity in logic programming in Chap(s). 2, 3 and 4, respectively.

Chapter 5, besides describing the modular logic language that is at core of this work called Contextual Logic Programming (CxLP), also presents a revised specification together with an optimisation framework obtained with abstract interpretation.

In Chap. 6 we combine consolidated approaches for temporal reasoning (in this case Temporal Annotated Constraint Logic Programming) and modularity (CxLP) into a single language. Although such a combination provides an expressive language, in Chap. 7 we propose a model where the usage of a module is influenced by temporal conditions. Moreover, the CxLP program structure is very suitable for integrating with temporal reasoning since it is quite straightforward to add the notion of *time of the*

[1] The use of an indefinite adjective was (unfortunately) on purpose.

context and let that time help in deciding whether a certain module is eligible or not to solve a goal.

In Chap. 8 we propose to augment the ISCO framework for constructing OIS with an expressive means of representing and implicitly using temporal information. This evolution builds upon the frameworks proposed in the previous chapters. Having simplicity as a design goal, we present a revised and stripped-down version of ISCO, keeping just some of the core features that we consider crucial in a language for the development of Temporal OIS.

We draw some concluding remarks and provide pointers for future work in Chap. 9.

In Appendix A we present a tutorial and a reference manual for the contextual part of GNU Prolog/CX. Finally, in Appendix B we briefly overview Constraint Logic Programming.

Part of this work has appeared before in joint publications with Prof. Salvador Abreu (my supervisor), Prof. Gabriel David and Prof. Daniel Diaz. I thank all of them for letting me use the following common work [NAD03, NAD04, ADN04, AN05, NA06b, AN06, ET06, NA06a, NA07d, NA07b, NA07a, NA07c].

Roadmap

There can be various reading paths for this thesis, in Table 1 we outline some of them.

Subject	Chapter
Survey on temporal reasoning	1 - 2 - 3
Survey on modularity	1 - 4 - 5 (optional)
Combining temporal reasoning and modularity	1 - 2.3 - 6 - 7
Temporal Information Systems	1 - 2.3 - 7 - 8
Brave reader	1 to 9

Table 1: Reading paths

Chapter 1

Introduction and Motivation

In this chapter we start with a brief overview of the concepts that are at the core of this work: time and modularity. We also provide a short description of Organisational Information Systems from a logic programming point of view. Subsequently we argue for the need to integrate modularity with temporal reasoning in order to provide a high level language in which to construct and maintain Organisational Information Systems.

1.1 Introduction

1.1.1 Time

Time is an elusive concept, studied across such diverse disciplines as physics, mathematics, linguistics, philosophy, etc. Each one provides an evolving perspective of Time: in physics, Newton defined Time as a dimension in which events occur in sequence whereas Einstein, in the special theory of relativity, stated "time intervals appear lengthened for events associated with objects in motion relative to an inertial observer" [Wik08].

In the last decades a great number of logic languages that deal with Time have been proposed. According to Van Benthem [Joh91], logic can be considered as a bridge between linguistics and mathematics since logic takes into consideration both the aspects of language and ontology. This author also points out that logics multiplicity of languages, theories of inference and formal semantics are adequate for the diverse intuitions inherent to the study of Time.

In a broad sense we can define a *temporal database* as a repository of temporal information [Cho94] therefore one can say that most database applications are supported by temporal databases. SQL [fS03] is often the language chosen for interacting with such databases. Although SQL is a very powerful language for writing queries, modifications

or constraints in the current state, it does not provide adequate support for the temporal counterparts: for instance the temporal equivalent to an ordinary query can be a very challenging task to express in SQL. To overcome such difficulties, several temporal data models, algebras and languages have been proposed. Moreover, McKenzie and Snodgrass [LEMS91] demonstrated that the design space for temporal algebras has, in some sense, been explored in that all combinations of basic design decisions have at least one representative algebra.

The logical and the database approach to Time are closely related: according to [BMRT02] temporal logic languages are responsible not only for the temporal databases theoretical foundations but also for their powerful knowledge representation and query languages.

Temporal reasoning plays an important role in many areas of Artificial Intelligence such as Natural Language, Planning, Agent-Based Systems, etc. Most approaches are restricted to the propositional case and follow an interval-based view originated in Allen's Interval Algebra [All83]. Moreover, temporal reasoning in AI is concerned with using additional assumptions (such as persistence or defaults) or special procedures (like persistence) to derive conclusions [Cho94].

1.1.2 Modularity

Module systems are an essential feature of programming languages, namely because besides structuring programs they also allow the development of general purpose libraries.

A modular extension to logic programming has been the object of research over the last decades. In a broad sense one can distinguish three different approaches to modularity: the algebraic, the logical and the syntactic. The algebraic approach started with work by O'Keefe [O'K85] and considers logic programs as elements of an algebra, whose operators are the operators for composing programs. The logical approach is based on work by Miller [Mil86, Mil89a], and extends the Horn language with logic connectives for building and composing modules. Finally, the syntactic approach (see [HF06] for a recent overview and a proposal of such approach) addresses the issue of the global and flat name space, dealing with the alphabet of symbols as a mean to partition large programs.

Contextual Logic Programming is modular extension of Horn clause logic proposed by Monteiro and Porto [MP89, MP93]. The CxLP *extension goal* can be regarded as a non-monotonic version of Miller *implication goal* [Mil86, Mil89a]. The *extension goal* is denoted by $\gg$ and $D \gg G$ (D is a set of clauses and G a goal) is derivable from a program P if G is derivable from $A \cup D$ and A is derivable from P, for some

finite set A of atoms for predicates not defined in D. Therefore $\gg$ provides a sort of lexical scoping for predicates: predicates in G which are defined in D are bound to such definitions, the others can be obtained from program P. Besides lexical scoping, CxLP also accounts for contextual reasoning, that is widely used for several Artificial Intelligence tasks such as natural language processing, planning, temporal reasoning, etc. Work by [AD03a] presents a revised specification of CxLP together with a new implementation for it and also explains how this language can be viewed as a shift into the Object-Oriented Programming paradigm.

1.1.3 Organisational Information Systems

Organisational Information Systems (OIS) have a lot to benefit from Logic Programming (LP) characteristics such as the rapid prototyping ability, the relative simplicity of program development and maintenance, the declarative reading which facilitates both development and the understanding of existing code, the built-in solution-space search mechanism, the close semantic link with relational databases, just to name a few. In [Por03, ADN04] we find examples of LP languages that were used to develop and maintain information systems.

Information System COnstruction language (ISCO) [Abr01] is a state-of-the-art logical framework for constructing Organisational Information Systems. ISCO is an evolution of the previous language DL [Abr00] and is based on a Constraint Logic Programming (CLP) framework to define the schema, represent data, access heterogeneous data sources and perform arbitrary computations. In ISCO, processes and data are structured as *classes* which are represented as typed[1] Prolog predicates. An ISCO class may map to an external data source or sink, such as a table or view in a relational database, or be entirely implemented as a regular Prolog predicate. Operations pertaining to ISCO classes include a *query* which is similar to a Prolog call as well as three forms of *update*.

1.2 Temporal (Logic-Based) OIS

Chomicki and Toman [CT98] stated that current Information Systems deal with more and more complex applications where, besides the static aspects of the world, one also has to model the *dynamics*, i. e., time, change and concurrency. This statement illustrates very clearly the importance of Time in OIS. It is our opinion that a language for OIS must have the capability of performing temporal representation and reasoning beyond the *ad hoc* approaches.

[1]The type system applies to class members, which are viewed as Prolog predicate arguments.

The amount of information handled by OIS is enormous and has increased by orders of magnitude over the last decades. Besides the already mentioned benefits of modularity, we consider that this fact by itself should be sufficient to justify that modularity is a *sine qua non* condition for designing and implementing OIS.

One common approach in combining modularity and time in a language is to consider them as independent, or orthogonal, features. In this work we start by presenting a logical based language that follows such guidelines.

Nevertheless, time and modularity can combine into a more fruitful environment, i. e. one where these features are more intertwined. We propose an extension of a modular logic language where temporal reasoning is deeply integrated, making the usage of a module itself time dependent. We consider this proposition to be natural and in fact largely employed in several domains where a given law, criteria application is time dependent.

Finally, we will show that the language above provides a sound basis for a framework to construct and maintain *Temporal Organisational Information Systems.*

Chapter 2

Temporal Reasoning in Artificial Intelligence

This chapter provides an overview of temporal reasoning in Artificial Intelligence. It starts by introducing the notion of model of time and temporal qualification. Afterwards it describes several constraint and modal logic based proposals for temporal reasoning. Finally, some theories for reasoning about actions and change are discussed.

2.1 Introduction

Temporal reasoning plays an important role in many areas of Artificial Intelligence such as Natural Language, Planning, Agent-Based Systems, etc. Most approaches are restricted to the propositional case and follow an interval-based view originated in Allen's Interval Algebra [All83] (see Sect. 2.3.1 for a description of such an algebra). Moreover, temporal reasoning in AI is concerned with using additional assumptions (such as persistence or defaults) or special procedures (like persistence) to obtain conclusions [Cho94]. For further reading on this subject see for instance [Vil94, CM00, Aug01, FGV05].

This chapter, for self-containment reasons, presents a general survey of temporal reasoning in AI and is organised as follows: Sect. 2.2 starts out by specifying several key concepts pertaining to the foundations of temporal reasoning such as *temporal ontology, topology* and *qualification.* Afterwards, we present three subfields of temporal representation and reasoning in AI: constraint-based temporal reasoning (Sect. 2.3), temporal modal logic (Sect. 2.4) and reasoning about actions and change (Sect. 2.5). The chapter ends with a brief summary of the concepts and approaches described therein.

2.2 General Notions of Time

In this section we briefly describe the notions of model of time and temporal qualification.

2.2.1 Model of Time

To define a model of time we need to define not only the *time ontology* but also the *time topology*. By *time ontology* we mean the class or classes of objects time is made of, i.e. one has to choose between *time points* and *intervals* as the basic temporal entities.

Regarding the *time topology* we can consider different structures for time, namely whether it is:

- discrete, dense or continuous;
- bounded, partially bounded or unbounded;
- linear, branching, circular or with a different structure.

Moreover, it is helpful to know what kind of properties the structure of temporal individuals have as a whole, i.e.:

- are all individuals comparable by the order relation (connectedness)?
- are all individuals equal (homogeneity)?
- is it the same to look at one side or at the other (symmetry)?

2.2.2 Temporal Qualification

Temporal qualification refers "to the way logic is used to express that temporal propositions are true or false at different times" [RV05] and is by itself a very prolific field of research.

Besides modal logic proposals, from a first–order point of view we can consider the following methods for temporal qualification: temporal arguments, token arguments, temporal reification and temporal token reification. For a brief description of the different proposals please consider Fig. 2.1 taken from [RV05]. Temporal reification assigns a special status to time and allows quantification over propositions. The main criticism made to reification is that such an approach requires a *sort structure*

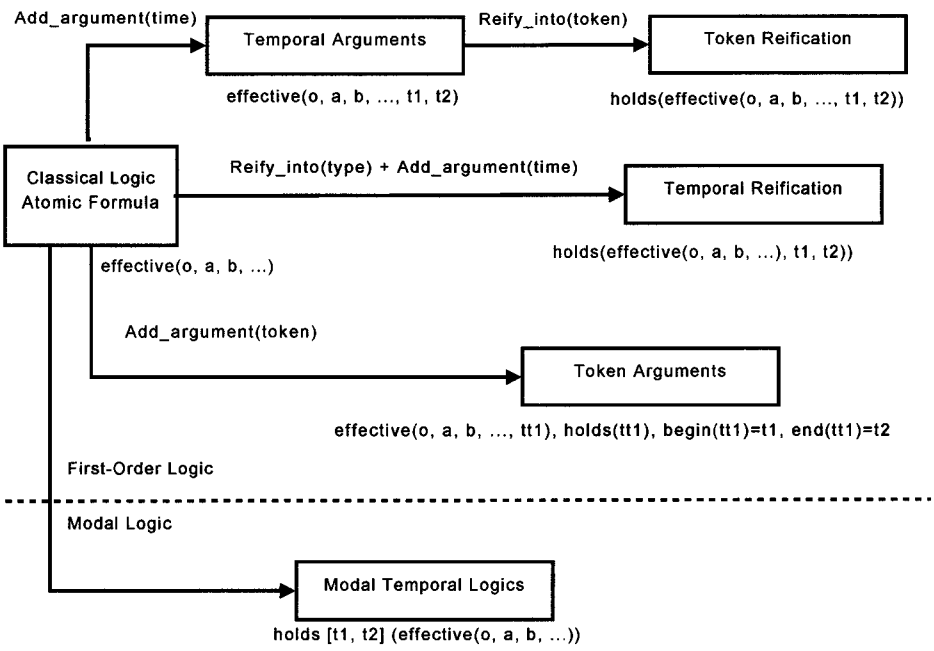

Figure 2.1: Temporal qualification methods

to distinguish betewen terms that denote real objects of the domain (terms of the original object language) and terms that denote propositional objects (propositions of the original object language).

2.3 Constraint-Based Temporal Reasoning

According to [Pao] constraint–based approaches mainly focus on the definition of a representation formalism and of reasoning techniques to deal specifically with temporal constraints between temporal entities, independently of the events and states associated with them. Following these guidelines, a class of Constraint Satisfaction Problem (CSP) was defined, called Temporal Constraint Satisfaction Problem (TCSP), where variables represent time and constraints represent temporal relations between them. Different TCSPs are defined depending on the time entity that variables can represent, namely time points, time intervals, durations (i.e. distances between time points) and the class of constraints, namely qualitative, metric or both [SV98]. Each class of constraints is characterised by the underlying set of *basic temporal relations.*[1]

TCSP constraints are binary and $C_{ij} = \{r_1, \ldots, r_k\}$ is a constraint[2] between the variables X_i and X_j where each r_i is a *basic temporal relation.* A *singleton labelling*

[1]The elements of all basic temporal relations are mutually exclusive and their union is the universal constraint.

[2]The set is interpreted as a disjunction of relations.

assigns an r_i to the pair X_i, X_j, and the *solutions* of a TCSP are its consistent singleton labelling.

The main techniques to find a solution to the general problem are path-consistency and backtracking algorithms. Finally, there are typically two tasks when working with TCSP: deciding consistency and answering queries about scenarios that satisfy all constraints.

For a comprehensive survey on this subject see for instance [SKD94, Kou95, SV96, Sch98, SV98]

2.3.1 Qualitative Temporal Constraints

Qualitative temporal constraints deal with the relative position between time points or intervals.

Allen's Interval Algebra

The Interval Algebra (IA) was proposed by James F. Allen [All83]. In this work, domain elements are temporal intervals[3] and constraints are built using the thirteen basic relations between intervals: *before, after, meets, met by, equal, overlaps, overlapped by, during, contained by, starts, started by, finishes, finished by.* The operations on those relations include composition and Boolean operators.

Intervals are used to qualify *properties, events* and *processes.* We say that a *property* holds over an interval if and only if it holds for all its subintervals whereas a *proccess* occurs over an interval if it occurs in at least one of its subintervals. Events occurs over the smallest possible interval [Rib93].

In [VKvB90] it is shown that the satisfiability of Allen's algebra is NP-complete. The study of maximal tractable subclasses started with Nebel and Bürckert (ORD-Horn algebra [NB94]) and recently Krokhin et. al. [KJJ03] showed that the IA contains exactly eighteen maximal tractable subalgebras. Moreover, they also proved that reasoning in any fragment not entirely contained in one of the these subalgebras is NP-complete.

[3]Besides intervals, the only other structural property defined is that time is linear. Everything else is set by the user according to the application.

Vilain and Kaut'z Point Algebra

The Point Algebra was introduced by Vilain and Kautz [VK86]. In their proposal, the domain elements are the temporal points and define the three basic relations that can hold between temporal points, i.e., *before, equal* and *after* $(<, =, >)$. Moreover, the Point Algebra defines two operations between these point-point relations: *composition* and *intersection.*

Regarding complexity, the full point algebra is tractable [VK86, VKvB90, Hir97].

Interval-Point Algebra

The Interval-Point algebra was proposed by Vilain in [Vil82]. In this algebra variables stand for time points or intervals and the only type of relations are between between a point and an interval. Moreover, since in an interval $[a_1, a_2]$ we have $a_1 < a_2$, there are only five possible relations: `before`, `starts`, `during`, `finishes` and `after`.

The complexity of deciding satisfiability in this algebra is NP-complete [Mei96].

Qualitative Algebra

Meiri [Mei96, Mei91] proposes a combination of the all the preceding algebras: Allen's Interval Algebra, the point algebra and the interval-point algebra. As expected, this proposal handles both time points and time intervals and can relate points with points, points with intervals and intervals with intervals.

In [JK04] the authors identify all tractable fragments of this algebra and also prove that all other fragments are NP-complete.

2.3.2 Metric Point Constraints

The metric point constraints proposal is based on the notion of time points and the distance between them, i.e. variables represent time points and the allowed temporal relations is the set of intervals of *time-structure.* A constraint has the form $C_{ij} = \{[a_1, b_1], \ldots, [a_k, b_k]\}$ and stands for[4]:

$$(a_1 \leq X_j - X_i \leq b_1) \vee \ldots \vee (a_k \leq X_j - X_i \leq b_k)$$

[4]The representation was taken from [SV98], replacing the conjunction by a disjunction that was probably a typo.

There are three operators for the metric constraints: *inverse, intersection* and *composition.* With respect to complexity we have that deciding consistency and computing a solution of a metric point constraint problem is NP-complete [DMP91]. Finally, [SV98] describes the three known relation based tractable classes, i.e. Simple Temporal Problems (STP), STP with inequation constraints (for continuous domains only) and Star TCSPs.

2.3.3 Hybrid Approaches

Meiri [Mei96] proposed a very expressive constraint language in which both qualitative and quantitative/metric constraints can be represented, this way allowing the representation and processing of most types of constraints. This proposal combines the *Qualitative Algebra* (Sect. 2.3.1) with *Metric Point Constraints* (Sect. 2.3.2).

Besides this general proposal, Meiri studied other subclasses that were also found to be intractable:

- qualitative point with unary metric;
- qualitative interval with unary metric.

2.3.4 Programming Languages

This section describes a short list of languages for temporal reasoning that, besides being constraint-based, have some affinity to (Constraint) Logic Programming.

Temporal Prolog

Temporal Prolog [Hry93] can be regarded as a "synthesis of temporal logic and of the Constraint Logic Programming paradigm, in which temporal constraints are formulated". Temporal Prolog has several objectives, namely: the temporal logic inherent to the language should be intuitive and efficient, easily integrated in a LP paradigm and easy to implement on top of the Prolog interpreters. Following these guidelines, Temporal Prolog proposed a first-order *reified* logic where `HOLDS(P,T)` means that the statement (e.g. a Prolog clause) `P` holds (i.e., is true) in interval `T` and `HOLDS(P)` means that `P` holds without limitation to a temporal interval. The interval-based axioms of Temporal Prolog are the following:

- `HOLDS(A, S) & subinterval(T, S)` $\rightarrow$ `HOLDS(A, T).`

- HOLDS(A) $\rightarrow$ ($\forall$ T) HOLDS(A, T).
- HOLDS(A, T) & HOLDS(B, T) $\rightarrow$ HOLDS(A & B, T).
- HOLDS(A, U) & HOLDS(B, V) & union(U, V, T) $\rightarrow$ HOLDS(A $\vee$ B, T).
- HOLDS(A, S) & HOLDS($\neg$ A, T) $\rightarrow$ disjoint(S, T).

Horn Temporal Reference Language

Horn Temporal Reference Language [Pan95] is also a temporal extension of Prolog. In this language, atoms can have temporal labels to express their validity. It allows two types of extended atoms: events (not necessarily true over every subinterval) and properties (hold over every subinterval). Moreover, it provides inference rules for these extended atoms together with a transformation from this language to Constraint Logic Programming.

Temporal Annotated Constraint Logic Programming

Temporal Annotated Constraint Logic Programming (TACLP) [Frü93, Frü94b, Frü96] is presented as an instance of Annotated Constraint Logic (ACL) [Frü96], suited for reasoning about time. ACL generalises basic first–order languages with a distinguished class of terms called *constraints*, and a distinguished class of terms, called *annotations*, used to label a formula.

TACLP allow us to reason about qualitative and quantitative, definite and indefinite temporal information using time points and time periods as labels for atoms [RF00a]. This section presents a brief overview of TACLP that follows closely Sect. 2 of [RF00a]. For a more detailed explanation of TACLP see for instance [Frü96].

We consider the subset of TACLP where time points are totally ordered, sets of time points are convex and non-empty, and only atomic formulas can be annotated. Moreover clauses are free of negation.

Time can be discrete or dense. Time points are totally ordered by the relation $\leq$. We call the set of time points D and suppose that a set of operations (such as the binary operations $+, -$) to manage such points, is associated with it. We assume that the time-line is left-bounded by the number 0 and open to the future (∞). A *time period* is an interval $[r, s]$ with $0 \leq r \leq s < \infty$, $r \in D$, $s \in D$ and represents the convex, non-empty set of time points $\{t \mid r \leq t \leq s\}$. Therefore the interval $[0, \infty[$ denotes the whole time line.

Definition 1 (Annotated Formula) *An annotated formula is of the form $A\alpha$ where A is an atomic formula and α an annotation. Let t be a time point and I be a time period:*

(at) *The annotated formula* **A at t** *means that A holds at time point t.*

(th) *The annotated formula* **A th I** *means that A holds throughout I, i.e. at every time point in the period I.*

A th-annotated formula can be defined in terms of an at-annotated as: $A \text{ th } I \Leftrightarrow \forall t\ (t \in I \rightarrow A \text{ at } t)$

(in) *The annotated formula* **A in I** *means that A holds at some time point(s) in the time period I, but there is no knowledge when exactly. The in annotation accounts for indefinite temporal information.*

An in-annotated formula can also be defined in terms of an at-annotated as: $A \text{ in } I \Leftrightarrow \exists t\ (t \in I \wedge A \text{ at } t)$.

The set of annotations is endowed with a partial order relation $\sqsubseteq$ which induces a lattice. Given two annotations α and β, the intuition is that $\alpha \sqsubseteq \beta$ if α is "less informative" than β in the sense that for all formulas A, $A\beta \Rightarrow A\alpha$.

In addition to *Modus Ponens*, TACLP has the following two inference rules:

$$\frac{A\alpha \quad \gamma \sqsubseteq \alpha}{A\gamma}\ \text{rule}(\sqsubseteq) \qquad \frac{A\alpha \quad A\beta \quad \gamma = \alpha \sqcup \beta}{A\gamma}\ \text{rule}(\sqcup)$$

The rule ($\sqsubseteq$) states that if a formula holds with some annotation, then it also holds with all annotations that are smaller according to the lattice ordering. The rule ($\sqcup$) says that if a formula holds with some annotation and the same formula holds with another annotation then it holds in the least upper bound (denoted by $\sqcup$) of the annotations. TACLP provides a constraint theory (detailed in Sect. 6.3).

A TACLP *program* is a finite set of TACLP clauses. A TACLP *clause* is a formula of the form $A\alpha \leftarrow C_1, \ldots, C_n, B_1\alpha_1, \ldots, B_m\alpha_m$ $(m, n \geq 0)$ where A is an atom, α and α_i are optional temporal annotations, the C_j's are the constraints and the B_i's are the atomic formulas. Frühwirth [Frü96], besides providing a meta-interpreter for TACLP clauses, also presents a compilation scheme that translates TACLP into Constraint Logic Programming. Finally, Raffaetà and Frühwirth [RF00b, RF00a] provide both an operational semantics and a fix-point one for TACLP.

2.4 Temporal Modal Logic

In a succinct way we can say that Modal Logics can be regarded as the logics of qualified truth [Che80]. In this logic besides standard logical symbols one also has modal operators that are used to qualify formulas in the following way: if A is a formula and ∇ is a modal operator then ∇A is a formula.

Although the term Temporal Logic is used in a broad sense when referring to any approach that deals with temporal information within a logic framework, it has a more specific definition related to the Modal Logic approach to temporal information [DMG94, Gal08]. Tense Logic [Pri57, Pri67, Pri68] defined by Arthur Prior is regarded as the seminal work on this field. Prior proposed the four modal operators described in Table 2.1. There are several formalisms that provide variations or extensions to the Tense Logic, such as:

- Event Logic [Gal87];
- TM [Rei89] of Reichgelt;
- Logic of time intervals [HS91] of Halpern and Shoham.

See [Aug01] for an overview on the formalisms above.

Symbol	Expression Symbolised
G	"It will always be the case that ..."
F	"It will at some time be the case that ..."
H	"It has always been the case that ..."
P	"It has at some time been the case that ..."

Table 2.1: Tense logic modal operators

2.4.1 Temporal Logic Programming

This section follows the survey of Gergatsoulis in [Ger01] that states that the basis for developing temporal logic programming languages are syntactic subsets of temporal and modal logic which present well defined computational behaviour. One can consider two broad approaches to the specification of these languages: linear-time and branching-time temporal logic programming languages.

The following languages use linear-time:

- Templog [AM89] extends Horn logic programming with the operators ○ (next), □ (always) and ◇ (eventually). As an illustration consider the Fibonacci sequence using Templog:

```
fib(0).
○fib(1).
□(○ ○ fib(Z) <- fib(X), ○fib(Y), Z is X+Y).
```

- Chronolog [Org91] has two temporal operators *first* and *next* where *first* stands for the first moment in time and *next* to the next moment in time. The following is a simple example of this language:

```
first light(green).
next light(amber) <- light(green).
next light(red) <- light(amber).
next light(green) <- light(red).
```

- Disjunctive Chronolog [GRP96] that combines Chronolog with Disjunctive Logic Programming.

- Chronolog(MC) [LO96] is an extension of Chronolog based on *Clocked Temporal Logic* where predicates are associated with *local clocks.*

- Metric Temporal Logic Programming (MTL) [Brz98] time can be either discrete or dense. MTL has two temporal operators $\Box_I$ and $\Diamond_I$ defined as: $\Box_I A$ is true if A is true in all moments in the interval I and $\Diamond_I A$ is true if A is true in some moment in the interval I. As an illustration of a simple fact in in this language consider:

$\Box_{[2005,2007]}$`salary(peter, 55000).`

Cactus [RGP97] is a proposal of a branching-time language that supports two main operators: the temporal operator *first* which refers to the beginning of time and the temporal operator $next_i$ which refers to the i-th child of the current moment.

2.5 Reasoning About Actions and Change

Reasoning about actions and change studies the evolution of (a portion of) the world as the result of the occurrence of a set of actions and/or events [CM00]. The main mechanism of this field is *temporal projection*, which can be further refined as:

- *forward temporal projection* that can be regarded as inferring consequences of actions having some knowledge of what is currently believed. Usually this is performed by *deduction.*

- *backward temporal projection* that can be regarded as inferring explanations of given situations having some knowledge of what is currently believed. Usually this is performed by *induction.*

For a comprehensive study on inference of temporal knowledge see for instance [Rib93].

According to [CM00], temporal projection has to deal with three main problems:

- the *ramification problem*: concerns the specification of the effects of a given event;

- the *qualification problem*: concerns the specification of the conditions under which an event actually produces the expected effects;

- the *frame problem*: its related with the ones above and its the problem of determining what stays the same about the world as time passes and actions are performed.

Next we survey some of the more relevant theories of action and change.

2.5.1 The Situation Calculus

The Situation Calculus (SC) proposed by McCarthy and Hayes [MH69] was the first formal representation of time in Artificial Intelligence. This formalism allows one to model the evolution of a world and, although there is no explicit representation of time, *situations* are used to model the flow of time. Besides situations, the SC also specifies *fluents* and *actions*:

- *fluents* are functions and predicates that change over time;

- *actions* stand for the possible actions that can be performed in the modeled world.

Except for the initial situation (usually denoted by S_0), all other situations are generated by applying an action to a situation. As a simple illustration taken from the *blocks world*, consider the (reified) fluent `on(A, B)` stating that block `A` is on top of block `B` and the auxiliary predicate `HoldsAt` to specify when the fluents are true, one can say: `HoldsAt(on(A, B), result(move(A, B), S))` to state that its true that `A` is on top `B`

in the situation that results from moving `A` to `B` in situation `S`. Using predicates instead of functions, i.e. in a non-reified form we have `on(A, B, result(move(A, B), S))`.

Finally, the main objection to SC comes from the fact that is impossible to model concurrent actions.

2.5.2 The Event Calculus

The Event Calculus (EC) formalism was proposed by Kowalski and Sergot [KS86] and as the name states, the primitive objects are *events*. In the EC, time is explicit and *events* are somewhat similar to the *actions* of SC. An EC fluent holds at time points and there is an axiom to allow us to reason about intervals of time[5]: a fluent is true at a point in time if an event initiated the fluent at some time in the past and was not terminated by an intervening event [RN03].

The relation `Initiates` (`Terminates`) expresses that the occurrence of an event `e` at a time point `t` causes a fluent `f` to become (cease to be) true. There is also the relation `Happens` that is used to state that an event `e` happens at a time `t`. As an example, `Happens(TurnOff(LightSwitch), 0:00)` states that the lightswitch was turned off at exactly `0:00`.

Finally, Kowalski and Sadri [KS97] proved that the Event Calculus and the Situation Calculus are equivalent.

2.5.3 Temporal Nonmonotonic Reasoning

The research on temporal nonmonotonic reasoning was *triggered* by the *Yale Shooting Problem* [HM87]. Although this is a (very broad) field of research reaching beyond the focus of the present work, for completeness reasons we decided to present a brief overview (that follows [Aug01]) of several key formalisms in this area.

Shoham's Non-Monotonic Temporal Logic

Shoham's non-monotonic temporal logic [Sho88] is based on model preference and uses *Chronological Ignorance* as a criterion to define a partial order between the models, i.e. it prefers those models where a fact holds as late as possible.

[5] In the EC an interval stands for the set of points between the interval bounds.

Extended Situation Calculus

In [Pin94] Pinto adds explicit time to Situation Calculus and addresses several problems such as: representation of concurrent and complex action; reasoning on a continuous ontology; easy reference to dates; etc.

Defeasible Temporal Reasoning

In a broad sense one can say that argumentation systems allow us to reason about a changing world where the available information is incomplete or unreliable. There are several examples of argumentation system that embed temporal reasoning based on intervals, on instants or both (see [Aug01] for an overview).

2.6 Conclusions

We started this section by describing the notions of model of time and of temporal qualification. We then reviewed several constraint-based and modal approaches to temporal reasoning, with a stronger emphasis on the former since the temporal representation and reasoning followed throughout this work is constraint-based. Although, on a first look, first-order and modal approaches can be regarded as radically different, Galton noted that they rely on the common use of a first–order language: the former uses it as the proper representation language and the later as a model theory. Finally, we briefly sketched the foundational theories for reasoning about actions and change.

The research on temporal representation and reasoning, even restricted to the one that "fits" under the domain of artificial intelligence is so vast that a chapter such as this one can only lightly brush over some aspects. For a more systematic and comprehensive approach we suggest [FGV05].

Chapter 3

Temporal Databases

This chapter reviews the main concepts concerning temporal databases, namely temporal data semantics, models and languages. Moreover, besides some considerations about the design of these databases it also describes several temporal database products.

3.1 Introduction

In a broad sense we can define a *temporal database* as a repository of temporal information [Cho94] therefore one can say that most database applications are supported by temporal databases. SQL [fS03] is often the language chosen for interacting with such databases. Although it is a very powerful language for writing queries, modifications or constraints in the current state, it does not provide adequate support for the temporal counterparts: for instance, expressing the temporal equivalent of an ordinary query can be a very challenging task in SQL. To overcome such difficulties, several temporal data models, algebras and languages have been proposed.

McKenzie and Snodgrass [LEMS91] have shown that the design space for temporal algebras has in some sense been explored in that all combinations of basic design decisions have at least one representative algebra.

In this section we present a brief overview of the temporal data semantics (Sect. 3.2) together with the models and languages (Sect. 3.3) that we consider more significant. For a more in depth study on this subject the reader may refer to [TCG+93, DD02]. Since the practical aspects of databases are highly important, we also review several products that extend regular database management system in order to include temporal capabilities (Sect. 3.5). Although most of these products only consider the time between the insertion and the removal of a fact from the database some applications already

suport the time when the fact is true in the modeled reality.

3.2 Temporal Data Semantics

According to [Jen00] a database models and records information about a part of reality (modeled reality) where the different aspects of this reality are represented by database entities. In temporal databases, the facts recorded by the database entities can have an associated *time* and this is usually defined as *valid time* or as *transaction time*:

Definition 2 (Valid Time) *The valid time of a fact is the collected times - possibly spanning the past, present, and future - when the fact is true in the modeled reality.*

Definition 3 (Transaction Time) *The transaction time of a database fact is the time spanning between when it was inserted into the database and when it was logically removed from the database.*

Unlike the valid time, the transaction time may be associated with any database entity (not only with facts) and captures the time-varying states of the database. Applications that demand accountability or "traceability" rely on databases that record transaction time. Moreover, data consistency is ensured since the transaction timestamps are maintained automatically by the DBMS.

There is no single answer on how to perceive time in reality and how to represent time in a database. As mentioned in Sect. 2.2, to define a model of time we need to consider not only the time *ontology* but also the time *topology*. In databases, a finite and discrete time domain is typically assumed. Most often, time is assumed to be totally ordered, but various partial orders have also been used.

3.3 Temporal Data Models and Languages

According to [Dat99] a data model is an abstract, self-contained, logical definition of the objects, operators, and so forth, that together constitute the abstract machine with which users interact. In the relational data model, the relations are the objects and data is operated on by means of a relational calculus or algebra, with equivalent expressive power.

Several proposals for extending the relational data model to incorporate the temporal dimension of data have appeared in the literature. These proposals have differed

considerably in the way that the temporal dimension has been incorporated both into the structure of the extended relations of these temporal models and into the extended relational algebra or calculus that they define.

Since there several dozen temporal data models, instead of providing a detailed description of each one we present a list with their names, main properties and references. A similar approach will be followed in overviewing the temporal languages in Sect. 3.3.2. The only exception is the language TSQL2 [IABC+95] and its temporal data model (Bitemporal Conceptual Data Model), since this language is regarded as a consensual temporal extension of SQL-92.

3.3.1 Temporal Data Model

A useful taxonomy for temporal data models was introduced by Clifford et. al. [CCT94, CCGT95] who classified them into two main categories: *temporally ungrouped* (tuple time stamping) and *temporally grouped* (attribute time stamping) data models. To illustrate these two categories please consider tables 3.1 and 3.2 taken from [CCT94], to represent a temporal ungrouped and grouped, respectively, version of an *employee* relation (in this modeled reality, employee Tom changes his name to Thomas at time 3).

EMPLOYEE			
NAME	DEPT	SALARY	time
Tom	Sales	20K	0
Tom	Finance	20K	1
Tom	Finance	20K	2
Thomas	MIS	27K	3
Jim	Finance	20K	1
Jim	MIS	30K	2
Jim	MIS	40K	3
Scott	Finance	20K	1
Scott	Sales	20K	2

Table 3.1: Prototypical temporally ungrouped employee relation

EMPLOYEE			
NAME	DEPT	SALARY	lifespan
$0 \rightarrow$ Tom	$0 \rightarrow$ Sales	$0 \rightarrow$ 20K	

continued on next page

continued from previous page

NAME	DEPT	SALARY	lifespan
1 → Tom	1 → Finance	1 → 20K	
2 → Tom	2 → Finance	2 → 20K	
3 → Thomas	3 → MIS	3 → 27K	{0, 1, 2, 3}
1 → Jim	1 → Finance	1 → 20K	
2 → Jim	2 → MIS	2 → 30K	
3 → Jim	3 → MIS	3 → 10K	{1, 2, 3}
1 → Scott	1 → Finance	1 → 20K	
2 → Scott	2 → Sales	2 → 20K	{1, 2}

Table 3.2: Prototypical temporally grouped employee relation

These authors also show that, although the temporally ungrouped models are less expressive than the grouped ones, there is a grouping mechanism for capturing the additional semantic power of temporal grouping. Moreover, they propose a metric of historical relational completeness as a basic for determining the expressive power of the query languages over these models.

Several data models use intervals in timestamps but that doesn't mean that such models are interval-based: sometimes intervals are used as shorthands for time points. In [BBJ98] the authors emphasise that the notions of point- and interval-based timestamps must be defined at the semantical level. Moreover, besides presenting a formal definition for these notions they also evaluate several temporal data models.

Table 3.3 (taken from [JSS95]) lists several influential temporal data models. For a concise description of these models, together with their comparison, the interested reader is referred to [JSS95].

Data Model	Identifier	Time Dimensions	Reference
Temporally Oriented Data Model	valid	Ariav	[Ari86]
Time Relational Model	both	Ben-Zvi	[BZ82]
-	valid	Brooks	[Bro56]
Historical Relational Data Model	valid	Clifford	[CC87]
Homogeneous Relational Model	valid	Gadia	[Gad88]
Heterogeneous Relational Model	valid	Yeung	[GY88]
DM/T	transaction	Jensen	[JMR91]
Legol 2.0	valid	Jones	[JMS79]
Data	transaction	Kimball	[Kim78]

continued on next page

continued from previous page

Data Model	Identifier	Time Dimensions	Reference
Temporal Relational Model	valid	Lorentzos	[Lor88]
Temporal Relational Model	valid	Navathe	[NA89]
HQL	valid	Sadeghi	[Sad87]
HSQL	valid	Sarda	[Sar90b]
Temporal Data Model	valid	Segev	[SS87]
TQuel	both	Snodgrass	[Sno87]
Postgres	transaction	Stonebraker	[SK91]
HQuel	valid	Tansel	[TA86]
Accounting Data Model	both	Thompson	[Tho91]
Time Oriented Data Base Model	valid	Wiederhold	[GWW75]

Table 3.3: Temporal data models

Bitemporal Conceptual Data Model

The Bitemporal Conceptual Data Model (BCDM) [JS96] tries to maintain the simplicity of the relational model and capture the temporal aspects of the facts stored in a database. One possible definition of the BCDM according to [Jen00] is:

Definition 4 (Bitemporal Conceptual Data Model) *both valid and transaction times are supported and the relations are coalesced.*[1] *Moreover, the value* now *and* until changed *are introduced for valid time and transaction time, respectively.*

This is a conceptual model where no two tuples with mutually identical explicit attribute values are allowed, i.e. the full history of a fact is contained in exactly one tuple. To illustrate this model, please consider a relation recording employee/department information, taken from [JSS95]: employee Jake was hired by the company in the shipping department for the interval from time 10 to time 15, and this fact became current in the database at time 5. Afterwards, the personnel department discovers that Jake had really been hired from time 5 to time 20, and the database is corrected beginning at time 10. Table 3.4 shows the corresponding bitemporal element.

Emp	Dept	T
Jake	Ship	{(5, 10), ..., (5, 15), ..., (9, 10), ...,(9, 15), (10, 5), ..., (10, 20), ...}

Table 3.4: BCDM tuple

[1] Value equivalent tuples with the same non-timestamp attributes and adjacent or overlapping time intervals are merged.

3.3.2 Temporal Languages

McKenzie and Snodgrass [LEMS91] provided a set of 26 criteria for evaluating temporal algebras. Moreover, they also show that, out of these 26 criteria, seven criteria are conflicting, i.e. no algebra can satisfy all 26 criteria. Finally, the authors evaluate 12 algebras (Jones [JMS79], Ben-Zvi [BZ82], Navathe [NA89], Sadeghi [Sad87], Sarda [Sar90a], Clifford [CC87], Tansel [TA86], Gadia [Gad88], Yeung [GY88], Lorentzos [LJ88], McKenzie [LEM88], Tuzhilin [TC90]) that extend the snapshot algebra to support valid and/or transaction time. The result of such evaluation is depicted in Tables 3.5 and 3.6. The possible table entries are: Y - satisfies criterion, P - partial compliance, N - criterion not satisfied, NA - applicable, ? - not specified, O - see [LEMS91].

		Ben-Zvi	Clifford	Gadia	Yeung	Jones	Lorentzos
	Conflicting Criteria						
1	All attributes in a tuple are defined for same interval(s)	Y	N	Y	N	Y	N
5	Each set of legal tuples is a legal relation	Y	N	Y	N	Y	Y
15	Restricts relations to first normal form	Y	N	N	N	Y	Y
16	Supports a 3-D view of historical state and operations	N	N	N	?	N	N
17	Supports basic algebraic equivalences	Y	P	Y	?	P	Y
23	Tuples are time-stamped	Y	P	N	N	Y	N
24	Unique representation for each temporal relation	N	N	N	Y	N	N
	Compatible Criteria						
2	Consistent extension of the snapshot algebra	Y	Y	Y	?	Y	Y
3	Data periodicity is supported	N	N	N	N	N	Y
4	Each collection of legal attribute values is a legal tuple	N	N	N	Y	N	N

continued on next page

continued from previous page

		Ben-Zvi	Clifford	Gadia	Yeung	Jones	Lorentzos
6	Formal semantics are well defined	P	Y	Y	P	N	Y
7	Has the expressive power of a temporal calculus	P	?	Y	P	?	?
8	Includes aggregates	Y	N	P	P	P	Y
9	Incremental semantics defined	N	N	N	N	N	N
10	Intersection, Θ-join, natural join, and quotient are defined	P	P	P	N	N	N
11	Is, in fact, an algebra	Y	N	Y	N	Y	Y
12	Model doesn't require null attribute values	Y	N	Y	Y	Y	Y
13	Multidimensional timestamps are supported	N	N	N	Y	N	N
14	Reduces to the snapshot algebra	Y	P	Y	P	Y	P
18	Supports relations of all four classes	P	P	P	Y	P	P
19	Supports rollback operations	P	N	N	Y	N	N
20	Supports multiple stored schemas	N	N	N	N	N	N
21	Supports static attributes	N	N	N	Y	N	Y
22	Treats valid time and transaction time orthogonally	Y	?	?	Y	?	?
25	Unisorted (not multisorted)	N	N	N	Y	Y	Y
26	Update semantics are specified	P	N	N	N	N	N

Table 3.5: Evaluation of algebras against criteria

		McKenzie	Navathe	Sadeghi	Sarda	Tansel	Tuzhilin
	Conflicting Criteria						
1	All attributes in a tuple are defined for same interval(s)	N	Y	Y	Y	N	O
5	Each set of legal tuples is a legal relation	N	N	N	Y	Y	NA
15	Restricts relations to first normal form	N	Y	Y	N	N	N
16	Supports a 3-D view of historical state and operations	Y	N	N	N	?	N
17	Supports basic algebraic equivalences	P	?	Y	?	P	Y
23	Tuples are time-stamped	N	Y	Y	Y	N	NA
24	Unique representation for each temporal relation	Y	Y	Y	N	N	NA
	Compatible Criteria						
2	Consistent extension of the snapshot algebra	Y	?	Y	?	?	Y
3	Data periodicity is supported	N	N	N	N	N	N
4	Each collection of legal attribute values is a legal tuple	N	N	N	Y	Y	NA
6	Formal semantics are well defined	Y	P	P	P	P	Y
7	Has the expressive power of a temporal calculus	Y	P	P	P	Y	Y
8	Includes aggregates	Y	N	N	N	Y	N
9	Incremental semantics defined	Y	N	N	N	N	N
10	Intersection, Θ-join, natural join, and quotient are defined	Y	P	N	N	N	Y
11	Is, in fact, an algebra	Y	P	?	?	Y	Y
12	Model doesn't require null attribute values	Y	Y	Y	Y	Y	Y

continued on next page

continued from previous page

		McKenzie	Navathe	Sadeghi	Sarda	Tansel	Tuzhilin
13	Multidimensional timestamps are supported	N	N	N	N	N	NA
14	Reduces to the snapshot algebra	P	Y	Y	Y	P	Y
18	Supports relations of all four classes	Y	P	P	P	P	P
19	Supports rollback operations	Y	N	N	N	N	N
20	Supports multiple stored schemas	Y	N	N	N	N	N
21	Supports static attributes	Y	Y	Y	N	Y	Y
22	Treats valid time and transaction time orthogonally	P	?	?	?	?	?
25	Unisorted (not multisorted)	N	N	Y	N	Y	Y
26	Update semantics are specified	Y	N	N	N	N	N

Table 3.6: Table 3.5 (continued)

TSQL2 and SQL/Temporal

TSQL2 [IABC+95] is a consensual temporal extension to the SQL-92 language standard [Int92]. The relations are similar to TQuel [Sno87] except that facts are timestamped not with (maximal) intervals but with *finite unions* of maximal intervals. A TSQL2 fact has exactly one timestamp and there is a temporal algebra to give special meaning to those timestamps. It was one of the design goals to make the format of timestamps irrelevant, i. e. there is no commitment to a specific temporal domain and consequently it does not allow testing for equality of time instants. TSQL2 is a point-based data model. [IABC+95] presents an extensive discussion of the design decisions, including an in-depth comparison with the other temporal query languages.

In 1994 the specification of TSQL2 was published which later result in proposals for an extension to SQL3 called SQL3/Temporal (see [Sno]). The formulation of SQL/Temporal has three very important requirements:

- upward compatibility;
- temporal upward compatibility;

- sequenced valid semantics.

Upward compatibility guarantees that (non-historical) legacy application code will continue to work without change when migrating, and temporal upward compatibility in addition allows legacy code to coexist with new temporal applications following the migration. A logical consequence of the temporal upward compatibility is that timestamps are implemented as hidden columns. Therefore one can conceptualise temporal tables as being special "views" on conventional tables which include explicit timestamp columns.

Sequenced valid semantics defines that SQL/Temporal must offer, for each query in SQL3, a temporal query that "naturally" generalises the initial query.

Finally, the full temporal functionality normally associated with a temporal language is added, specifically, non-sequenced temporal queries, assertions, constraints, views, and modifications. These additions include temporal queries and modifications that have no syntactic counterpart in SQL3.

SQL/Temporal has three different semantics for the queries, namely:

- temporally upward compatible: the query is evaluated only on the current state;
- sequenced: the query is effectively evaluated on each state independently[2];
- non-sequenced: the query is evaluated at the specified state.

The semantics above apply orthogonally to valid time and transaction time.

For a better understanding of the proposed language we use a motivational example from [Sno07]. Consider the following table schema:

Employees	SSN	FirstName	LastName	Birthdate

Positions	PCN	Jobtitle

Incumbents	SSN	PCN	FromDate	ToDate

Salary	SSN	Amount	FromDate	ToDate

With the schema above, consider a SQL-92 sequenced query to provide the salary and position history for all employees:

```
SELECT S.SSN, Amount, PCN, S.FromDate, S.ToDate
```

[2]Intuitively, a sequenced query is the temporal analogue of a query on the current state.

```
FROM Salary S, Incumbents I
WHERE S.SSN = I.SSN
AND I.FromDate < S.FromDate AND S.ToDate <= I.ToDate
UNION ALL
SELECT S.SSN, Amount, PCN, S.FromDate, I.ToDate
FROM Salary S, Incumbents I
WHERE S.SSN = I.SSN
AND S.FromDate >= I.FromDate
AND S.FromDate < I.ToDate AND I.ToDate < S.ToDate
UNION ALL
SELECT S.SSN, Amount, PCN, I.FromDate, S.ToDate
FROM Salary S, Incumbents I
WHERE S.SSN = I.SSN
AND I.FromDate >= S.FromDate
AND I.FromDate < S.ToDate AND S.ToDate < I.ToDate
UNION ALL
SELECT S.SSN, Amount, PCN, I.FromDate, I.ToDate
FROM Salary S, Incumbents I
WHERE S.SSN = I.SSN
AND I.FromDate > S.FromDate AND I.ToDate < S.ToDate
```

The size and complexity of the query above has to due with the different possible relations between the intervals (`[FromDate, ToDate]`) of tables `Salary` and `Incumbents`. In SQL/Temporal and assuming that `Incumbents` and `Salary` are valid time tables, this query reduces to:

```
VALIDTIME SELECT S.SSN, Amount, PCN
          FROM Incumbents I, Salary S
          WHERE S.SSN = I.SSN
```

Nevertheless, although this language was accepted by the ANSI committee, from the ISO committee point of view the SQL/Temporal is in *limbo*, at the time of this writing. This is mainly because of criticisms that appeared during its ISO discussion: in [DD05] the authors take a brief look at TSQL2 and compare this language with the one proposed in [Dat99, DD02]. According to the authors the major flaw is that TSQL2 involves "hidden attributes"[3] therefore leading to a major departure from *The Information Principle* which states that all information in the database should be represented in one and only one way, namely, by means of relations. This uniformity carries with it uniformity of access mode (all data in a table is accessed by reference to its columns names) and uniformity of description (to study the structure of a table, one needs only to examine the description).

[3]TSQL2 valid and transaction time columns are always hidden by definition.

Moreover, the authors also consider that the concept of statement modifiers to be logically flawed. Finally, regarding temporal upward compatibility of TSQL2, these authors reject the very idea that the goal might be desirable, let alone achievable.

In [DD02] Date et. al deal with the problems of data representing beliefs that hold throughout given intervals and propose a foundation for the inclusion of support for temporal data in a truly relational database management system where additional operators on relations and relation variables having interval-valued attributes are definable in terms of existing operators and constructs.

3.4 Temporal Databases Design

With respect to their logical design, temporal databases have higher needs for design guidelines nevertheless concepts such as normalisation are not directly applicable to temporal data modelling. One approach considered applying the concepts to all the snapshots of a temporal relation, but this isn't truly temporal as it applies to each snapshot in isolation.

Semantic modelling is traditionally done through a high-level conceptual design model such as the Entity-Relationship model. Although these models are quite understandable and natural, the introduction of temporal issues make them cluttered (for a survey on this subject the reader is referred to [GJ99]).

3.5 Temporal Database Products

In this section we present a brief overview of all the products (at least to our knowledge) that allow some sort of temporal support in the context of databases. As we will see, most of the products provide only (some sort of) transaction time support. Oracle Workspace Manager and TimeDB are the only ones with valid time support. The reviews about Log Explorer, Time Navigator, Data Propagator and FlashBack Queries follow closely [Sno].

3.5.1 Log Explorer

Log Explorer [Lum] is a product from Lumigent that allows the analysis of SQL Server logs. Log Explorer gives the ability to view the evolution of rows over time and then selectively recover modified, deleted, dropped, or truncated data, exporting data for follow-up analysis and reporting (on both the relational data and the schema).

3.5.2 Time Navigator

Time Navigator [Ate] from Atempo is a high performance online backup and recovery solution for heterogeneous environments, including several major DBMS such as Oracle, Microsoft SQL Server, DB2, Sybase and MySQL. It builds a sliced repository of a database, thereby enabling image-based restoration of a past slice.

3.5.3 Data Propagator

DataPropagator [IBM] from IBM can use data replication of a DB2 log to create both before and after images of every row modification to create a transaction-time database that can be later queried.

3.5.4 SQL:2003

SQL:2003 [EMK+04], is at the time of writing, the most recent revision of the SQL standard. The full length definition of this language can be obtained from the ISO documents [fS03], but since we are interested in the temporal aspects, below we present a brief overview of them.

In SQL:2003 there are three datetime types, each of which specifies values comprising datetime fields:

- TIME comprises values of the datetime fields HOUR, MINUTE and SECOND (possibly WITH TIME ZONE). TIME is a valid time of day.
- DATE comprises values of the datetime fields YEAR (between 0001 and 9999), MONTH, and DAY. DATE is a valid Gregorian date.
- TIMESTAMP comprises values of the datetime fields YEAR (between 0001 and 9999), MONTH, DAY, HOUR, MINUTE and SECOND (possibly WITH TIME ZONE).

For the data types above there are several comparison predicates. Besides these elementary types SQL:2003 also has an interval type to represent the duration of a period of time. Moreover, we can consider two classes of intervals. One class, called year-month intervals, has a datetime precision that includes a YEAR field or a MONTH field, or both. The other class, called day-time intervals, has an express or implied interval precision that can include any set of contiguous fields other than YEAR or MONTH.

Besides intervals and datetime arithmetic there is an explicit CAST between datetime types and character string type. Finally, there are several time-varying system variables such as CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP, LOCALTIME, LOCALTIMESTAMP.

Although SQL:2003 isn't really a temporal database language, in fact almost all the features above are legacy from previous SQL versions, we decided to include a description of this language due to its widespread.

3.5.5 Oracle

Oracle DBMS support for temporal data goes far beyond the one we saw in the previous section (SQL:2003). Among the features/tools related with temporal information, below we describe the ones that we consider more relevant for this work.

FlashBack Queries

Oracle 9i flashback queries allows the application to access prior transaction time states of the database. Oracle 10g extends these queries to retrieve all the versions of a row between two transaction times and allows tables and databases to be rolled back to a previous transaction time, discarding all changes after that time.

Workspace Manager

Workspace manager appeared on Oracle database 10g and according to [Cor05] allows current, proposed and historical values for data in the same database using workspaces. A workspace[4] is a virtual environment that logically groups collections of changes that are physically contained in one or more version-enable tables. Version-enabled tables are tables where all rows can support multiple versions of the data. Since this versioning is invisible to the users of the database, DML statements continue to work in the usual way. Moreover, version-enabled tables have the history option, adding a transaction time timestamp every time the row is changed and allowing users in a workspace to go back to any point in time and view the entire database from a perspective of changes made in that workspace.

Finally, its possible to associate a valid time to stored data. To use such valid time, the user sets a valid time in his session context before executing a query. Afterwards, all queries only return versions stamped with a valid time that falls within the valid time set for the session context.

[4]Unless specified, the default user workspace is the current one.

3.5.6 TimeDB

Instead of being integrated with the internal modules of a DBMS, TimeDB [Ste05] provides a software layer between the user-applications and a conventional (atemporal) DBMS. TimeDB supports a temporal version of SQL called ATSQL2 [SBJS97] by translating temporal SQL statements into standard SQL statements which are then evaluated using a regular database management system. The language supports valid and transaction time, along with temporal queries, insert, update and delete statements. Finally, it has temporal tables (along with temporal constraints and assertions) and views.

3.6 Conclusions and Future Pointers

It is an acknowledged fact that most (if not all) database applications have temporal issues. In this chapter we saw that the research community has provided several temporal data models and languages. Therefore one question arises: why do most DBMS only have basic temporal support?

One possible answer to this question comes from the fact that even the more elementary notions of temporal databases (valid and transaction time) are quite complex leading to a departure from one of the reasons that made the relational model so widely accepted: its simplicity.

Another possible answer has to do with legacy issues. Most applications deal with the temporal issues in an ad-hoc way. When migrating to a temporal DBMS, some applications should also change in order to benefit from the new temporal mechanisms, but legacy applications are not very likely to change.

Wang et. al. [WZZ05] advocated that SQL:2003 and the XML/XQuery standards have actually enhanced our ability to support temporal applications in commercial database systems. To this end, they showed that the integration of SQL and XML allowed the management of transaction time information and stated that this approach could be extended to valid time and bitemporal databases.

The area of spatiotemporal databases is becoming increasingly important (see [AR99, SN04]). There is a strong need to support objects with extents in space and in time, therefore a whole new field of (database) applications will be available which require support for these features.

Multimedia presentations and virtual reality scenarios are in fact special breeds of spatiotemporal databases. Also the new area of temporal data mining is emerging.

As a concluding remark one would like to say that temporal databases do require a new way of thinking about information and there are still several directions for future developments.

Chapter 4

Modular Logic Programming

This chapter provides an overview of the different logic programming approaches to modularity, namely the algebraic, the logical and the syntactic approach. Afterwards, the relations between logic and objects are briefly described.

4.1 Introduction

Module systems are an essential feature of programming languages, namely because besides structuring programs they also allow the development of general purpose libraries, therefore code re-use.

A modular extension to logic programming has been subject of research over the last decades. In a broad sense one can distinguish three different approaches to modularity: the algebraic, the logical and the syntactic. The algebraic approach started with work by O'Keefe [O'K85] and considers logic programs as elements of an algebra, whose operators are the operators for composing programs. The logical approach is based on a work of Miller [Mil86, Mil89a], and extends the Horn language with logic connectives for building and composing modules. Finally, the syntactic approach (see [HF06] for a recent overview and proposal of such approach) addresses the issue of the global and flat name space, dealing with the alphabet of symbols as a means to partition large programs.

In this chapter, and for completeness reasons, we present a brief overview of these approaches where the algebraic (Sect. 4.2) and logical (Sect. 4.3) overview follows closely [LM94, BLM94]. Although Contextual Logic Programming fits into the logical category, due to the relevance to this work we decided to present a very brief presentation here and dedicate an entire chapter (5) to its description.

This chapter is organised as follows: in Sect(s). 4.2, 4.3 and 4.4 we review the algebraic,

the logical and the syntactic approach, respectively. In Sect. 4.5 we relate with Object-Oriented concepts and frameworks. Finally, in Sect. 4.6 we state some conclusions.

4.2 Algebraic Approach

The algebraic approach started with the work of O'Keefe [O'K85] and the main idea behind this proposal is that a logic program should always be understood as part of a system of programs. He provided an algebraic approach where a logic program was viewed as an element of an algebra and composition operators over that algebra. This program composition approach has several benefits, namely:

- it is a powerful tool for structuring programs without any need to extend the theory of Horn clauses;
- it supports the re-use of the same program within different composite programs;
- it is highly flexible since new composition mechanisms can be achieved by inserting the corresponding operator in the algebra or by combining the existing ones;
- it allows one to model powerful forms of encapsulation and information hiding when coupled with mechanisms for specifying the interfaces between components.

4.2.1 The Algebra of Programs and Its Operators

The programs of the algebra are regular definite logic programs and we will consider three algebraic operators: union ($\cup$), closure (*) and overriding-union ($\lhd$). The union of two programs stands for taking the set-theoretic union of their clauses; the closure of a program makes the resulting program visible to others only in terms of its logical consequences (encapsulating the program's intensional knowledge); the overriding union of P and Q ($P \lhd Q$) restricts the union of those programs to the case where the definitions in P override the corresponding definitions provided in Q.

These operators composition will be denoted with the extension formulas defined by the following productions:

$$E ::= P \mid E \cup E \mid E^* \mid E \lhd E$$

where P stands for the name of a logic program.

The immediate consequence operator can be taken as the denotation of a program in order to provide the semantics for the programs and for the composition operators (see [BLM94] for the detailed definition).

Since the notion of program equivalence induced by the immediate consequence operator is essentially operational, other more abstract semantics can also be provided (see [BLM94]).

Finally, operators can be composed in order to obtain more complex scope policies. Typical examples are the operators for nested composition and static or dynamic inheritance-based compositions.

4.3 Logical Approach

One criticism made to the algebraic approach is that the modular composition chosen for a top-level goal is used for all its sub-goals, i.e. it's not possible to dynamically modify the structure of modules and evaluate a sub-goal in a collection of modules different from the one associated with the top-level goal. The logical approach overcomes this restriction by enriching the language with operators for building and composing modules that modify the language's evaluation procedure.

Following [BLM94] the operational semantics of the logical extensions proposed in this section are defined proof theoretically where the associated proof-relation will be presented in terms of the corresponding inference system in the sequent calculus. Each sequent is denoted as pairs of the form $\Delta \vdash \Gamma$ where the antecedent Δ and the succedent Γ stand for sets of formulas and such sequent states that there exists a proof from the antecedent Δ to some of the formulas in the succedent Γ.

Proofs are defined constructively by composing inference figures of the form:

$$\frac{\textit{upper sequent}(s)}{\textit{lower sequent}}$$

4.3.1 Embedded Implications

The seminal work of Miller [Mil86] proposed a notion of modular programming based on the use of *embedded implications* $D \supset G$ where D and G stand respectively for definite clauses and goals. The Horn clauses extended with *implication goals* can be defined as:

$$
\begin{aligned}
D &::= A \mid D \wedge D \mid \forall x D \mid G \supset A \\
G &::= \top \mid A \mid G \wedge G \mid \exists x G \mid D \supset G
\end{aligned}
$$

where $\top$ and A denote, respectively, the distinguished formula *true* and an atomic formula. The intuition behind implication goal is that querying a program $\mathcal{P}$ with the goal $D \supset G$ amounts to requesting that the proof of G to be drawn from $\mathcal{P}$ by assuming D as further hypothesis. This mechanism is similar to the program composition presented in Sect. 4.2.1, nevertheless the main difference between them is that in the former the program structure is static during the evaluation of a goal and in the latter it is dynamic, as intended.

The implication goal can be formalised by the following inference rule:

$$\text{(AUGMENT)}\ \frac{\mathcal{P} \cup D \vdash G}{\mathcal{P} \vdash D \supset G}$$

The inference rules for the remaining goals are the usual ones:

$$\text{(SUCCESS)}\ \frac{}{\mathcal{P} \vdash \top} \qquad \text{(AND)}\ \frac{\mathcal{P} \vdash G_1 \ \mathcal{P} \vdash G_2}{\mathcal{P} \vdash G_1 \wedge G_2}$$

$$\text{(INSTANCE)}\ \frac{\mathcal{P} \vdash G[x/t]}{\mathcal{P} \vdash \exists x G} \qquad \text{(BACKCHAIN)}\ \frac{\mathcal{P} \vdash G}{\mathcal{P} \vdash A}$$

with the condition $G \supset A \ \in [\mathcal{P}]_\Sigma$ for backchain.

In this approach, modules can be introduced as named collection of clauses and programs can be regarded as collections of modules.

Encapsulation and Scope

Embedded implications also allow us to model forms of encapsulation and scope. To illustrate it let us consider the list-reverse predicate taken from [BLM94]:

Example 1 *List-reverse predicate with embedded implications:*

$$\begin{aligned} \forall x, y\ rev(x, y) \ \leftarrow\ & \{\forall l\ rev_1([\,], l, l). \\ & \forall x, l_1, l_2, k\ rev_1([x|l_1], l_2, k) \leftarrow rev_1(l_1, l_2, [x|k]). \\ & \} \supset rev_1(x, y, [\,]) \end{aligned}$$

In this example the definition of the auxiliary predicate (rev_1) is encapsulated in the embedded implications. Therefore, the clauses of rev_1 are local to the definition of the rev predicate.

Parametric Modules

Using embedded implications with free variables one can also account for the notion of parametric modules. As an illustration, consider a module named A that contains the clause $\exists x\ (D(x) \supset G(x))$. Since the variable x is free, we can refer to this module by $M_A(x)$, i.e. the arguments for the module name designate the parameters of this module.

Variable Inheritance

Embedded implications with free variables can also be employed to model forms of variable inheritance between nested scopes:

Example 2 *List-reverse predicate with free variables:*

$$\begin{aligned} \forall x, y\ rev(x, y) \ \leftarrow\ & \{rev_2([\,], y). \\ & \forall x, l_1, l_2\ rev_2([x|l_1], l_2) \leftarrow rev_2(l_1, [x|l_2]). \\ & \} \supset rev_2(x, [\,]) \end{aligned}$$

In this example, the *binding* of the variable y is propagated from the nested definition to the outer definition. Moreover, variable x of the rule in the nested definition is different from variable x in the outer definition.

4.3.2 Lexical Scoping

Giordano et al. [GMR88] provide a notion of *lexical scope* that allows one to determine the set of formulas for reducing each goal by looking at the syntactic structure of a pro-

gram. These authors consider a different interpretation for the embedded implications: assume that $\mathcal{P}^*$ is the set of atomic consequences of $\mathcal{P}$, i.e.

$$\mathcal{P}^* = \{A \mid A \text{ is atomic and } \mathcal{P} \vdash A\}$$

the following rule formalises their proposal:

$$\frac{\mathcal{P}^* \cup D \vdash G}{\mathcal{P} \vdash D \supset G}$$

The main difference w.r.t. the (AUGMENT) (see page 40) rule is that the body of D depends on $\mathcal{P}$ but not vice-versa. Therefore all the references to a predicate in the outer scope $\mathcal{P}$ can be bound *lexically* to the definitions occurring in that scope.

Finally, in order to avoid the (potentially expensive) computation of $\mathcal{P}^*$, Giordano et al. provided an equivalent proof system with a more complex structure for sequents and where the antecedents of a sequent forms a *stack* (as opposed to *set*) of clauses.

$$(\text{AND}_{\text{stk}}) \quad \frac{\mathcal{S} \vdash_{stk} G_1 \quad \mathcal{S} \vdash_{stk} G_2}{\mathcal{S} \vdash_{stk} G_1 \wedge G_2} \qquad (\text{INSTANCE}_{\text{stk}}) \quad \frac{\mathcal{S} \vdash_{stk} G[x/t]}{\mathcal{S} \vdash_{stk} \exists x G}$$

$$(\text{BACKCHAIN}_{\text{stk}}) \quad \frac{\mathcal{P}_i \mid \cdots \mid \mathcal{P}_1 \vdash_{stk} G}{\mathcal{P}_n \mid \cdots \mid \mathcal{P}_1 \vdash_{stk} A} \qquad (\text{AUGMENT}_{\text{stk}}) \quad \frac{D \mid \mathcal{S} \vdash_{stk} G}{\mathcal{S} \vdash_{stk} D \supset G}$$

The conditional stipulation of ($\text{BACKCHAIN}_{\text{stk}}$) is that the clause $G \supset A$ used to backchain on A belongs to $\mathcal{P}_i$, the top of the stack in the antecedent of the upper sequent. Moreover, one notices that if $i < n$, this inference rule shrinks the program stack therefore reducing the definitions available for subsequent backchaining steps (in the same nesting level). On the opposite side, the ($\text{AUGMENT}_{\text{stk}}$) rule increases the stack size by pushing the new scope D on top of the current stack.

As we shall see, a similar approach is used for the proof system of Contextual Logic Programming (see Sect. 4.3.4).

4.3.3 Closed Scope Mechanisms

Here we provide an even stronger notion of scope where the meaning of the nested scope D doesn't depend on the meaning of the outer scope associated with $D \supset G$ (as it happens in all the previous proposals). The following inference rule accounts for such a notion of scope:

$$\frac{D \vdash G}{\mathcal{P} \vdash D \supset G}$$

The semantics of this rule is the same as that of the *demo* predicate defined by Bowen and Kowalski in [BK82].

4.3.4 Contextual Logic Programming

Although we provide a very thorough description of Contextual Logic Programming (CxLP) in Chap. 5, for completeness of the comparison with other logical approaches, we present a brief description of this language. The interpretation of the implication goal in CxLP also models a lexical notion of scope, nevertheless the difference of the evaluation of the implication goal $D \supset G$ (or $D \gg G$ in the CxLP notation) is that the extension of the search space is non-monotonic, i.e. the definitions coming from the nested scope D override the corresponding definitions provided by the outer scope. To model this behaviour we can consider the following inference rule:[1]

$$(\text{AUGMENT}_{\gg}) \ \frac{D \lhd \mathcal{P}^* \vdash G}{\mathcal{P} \vdash D \gg G}$$

From the interpretation of this rule we can state that the nested scope D depends on the outer scope $\mathcal{P}$ only for those definitions which are not local to D.

Using proof rules similar to the ones we saw for $\vdash_{stk}$ (see Sect. 4.3.2) we can describe the provability relation for CxLP (denoted by $\vdash_{\gg}$) by:

$$(\text{BACKCHAIN}_{\gg}) \ \frac{\mathcal{P}_i \mid \cdots \mid \mathcal{P}_1 \vdash_{\gg} G}{\mathcal{P}_n \mid \cdots \mid \mathcal{P}_1 \vdash_{\gg} A}$$

imposing that $\mathcal{P}_i$ be the *top-most* component of $\mathcal{P}_n \mid \cdots \mid \mathcal{P}_1$ which contains a definition $(G \supset A)$ for the predicate symbol of A (in $\vdash_{stk}$ the choice of $\mathcal{P}_i$ was non-deterministic).

Returning to the list-reverse given in Example 2, consider the CxLP version:

Example 3 *List-reverse in Contextual Logic Programming*

$$\begin{aligned} \forall x, y \ rev(x, y) \ \leftarrow \ & \{rev([\,], y). \\ & \forall x, l_1, l_2 \ rev([x|l_1], l_2) \leftarrow rev(l_1, [x|l_2]). \\ & \} \gg rev(x, [\,]) \end{aligned}$$

The program of the example above is very similar with the one that we saw with in Example 2 (see page 41). The difference is that now there is no need to rename the nested predicate since the override semantics ensures that the inner definition of $rev/2$ is the one that is used to solve $rev(x, [])$.

[1] The overriding-union operator ($\lhd$) was defined in Sect. 4.2.1.

Overriding and Dynamic Scope

Mello et al. [MNR89] proposed an alternative semantics for extension goals that combines the overriding semantics of CxLP with dynamic scope. This semantics can be described by the following proof rule:

$$\frac{D \lhd \mathcal{P} \vdash G}{\mathcal{P} \vdash D \gg G}$$

In the rule above $\mathcal{P}$ stands for a set (as opposed to a stack) of clauses, where the dependency between D and $\mathcal{P}$ is again bi-directional but constrained by virtue of the restricted form of union provided by the overriding-union operator ($\lhd$).

4.3.5 Lexical Scoping as Universal Quantification

In [NM88] the authors propose an extension of the language presented in Sect. 4.3.1 in order to incorporate universal quantification of goals, i.e.

$$\begin{aligned} D &::= A \mid D \wedge D \mid \forall x D \mid G \supset A \\ G &::= \top \mid A \mid G \wedge G \mid \exists x G \mid D \supset G \mid \forall x G \end{aligned}$$

The formulas built according to the rules above are called *Hereditary Harrop Formulas.*

Due to the universal quantification, the sequent proofs for this language need to represent the domain, i.e. are of the form $\Sigma; \mathcal{P} \vdash G$ where Σ is the signature of the current domain. Additionally to the sequents we need to add the following rule to the proof system:

$$(\text{GENERIC}_\forall) \quad \frac{\Sigma + \{c\}; \mathcal{P} \vdash_\forall G[x/c]}{\Sigma; \mathcal{P} \vdash_\forall \forall x G}$$

where $c \notin \Sigma$, i.e. ($\text{GENERIC}_\forall$) extends the current signature with a new constant (c). Therefore, universally quantified embedded implications provide a mechanism to introduce constants with local scope.

Finally, Miller and Nadathur [NM88, Mil89b] also propose an extension called *Higher-Order Hereditary Harrop Formulas* that allows universal quantifiers over the predicate symbols that occur in the embedded implications. This predicate quantification encompasses an overriding mechanism similar to the one that we saw with the operator $\gg$ of Contextual Logic Programming.

The λProlog Module System

Higher-Order Hereditary Harrop Formulas is the underlying logical foundation of the declarative language λProlog [NM88]. Each module system of this language [Mil93] contains three parts: header, preamble and declarations and clauses. The header stated the module name being defined, the preamble (optional) declares the other modules that can be accumulated or imported and the remaining part of the module can contain signatures declarations and program clauses. Modules can *accumulate* or *import* other modules. If a module `mod1` contains the line

```
accumulate mod2 mod3.
```

the intended meaning is that the clauses in `mod2` and `mod3` are made available at the end of the clauses in `mod1`.

If a module `mod1` contains the line

```
import mod2 mod3.
```

modules `mod2` and `mod3` are made available (via implications) during the search for proofs of the body of clauses listed in `mod1`. To better grasp the dynamic semantics of *import* vs *accumulate* see [Mil93].

There are multiple implementations of this language, Teyjus[2] being the most actively supported.

4.4 Syntactic Approach

Initially Prolog systems had a global and flat predicate name space, leading to several difficulties specially when developping large applications. Modules in current Prolog systems address this issue. Paradoxically, although there is a standard for Prolog modules [fS00] almost each Prolog system has its own and incompatible module specification.

There are two types of modules systems: name-based and predicate-based. In the former each atom is tagged with the module name, with the exception of those atoms that are defined as public. In the latter, each module has its own independent predicate name space.

One great disadvantage of the name-based approach is that *all* data is local to a module, therefore atom `foo` from module `m1` is different from atom `foo` in module `m2`. Of course

[2]See http://teyjus.cs.umn.edu/.

that this feature poses several difficulties, even for developping simple programs based on modules.

One well known problem of an independent name space for each module (predicate-based approach) is related to meta–predicates such as `call(Goal)` that have to know in which module the `Goal` must be resolved. Moreover, the call predicate interferes with the protection of the code, an essential task of a module system. In [HF06] the authors take a further step down and distinguish between the *called module code protection* (only the visible predicates of a module can be called from the outside) from the *calling module code protection* (the called module does not call any predicate of the calling module, as they are not visible[3]). Furthermore, they classify several Prolog systems in terms of called/calling module code protection and propose a formal module system with both forms of code protection.

In this section besides the ISO standard for modules we also present a brief overview of several Prolog systems that address the issue of modularity. As we shall see, almost all these systems are predicate-based,[4] i.e. they prefer to have global data and deal in some way with the problem of meta-predicates. Wielemaker [Wie97] points out a justification for this (almost) unanimous preference: it is much more frequent to pass data across modules in program than writing meta-predicates that have to be used across modules.

4.4.1 Prolog Modules: the ISO Standard

According to the ISO standard proposal for Prolog modules [fS00], a module is a named collection of procedures and directives together with provisions to export some of the procedures and to import and re-export procedures from other modules. A module name `M` can be used to qualify terms `T`, leading to a term whose principal functor is `(:)/2`, i.e. `M:T`.

Since this is a predicate-based approach a special attention is given to meta-predicates. Namely, it defines the *calling context* as the name of the module from where a call is made together with the notion of meta-procedure and meta-variable where the former is a procedure whose actions depend on the calling context and the latter is a variable used as argument of a meta-procedure which will be subject to module name qualification when the meta-procedure is activated.

The standard also defines that procedures can be (re-)exported and, of course, imported by modules. Moreover, such actions can be selective meaning that only specified

[3]Nevertheless one must consider the exception where calling a non-exported predicate is indeed the intended behaviour as it happens for instance in the implementation of the predicate `forall/2`.

[4]The exception to the rule is the name-based XSB.

procedures are exported/imported.

Moreover, two important notions are defined in this standard: *accessibility* and *visibility* of procedures. A procedure is *visible* in a module `M` if it can be activated from `M` without using qualification. A procedure is *accessible* if it can be activated with module name qualification.

Finally, ISO doesn't provide any sort of module code protection although it allows an extension that *hides* certain procedures defined in a module `M` so that they cannot be activated, inspected or modified except from within a body of the module `M`.

4.4.2 Implementations

Ciao Prolog

The Ciao Prolog System [BCC+97] is a predicate-based Prolog where the predicates visible in a module are those defined in that module, plus the predicates imported from other modules (these predicates must always be referred with the module name as prefix, i.e. `Module:Predicate`). The default module of a given predicate name is the local one if the predicate is defined locally, else it is the last module from which the predicate is imported. Finally, only predicates exported by a module can be imported into other modules. All predicates defined in files with no module declaration belong to a special module called `user`, and all are implicitly exported. Every `user` or module file implicitly imports all the builtin modules. These aspects accommodate compatibility with traditional module-less Prolog.

Before calling meta-predicates, Ciao Prolog translates the meta-arguments into an internal representation containing the goal and the context in which the goal must be called, therefore correctly selecting the context in which the meta-data must be called. Ciao observes both forms of module code protection, i.e. called and calling module code protection.

ECL^{i}PS^e

According to [Aea07] ECL^{i}PS^e (ECL^{i}PS^e Common Logic Programming System) is a Prolog based system whose aim is to serve as a platform for integrating various Logic Programming extensions, in particular CLP. It provides a module system that follows most of the ISO directives, namely it supports (re)exporting and importing predicates. An exported predicate is accessible everywhere although it may require explicit module name qualification via `:/2`. Meta-predicates are declared by means of the `:- tool` directive. Besides the usual qualification, the system also provides the

construct `call(Goal)@Module` that (in some situations) can be used to access non-exported predicates. Nevertheless, once a module implementation is stable its possible to *lock it* and after that not even `call(Goal)@Module` can access to the hidden parts of the module, i.e. locked modules provide both forms of code protection.

Mercury

Mercury [F. 06] is an efficient, purely declarative logic programming language. It is touted by its proponents as a successor of Prolog that is a syntax for first-order logic augmented with types and modes. Mercury supports programming-in-the-large: the module system is flat and loosely based on the module system of imperative languages such as Modula-2. A module:

- starts with a module declaration directive,
- is followed by an interface section where the exported items are declared,
- ends with an implementation section that contains the definitions of the exported entities and the declarations and definitions of entities used only within the module.

Mercury protects module code: a module cannot get access in any way to the entities private to another.

SICStus Prolog

The module module system of SICStus Prolog [The07] is modeled after the Quintus Prolog module [Lab03] and is described as:

- procedure based;
- flat, i.e. all modules are visible to one another;
- non-strict, i.e. the normal visibility rules can be overridden by special syntax `(M:P)`.

Due to non-strictness there is no called module code protection: the special syntax allows calling non-exported predicates of a given module. Although it is possible to declare meta-predicates and corresponding meta-arguments that must be module name expanded, one can use a similar expansion to access any predicate of the calling module, therefore there is no calling module code protection in SICStus.

SWI-Prolog

SWI-Prolog [Wie97] has a module system where predicates can be exported (using the module directive) and imported (by means of predicates `use_module/[1,2]` or `import/1`). It has two special modules: *system* and *user*. The first module contains all built-in predicates and the second forms the initial workspace of the user. Moreover, to avoid explicitly importing predicates such as system predicates, it is assigned a *default module* to every module: the default module of *user* is *system* and all other modules import from *user*.

In order to work with meta-predicates in modules, SWI-Prolog introduces the notion of *context module* for active goals: by default is the definition module[5] of the predicate running the goal; for meta-predicates is the context module of the goal that invoked them. This mechanism is called *module transparent* and differs from the `meta_predicate/1` of SICStus (see [Wie97] for a comparison). This module system doesn't ensure any kind of module code protection.

XSB

XSB's [SSW+07] module system is *flat* (no module nesting), *file based* (one module per file) and *name-based* where any symbol in a module can be imported, exported or be a local symbol. Moreover, all non-predicate symbols are assumed to be global whereas predicate symbols are assumed local to that model unless declared otherwise. Nevertheless, symbols cannot be exported.

A file is treated as a module if it has one or more *export declarations*. Since there is no module directive, the module name is equal to the base file name.

Finally, XSB fully satisfies the code protection property: the meta-call of a term corresponds to the call of the predicate of the same symbol and arity as the module where the term has been created.

YAP

YAP [YAP06] is a high-performance Prolog compiler that implements a module system compatible with the Quintus Prolog [Lab03] modules. Like SICStus, the module system is predicate-based, flat and non-strict. Therefore, YAP has no module code protection. There are two special modules: default module *user* to where new predicates belong and *primitive* module to where system predicates belong. Besides `module/{1,2,3}` declaration, YAP has also `meta_predicate/1` directive to state that some arguments

[5] Module in which the predicate was originally defined.

of a procedure are goals, clauses or clauses heads, and that these arguments must be expanded to receive the current source module.

4.5 Logic and Objects

Modularity provides concepts that are similar to some of the core ideas in Object-Oriented programming: for instance, [AD03b] shows how *contexts* (see Sect. 4.3.4) can be used as *objects*.

Due to this proximity between Object-Oriented and Modularity, for completeness reasons we decided to include a description of how embedded implications can be used to model Object-Oriented concepts such as inheritance. Finally, in order to provide a practical point view, we briefly describe an object-oriented extension of Prolog called Logtalk.

4.5.1 Object-Oriented Programming and Embedded Implications

According to [BLM94][6] the integration of OO and logic programming paradigms can be considered from two different approaches:

1. one approach started with the work of Aït-Kaci and Nasr on LOGIN [AKN86]. This language can be described as a logic language with inheritance where classes and objects are represented as compound terms whose arguments designate the objects attributes.

2. the other approach started with McCabe's Class Template Language [McC92] and is based on the idea of representing an object as a *first-order* logic theory.

Following the second approach, classes can be introduced as parametric modules whose parameters act as (stateless) instance variables in conventional OO languages. Moreover, a message $\overline{O} : G$ requesting that G be evaluated in object $\overline{O}$ can be modeled as:

$$\frac{\overline{O} \vdash_{oo} G}{O \vdash_{oo} \overline{O} : G}$$

[6] This subsection follows closely this reference.

Inheritance

Considering the approach that represents an object as a first-order logic theory, Monteiro and Porto in [MP90] divide inheritance into two kinds: semantic and syntactic. In order to better define these types of inheritance consider two units u and v such that `u isa v`. With the semantic (also called relational or predicate) we can consider that the compositionality is at the semantic level by using relations from the model of v in order to construct the model of u. If the inheritance is syntactic, one must compose syntactic definitions of v and u in order to build the model of u.

Another orthogonal concept also related to inheritance presented in [MP90] was the mode of inheritance: extension or overriding. These concepts are similar to the algebraic operators union ($\cup$) and overriding-union ($\lhd$) presented in Sect. 4.2.1.

One can also use embedded implications to model inheritance, where the message-sent $O : G$ causes the evaluation of G to take place not simply in O but in the program obtained by the composition of O with all of its ancestors in the object hierarchy. Consider the hierarchy O_n *isa* O_{n-1} ... O_2 *isa* O_1 and the message-sent $O_J : G$. A system with syntactic overriding inheritance is modeled by means of the following proof rule:

$$\frac{O_j \lhd O_{j-1} \lhd \ldots \lhd O_1 \vdash_{oo} G}{H \vdash_{oo} O_j : G}$$

where H is the current object hierarchy.

A system with semantic overriding inheritance can be modeled by the rule:

$$\frac{(O_j \lhd (O_{j-1} \lhd (\ldots \lhd O_1^*)\ldots)^*)^* \vdash_{oo} G}{H \vdash_{oo} O_j : G}$$

4.5.2 Logtalk

Logtalk [Mou03] is an object-oriented extension of Prolog. As expected, this language allows the definition of several namespaces. It supports both prototype and class-based system and objects can have multiple independent hierarchies. Inheritance and object predicates can be private, protected or public. Objects can have parameters [Mou00], nevertheless the access to parameter values is through a built-in method.[7] Although private predicates enable calling module code protection, meta-predicates can access the private predicates of the calling object, therefore there is no calling module code protection.

[7]Instead of making the parameters scope global over the whole object.

4.6 Conclusions

In this chapter we presented an overview of the main approaches to modularity in Logic Programming. Namely, we described the *algebraic approach* that considers logic programs as elements of an algebra whose operators are operators for composing programs; the *logical approach* that enriches the language with operators for building and composing modules that modify the language evaluation procedure and the *syntactical approach* that addresses the issue of the global and flat name space, dealing with the alphabet of symbols as a mean to partitionate large programs. To further perspective this issue we also included the *Object-Oriented approaches* in LP, since OO and modularity intersect in several ways. Finally, although most implementations/systems herein described follow the syntactic approach, there are also examples of systems that use the OO or the logical approach.

Chapter 5

Contextual Logic Programming

This chapter presents a detailed overview of Contextual Logic Programming, namely its language, operational and declarative/fix-point semantics. It also describes several extensions to the base language and an optimisation through abstract interpretation. Finally, the virtual-machine-based implementations CSM (Contexts as SICStus Modules) and GNU Prolog/CX are reviewed.

5.1 Introduction

Contextual Logic Programming (CxLP) is a modular extension of Horn clause logic proposed by Monteiro and Porto [MP89, MP93]. The CxLP "extension goal" can be regarded as a *non-monotonic* version of Miller's "implication goal" [Mil86, Mil89a].[1] The extension goal is denoted with the $\gg$ operator and $D \gg G$ (pronounced extend with D for goal G) is derivable from a program P if G is derivable from $A \cup D$ and A is derivable from P, for some finite set A of atoms for predicates *not defined* in D. Therefore $\gg$ provides a sort of lexical scoping for predicates: predicates in G which are defined in D are bound to those definitions, the others can be obtained from program P. Besides lexical scoping, CxLP also accounts for contextual reasoning that is widely used for several Artificial Intelligence tasks such as natural language processing, planning, temporal reasoning, etc.

Work by Abreu and Diaz [AD03a] presented a revised specification of CxLP together with a new implementation for it and explained how this language could be seen as a shift into the Object-Oriented Programming paradigm.

In this chapter we start by defining the syntax of the language (Sect. 5.2), afterwards we present its operational (Sect. 5.3) and corresponding fix-point semantics (Sect. 5.4).

[1] See Sect. 4.3 for an overview of logical approaches to modularity.

Next a possible optimisation through abstract interpretation is given (Sect. 5.6). Finally, we briefly review two proposed systems for CxLP (Sect. 5.7), namely CSM (Contexts as Sicstus Modules) [NO93, NO] and GNU Prolog/CX [AD03a].

5.2 The CxLP Language

The vocabulary of Contextual Logic Programming, besides the usual sets of variables, constants, functions and predicates symbols also contains a set of *unit* names. Although we are going to see a formal definition of unit for now assume that it stands for a finite set of Horn clauses, i.e. the unit represents the concept of "module".

Definition 5 (Vocabulary of CxLP) *The vocabulary of Contextual Logic Programming contains the following finite and pairwise disjoint sets:*

- *Var of variables*
- *Fun of functions*
- *Pred of predicates*
- U_n *of unit names*

Terms, atomic formulas and clauses are defined in the usual way, with the only difference that clauses can have *extension formulas* in their bodies:

Definition 6 (Extension Formula) *An extension formula is a formula* $u \gg G$ *where* u *is a unit name* ($u \in U_n$) *and* G *is a finite conjunction of atomic or extension formulas.*

Using the definitions above we are now in position to provide a more formal definition of units:

Definition 7 (Unit) *A unit is a formula of the form* $u : U$, *where* $u \in U_n$ *and* U *is a finite set of clauses. We call* u *the name of the unit and* U *its body (also denoted by* $|u|$*). Moreover, the set of predicates defined in* U *is denoted by* $||u||$ *(the sort of* u*).*

Finally, a *system of units* is a set $\mathcal{U}$ of units such that no two distinct units in $\mathcal{U}$ have the same name.

5.3 Operational Semantics

The system of units that we saw above is a static concept of a program. The dynamic view of a program is the derivation of formulas in contexts. Context names are arbitrary sequences of unit names ($C_n = U_n{}^*$). The empty context is represented by λ and the context that results from extending the context $C \in C_n$ with unit $u \in Un$ is represented by uC.

The operational semantics is presented by means of derivations. For self-containment reasons, we explain briefly what is a derivation and this description follows closely the one presented in [MP93].

Derivations are defined in a a declarative style, by considering a derivation relation and introducing a set of inference rules for it. A tuple in the derivation relation is written as:

$$\mathcal{U}, C \vdash G[\theta]$$

where $\mathcal{U}$ is a system of units, C a context name, G a goal and θ a substitution. Since the system of units remains the same for a derivation, we will omit $\mathcal{U}$ in the definition of the inference rules. Each inference rule has the following structure:

$$\frac{Antecedents}{Consequent}\{Conditions$$

The *Consequent* is a derivation tuple, the *Antecedents* are zero, one or two derivation tuples and *Conditions* are a set of arbitrary conditions.

The inference rules can be interpreted in a declarative or operational way. In the declarative reading we say that the *Consequent* holds if the *Conditions* are true and the *Antecedents* hold. From a operational reading we get that if the *Conditions* are true, to obtain the *Consequent* we must establish the *Antecedents*. A derivation is a tree such that:

1. any node is a derivation tuple;
2. in all leaves the goal is null;
3. the relation between any node and its children is that between the consequent and the antecedents of an instance of an inference rule;
4. all clause variants mentioned in these rule instances introduce new variables different from each other and from those in the root.

The operation of the contextual logic system is as follows: given a context C and a goal G the system will try to construct a derivation whose root is $C \vdash G\ [\theta]$, giving θ as the result substitution, if it succeeds. The substitution θ is called the *computed answer substitution.*

We may now enumerate the inference rules which specify computations. These rules are based on the ones presented in [MP89], that can be regarded as the base for the ones in [MP93]. Together with each rule we will also present its name and a corresponding number. Moreover, the paragraph after each rule gives an informal explanation of how it works.

Null goal

$$\frac{}{C \vdash \Delta[\epsilon]} \tag{5.1}$$

The null goal is derivable in any context, with the *empty* substitution ϵ.

Conjunction of goals

$$\frac{C \vdash G_1[\theta] \qquad C \vdash G_2\theta\ [\sigma]}{C \vdash G_1, G_2[\theta\sigma]} \tag{5.2}$$

To derive the conjunction derive one conjunct first, and then the other in the same context with the given substitutions.

Reduction

$$\frac{uC \vdash (G_1, G_2 \cdots G_n)\theta[\sigma]}{uC \vdash G[\theta\sigma]} \left\{ \begin{array}{l} H \leftarrow G_1, G_2 \cdots G_n \in |u| \\ \theta = \mathrm{mgu}(G, H) \end{array} \right. \tag{5.3}$$

If the unit at the top of the context (u) defines the predicate of the atomic formula G, reduce such formula in the unit and derive the body of the clause used in the reduction.

Extension Formula:

$$\frac{uC \vdash G[\theta]}{C \vdash u \gg G[\theta]} \tag{5.4}$$

To derive an extension formula, derive the "inner" formula in the context obtained by extending the current one with the unit name in the extension formula.

Context traversal:

$$\frac{C \vdash G[\theta]}{uC \vdash G[\theta]} \{ \; name(G) \notin ||u|| \tag{5.5}$$

When none of the previous rules applies remove the top element of the context, i.e. resolve goal G in the *supercontext*, i.e. the context that results from removing the top unit from the current context.

5.3.1 Application of the Rules

It is rather straightforward to check that the inference rules are mutually exclusive, leading to the fact that given a derivation tuple $C \vdash G[\theta]$ only one rule can be applied. Moreover, the operational semantics of a goal G is the set of all substitutions θ such that $\lambda \vdash G[\theta]$.

Finally, in [MP89] besides the top-down derivation above there is also a bottom-up one. Since both approaches were proven equivalent and the bottom-up is similar to the declarative reading of the inference rules presented, we considered the inference rules above to be sufficient.

5.4 Declarative/Fix-Point Semantics

Similarly to the Herbrand interpretation of a first-order language, in CxLP the domain of an interpretation is the Herbrand universe H, each function of arity n is interpreted as a function from H^n to H. Nevertheless, instead of considering an interpretation as an assignment of a subset of the Herbrand base B (set of all ground atomic formulas) to every predicate symbol, CxLP interpretations associate unit names with functions from $\wp(B)^2$ to $\wp(B)$. For instance, $u \gg G$ is true in a subset S of the Herbrand base, from here on called *situation*, if every formula in G is true in the situation obtained from S by the transformation associated with u.

More formally, and considering $P \subseteq Pred$, $S \subseteq B$, S_P the restriction of S to P, i.e. the set of all $p(t_1, \ldots, t_n) \in S$ such that $p \in P$. Considering also $S_{-P} = S_{Pred-P}$, an interpretation I is an assignment, for each $u \in Un$, of a continuous function:

$$u_I : \wp(B_{-||u||}) \rightarrow \wp(B_{||u||})$$

[2] $\wp(B)$ is the powerset of B.

Moreover, the *update* of $S \subseteq B$ by $u \in Un$ is:

$$S[u_I] = S_{-||u||} \cup u_I(S_{-||u||})$$

Denoting $S \models_I f$ by the fact that f is *true* in S with respect to I, the relation $\models_I$ is defined by:

- $S \models_I u : U$ if and only if $S[u_I] \models_I U$
- $S \models_I H \leftarrow B$, where $H \leftarrow B$ is ground, if and only if $S \models_I H$ whenever $S \models_I B$
- $S \models_I u \gg G$ if and only if $S[u_I] \models_I G$
- $S \models_I g$ where g is a ground atomic formula, if and only if $g \in S$

Similarly, an interpretation I is a model of a system of units $\mathcal{U}$ if every unit in $\mathcal{U}$ is true in every situation with respect to I. Monteiro and Porto also proved that any system of units has a minimal model and this model could also be obtained by means of a fix-point operator (for the description of this operator see [MP89]). Finally, they also showed that the operational semantics (see Sect. 5.3) and the declarative presented herein are equivalent.

5.5 Extensions

Several extensions were proposed to the basic theory above. In this section we present a brief description of the extensions that are relevant for this work and that were the subject of a recent revision of CxLP [AD03a]. A more formal and complete description of all the extensions (including extensions for predicate hiding, two-level contexts, etc) can be found in [MP89] and [MP93].

Parametrised Units

Unit parameters act as "global variables" for units and these parameters are encoded as arguments of a term whose main functor is the unit name. Therefore instead of referring to a unit with an atom we can use terms whose main function is the unit name, such terms are called *unit descriptor*. To account for this extension, several inference rules of Sect. 5.3 need an extra unification since now unit descriptors (and therefore contexts) can have variables. For instance, the rule for conjunction of goals becomes:

$$\frac{C \vdash G_1[\theta] \qquad C\theta \vdash G_2\theta\ [\sigma]}{C \vdash G_1, G_2[\theta\sigma]} \tag{5.6}$$

Since C may contain variables in unit descriptors that may be bound by the substitution θ obtained from the derivation of G_1, we have that θ must also be applied to C in order to obtain the updated context in which to derive $G_2\theta$.

Context switch

$$\frac{C' \vdash G[\theta]}{C \vdash C' :< G[\theta]} \tag{5.7}$$

The purpose of this rule is to allow execution of a goal in an arbitrary context, independently of the current context. This rule causes goal G to be executed in context C'.

Context inquiry

$$\frac{}{C \vdash :< X[\theta]} \left\{ \; \theta = \text{mgu}(X, C) \right. \tag{5.8}$$

To complement the context switch operation, there is an operation which fetches the current context. This rule recovers the current context C as a term and unifies it with term X, so that it may be used elsewhere in the program.

5.6 Optimisations

One way to classify modular languages, according to [BCLM98] concerns the way in which the set of clauses that define a goal are found. Considering the evaluation of a goal g in a context $C = [u_n, \ldots, u_1]$, we have:

- **Dynamic scope rules**: is a system in which the definition associated with each predicate call is given by the clauses for g contained in the whole context C, regardless of the unit where the call occurs.
- **Static scope rules**: is a system in which the definition associated with predicate call g occurring in the unit u_i, is given by the clauses for g contained in the sub-context $[u_i, \ldots, u_1]$. The definitions visible from a unit u are therefore those found in u and its ancestors in the context.

CxLP is a system with static scope rules, and a useful optimisation for these systems is partial deduction or partial evaluation. One possible approach can be found in [BLM93] where the authors assume that each goal goal is evaluated in a context and therefore propose a transformation that's not only a function of the goal, but also of the initial

context where this goal has to be evaluated. This program transformation leads to a new, still modular program where some of (possibly all) the modules occurring in the initial context are replaced by the specialised version.

Nevertheless, to make use of a partial evaluation based approach, one has to know beforehand what the initial context and goal will be. If this knowledge is absent, this optimisation cannot be applied.

For dynamic scope rules systems the optimisation proposed in [CLM96] is based on a bottom–up abstract interpretation [CC92]. In that proposal an abstract transformation function is presented, which returns sets of pairs $\langle c, p\rangle$ where c is a *minimal* context for predicate p, i.e. any successfull derivation for an atom whose predicate is p must must contain (at least) one of those minimal contexts. Moreover, this condition is necessary but not sufficient since the computation with a minimal context may still fail, just not because of undefined predicates.

5.6.1 Abstract Interpretation for Static Scope Systems

Although the proposal of [CLM96] was designed for dynamic scope rules systems like the language of *embedded implications* (see Sect. 4.3.1), in this section we are going show that it can be applied to a static scope system such as CxLP. Before formally developing our proposal, we start with the following motivating modular program:

```
:- unit(u1).        :- unit(u2).        :- unit(u3).
p :- q.             q :- r.             r.
```

From the abstract analysis of [CLM96] we get pairs $\langle c, p\rangle$ where c stands for the minimal context for predicate p. In this case we have:

$$\langle\{\mathrm{u1}, \mathrm{u2}, \mathrm{u3}\}, \mathrm{p}\rangle \cup \langle\{\mathrm{u2}, \mathrm{u3}\}, \mathrm{q}\rangle \cup \langle\{\mathrm{u3}\}, \mathrm{r}\rangle$$

Although in CxLP the order in which the units appears in the context is important, we can state that the computation for `p` in a context without units `u1`, `u2` and `u3` will certainly lead to a failure. Therefore, we claim that this method is also suitable for optimising CxLP. Nevertheless, as the reader might have noticed, this way we also allow the evaluation of `p` in contexts like `[u2, u1, u3]` or `[u3, u1, u2]` and this ought to be avoided with a finer pruning mechanism.

To prove that this method can be applied to a static scope language we rely upon the unifying formalisation presented in Sect. 4.3 (page 39) for *embedded implications* and CxLP. Returning to the formalisation of that chapter can be misleading since there the

provability relation for *embedded implications* and CxLP was $\vdash$ and $\vdash_{\gg}$, respectively, and in this chapter $\vdash$ stands for the provability of CxLP. The reason for that has to do with the fact that in Sect. 4.3 there were several logical approaches (embedded implications being the first); in this chapter, CxLP is the only language described. Nevertheless, since we need the unifying framework of Sect. 4.3 we will return to that notation.

Proposition 1 *Given a goal G, a* stack *of clauses $\mathcal{P}$ and the set $\mathcal{P}'$ that results from the union of all the clauses in the stack $\mathcal{P}$, we have*

$$\mathcal{P} \vdash_{\gg} G \Rightarrow \mathcal{P}' \vdash G$$

Proof: $\mathcal{P} \vdash_{\gg} G$ is true if there is a tree of derivation tuples that starts with $\mathcal{P} \vdash_{\gg} G$, the relation between two consecutive tuples is that between the antecedents and the consequent of the inference rules above and where the goal of the last tuple is null.

The basic idea for this proof is to construct a derivation tree for $\mathcal{P}' \vdash G$ using the existing (by hypothesis) derivation for $\mathcal{P} \vdash_{\gg} G$. More specifically, we are going to replace each inference rule used in the CxLP derivation by the equivalent embedded implications rule.

Therefore the tree for $\mathcal{P}' \vdash G$ has $\mathcal{P}' \vdash G$ as its root and to build the children of any node consider the inference rule that was applied in the tree for $\mathcal{P} \vdash_{\gg} G$.

The rules for (SUCCESS$_{\gg}$), (AND$_{\gg}$) and (INSTANCE$_{\gg}$) can be trivially replaced by the identical counterparts (SUCCESS), (AND) and (INSTANCE).

Both versions of the Augment have no conditions, therefore instead of (AUGMENT$_{\gg}$) we can use (AUGMENT), with the proper substitutions, i.e. if $\mathcal{P}' \vdash D \supset G$ is the parent, then its child is $\mathcal{P}' \cup D \vdash G$ (instead of $D \mid \mathcal{P} \vdash G$).

It remains to verify we can replace (BACKCHAIN$_{\gg}$) by (BACKCHAIN). For that, one must notice that the set of rules in the embedded implications increases continuously. If (BACKCHAIN$_{\gg}$) was used then there is a component in the stack that contains a rule $G \supset A$, therefore from the monotony of embedded implications set of clauses, we have that that $G \supset A$ also belongs to the set that results from that stack, i.e. (AUGMENT) condition is verified and therefore it can be used. □

In Table 5.1 we find a CxLP derivation and the corresponding embedded implications one that illustrates the proof procedure considered above.

CxLP	Embedded implications
$\lambda \vdash_{\gg} \{r\} \gg (\{q : -r\} \gg q)$	$\lambda \vdash \{r\} \supset (\{q : -r\} \supset q)$
$\{r\} \mid \lambda \vdash_{\gg} \{q : -r\} \gg q$	$\{r\} \cup \lambda \vdash \{q : -r\} \supset q$
$\{q : -r\} \mid \{r\} \mid \lambda \vdash_{\gg} q$	$\{q : -r\} \cup \{r\} \cup \lambda \vdash q$
$\{q : -r\} \mid \{r\} \mid \lambda \vdash_{\gg} r$	$\{q : -r\} \cup \{r\} \cup \lambda \vdash r$
$\{r\} \mid \lambda \vdash_{\gg} \top$	$\{q : -r\} \cup \{r\} \cup \lambda \vdash \top$
$\{r\} \mid \lambda \vdash_{\gg}$	$\{q : -r\} \cup \{r\} \cup \lambda \vdash$

Table 5.1: Derivation: CxLP and embedded implications

It follows from Proposition 1 that $\mathcal{P}' \not\vdash G \Rightarrow \mathcal{P} \not\vdash_{\gg} G$. Therefore, consider that we want to verify if atomic goal G is true in the stack (context) $\mathcal{P}$, i.e. $\mathcal{P} \vdash_{\gg} G$ and that p is the predicate of G. Furthermore, suppose also that $\mathcal{P}'$ is the (embedded implications) program that results from the union of all the clauses in the $\mathcal{P}$. If $\mathcal{P}'$ doesn't contains any minimal context for p, them $\mathcal{P}' \not\vdash G$ and therefore $\mathcal{P} \not\vdash_{\gg} G$.

5.7 Implementations

There are two main approaches to implement CxLP in a logic programming framework:

- Prolog-based
- Virtual-machine-based

Meta-interpretation and translation to Prolog are two widely used techniques in Prolog-based approaches. Although these systems are rather simple to implement, due to efficiency reasons they aren't used to build real-world applications. Virtual-machine-based implementations although more elaborate in development, are more efficient. For a comparison between the approaches see for instance [DLM+92].

In this section we present a brief overview of the more relevant, at least to our knowledge, virtual-machine-based implementations.

5.7.1 CSM (Contexts as SICStus Modules)

The Contexts as SICStus Modules (CSM) system [DNO92] can be regarded as an enhanced virtual-machine that exploits the program representation and the predicate addressing mechanism of a modular Prolog programming system. As the name states the modular Prolog system chosen was SICStus (see Sect. 4.4.2) and in this implementation, contexts are first-class objects which can be referred to by logic variables. Besides

the basic CxLP theory, CSM also implements two-level contexts, context enquiry and context switch.

In [NO, NO93] the authors extend the implementation above in order to include object-oriented abstractions and mechanisms such as state modification. In order to incorporate the notion of update they make a clear distinction between deductive and updating phases: the deductive activity isn't affect by the update actions it produces; updates are backtrackable and just take place when the corresponding demonstrations ends.

5.7.2 GNU Prolog/CX

GNU Prolog/CX [AD03a] is a WAM-based approach, i.e. extends the standard Warren Abstract Machine (WAM) [War83] with a minimal set of instructions and new data structures in order to provide the CxLP characteristics. For a prior WAM-based approach to CxLP see [LMN92].

Besides the basic CxLP theory, GNU Prolog/CX presents several extensions such as units arguments, two-level contexts, context enquiry and context switch. A full detailed tutorial (A.1) together with its reference manual (A.2) can be found in Appendix A.

In [AD03a] the authors show that this compiler incurs a low overhead when compared to regular GNU Prolog when the CxLP extensions are not being used. Moreover, they also compare it to CSM (see Sect. 5.7.1) and state that the relative performance is much better, allowing GNU Prolog/CX to have *deeper* contexts.

Finally, this system was used to represent and query ontologies [LFA07], to develop University Information Systems [ADN04] and applied for temporal representation and reasoning (see Sect(s). 6.5 and 7.5, respectively).

University Employees

Using the syntax of GNU Prolog/CX consider a unit named `employee` to represent some basic facts about university employees, using `ta` and `ap` as an abbreviation of teaching assistant and associate professor, respectively:

```
:-unit(employee(NAME, POSITION)).

name(NAME).
position(POSITION).

item :- employee(NAME, POSITION).
```

```
employee(bill, ta).
employee(joe, ap).
```

The main difference between the example above and a plain logic program is the first line that declares the unit name (`employee`) along with the unit arguments (`NAME, POSITION`). Unit arguments help avoid the annoying proliferation of predicate arguments, which occur whenever a global structure needs to be passed around. A unit argument can be interpreted as a "unit global" variable, i.e. one which is shared by all clauses defined in the unit. Therefore, as soon as a unit argument gets instantiated, all the occurrences of that variable in the unit are replaced accordingly.

For instance if the variable `NAME` gets instantiated with `bill`, it is as if the following changes had occurred:

```
:-unit(employee(bill, POSITION)).

name(bill).
position(POSITION).

item :- employee(bill, POSITION).
employee(bill, ta).
employee(joe, ap).
```

Suppose also that each employee's position has an associated index (integer) that will be used to calculate the salary. Such a relation can be easily expressed by the following unit `index`:

```
:- unit(index(POSITION, INDEX)).

position(POSITION).
index(INDEX).

item :-
        index(POSITION, INDEX).

index(ta, 12).
index(ap, 20).
```

With the units above we can build the program `P = {employee, index}`.

Given that in the same program we can have two or more units with the same name but different arities, to be more precise besides the unit name we should also refer its arity i.e. the number of arguments. Nevertheless, since most of the times there is no ambiguity, we omit the arity of the units, without loss of generality. If we consider that

`employee` and `index` designate sets of clauses, then the resulting program is given by the union of these sets.

Contexts are implemented as lists of unit descriptors and each computation has a notion of its *current context*. The program denoted by a particular context is the union of the predicates that are defined in each unit. Moreover, we resort to the *override semantics* to deal with multiple occurrences of a given predicate: only the topmost definition is visible.

To construct contexts, we have the *context extension* operation denoted by `:>`. The goal U `:>` G extends the *current context* with unit `U` and resolves goal `G` in the new context. For instance to obtain Bill's position we could do:

```
?- employee(bill, P) :> item.

P = ta
```

In this query we extend the initial empty context `[]`[3] with unit `employee` obtaining context `[employee(bill, P)]` and then resolve query `item`. This leads to `P` being instantiated with `ta`.

Suppose also that the employee's salary is obtained by multiplying the index of his position by the base salary. To implement this rule consider the unit `salary`:

```
:-unit(salary(SALARY)).

item :-
        position(P),
        [index(P, I)] :< item,
        base_salary(B),
        SALARY is I*B.

base_salary(100).
```

The unit above introduces a new operator (`:<`) called *context switch*: goal `[index(P, I)] :< item` invokes `item` in context `[index(P, I)]`. To better grasp the definition of this unit consider the goal:

```
?- employee(bill, P) :> (item, salary(S) :> item).
```

Since we already explained the beginning of this goal, lets see the remaining part. After `salary/1` is added, we are left with the context `[salary(S), employee(bill,ta)]`.

[3]In the GNU Prolog/CX implementation the empty context its not entirely *empty* since it contains all the standard Prolog built-in predicates such as `=/2`.

The second `item` is evaluated and the first matching definition is found in unit `salary`. Goal `position(P)` is called and because there is no rule for this goal in the current unit (`salary`), a search in the context is performed. Since `employee` is the topmost unit that has a rule for `position(P)`, this goal is resolved in the (reduced) context `[employee(bill, ta)]`. In an informal way, we queried the context for the position of whom we want to calculate the salary, obtaining `ta`. Next, the index corresponding to such position is computed, i.e. `[index(ta, I)] :< item` obtaining `I = 12`. Finally, to calculate the salary, we just need to multiply the index by the base salary, getting `S = 1200` as answer and `[salary(1200), employee(bill, ta)]` as the final context.

5.8 Conclusions

In this chapter we reviewed the modular logic language called Contextual Logic Programming, namely its syntax and semantics (not only the operational but also the declarative/fix-point semantics). This thorough overview is motivated by the fact that CxLP is the modular logic language used as a basis for this work.

We showed one possible optimisation of CxLP by means of abstract interpretation. It is our opinion that implementing this optimisation (see [CLM96]) would incur a low overhead and allow any implementation to avoid resolving goals bound to fail because of undefined predicates.

Finally, we reviewed the implementations CSM and GNU Prolog/CX. A much deeper description of GNU Prolog/CX is given (including a tutorial and reference manual in Appendix A) because this system encompasses more features of CxLP and has better performance than CSM.

Chapter 6

Temporal Reasoning in a Modular Language

This chapter merges the languages Contextual Logic Programming and Temporal Annotated Constraint Logic Programming. It presents the constraint theory of this combination together with its operational semantics. Afterwards, an interpreter together with a compilation scheme are described. Finally, some comparisons with the related approach Multi-theory Temporal Annotated Constraint Logic Programming are brought forward.

6.1 Introduction

It is our belief that not only is modularity a mechanism essential for real world applications, but also that the ability to represent and reason about temporal information is obligatory for the vast majority of domains. Therefore the combination of these two paradigms comes as a natural requisite for any language suitable for designing and implementing Organisational Information Systems.

In this chapter we propose we propose to merge consolidated proposals for temporal reasoning and modularity into a common language. More specifically, we combine Temporal Annotated Constraint Logic Programming (an overview of TACLP was presented in Sect. 2.3.4) with Contextual Logic Programming (an overview of CxLP was presented in Chap. 5).

The choice for the modular logic language CxLP was motivated by the recent work of Abreu and Diaz [AD03b] that presented a revised specification of CxLP, together with a native-code compiler for it, GNU Prolog/CX, itself based on GNU Prolog [DC01]. Besides providing a usable initial performance, this proposal introduces an enhancement over the original Contextual Logic Programming of Monteiro and Porto which

has far-reaching consequences: the parametrisation of units and the use of unit-wide logic variables.

TACLP was chosen because the annotations make time explicit but avoid the proliferation of temporal variables and quantifiers of the first-order approach [BMRT01]. Also of great importance was the fact that, besides an interpreter, there is also a compiler from TACLP to CLP, allowing its use in real world applications. Finally, the language Multi-theory Temporal Annotated Constraint Logic Programming (MuTACLP) (see Sect. 6.6.1) is one of the few languages that combines temporal reasoning (TACLP also) with a mechanism for structuring programs and combining different knowledge source in the style of Brogi et al. [BMPT94]. Therefore, taking preference for TACLP makes the comparisons with similar approaches more clear.

This chapter is organised as follows: we start by building in a incremental way the language that extends CxLP with temporal annotations. We also present several illustrative examples (Sect. 6.2). Afterwards, we deal with the semantics of this proposal, more specifically, the constraint theory (Sect. 6.3) and the operational semantics (Sect. 6.4). An implementation with an instantiation for the time point domain follows in Sect. 6.5. Then we give some comparisons between our framework and related work (Sect. 6.6) and finally, we present some conclusions in Sect. 6.7.

6.2 CxLP with Temporal Annotations

As mentioned, Temporal Annotated Constraint Logic Programming is an instance of annotated constraint logic (ACL), designed for temporal reasoning. The inference rules for annotated formulas were given in Sect. 2.3.4 (page 13).

Following an incremental approach, in order to combine temporal annotations with Contextual Logic Programming we start by adding constraints to CxLP turning into what can be designated as CCxLP (Constraint Context Logic Programming), i.e. we add a distinguished class of predicates called *relational constraints* and a distinguished class of of interpreted functions called *functional constraints.* Afterwards, we allow formulas to be annotated with a distinguished class of constraint terms called *annotation terms* obtaining what will be called Annotated Constraint Contextual Logic Programming. In this phase, as in ACL, we have the partial ordering $\sqsubseteq$ as a relational constraint and the least upper bound $\sqcup$ as a functional constraint, along with a constraint theory (see [Frü94b]) as well as inference rules for annotations (already presented in Sect. 6.6.1). Instantiating to the temporal case, we consider the annotated formulas `A at t` as meaning that the formula `A` is true at time point `t`[1] and relate

[1] "Indivisible, duration-less instant or moment in time" [Frü94b]

periods to convex sets of points by the annotations `A th I` and `A in I`, meaning that `A` holds throughout the set `I` (i.e. at *every* time point in `I`) and that `A` holds at *some* time point(s) in `I`, respectively. The resulting formalism will be called *Temporal Annotated Constraint Contextual Logic Programming* (or TACCxLP for short).

Leaving aside the (general) constraints and focusing only on the temporal aspects, from a syntactical point of view we add the possibility of CxLP atoms having one of the following temporal annotations: `{at t, th I, in I}`. As an illustration of this extension consider one possible temporal version of the unit `employee` from the example presented in Sect. 5.7.2 (page 63):

```
:- unit(employee).

employee(bill, ta) th [2004, inf].
employee(joe, ta) th [2002, 2006].
employee(joe, ap) th [2007, inf].
```

The main difference is that now each basic fact has one temporal annotation. For instance the last fact of `employee` states that `joe` is an `ap` (associate professor) since 2007 (the `inf` stands for ∞, i. e. a time point that is later than any other). Moreover, in this case there is no use for unit arguments. In a similar way we can define unit `index` to account for the temporal evolution of the careers salary index (which could be updated to keep up with inflation):

```
:- unit(index).
index(ta, 10) th [2000, 2005].
index(ta, 12) th [2006, inf].
index(ap, 19) th [2000, 2005].
index(ap, 20) th [2006, inf].
```

The temporal version of unit `salary` has the following structure:

```
:- unit(salary).
salary(N, S) th J :-
                [employee] :< employee(N, P) th J,
                [index] :< index(P, I) th J,
                base_salary(B), S is B*I.

base_salary(100).
```

and states that the rule to calculate the salary is time dependent: the salary through a given interval J is calculated by querying the employee position and corresponding index during such interval. Then, the salary is obtained by multiplying this index by

the base salary.[2] As an illustration consider the following goal to discover `joe`'s salary `in` [2005, 2007]:

```
?- salary :> salary(joe, S) in [2005, 2007].

S = 1000; S = 1200; S = 2000
```

The reader should notice that the salary of a *teaching assistant* (`ta`) was raised in 2006 (`S = 1000` and `S = 1200`) and that `joe` achieved the position of *associate professor* (`ap`) in 2007 (`S = 2200`).

6.3 Constraint Theory

The constraint theory is identical to the one of TACLP [BMRT02], i.e. a constraint domain for time points includes suitable constants for time points, function symbols for operations on time points (e.g., `+,-, ...`) and the predicate symbol $\leq$, modelling the total order relation on time points. This constraint domain is extended to a constraint domain $\mathcal{A}$ for handling *annotations*, by enriching the signature with function symbols `[., .], at, th, in,` $\sqcup$ and the predicate symbol $\sqsubseteq$. Moreover, assuming $r_1 \leq s_1$, $s_1 \leq s_2$ and $s_2 \leq r_2$, the axioms for the predicate symbol $\sqsubseteq$ can be summarised as:

$$in[r_1, r_2] \sqsubseteq in[s_1, s_2] \sqsubseteq in[s_1, s_1] = at\ s_1 = th[s_1, s_1] \sqsubseteq th[s_1, s_2] \sqsubseteq th[r_1, r_2]$$

Finally, the axioms of the least upper bound $\sqcup$ (function symbol) can be restricted to:[3]

$$th[s_1, s_2] \sqcup th[r_1, r_2] = th[s_1, r_2] \Leftrightarrow s_1 < r1, r_1 \leq s_2, s_2 < r_2$$

6.4 Operational Semantics

To obtain the operational semantics of this language one needs to replace the inference rules ($\sqsubseteq$) and ($\sqcup$) of Sect. 2.3.4 by their contextual versions, i.e. instead of annotated atoms $A\alpha$ we must have $C \vdash A\alpha$.

$$\frac{C \vdash A\alpha \quad \gamma \sqsubseteq \alpha}{C \vdash A\gamma}\ \text{rule}\,(\sqsubseteq_C) \qquad \frac{C \vdash A\alpha \quad C \vdash A\beta \quad \gamma = \alpha \sqcup \beta}{C \vdash A\gamma}\ \text{rule}\,(\sqcup_C)$$

[2] Of course that it would be reasonable to annotate the base salary. However, doing so wouldn't bring any novelty to the problem.

[3] The least upper bound only has to be computed for overlapping *th*–annotations.

Moreover, the inference rules of CxLP must be adapted to the case where literals can be annotated. These rules can be obtained in a rather straightforward manner from the ones for CxLP (see Sect. 5.3 on page 55) by substituting each goal G by its annotated version $G\alpha$.

6.5 Interpreter and Compiler

Meta-interpretation and compilation are two widely used techniques to implement logic languages. Frühwirth [Frü96] provides both an interpreter and a compiler for TACLP where the compiler transforms TACLP into CLP. Moreover, the same author in [Frü94b] presents an efficient optimised interpreter for TACLP were only atomic formulas can be annotated.

An obvious implementation of the proposed language can be obtained with a slight modification of the compiler (interpreter) for TACLP in order to handle context operations (leaving them unchanged). Therefore, to get a full blown implementation it is sufficient to define the constraint domain $\mathcal{C}$ for the TACLP, i.e. one has to define the time point domain.

6.5.1 Time Point Domain

The main motivation behind our temporal data model was to be able to represent a wide variety of notions of time, not only common ones such as the 24–Hour timekeeping system or the Gregorian calendar but also more specialised ones such as time in digital circuits. As we will see in this section, Constraint Logic Programming [JL87] specialised to Finite Domains (CLP(FD)) and reified Booleans (CLP($\mathcal{B}$)) turns out to be a very suitable framework: the size of the temporal variables domain admits an efficient implementation with Finite Domains and the reified Booleans allows us to express elaborated temporal constraints. Moreover, since CLP(X) is so pervasive to this work, an overview of the main characteristics of this paradigm can be found in Appendix B.

Although constraints are widely used for temporal reasoning, their application in temporal information representation is rare. One of the exceptions is the work of Kabanza et. al [KSW90] with infinite temporal information. Here we present another way of using constraints to represent temporal data: by interpreting a Constraint Satisfaction Problem as an intensional way of specifying a set of time points, i.e. as a time point domain:

Definition 8 (Time Point Domain) *A time point domain is a Constraint Satisfaction Problem on finite domains.*

If n is the number of variables of the CSP we say that the domain is n-dimensional. A time point is a tuple that is solution of the CSP:

Definition 9 (Time Point) *Given a time point domain D on variables $\{X_1, \ldots, X_n\}$ we say that the tuple $X = (x_1, \ldots, x_n)$ is a time point of D if the assignment $\{X_1 = x_1, \ldots, X_n = x_n\}$ satisfies all the constraints of D.*

With this representation the domain D_{24} can be formulated as:

Example 4 (24–Hour) *The CSP $D_{24} = H \in [0, 23] \wedge M \in [0, 59]$ on variables $\{H, M\}$ is a time point domain representing the 24–Hour system. The tuples $x = (10, 30)$ and $y = (13, 20)$ are examples of time points of this domain.*

Finally it is rather straightforward, although not as trivial as the 24–Hour, to use a CSP to represent a calendar (Gregorian, Julian, etc).

Considering there are benefits in dealing with each of the tuple members instead of the tuple as a whole, we modified the temporal variables to accommodate this:

Definition 10 (Time Variable) *We say that X is a time variable of the n-dimensional time point domain D iff X is a n-tuple of CLP(FD, $\mathcal{B}$) variables.*

For instance $X_1 = (H_1, M_1)$ and $X_2 = (H_2, M_2)$ are variables of D_{24}. With this representation for variables it is easy to express constraints that correspond to the equal, before and after relations. For instance, $X_1 < X_2$ corresponds to the constraint $H_1 < H_2 \vee (H_1 = H_2 \wedge M_1 < M_2)$. Note that this constraint may be expressed using CLP(FD, $\mathcal{B}$).

Finally, since GNU Prolog/CX, besides providing the CxLP primitives, also has a constraint solver over finite domains and reified booleans (CLP(FD, $\mathcal{B}$)), making the implementation of this language relatively straightforward.

Relations Between Different Time Point Domains

Applications such as Heterogeneous Information Systems have to deal with temporal data described in different domains. There are several reasons for this state of things such as:

- information relative to different timezones or calendars (Julian, Gregorian and Chinese lunar calendar);

- information from different domains that *share* temporal units. For instance a newspaper is characterised by $\{Year, Month, Day\}$ and a specific news can be further described by $Hour$ and $Minute$.

Therefore there is a strong need to establish (whenever possible) temporal relations between these different time domains. To have this capability in our framework we just need to add a *function* that converts a time point from one domain to another.

A simple illustration of such function is the one that relates the 24–Hour domain with seconds (denoted by D_{24_s}) and the D_{24}:

$$\begin{array}{rccc} f: & D_{24_s} & \longrightarrow & D_{24} \\ & (h, m, s) & \longrightarrow & (h, m) \end{array}$$

Another basic example is the one derived from the problem of conversion of timezones. For that consider the domain D_{+1} on variables $\{Y, Mo, D, H, Mi\}$ for time in the timezone GMT+1 and a similar D_{+2} to express time in the timezone GMT+2. The following function converts time points from D_{+1} to D_{+2}:

$$\begin{array}{rccc} g: & D_{+1} & \longrightarrow & D_{+2} \\ & (y, mo, d, h, mi) & \longrightarrow & \begin{cases} (y, mo, d, h+1, mi) & \text{if } h < 23 \\ (y, mo, d+1, 0, mi) & \text{if } d < max\_day(y, mo) \\ \ldots \end{cases} \end{array}$$

Please notice that, although the inverse of g is still a function, this doesn't happen for f because, to each time point of D_{24} corresponds a set of time points of D_{24_s}: for instance, to the point $(14, 20)$ corresponds the set of points $\{(14, 20, 0), \ldots, (14, 20, 59)\}$.

If there is at least one conversion function between two domains, it is easy to relate ($=, <$ or $>$) points on these domains. The procedure its quite simple and can be described in two steps: convert the points to the same domain and apply the corresponding domain relation.

As an illustration, consider the point $(16, 20)$ of D_{24} and $(15, 30, 12)$ from D_{24_s}. The relation $(15, 30, 12) < (16, 20)$ its valid because $f(15, 30, 12) = (15, 30)$ and $(15, 30) < (16, 20)$ in the domain D_{24}.

6.6 Related Work

In this section we describe other formalisms that combine temporal reasoning with modularity. Since Multi-theory Temporal Annotated Constraint Logic Programming

(MuTACLP) is the language that is nearer to our proposal, a more indepth comparison with this language is presented.

6.6.1 MuTACLP

MuTACLP [BMRT02] is a language for modelling and handling temporal information together with some basic operators for combining different temporal knowledge bases. In this language, temporal information can be naturally represented and handled and, at the same time, knowledge can be separated and combined by means of meta-level composition operators. For a more comprehensive reading see [MRT97, MRT99, MRT99, Raf00, MNRT00, BMRT01].

The operators for combining theories follows the work of Brogi et al. [BMPT94]. The language of program expressions *Exp* is defined by the following abstract syntax:

$$Exp \ ::= \ Pname \mid Exp \cup Exp \mid Exp \cap Exp$$

where *Pname* is the syntactic category of constant names for plain programs.

The main difference towards our proposal is that, instead of the programming-in-the-large approach for modularity followed by MuTACLP, we resort to the programming-in-the-small of CxLP. In the following section, and for a better comparison, we illustrate both languages proposals for a specific legal reasoning problem.

Example: the British Nationality Act

The following example was taken from the British Nationality Act and was presented in [BMRT02] to exemplify the language MuTACLP. The reason to use an existing example is twofold: not only do we consider it to be a simple and concise sample of legal reasoning but also because this way we can give a more thorough comparison with MuTACLP. The textual description of this law can be given as a person `X` obtains the British Nationality at time `T` if all the conditions below are true:

- `X` is born in the UK at the time `T`
- `T` is after the commencement
- `Y` is a parent of `X`
- `Y` is a British citizen or resident at time `T`.

Assuming that *Jan 1 1955* is the commencement date of the law, the MuTACLP program for this problem is:

```
BNA:
get_citizenship(X) at T <- T >= Jan 1 1955,
                           born(X, uk) at T,
                           parent(Y, X) at T,
                           british_citizen(Y) at T.

get_citizenship(X) at T <- T >= Jan 1 1955,
                           born(X, uk) at T,
                           parent(Y, X) at T,
                           british_resident(Y) at T.
```

Moreover, those authors also assume a separate program to encode the data for a given person John whose parent Bob is a British citizen since *Sep 6 1940*:

```
JOHN:
born(john, uk) at Aug 10 1969.
parent(bob, john) th [T, inf] <- born(john, _) at T.
british_citizen(bob) th [Sep 6 1940, inf].
```

Finally, the citizenship of John can be queried as:

```
demo(BNA ∪ JOHN, get_citizenship(john) at T)
```

and the result is `T = Aug 10 1969`.

Our proposal to solve this problem is a more modular way since we will opt to define one unit for each *predicate*. More specifically, unit `born/2` represents the name and country where a person was born:

```
:- unit(born(Name, Country)).
born(john, uk) at 'Aug 10 1969'.
item at T :- born(Name, Country) at T.
```

It this unit (and in the subsequent one) we present dates as atoms such as `'Aug 10 1969'`, this is only for the reader's confort since, as mentioned in Sect. 6.5 we resort to CLP(FD, $\mathcal{B}$) to implement the temporal elements. Moreover, in order to provide an implementation coherent with the current CxLP proposal we also define predicate `item` to instantiate the units arguments.

The unit `british_citizen/1`[4] states when a person started to be a British citizen:

[4]In a similar way we could have defined an analogous unit `british_resident/1`.

```
:- unit(british_citizen(Name)).
british_citizen(bob) th ['Sep 6 1940', inf].
item th T :- british_citizen(Name) th T.
```

and unit `parent(Parent, Son)` expresses the parent/child relationship:

```
:- unit(parent(Parent, Son)).
parent(bob, john).
item :- parent(Parent, Son).
```

Remembering that *Jan 1 1955* is the commencement date of the law and that a person `X` obtains the British Nationality at time `T` if all the conditions below are true:

- `X` is born in the UK at the time `T`
- `T` is after the commencement
- `Y` is a parent of `X`
- `Y` is a British citizen or resident at time `T`.

one formalisation of this law in our language is:

```
:- unit(bna).

get_citizenship(X) at T :-
        born(X, uk) :> item at T,
        'Jan 1 1955' #=< T,
        parent(Y, X) :> item,
        (british_citizen(Y) :> item at T
         ;
         british_resident(Y) :> item at T).
```

The explanation of this rule is quite simple because each line of the clause body corresponds to and is presented in the same order as in the textual description of the law.

Finally, the goal `bna :> get_citizenship(john) at T` also yields `T = Aug 10 1969`.

6.6.2 Other Approaches

This combination of modularity and temporal reasoning is not frequent on logical temporal languages. Two exceptions are the language Temporal Datalog by Orgun [Org96]

and the work on amalgamating knowledge bases by Subrahmanian [Sub94]. Temporal Datalog introduces the concept of module which, however, seems to be used as a means for defining new non-standard algebraic operators, rather than as a knowledge tools. On the other hand, the work on amalgamating knowledge bases offers a multi-theory framework, based on *annotated logics*, where temporal information can be handled, but only a limited interaction among the different knowledge sources is allowed: essentially a kind of message passing mechanism allows one to delegate the resolution of an atom to other databases.

6.7 Concluding Remarks

In this chapter we added temporal reasoning capabilities to a modular logic language. More specifically, we joined the Temporal Annotated Constraint Logic Programming with Contextual Logic Programming. The aim of this proposal is to have both formalisms in the same language, without defining any sort of relation between temporal reasoning and modularity.

Besides defining the language, we also provided an operational semantics. Moreover, we sketched a compiler for the proposed language and specified a possible implementation for a specific time point domain. Finally, we reviewed related work on this subject, in particular, the MuTACLP language that also combines the chosen temporal formalism with another modular logic language.

Chapter 7

Temporal Contextual Logic Programming

This chapter presents a temporal extension of CxLP, called Temporal Contextual Logic Programming (TCxLP). It describes the language syntax, operational semantics and a procedure that computes the last upper bound of the temporal annotations. Afterwards, an interpreter and a compilation scheme are given. Finally, this language is applied to several domains and compared with similar approaches.

7.1 Introduction

One possible way of devising a language with modularity and temporal reasoning is to consider that these two characteristics can co-exist without any direct relationship. This was the approach followed in Chap. 6. Nevertheless we can also conceive a scenario where those concepts are more integrated or more related. In this chapter we take the interaction between modularity and time to the next level by considering that the usage of a module is influenced by temporal conditions. The intuition for this proposal came from common sense reasoning where it is usual to consider that the *usage* of (part of) *knowledge* (law, criteria) can be time dependent.

Contextual Logic Programming structure is very suitable for integrating with temporal reasoning since, as we shall see, it is quite straightforward to add the notion of *time of the context* and let that time help in deciding if a certain module is eligible or not to solve a goal. Regarding the temporal paradigm we shall continue to use TACLP, not only because of the reasons enumerated in Sect. 6.1, but also because, this way, the original contribution stands out.

This chapter is organised as follows: we start incrementally building a language that extends CxLP with temporal annotations (Sect. 7.2). An algorithm to compute the

least upper bound of annotations is described in Sect. 7.3. Afterwards, we deal with the semantics of this proposal, more specifically, the constraint theory and the operational semantics (Sect. 7.4). An implementation with an instantiation for the time point domain (see page 71) follows in Sect. 7.5. Applications to several domains is presented in Sect. 7.6 and comparisons between our framework and related work follows (Sect. 7.7). Finally, we provide some conclusions in Sect. 7.8.

7.2 Language of TCxLP

The basic mechanism of CxLP is called *context search* and can be described as follows: to solve a goal G in a context C, a search is performed until the topmost unit of C that contains clauses for the predicate of G is found. We propose to incorporate temporal reasoning into this mechanism. To accomplish this we add temporal annotations to not only the dynamic part of CxLP (*contexts*) but also to the static one (*units*) and it will be the relation between those two types of annotations that will determine if a given unit is eligible to match a goal during a context search.

7.2.1 Annotating Units

Considering the initial unit definition (i.e. without parameters), adding temporal annotations to these units could be done simply by annotating the unit name. Using an extended GNU Prolog/CX syntax, an annotated unit could be defined as:

```
:- unit(foo) th [1,4].
```

Nevertheless, units with arguments ought to allow for a refinement of the temporal qualification, i.e. we can have several qualifications, one for each possible argument instantiation. As an illustration consider the following example:

Example 5 *Temporal annotated unit* `bar/1`*:*

```
:- unit(bar(X)).
bar(a) th [1,2].
bar(b) th [3,4].
```

where the annotated facts state that unit `bar` with its argument instantiated to `a` has the annotation `th [1,2]` and with its argument instantiated to `b` has the annotation `th [3,4]`, respectively. Moreover, it should be clear that this is more expressive than the proposal at the beginning of this section since it is still possible to annotate the

unit most general descriptor (for the definition of unit descriptors see *Parametrised Units* in Sect. 5.5) :

```
:- unit(foo).
foo th [1,4].
```

As we saw in the example above, the unit descriptor is the same as the annotated predicate, `foo`. Such overloading is intentional and makes it easier to express whether a given unit (descriptor) in a context satisfies a temporal condition (annotated predicate). This relation will be made precise when we present the operational semantics (see inference rule 7.5 in page 88).

Therefore we propose to temporally annotate the unit descriptors and those annotated facts are designated as the unit *temporal conditions.*

7.2.2 Temporal Annotated Contexts

The addition of time to a context is rather simple and intuitive: instead of a sequence of unit descriptors C we now have a temporally annotated sequence of units descriptors $C\Delta$, where Δ is a temporal annotation of the type `at t, th I` or `in I` (see page 13). This annotation Δ is called the *time of the context* and by default contexts are implicitly annotated with the current time. As the name implies, the *time of the context* specifies the time when a given context should be considered, making all the computations relative to that time.

Recall that in GNU Prolog/CX a context is implemented as list of units descriptors such as `[u1(X), u2(Y,Z)]`: a temporal annotated context can be for instance `[u1(X), u2(Y,Z)] th [1,4]`.

7.2.3 Relating Temporal Contexts with Temporal Units

Although the relation between temporal annotated contexts and units will be made precise when we present the semantics (Sect. 7.4), in this section we illustrate what we mean when we say that the relation between those two types of annotations (context and units) determines if a given unit is eligible to match a goal during a context search.

Contexts for a Simple Temporal Unit

Consider the unit `bar/1` of Example 5 (page 80). Roughly speaking, this unit will be eligible to match a goal in a context like:

```
[..., bar(a), ...] in [1,4]
```

if "`bar(a) in [1,4]`" can be proved.

Since one of the unit temporal conditions is "`bar(a) th [1,2]`" and we known that `in [1,4]` $\sqsubseteq$ `th [1,2]`, then by the inference rule ($\sqsubseteq$) (see page 14) one can derive "`bar(a) in [1,4]`". In a similar way we say that this unit its not eligible in the following context (recall that `th [3,6]` $\not\sqsubseteq$ `th [3,4]`):

```
[..., bar(b), ...] th [3,6]
```

Conversely, suppose that unit `bar/1` has some definition for the predicate of an atomic goal `G`, then in order to use this definition in a goal like:

```
?- [bar(X), ...] in [1,4] :< G
```

besides the instantiations for the variables of `G`, one also obtains bindings for the unit arguments:

`X = a` or `X = b`.

University Employees Example: Temporal Version

Revisiting the University employees example (already seen in page 63 to illustrate CxLP and in page 68 to exemplify CxLP with temporal annotations), unit `employee` with temporal information can be written as:

```
:- unit(employee(NAME, POSITION)).

name(NAME).
position(POSITION).

item.

employee(bill, ta) th [2004, inf].
employee(joe, ta) th [2002, 2006].
employee(joe, ap) th [2007, inf].
```

This way it is possible to represent the *history* of the employees positions: `joe` was a teaching assistant (`ta`) between 2002 and 2006. The same person is a associate professor (`ap`) since 2007. Moreover, in this case the rule for predicate `item/0` doesn't need to be "`item :- employee(NAME, POSITION).`" because the goal `item` is true only if the unit is (temporally) eligible and, for that to happen, the unit arguments must be

instantiated. To better understand this example, consider the goal that queries joe's position throughout [2005,2006]:

```
?- [employee(joe, P)] th [2005,2006] :< item.
```

the evaluation of item is true as long as the unit is eligible in the current context, and this happens when P is instantiated with ta (teaching assistant), therefore we get the answer P = ta.

In a similar way we can define a temporal version of unit index/2 as:

```
:- unit(index(POSITION, INDEX)).

position(POSITION).
index(INDEX).

item.

index(ta, 10) th [2000, 2005].
index(ta, 12) th [2006, inf].
index(ap, 19) th [2000, 2005].
index(ap, 20) th [2006, inf].
```

to express the evolving index associated with each position. Unit salary can be defined as:

```
:- unit(salary(SALARY)).
item :-
        position(P),
        index(P, I) :> item,
        base_salary(B),
        SALARY is B*I.

base_salary(100).
```

There is no need to annotate the goals position(P) or index(P, I) :> item since these are evaluated in a context which already has the same temporal annotation. To find out joe's salary in 2005 we can say:

```
?- [salary(S), employee(joe, P)] at 2005 :< item.
```

which yields the bindings:

```
P = ta
S = 1000
```

Since `salary` is the topmost unit that defines a rule for `item/0`, the body of the rule for this predicate is evaluated. In order to use the unit `employee(joe, P)` to solve `position(P)`, the unit must satisfy the temporal conditions "`at 2005`", that in this case means instantiating `P` with `ta`, therefore we obtain `position(ta)`. A similar reasoning applies for goal "`index(ta, I) :> item`", i.e. this `item` is resolved in context "`[index(ta, 10), salary(S), employee(joe, ta)] at 2005`". The remainder of the rule body is straightforward, leading to the answer substitution `P = ta` and `S = 1000`.

7.3 Computing the Least Upper Bound

One obvious drawback stemming from incorporating temporal reasoning into context search is the extra computation that results from processing the units/contexts temporal annotations. Moreover, since context search is a basic mechanism, extra care must be taken to minimise this overhead.

In order to lighten the process, we propose to calculate, at compile time, the least upper bound ($\sqcup$) of the units descriptors temporal annotations. This way context only needs to verify the satisfaction of the relation between *annotations* of the context and that of the units, i.e. only relation $\sqsubseteq$ is left for runtime.

According to [BMRT02] it suffices to consider the least upper bound (or lub for short) for time periods that produce another *different meaningful* time period, i.e. one may consider only overlapping time periods that do not include one another (see Sect. 6.3). In this section we present a procedure that iterates over the set of each unit th-annotated descriptors, calculating the lub of unifiable atoms with th-overlapping intervals, inserting the output of the computation and (possibly) removing (some of) the inputs. This procedure is presented in an incremental way, starting with ground annotated unit descriptors and afterwards we handle the non ground case.

7.3.1 Ground Temporal Conditions

In this case the set of temporally annotated unit descriptors is not only finite (by definition) but also ground. Before giving a formal description let us consider an example that provides the general intuition behind this procedure. To that purpose, consider the unit `baz/1`:

```
:- unit(baz(X)).
baz(a) th [1,4].
baz(a) th [3,7].
```

```
...
```

Since `th [1,4]` and `th [3,7]` are two overlapping th-annotations for the same unit descriptor (`baz(a)`), we compute their lub, obtaining `th [1,7]`. Therefore, we can replace these two temporal conditions by a more general one:

```
:- unit(baz(X)).
baz(a) th [1,7].
...
```

A formal description of this procedure is given in Algorithm 1 (page 85). Basically, the algorithm iterates over the set of units of a given program and for each pair of th-annotated temporal conditions that have the same unit descriptor (u) and overlapping intervals (I_1 overlaps I_2), remove that pair from the unit temporal conditions and insert one with the same unit descriptor and the annotation that results from the computing the lub of those annotations, i.e. insert a new temporal condition that subsumes the previous two.

One can easily see that this procedure terminates not only because the set of units is finite (for loop) but also due to the fact that each iteration of the while loop decreases the size of the said set, the set of th-annotated temporal conditions of a unit.

Algorithm 1 Computing the least upper bound of ground th-annotations

Let $\{u_1, \ldots, u_n\}$ be the set of units of a given program P and Th_{u_i} the set of th-annotated temporal conditions of unit u_i (that, by hypothesis, are all ground):

1. For each unit $u_i \in P$
   (a) while ($\exists$ u th I_1, u th $I_2 \in Th_{u_i}$: I_1 overlaps I_2)
      i. remove u th I_1 and u th I_2 from Th_{u_i}
      ii. insert u th $(I_1 \sqcup I_2)$ into Th_{u_i}

7.3.2 Non Ground Temporal Conditions

In this section we provide a generalisation of Algorithm 1 to non ground th-annotated temporal conditions. As a simple illustration consider unit `foo/2`:

```
1 :- unit(foo(X, Y)).
2 foo(_, b) th [1, 3].
3 foo(a, _) th [4, 6].
4 foo(a, b) th [5, 8].
```

Previously, a unit u with parameter variables $\overrightarrow{p}$ was a set of clauses, for which all variables in $\overrightarrow{p}$ are implicitly existentially quantified over all clauses of the unit. The same existential quantification also applies to the unit temporal conditions. Therefore, in the example above there is no need to name the variables in the temporal conditions (lines 2 and 3) since the first argument of `foo` already coincides with `X` and the second with `Y`, i.e. the unit arguments.

Before describing the procedure for the non ground case, we have yet to define what we mean by an annotated atom being subsumed by a set of the same elements:

Definition 11 ($\vdash_{\sqsubseteq}$) *Consider that S is a set of temporal annotated atoms. We say that A th I is subsumed by the set S and denote by $S \vdash_{\sqsubseteq} A$ th I if and only if there is a B th J $\in$ S and a substitution θ such that $A = B\theta$ and $I \sqsubseteq J$, i.e. if there is an annotated atom in S that is at least as informative as A th I.*

The procedure for this case is formally presented as Algorithm 2 (page 86) and can be described as: for every unit, find two th-annotated temporal conditions that overlap (I_1 overlaps I_2) and whose unit descriptors are unifiable (θ =mgu(U_1, U_2)). Them (possibly) remove those temporal conditions and insert a new one with the annotation that results from computing the lub ($I_1 \sqcup I_2$) and *unified* descriptor ($U_1\theta$). Moreover, the body of the *while* loop is executed if the temporal condition that can be generated isn't already subsumed by the set of th-annotated temporal conditions ($Th_{u_i} \not\vdash_{\sqsubseteq}$ ($U_1\theta$) th($\mathrm{I}_1 \sqcup \mathrm{I}_2$)).

Algorithm 2 Computing the least upper bound th-annotations

Let $\{u_1, \ldots, u_n\}$ be the set of units of a given program P and Th_{u_i} the set of th-annotated temporal conditions of unit u_i:

1. For each unit u_i
   (a) while ($\exists$ U_1 th I_1, U_2 th $I_2 \in Th_{u_i}$, θ = mgu(U_1, U_2): I_1 overlaps I_2 and $Th_{u_i} \not\vdash_{\sqsubseteq}$ ($U_1\theta$) th($\mathrm{I}_1 \sqcup \mathrm{I}_2$)
      i. if U_1 (U_2) is ground, remove U_1 th I_1 (U_2 th I_2) from Th_{u_i}.
      ii. insert $U_1\theta$ th ($I_1 \sqcup I_2$) into Th_{u_i}.

For a better understanding, let us consider unit `foo/2` above. According to this algorithm one possible evolution of the set $Th_{foo/2}$ is:

1. {`foo(_, b) th [1, 3], foo(a, _) th [4, 6], foo(a, b) th [5, 8]`}.
2. {`foo(_, b) th [1, 3], foo(a, _) th [4, 6], foo(a, b) th [1, 6], foo(a, b) th [5, 8]`}.

3. {foo(_, b) th [1, 3], foo(a, _) th [4, 6], foo(a, b) th [1, 8]}.

The first iteration computes the lub of the {foo(_, b) th [1, 3]} and {foo(a, _) th [4, 6]}, yielding {foo(a, b) th [1, 6]}. In the next iteration, the lub of the inserted annotation ({foo(a, b) th [1, 6]}) and {foo(a, b) th [5, 8]} is computed, yielding {foo(a, b) th [1, 8]}.

Like the algorithm for the ground case, we also iterate over a finite set of units (for loop), nevertheless showing the termination of the while loop requires more explanation since now the set of th-annotated temporal conditions may not decrease. In fact, as we saw in the example above, the first iteration increases the set and the second decreases. Before explaining the termination the reader should keep in mind that the unit descriptor is a finite term and the set of temporal conditions is also finite. Moreover, the temporal periods of the th-annotations are of the form [T1, T2] where T1 and T2[1] are integers, greater or equal than 0, with T2 $\geq$ T1. Therefore, the number of temporal conditions that can result from combining overlapping intervals or by specialising descriptors is finite. Since each iteration of the while loop introduces a new temporal condition (not subsumed by the others) of a finite set, the loop terminates.

7.4 Operational Semantics

Similar to the operational semantics of Contextual Logic Programming (see Sect. 5.3) and as usual in logic programming, we present the operational semantics by means of derivations. We will name and enumerate the inference rules which specify computations. Finally, the paragraph after each rule gives an informal explanation of how it works.

7.4.1 Inference Rules

To define the operational semantics we consider the following notation: C and C' are contexts; T_u is the set of temporal conditions of unit u; θ, σ and ϵ are substitutions; Δ and Λ are temporal annotations and $\emptyset$ and G are goals. Moreover, we also assume that Algorithm 2 (page 86) has been applied to the subset of th-annotated atoms of T_u.

Null goal

$$\frac{}{C\Delta \vdash \emptyset\ [\epsilon]} \tag{7.1}$$

[1] With the possible exception of T2 = inf.

The null goal is derivable in any temporal annotated context, with the *empty* substitution ϵ as result.

Conjunction of goals

$$\frac{C\Delta \vdash G_1\ [\theta] \qquad C\Delta\theta \vdash G_2\theta\ [\sigma]}{C\Delta \vdash G_1, G_2\ [\theta\sigma\lceil vars(G_1, G_2)]} \tag{7.2}$$

To derive the conjunction, derive one conjunct first and then the other, in the same context with the given substitutions.[2]

Since C may contain variables in unit descriptors that may be bound by the substitution θ obtained from the derivation of G_1, we have that θ must also be applied to $C\Delta$ to obtain the updated context in which to derive $G_2\theta$.

Context inquiry

$$\frac{}{C\Delta \vdash :< C'\Lambda\ [\theta]} \begin{cases} \theta = \text{mgu}(C, C') \\ \Lambda \sqsubseteq \Delta \end{cases} \tag{7.3}$$

In order to make the context switch operation (inference rule 7.4) useful, there needs to be an operation which fetches the context. This rule recovers the current context C as a term and unifies it with term C', so that it may be used elsewhere in the program. Moreover, the annotation Λ must be equal to or less informative than the annotation Δ ($\Lambda \sqsubseteq \Delta$).

Context switch

$$\frac{C'\Lambda \vdash G\ [\theta]}{C\Delta \vdash C'\Lambda :< G\ [\theta]} \tag{7.4}$$

The purpose of this rule is to allow execution of a goal in an arbitrary temporal annotated context, independently of the current annotated context. This rule causes goal G to be executed in context $C'\Lambda$.

Reduction

$$\frac{(uC\Delta)\theta \vdash (G_1, G_2 \ldots G_n)\theta\ [\sigma]}{uC\Delta \vdash G\ [\theta\sigma\lceil vars(G)]} \begin{cases} H \leftarrow G_1, G_2 \ldots G_n \in |u| \\ \theta = \text{mgu}(G, H) \\ T_u \vdash_{\sqsubseteq} u\Delta \end{cases} \tag{7.5}$$

This rule expresses the influence of temporal reasoning in the context search mechanism and can be regarded as the temporal version of rule 5.3 (page 56).

[2]The notation $\delta\lceil V$ stands for the restriction of the substitution δ to the variables in V.

Informally it can be described as: when a goal (G) has a definition ($H \leftarrow G_1, G_2, \ldots G_n \in |u|$ and $\theta = \text{mgu}(G, H)$) in the topmost unit ($u$) of the annotated context ($uC\Delta$) and the unit temporal conditions subsume the "time of the context" ($T_u \vdash_{\sqsubseteq} u\Delta$), to derive the goal we must call the body of the matching clause, after unification. The main difference towards the non-temporal version is that we now also check whether the unit is "temporally eligible".

Context traversal:

$$\frac{C\Delta \vdash G[\theta]}{uC\Delta \vdash G[\theta]} \left\{ \; pred(G) \notin \overline{u} \right. \tag{7.6}$$

When none of the previous rules apply and, in particular, when the predicate of G isn't defined in the predicates of u ($\overline{u}$), remove the top element of the context and proceed, i.e. resolve goal G in the supercontext.

Application of the Rules

It is straightforward to verify that the inference rules are mutually exclusive, leading to the fact that, given a derivation tuple $C\Delta \vdash G[\theta]$, only one rule can be applied.

7.5 TCxLP Compiler and Interpreter

In this section we propose a source-to-source program transformation that allows us to convert from Temporal Contextual Logic Programming into CxLP where the unit descriptors can have temporal annotations. Since after this transformation we obtain (a subset of) the language proposed in Chap. 6 we can use the compiler (or interpreter) described in Sect. 6.5 on the resulting program. This way we obtain a compiler (or interpreter) for TCxLP.

7.5.1 From TCxLP to CxLP+TACLP

Our goal is to convert a TCxLP program into CxLP (with annotations), such that this last program expresses the behaviour of context search being influenced by temporal reasoning. Informally, we add to every unit a sort of "frontend" that makes the unit body (the "backend") accessible only if the unit temporal conditions are satisfied by the time of the context, otherwise context search must continue, bypassing the unit.

In the first step of Algorithm 3, throughout renaming the predicates we define the "backend". The access to the renamed predicate (`p'`) is controlled by means of the

Algorithm 3 Program transformation: from TCxLP to CxLP+TACLP

Let $\mathcal{P} = \{u_1, \ldots, u_n\}$ be a TCxLP program and $||\mathcal{P}|| = \bigcup_{i=1}^{n} ||u_i||$, i.e. $||\mathcal{P}||$ is the set of predicates defined in all units. Consider also the set of predicates S such that:

- $\#S = \#||\mathcal{P}||$
- $S \cap ||\mathcal{P}|| = \emptyset$
- for each predicate `p` $\in ||\mathcal{P}||$ there is a `p'` $\in S$

For each unit u_i and predicate `p` defined in this unit:

1. replace each occurrence of `p` by `p'`;
2. insert into u_i a new clause

   ```
   pd :- :< [ud|C] Δ, (ud Δ -> p'd ; C Δ :< pd)
   ```

   where `pd` (`p'd`) and `ud` are the predicate `p` (`p'`) and unit `u` descriptors, respectively.

---

clause introduced in the second step: to resolve a goal (`pd`) whose main functor is `p`, query the current temporal context (`:< [ud|C] Δ`) and if the unit satisfies the temporal conditions (`ud Δ`) then access the "backend" predicate (`p'd`), otherwise continue the context search for the given goal (`C Δ :< pd`).

As an illustration of the procedure above, the following unit

```
:- unit(foo(A,B)).

p(X) :- q(X).
q(a).
```

is transformed into

```
:- unit(foo(A,B)).

p'(X) :- q'(X).
q'(a).

p(X) :- :< [foo(Y, Z)|C] T,
        (foo(Y, Z) T -> p'(X) ; C T :< p(X)).
q(X) :- :< [foo(Y, Z)|C] T,
        (foo(Y, Z) T -> q'(X) ; C T :< q(X)).
```

For reading purposes, in the program above we used an abstract version of the generated code, since `C T` or `:< [foo(Y, Z)|C] T` isn't valid CxLP code.[3]

7.6 Application Examples

We now present a few examples which will sustain the adequacy of TCxLP. To ease the reading, in this section we present dates by atoms such as `'Aug 10 1969'`. Nevertheless, as mentioned in Sect. 6.5 (page 71), we resort to CLP(FD, $\mathcal{B}$) to implement the temporal elements. Moreover, unless stated otherwise, we assume that all temporal units implement the fact `item`.

7.6.1 Management of Workflow Systems

Workflow management systems (WfMS) are software systems that support the automatic execution of workflows. Although time is an important resource here, the time management offered by most of these systems must be handled explicitly and is rather limited. Therefore, automatic management of temporal aspects of information is an interesting and growing field of research [CP03, CP04, CP06, MMZ06]. Such management can be defined not only at the conceptual level (for instance changes defined over a schema) but also at run time (for instance workload balancing among agents).

The example used to illustrate the application of our language to WfMS is based on the one given in [CP04] and can be described as the process of enrolment of graduate students applying for PhD candidate positions. In the first proposal of the process model, from September 1^{st}, 2008, any received application leads to an interview of the applicant (see workflow on the left of Fig. 7.1). After September 30^{th}, 2008, the process model was refined and the applicants CVs must be analysed first: only applicants with an acceptable CV will be interviewed (see workflow on the right of Fig 7.1).

One of the functionalities performed by the workflow engine is selecting the successor task. Combi and Pozzi in [CP04] implemented their functionality by means of a trigger `FindSuccessor`. Since the *active* features of this trigger are outside the scope of this work, we are going to illustrate in our language the temporal aspects, recurring to the units: `task_history`, `case_history`, `work_task` and `next_task`. Below, for each unit we present a short explanation together with its implementation.

Unit `task_history` has the case identifier, the name of the tasks that were effectively

[3] In the actual implementation we use special units to represent the temporal information of the context.

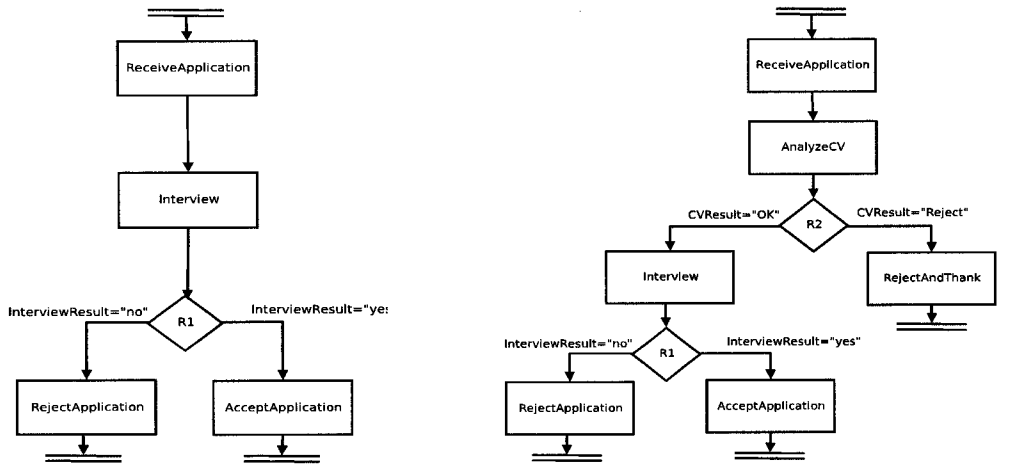

Figure 7.1: The student enrolment process model: initial proposal (left) and refinement (right)

executed for that case along with the executing agent.

```
:- unit(task_history(CaseID, Task, FinalState, Agent)).

task_history(27, receiveApplication, completed, r01)
            th ['Sep 9 2008', 'Sep 9 2008'].
task_history(89, receiveApplication, completed, b01)
            th ['Oct 3 2008', 'Oct 3 2008'].
```

Unit `case_history` has the unique identifier for every case together with the responsible agent.

```
:- unit(case_history(CaseId, Schema, Responsible)).

case_history(27, studentEnrollment, r01)
            th ['Sep 9 2008', 'Sep 27 2008'].
case_history(89, studentEnrollment, b01)
            th ['Oct 3 2008', 'Oct 31 2008'].
```

Unit `work_task` has the name of the task and the role required by the agent for the execution.

```
:- unit(work_task(Schema, Task, Role)).

work_task(studentEnrollment, receiveApplication,
          secretary) th ['Sep 1 2008', inf].
work_task(studentEnrollment, interview,
          committeeMember) th ['Sep 1 2008', inf].
```

Unit **next_task** provides a simplified version[4] of the integration of tables **RoutingTask**, **Next** and **AfterFork** [CP04]. This unit, besides the successor for every task, also has a condition that must be satisfied in order to allow such a transition to take place. The first temporal qualification states that for the student enrolment, the **next task** after receiving an application is doing an interview, but this is only valid between 'Sep 1 2008' and 'Sep 30 2008'.

```
:- unit(next_task(Schema, Task, NextTask, Condition)).

next_task(studentEnrollment, receiveApplication,
        interview, _) th ['Sep 1 2008', 'Sep 30 2008'].
next_task(studentEnrollment, receiveApplication,
        analyzeCV, _) th ['Oct 1 2008', inf].
next_task(studentEnrollment, analyzeCV,
        interview, cvresult(ok)) th ['Oct 1 2008', inf].
```

Recalling that all the above units implement the fact **item**, consider the goal:

```
?- ([] at 'Sep 4 2008') :>
        next_task(studentEnrollment,
                receiveApplication, N, _) :> item.

N = interview
```

i.e., at 'Sep 4 2008', the next task after receiving an application is an interview. The same query could be done without the explicit time:

```
?- next_task(studentEnrollment, receiveApplication,
             N, _) :> item.

N = analyzeCV
```

Recall that, if nothing is said about time, we assume we are in the present time (after 'Sep 30 2008') and, according to the refined workflow, the next task must be to analyse the CV.

Finally, using the units above, we can simulate the behaviour of the **FindSuccessor** trigger in the following way:

```
1 [case_history(CaseId, Schema, _)] th[L, _] :<
2     item, fd_min(L, Lmin),
3 [] at Lmin :>
4     next_task(Schema, Task, NextTask, Conditions)
```

[4]To obtain the full representation we just need to handle the task conditions.

```
5        :> (item,
6             work_task(Schema, NextTask, Role) :> item)
```

The first two lines query when the case started, allowing us to know which version of the workflow must be applied.[5] Then the context time is set to the lower bound and the next task, along with the conditions for this transition to take place are specified in lines 4 and 5. In the last line, the role of the employee which will execute this task is given.

7.6.2 Legal Reasoning

Legal reasoning is a very prolific field in which to illustrate the application of temporal languages. Not only is a modular approach very suitable for reasoning about laws but also time is pervasive in their definition. To illustrate the use of TCxLP in the legal reasoning domain, we return to the *British Nationality Act* already exploited in Sect. 6.6.1 (page 74).

The solution below presents several similarities with the CxLP with temporal annotations. The intention of focus on the differences between the proposals.

The TCxLP solution also has a unit `born/2`:

```
:- unit(born(Name, Country)).
born(john, uk) at 'Aug 10 1969'.
```

Although it might seem that this unit is almost identical, in the case of TCxLP (and remembering that all temporal units implement the fact `item`), now we can ask when John was born as:

```
?- [born(john, uk)] at T :< item.
```

in this case the context has the temporal informations, as opposed to the CxLP with annotations query:

```
?- [born(john, uk)] :< item at T.
```

Before presenting the rule for the nationality act we still need to state some facts about who is a British citizen along with who is parent of whom:

```
:- unit(british_citizen(Name)).

british_citizen(bob)
     th ['Sep 9 1940', inf].
```

[5] `fd_min(L, Lmin)` succeeds if `Lmin` is the minimal value of the current domain of `L`.

```
:- unit(parent(Parent, Son)).

parent(bob, john)
     th ['Aug 10 1969', inf].
```

To declare that the commencement date is 'Jan 1 1955', consider the following unit:

```
:- unit(bna).
bna th ['Jan 1 1955', inf].
```

Now we can give a formulation of this law in our language:

```
1 [] at T :< (born(X, uk) :> item,
2              parent(Y, X) :> item,
3              bna :> item,
4              (british_citizen(Y) :> item;
5               british_resident(Y) :> item)).
```

Notice that, by making use of the time of the context (T), there is no need to explicitly mention this anywhere else. This is the main difference w.r.t. CxLP with temporal annotations versions. As an informal description of this rule, we can say that we start by defining the time of the context as the time when the person was born (1). Afterwards, we query the name of the parent (2) and if the bna law can be applied (3) in this temporal context. Finally, we ask whether the parent is a British citizen (4) or a British resident (5).

7.6.3 Vaccination Program

Vaccinations programs are a very prolific domain for temporal reasoning. Not only is the inoculation of a given vaccine (doses) time dependent, but also the vaccination plans evolve throughout time. In this section we illustrate the use of TCxLP to reason about changes that occurred in a vaccination program. Although we are going to report our example to the Portuguese case, we do so without loss of generality.

In Portugal, a new National Vaccination Program was introduced in the year 2006. Several changes regarding the previous program (that started in 2000) were made, namely the introduction of the Meningococcal C vaccine. In Table 7.1 we can see a (partial) representation of this vaccination program.

	Age			
Vaccine against	0 (birth)	2 months	3 months	4 months
Tuberculosis	BCG			
Poliomyelitis		VIP 1		VIP 2
Diphteria-Tetanus-Whooping cough		DTPa 1		DTPa 2
Haemophilus influenza b		Hib 1		Hib 2
Hepatitis B	VHB 1	VHB 2		
Meningococcal C			MenC 1	

Table 7.1: Vaccination recommended scheme

One can regard it as set of facts that specify when one given vaccine should be inoculated, for instance the vaccines against Tuberculosis and Hepatitis B (first doses) should be inoculated at birth, Poliomyelitis (first doses), Diphtheria-Tetanus-Whooping cough (first doses), Haemophilus influenza b (first doses) and Hepatitis B (second doses) should be inoculated at the age of 2 months, etc.

Regarding vaccines, the most common questions made by a health care professional are:

- when was the last one?
- when should the next one be performed?

The answer to the former its quite straightforward whereas the latter entails knowing the vaccination program. To represent the scheme of Table 7.1 let us consider unit `vaccination_plan` where the temporal qualification of the facts represents their *validity*:

```
:- unit(vaccination_plan(Age, Vaccine)).
vaccination_plan(age(0,0), [bcg,
                  vhb(1)]) th [2000, inf].
vaccination_plan(age(0,2), [vip(1), dtpa(1), hib(1),
                  vhb(2)]) th [2000, inf].
vaccination_plan(age(0,3), [menC(1)]) th [2006, inf].
vaccination_plan(age(0,4), [vip(2), dtpa(2), hib(2)])
                  th [2000, inf].
```

As we can see, the inoculation of `menC` is valid only in temporal contexts after 2006 while the remaining ones are valid after 2000. To query about the next vaccine consider unit `next_vaccine/1`:

```
1 :- unit(next_vaccine(Vaccine)).
```

```
2 item :-
3  age(Person_Year, Person_Month),
4  vaccination_plan(age(Vac_Year, Vac_Month), Vaccine) :>
5         (item, (Vac_Year #> Person_Year #\/
6                 (Vac_Year #= Person_Year #/\
7                  Vac_Month #>= Person_Month))).
```

Predicate `item/0` queries the (temporal) context about the age of the person for whom it wants to compute the next vaccine (2), then it extends the context with unit `vaccination_plan` (3) and iterates over all (eligible) vaccines until it finds one that must be inoculated at or immediately after (4-7) (recall that the facts of this unit are ordered).

Finally, to illustrate, consider that unit `person(Name, Birth_Date)` implements predicate `age/2`:

```
?- [] at 'Aug 10 2006' :>
          person(john, 'May 1 2006') :>
                          (item, next(Vaccine) :> item).

Vaccine =  [menC(1)]
```

and if we replace all the occurrences of the year 2006 by 2000 in the query above, then the answer would be `Vaccine = [vip(2), dtpa(2), hib(2)]`.

7.7 Related Work

To the best of our knowledge, a temporal modular language where the usage of modules is influenced by temporal conditions is a novelty. The related work known on this subject is closer to the proposal of Chap. 6 where we already did a comparison with other approaches, in Sect. 6.6 (page 73).

Nevertheless, for completeness reasons, we discuss some (possible) relations with the language MuTACLP and the temporal architecture of Combi and Pozzi. Both comparisons have a similar structure: an applicational example is used to illustrate it.

In Sect. 7.6.2 (page 94) we provided the TCxLP version of the *British Nationality Act* example of Sect. 6.6.1 (page 74). As we can see, the use of contexts allows for a more compact writing where some of the annotations of the MuTACLP version are subsumed by the annotation of the context. For instance, the rules of the MuTACLP version for `get_citizenship` are far more verbose.

A similar reasoning applies when comparing the TCxLP solution for workflow management systems (see Sect. 7.6.1) with *relational frameworks* such as the one proposed by Combi and Pozzi in [CP04] where relational queries contained far more explicit references to time that the contextual version.

7.8 Conclusions

In this chapter we presented a temporal extension of CxLP where time influences the eligibility of a module to be used in solving a goal. We consider that such interaction between modularity and temporal reasoning comes naturally and, as far as we know, is original.

Besides the operational semantics for this language we proposed a procedure that computes the least upper bound of the units annotation at runtime.

An interpreter together with a compiler for this language were presented, forming the basis of a prototype implementation.

Several domains of applications were illustrated, namely legal reasoning, workflow management systems and medicine.

Chapter 8

Language for Temporal Organisational Information Systems

In this chapter we start with a revised and stripped-down version of the logical framework ISCO (Information System COnstruction language). We then describe a temporal extension together with a compilation scheme for this language.

8.1 Introduction

Organisational Information Systems (OIS) have a lot to benefit from Logic Programming (LP) characteristics such as a rapid prototyping ability, the relative simplicity of program development and maintenance, the declarative reading which facilitates both development and the understanding of existing code, the built-in solution-space search mechanism, the close semantic link with relational databases, just to name a few. In [Por03, ADN04] we find examples of LP languages that were used to develop and maintain OIS.

ISCO (Information System COnstruction language) [Abr01] is a logical framework for constructing Organisational Information Systems. ISCO is an evolution of the previous language DL [Abr00] and is based on a Constraint Logic Programming framework to define the schema, represent data, access heterogeneous data sources and perform arbitrary computations. In ISCO, processes and data are structured as *classes* which are represented as typed[1] Prolog predicates. An ISCO class may map to an external data source or sink like a table or view in a relational database, or be entirely implemented as a regular Prolog predicate. Operations pertaining to ISCO classes include a *query* which is similar to a Prolog call as well as three forms of *update*, i.e. ISCO implements Create, Read, Update and Delete (CRUD) access to Relational Database Management

[1]The type system applies to class members, which are viewed as Prolog predicate arguments.

System.

In this chapter we propose to extend ISCO with an expressive means of representing and implicitly using temporal information. As it should be expected, this evolution relies upon the frameworks proposed in Chap(s). 6 and 7. Moreover, having simplicity (syntactic, semantic, etc) as a guideline we present a revised and stripped-down version of ISCO, keeping just some of the core features that we consider indispensable in a language for Temporal OIS. Leaving out aspects such as access control [Abr02] does not mean they are deprecated, only not essential for the purpose at hand. The main motivation of this chapter is to provide a language suited to construct and maintain Temporal OIS where contexts aren't explicit but implicit (they are obtained throughout the compilation of this language).

This chapter is organised as follows: in Sect. 8.2 we review the ISCO language and in Sect. 8.3 we present its temporal extension ISTO. Section 8.4 discusses a compilation scheme for this language and Sect. 8.5 compares it with other approaches. Finally, in Sect. 8.6 we draw some conclusions.

8.2 Revising the ISCO Programming Language

In this section we present a revised and stripped-down version of ISCO focusing on what we consider the required features in a language to construct and maintain (Temporal) Organisational Information Systems.

8.2.1 Classes

When dealing with large amounts of data, one must be able to organise information in a modular way. The ISCO proposal for this point are *classes*. Although ISCO classes can be regarded as equivalent to Prolog predicates, they provide the homonymous OO concept, along with the related data encapsulation capability.

Before presenting a formal description of classes, let us see an ISCO class that handles data about persons, namely its name and social security number:

Example 6 *Class Person*

```
class person.
name: text.
ssn: int.
```

In this example after defining the class name to be `person`, we state its arguments and types: `ssn` (Social Security Number) type `int`(eger) and `name` type `text`.

Using a Definite Clause Grammar, the ISCO class syntax can be defined in the following way:

Definition 12 *Simple class syntax*

```
class --> class_head, arguments.

class_head --> [class NAME].

arguments --> [].
arguments --> argument, arguments.

argument --> [NAME : type].

type --> [int].
type --> [float].
type --> [bool].
type --> [text].
type --> [date].
```

From the definition above, we can see that a class is formed by a head and a (possibly empty) set of arguments. The class head starts with the keyword `class` and is followed by a descriptor. An argument definition its composed by its name concatenated with `":"` and an argument type (int, float, bool, text or date). This is the basic definition of a class that we will enrich it as long as we add other features.

8.2.2 Methods

By defining class arguments we are also implicitly setting class methods for accessing those arguments. For instance, in class `person` we have the predicates `ssn/1` and `name/1`. Therefore to query John's ssn, besides the (positional) Prolog goal `?- person(john, S).` we can also state:

Example 7 *John's social security number*

```
?- person(name(john), ssn(N)).

N = 123
```

Besides these implicit methods, there can also be explicit ones defined by means of regular Horn clauses. To allow for class methods, we enrich the rule for `class` as follows:

```
class --> class_head, arguments, horn_clauses.
```

As an illustration of an Horn clause, consider that argument `name` of class `person` is an atom with the structure 'SURNAME FIRST_NAME'. Therefore, to obtain the surname we add the following clause to the class definition:

```
surname(Surname) :-
            name(Name),
            atom_chars(Name, Name_Chars),
            append(Surname_Chars, [' '|_], Name_Chars),
            atom_chars(Surname, Surname_Chars).
```

8.2.3 Inheritance

Inheritance is another Object Oriented feature of ISCO that we would like to retain in our language. The reasons for that are quite natural since it allow us to share not only methods, but also data among different classes. The ISCO inheritance syntax is obtained by replacing the `class_head` of Definition 12 by:

```
class_head --> [class NAME], superclass.

superclass --> [].
superclass --> [: NAME].
```

Consider class `person` of Example 6 (page 100) and that we want to represent some facts (name, social security number and salary) about the employees of a given company. Therefore, we define `employee` to be a subclass of `person` with the argument `salary`:

Example 8 *Class employee*

```
class employee: person.
salary: int.
```

8.2.4 Composition

As we already saw, we have five basic types: `int`, `float`, `bool`, `text` and `date`. In order to have the OO feature of *composition*, we include ISCO possibility to re–use the class definitions as a data type. Therefore, the `type` syntactical definition becomes:

```
type --> basic_type.
type --> class_type.


basic_type --> [int].
basic_type --> [float].
basic_type --> [bool].
basic_type --> [text].
basic_type --> [date].

class_type --> [NAME].
```

Suppose that we want to deal with the employees home address and have the possibility of using the address schema in other places (suppliers, etc). For that we define a new class `address` and re–define the `employee` class adding a new argument (`home`) whose type is `address`:

Example 9 *Class employee with home address*

```
class address.
street: text.
number: int.
```

```
class employee: person.
home: address.
salary: int.
```

The access to compound types its quite natural, for instance suppose that we want to know John's street name:

Example 10 *John's home address*

```
?- employee(name(john), home(address(street(S)))).

S = upstreet
```

Actually, as the reader might see, there is no real need for the basic types since we could develop one class for each. Nevertheless, because they are quite intuitive to use, we decided to keep them.

8.2.5 Persistence

Having persistence in a Logic Programming language is a required feature to construct actual OIS; this could conceivably be provided by Prolog's internal database but is best

accounted for by software designed to handle large quantities of factual information efficiently, as is the case in relational database management systems. The semantic proximity between relational database query languages and logic programming languages have made the former privileged candidates to provide Prolog with persistence.

ISCO's approach for interfacing to existing RDBMS[2] involves providing declarations for an external database together with defining equivalences between classes and database relations. As an illustration, consider that the `employee` facts are stored in a homonymous table of a PostgreSQL [SK91] database named `db` running in the `localhost`:

Example 11 *External Databases*

```
external(db_link, postgres(db, localhost)).

external(db_link, employee) class employee: person.
home: address.
salary: int.
```

Since the database table has the same name as the class, the `employee` inside the `external` term above is optional. Finally, we can specify the full class definition:

Definition 13 *ISCO class syntax with external DB references*

```
class --> class_head, arguments, horn_clauses.

class_head --> external, [class NAME], superclass.

external --> [].
external --> [external(DB_LINK)].
external --> [external(DB_LINK, RELATION_NAME)].

superclass --> [].
superclass --> [: NAME].

arguments --> [].
arguments --> argument, arguments.

argument --> [NAME : type].
```

[2]ISCO access to frameworks beyond relational databases, such as LDAP directory services or web services is out of scope for the present work. We shall stick with to RDMBS only.

```
type --> basic_type.
type --> class_type.

basic_type --> [int].
basic_type --> [float].
basic_type --> [bool].
basic_type --> [text].
basic_type --> [date].

class_type --> [NAME].
```

8.2.6 Data Manipulation Goals

Under the data manipulation operations we include not only insertion, removal and update but also query operations, i.e. Create, Read, Update and Delete access to RDBMS. The (simplified) formal syntax of these operations can be described as:

Definition 14 *Goal Definition*

```
goal --> prolog_goal.
goal --> modification_goal.

modification_goal --> prolog_goal, [+].
modification_goal --> prolog_goal, [-].
modification_goal --> prolog_goal, [#], prolog_goal.
```

From the definition above, we have that a goal is either a Prolog goal or a modification goal. For illustrations of the first type of goals see Examples 7 and 10.

The modification goals (insert, delete and update) are based on simple queries, non backtrackable and all follow the tuple-at-a-time approach, which is more consistent with the usual Prolog operational semantics. As an example, suppose that we want to insert the employee `Smith`, update his address and then remove this employee:

Example 12 *Modification Goals*

```
?- employee(name(smith), ssn(111), salary(20),
            home(address(street(up), number(1)))) +.

?- employee(name(smith)) #
                (home(address(street(down), number(2)))).
```

```
?- employee(name(smith)) -.
```

This concludes our overview of the simplified ISCO language. In the next section we'll deal with temporal concerns.

8.3 The ISTO Language

In this section we provide a revised ISCO language with the ability to represent and implicitly use temporal information. This temporal evolution of the ISCO language will be called *ISTO* (Information System Tools for Organisations). The capability of representing temporal information will be achieved by extending ISCO classes to their temporal counterpart. As expected, in order to handle such temporal information ISTO includes temporal data manipulation operations.

8.3.1 Temporal Classes

In pursuing our goal of simplicity, temporal classes are given by introducing the keyword `temporal` before the keyword `class`. Therefore the class head syntax becomes:

```
class_head --> external, temporal, [class NAME],
               superclass.

temporal --> [].
temporal --> [temporal].
```

Facts of a temporal class have a temporal dimension, i.e. all tuples have associated a *temporal stamp*. In line with the other temporal languages proposed in this work, these stamps will stand for instants or intervals (their precise definition is given in Sect. 8.3.2).

As an illustration, class `employee` can be temporal (Example 9), since it makes sense that the salary and home address of a given employee evolves throughout time. On the other hand class `person` should continue atemporal since the facts it stores shouldn't evolve over time.

Example 13 *Temporal class employee*

```
temporal class employee: person.
home:address.
salary: int.
```

8.3.2 Temporal Data Manipulation

Here we present the temporal equivalent of the data manipulation goals described in Sect. 8.2.6 (page 105), i.e. we provide a temporal query, insertion, removal and update goals.

Again trying to keep the syntax as simple as possible we shall distinguish temporal operations from *regular* ones by adding the operator "@" followed by a temporal descriptor annotation. The interval representation is the standard one `[T1, T2]` and stands for all time points between `T1` and `T2` (where `T2` $\geq$ `T1`). Moreover, in order to be able to handle in-annotated information (see page 13), we also allow another type of intervals `[T1; T2]` to represent some time points (not necessarily all) between `T1` and `T2`, i.e. `[T1; T2]` is a subset of `[T1, T2]`. The ISTO temporal operations will be:

```
goal --> prolog_goal, temporal_condition.
goal --> modification_goal, temporal_condition.

temporal_condition --> [].
temporal_condition --> [ @ [Point , Point] ]
temporal_condition --> [ @ [Point ; Point] ]
```

Although an instant `T` can be represented by the interval `[T, T]`, to ease the reading we use just `T`. As an illustration consider that we want to know John's street name in the year 2007:

Example 14 *John's street name in the year 2007*

```
?- employee(name(john), home(address(street(S)))) @ 2007

S = upstreet
```

The goal above is a temporal version of the one we saw in Example 10 (page 103).

8.4 Compilation Scheme for ISTO

The ISTO compilation scheme is describe along the same incremental line that we saw in the language presentation. As expected, the compilation of the non-temporal part yields CxLP and from the temporal extension we target TCxLP.

8.4.1 Classes

The translation from an ISTO class to CxLP can be roughly described as: every class becomes a corresponding unit, with the class arguments transformed into unit arguments and predicates for accessing these arguments. For instance, class `person` of Example 6 (page 100) is compiled into the CxLP unit:

```
:- unit(person(OID, NAME, SSN)).

oid(OID).

name(NAME).

ssn(SSN).

item :- person(OID, NAME, SSN).
```

The unit argument `OID` stands for *Object Identifier* and as the name implies, is used to discriminate a given tuple in a class (and throughout the database in the corresponding superclass as we will see). The predicate `item/0` simply fetches facts from the internal database and binds the unit arguments. In the Sect. 8.4.5 (page 110) we see how to access external databases.

8.4.2 Methods

The compilation of Horn clauses is trivial since they remain unaltered. Regarding the implicit methods resulting from the class arguments, we already saw them in the previous section.

8.4.3 Inheritance

Since CxLP already has dynamic inheritance, it is rather simple to simulate the single static inheritance of ISTO. Take for instance the class `employee` of Example 8 (see page 102), subclass of `person`. Since `person` is a toplevel class, to use it we can simply do:

```
[person(OID, NAME, SSN)] :< (item, ...)
```

whereas to use `employee` we do:

```
[employee(OID, SALARY), person(OID, NAME, SSN)]
                                        :< (item, ...)
```

As mentioned previously, `OID` is used to *join* a class with its superclass. Finally, class `employee` of Example 8 is compiled as:

```
:- unit(employee(OID, SALARY)).

oid(OID).

salary(SALARY).

item :- employee(OID, SALARY),
        :^ item.
```

As we can see predicate `item/0` first invokes the local database and then calls `item` in the superclass, this way building the whole tuple. This definition requires the context to include the unit(s) corresponding to the superclass(es).

8.4.4 Composition

For the compound types the idea is similar to the one we saw with inheritance, i.e. we use an *identifier* to relate an argument to its compound type. In this case instead instead of using this identifier to perform a join operation, we use it as a *pointer*. To better understand let us see the compilation of the `employee` class with `home` argument (see Example 9 in page 103):

Example 15 *Compilation of employee with home address.*

```
:- unit(employee(OID, HOME_OID, SALARY)).

oid(OID).

home(address(GOAL)) :-
      [address(HOME_OID, STREET, NUMBER)] :< (item, GOAL).

salary(SALARY).

item :- employee(OID, HOME_OID, SALARY),
        :^ item.
```

In Example 10 (page 103) we saw that a query to an employee address has the structure `home(address(GOAL))`. The unit above explain how this query is handled: `GOAL` is resolved in a context with unit `address`. Moreover, the unit arguments

are obtained by binding the address object identifier with the `employee` argument `HOME_OID`, i.e. `HOME_OID` can be regarded as a pointer for the home address.

8.4.5 Persistence

Due to the possibility of having different backends, persistence related compilation can be more elaborate than what we are going to present. Nevertheless we consider the explanation below sufficient to grasp how it's done. Returning to the `employee` class, its modification into an external class presented in Example 11 (page 104) modifies the `item` predicate to:

```
item :- [isto_backend(db_link, employee)] :<
                    query(employee(OID, HOME_OID, SALARY)),
        :^ item.
```

8.4.6 Data Manipulation Goals

The compilation of the manipulation goals is quite simple, in fact we already saw how the query translation could be obtained when we presented the *Persistence*. Knowing that unit `isto_backend` also implements `insert/1`, `delete/1` and `update/2` it is simple to determine the compilation of the modification goals. For instance, the *removal of employee Smith* that we saw in Example 12 (page 105) is translated into:

```
1 ?- [employee(OID, HOME_OID, SALARY),
       person(OID, NAME, SSN)] :< (item, name(smith)),
3
     [isto_backend(db_link, employee)] :<
5              delete(employee(OID, HOME_OID, SALARY)),

7    [isto_backend(db_link, person)] :<
                       delete(person(OID, NAME, SSN)).
```

In the goal above we start by querying the whole tuple of the employee Smith (lines 1 and 2) and them we remove it from the `employee` backend (lines 4 and 5) and from the `person` backend (lines 7 and 8).

8.4.7 Temporal Classes

Before presenting the compilation of temporal classes and goals one must observe that non-temporal classes must behave as if they were valid throughout the entire time line.

Such a behaviour can be obtained simply by adding a fact to each nontemporal unit. For instance, to the unit `person` above it is sufficient to add the fact:

```
person(_, _, _) th [0, inf].
```

Let us now see the result of compiling the temporal class `employee` from Example 13 (page 106):

```
:- unit(employee(OID, HOME_OID, SALARY)).

oid(OID).

home(address(GOAL)) :-
     [address(HOME_OID, STREET, NUMBER)] :< (item, GOAL).

salary(SALARY).

item :- :^ item.
```

The difference from the compilation of the nontemporal version (see Example 15 in page 109) is that there is no need for predicate `item` to instantiate the unit arguments, since temporal context search will do that implicitly.

8.4.8 Temporal Data Manipulation Goals

The translation of a temporal query will result in a goal in a temporal context. The ISTO query of John's street name in 2007 (see Example 14 in page 107) is translated into:

```
?- [employee(OID, HOME_OID, SALARY),
      person(OID, NAME, SSN)] (at 2007) :<
           (item, name(john), home(address(street(S)))).
```

Introducing temporal modification goals needs further considerations. First of all, as mentioned, these goals are extra-logical. Moreover, since now the th-annotated facts can change at runtime, to use the (simplified) semantics of TCxLP one must ensure that the backend of a temporal class always stores the least upper bound of th-annotated unit temporal conditions. In order to guarantee that, every insertion of a th-annotated temporal condition must induce a recomputation of the least upper bound. As an illustration consider that John's `OID` is `1` and that is salary was `15000` between `2001` and `2006` and `20000` between `2007` and `2008`:[3]

[3]For simplicity reasons in this illustration we ignore the home address argument.

```
employee(1, 15000) th [2001, 2006].
employee(1, 20000) th [2007, 2008].
```

Suppose the following ISTO goal to remove this employee information between 2005 and 2006:

```
?- (employee(name(john)) -) @ [2005, 2006].
```

it changes the temporal conditions in the following way:

```
employee(1, 15000) th [2001, 2004].
employee(1, 20000) th [2007, 2008].
```

Finally, if we add the information that John's salary between 2005 and 2006 was 20000:

```
?- (employee(name(john), salary(20000)) +) @ [2005, 2006].
```

then the least upper bound must be recomputed, leading to:

```
employee(1, 15000) th [2001, 2004].
employee(1, 20000) th [2005, 2008].
```

8.5 Comparison with Other Approaches

The temporal timestamp of ISTO can regarded as the counterpart of the valid time that we saw in temporal databases (see Chap. 3). Although ISTO has no concept similar to the transaction time, we consider that one could implement this notion integrating a log in the `isto_backend` unit. However this capability is beyond the scope of the ISTO initial concerns. On the other hand, ISTO *contextual time* enables an expressivity that lacks in most database products with temporal support. Only in the Oracle Workspace Manager (see page 34) we find a concept (workspace) that is similar to our temporal context.

In this work we overviewed several logical languages that have modularity/OO, others that provide temporal reasoning or even persistence. ISTO is the only one that encompasses all those features.

8.6 Conclusions

In this chapter we presented a revision of a state-of-the-art logical framework for constructing OIS called ISCO. We proceeded to introduce an extension to this language called ISTO, that includes the expressive means of representing and implicitly

using temporal information. Together with a syntactical definition of ISTO we also presented a compilation scheme from this language into Temporal Contextual Logic Programming.

ISTO can take advantage of the experience gained from several real-world application developed in ISCO in order to act as backbone for constructing and maintaining Temporal OIS.

Chapter 9

Conclusions and Future Work

This short chapter presents the main conclusions of this work along with several pointers for future work.

9.1 Conclusions

In this work we started with a modular logic programming language called Contextual Logic Programming (CxLP) and presented a possible optimisation for it throughout abstract interpretation. Moreover, we coupled it with a consolidated temporal reasoning language called Temporal Annotated Constraint Logic Programming, for which two different paths were considered:

- one where those languages were (almost) independent, i.e. there was no implicit relation between Time and Modularity;
- another where Time is integrated with and affects Modularity, more specifically, it is the *time of the context* that helps decide if a given module is eligible or not for a computation.

For both approaches we described the language syntax and semantics (operational), provided an interpreter, a compilation scheme and illustrated its usage in various applicational domains.

It was not our goal to develop another temporal or modular theory, but to choose the ones that could have a natural integration within a logical framework. Although several criteria were used as a foundation for our choice, we would like to highlight *pragmatics*: the languages should be expressive enough to conveniently represent common or usual reasoning tasks in a temporal modular logic environment. Morever, the languages had to have a practical implementation.

Finally, and relying upon these formal languages, we extended a high level language for OIS with an expressive means of representing and implicitly using temporal information.

9.2 Future Work

The possibilities for future work are so vast that we must restrain a little and present just the ones that we would like to tackle in the near future:

- we would like to apply the ISTO language to real world problems. Perhaps, starting with University Information Systems, since this was the main application domain of its nontemporal predecessor.
- There are very promising developments in the fields of business intelligence and information retrieval which would stand to gain from the integration of temporal information.
- ISTO would also benefit from a logical semantics to the modification goals, by recurring to frameworks such as Transaction Logic [BK94].
- A declarative semantics for Temporal Contextual Logic Programming would provide a more solid theoretical foundation.
- The relation with other frameworks such as temporal XML [WZZ05] or Constraint Handling Rules [Frü94a] could be of great interest.
- Another appealing direction is looking into ways of integrating Logic Programming based tools like as ISTO into Content-Management Systems such as Plone or Mambo. It is our belief that these CMS will benefit from having plug-ins with the expressiveness provided by Constraint Contextual Logic Programming, namely unification and constraint solving.
- The decision of whether a given module is eligible or not for a computation could be extended in order to incorporate other concepts such as Space, etc.

Finally, we also consider that it would be interesting to use the logical frameworks described in this work to model OS-level applications such as the Mac OSX backup system, Time Machine.

Appendix A

GNU Prolog/CX

A.1 Tutorial

A.1.1 Unit Directive

A unit is program where the first term is the `unit/1` directive. The argument of this directive is a term, that in its simple form is just an atom that specifies the unit name. As an illustration consider the following unit defining the predicate *factorial*:

Example 16 *Unit factorial*

```
:- unit(factorial).

factorial(N, F) :-
        aux(N, 1, F).

aux(0, F, F).

aux(N, T, F) :-
        N > 0,
        T1 is T*N,
        N1 is N-1,
        aux(N1, T1, F).
```

The only difference between the program above and a regular Prolog program is the `unit/1` directive in the first line that declares the unit name to be `factorial`. The remaining code should be familiar to any Prolog programmer. Moreover, since we

have a predicate–based approach, other units can also define predicate `factorial/2` or `aux/3` that no collision will occur.

A.1.2 Unit Arguments

In Prolog its normal to find a proliferation of predicates arguments whenever a global structure is to be passed around. In the tail recursive version of factorial that we saw in Example 16, variable `F` is passed around over and over in `aux/3`, until it gets instantiated in the end. Moreover, this is a standard programming practice in *iterative/tail recursive* approaches to other predicates such as `reverse/2, sum_list/2`.

In Contextual Logic Programming we can avoid this proliferation, by considering some variables to be *global*. These variables are called *units arguments* and [AD03a] claims that they are an essential addition to this programming model. A unit argument can be interpreted as a sort of *unit global* variable, i.e., that is shared by all clauses defined in the unit. For that consider another version of unit `factorial` using unit arguments:

Example 17 *Unit factorial with arguments*

```
:- unit(factorial(N, F)).

item :-
        aux(N, 1).

aux(0, F).

aux(N1, Acc1) :-
        N1 > 0,
        Acc2 is Acc1* N1,
        N2 is N1-1,
        aux(N2, Acc2).
```

The reader should notice that in the unit directive, besides the unit name we also have unit arguments: variables `N` and `F`. These arguments are unit global variables (all the occurrences of `N` and `F` refer to the same variable), therefore we can avoid passing the (uninstantiated) result over and over again in the recursive call of `aux/2`: when the first argument of `aux/2` reaches 0, we instantiate the unit argument `F`. Moreover, predicate `item/0` assumes unit argument `N` is instantiated when the unit is invoked (we will see how to do this in the next section).

Finally, both units can be part of the same CxLP program because although they have the same name (`factorial`) they have different arities (`factorial/0` and `factorial/2`).

A.1.3 Context Extension

This section illustrates the usage of the units developed above. Let us start by obtaining the factorial of 6 recurring to the first unit:

```
?- factorial :> factorial(6, N).

N = 720
```

In an informal way we can read this goal as: reduce `factorial(6, F)` in the enriched environment with the definitions of the unit factorial.

In a similar way, we can use the unit `factorial/2` (with arguments) to solve the same problem:

```
?- factorial(6, N) :> item.

N = 720
```

In this case we can read it as reduce the goal `item` in the enriched environment with the definitions of `factorial/2`, instantiating the unit first argument to 6. In this case, this would mean that the rule for item was `item :- aux(6, 1).`

The operator `:>` is called *context extension*, and for now consider consider that as extending the GNU Prolog base predicates with the ones stated in the unit. This operator will become more clear below.

A.1.4 Current Context

Given a CxLP program we can combine at run-time its units leading to the notion of context. The compiler GNU Prolog/CX uses lists of units for representing contexts, where the empty list stands for the empty context.

To explicitly handle contexts, there is an operator `:<` called *current context*, where `:< C` unifies `C` with the current context. As a simple demonstration of this operator, consider a modified version of the query for the factorial of 6:

```
?- :< C_start,
   factorial :> (factorial(6, _F), :< C_fact),
   :< C_end.
```

```
C_end = []
C_fact = [factorial]
C_start = []
```

In this example we decided to use variable `_F` because we are only interested in the instantiation of the contexts. Let us start with the empty contexts `C_start` and `C_end`. They are empty because the place where they occur is not under the *scope* of any unit. The context `C_fact` on the contrary, is inside the environment of the unit `factorial`, therefore it should be no surprise that `C_fact = [factorial]`.

Moreover, and remembering that the empty list is a member of all the lists, it should be clear why `:>` is a context extension: we extended the empty context `[]` with the unit `factorial`, yielding the context `[factorial | [] ]`, i.e. `[factorial]`.

A similar modification can be done to obtain the contexts for the case of `factorial/2`:

```
?- :< C_start,
   factorial(6, _F) :> (item, :< C_fact),
   :< C_end.

C_end = []
C_fact = [factorial(6, 720)]
C_start = []
```

The difference from the previous example is the context `C_fact = [factorial(6, 720)]`, i.e. from a simple context inspection we know that the factorial of 6 is 720. Therefore, it should be clear why we state that units arguments make contexts more *transparent.*

A.1.5 Context Traversal

Until now we just saw simple contexts, namely empty or single contexts. Nevertheless, contexts can be more elaborated and as we are going to see, they are a natural way to represent dynamic inheritance. Dynamic inheritance allows objects to change and evolve over time. Since base classes provide properties and attributes for objects, changing base classes changes the properties and attributes of a class. More specifically, dynamic inheritance refers to the ability to add, delete or change parents from objects (or classes) at runtime.

To illustrate this feature we are going to build an example based on the *cd* (change working *d*irectory) of the Unix system V shell. Since this command is strongly connected not only to the *directory* tree but also to the *user* that is issuing it, we also

decided to represent these concepts.[1] Regarding the unit `user` we represented just the user name and its home directory inode:

Example 18 *Unit user*

```
:- unit(user(NAME, HOME)).

item :- user(NAME, HOME).

user(foo, 4).
user(bar, 5).
user(baz, 6).

name(NAME).
home(HOME).
pwd(HOME).
```

In this unit besides the `user/2` facts, we have one predicate to access each unit argument and another called `item` to instantiate the unit arguments using the facts retrieved from the database. Moreover, instead of the full path to the user home directory we decided to represent just the inode number of that directory. Using this number–based approach for directories, we represented a typical Unix filesystem as:

Example 19 *Unit fs (filesystem)*

```
:- unit(fs).

fs(1, /, nil).
fs(2, bin, 1).
fs(3, home, 1).
fs(4, foo, 3).
fs(5, bar, 3).
fs(6, work, 4).
fs(7, papers, 6).
```

In the unit above, each directory is represented by a triple: inode number, name and inode of its parent directory. This example represents the file system of Figure A.1.

[1] Since directory permissions didn't bring any advantage to illustrate concepts of CxLP we decided to omit them.

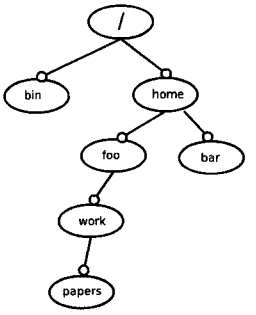

Figure A.1: File system

For instance, directory number 4 is */home/foo*: 3 (home) is the parent of 4 (foo) and 1 (/) is the parent of 3.

Let us see a goal to find out if the name of user home directory is the same as the user name. This can be achieved by a simple join between unit `fs` and unit `user`:

```
?- user(U, I) :> item, fs :> fs(I, U, _).

I = 4
U = foo ? ;

I = 5
U = bar
```

Having defined these two units we can now see the intended behaviour of the *cd* command. Suppose that user `foo` logs into the system and issues the command `cd work`, i.e. this command is executed in the *context* of `user(foo,_)` and in the filesystem seen above (`fs`). Using CxLP we can expect something like:

```
?- fs :> user(foo, _ ) :> (item, cd(work) :> check).
```

Therefore predicate `check/0` of unit `cd` must ask the context the working directory and then check if `work` is a subdirectory[2] of it. Therefore unit `cd` could be something like:

```
:- unit(cd(ARG)).

check :- pwd(WD),
         fs(_, ARG, WD).
```

Since predicate `pwd/1` isn't defined in this unit, is performed a search in the context for the topmost unit that defines this predicate. Thefore, although `pwd(WD)` was called

[2]We are just going to consider arguments that specify a direct subdirectory of the current one.

in the context `[cd(work), user(foo, 4), fs]` it is going to be solved in the reduced context `[user(foo, 4), fs]`, making variable `WD = 4`.

We are going to explain in detail these different types of contexts in section A.1.9 but for now imagine that the context is shortened until the topmost unit as a rule for the goal (of course that if none is found, the goal fails).

A similar reasoning applies to the other goal of this rule, i.e. `fs(_, work, 4)` is called in context `[cd(work), user(foo, 4), fs]` but since only unit `fs` has a rule for it, it is going to be solved in the context `[fs]`.

A.1.6 Context Switch

But what happens if another *cd* command follows as in:

```
?- fs :> user(foo, _ ) :> (item, cd(work) :>
                                  (check, cd(papers) :>
                                              check)).
```

The last `check/0` is solved in the context `[cd(papers), cd(work), user(foo, 4), fs]`, but this goal is going to fail because when it asks for `pwd(WD)` its going to obtain `WD = 4` (/home/foo) instead of `WD = 7` (/home/foo/work). This is because only unit `user` has a rule for `pwd/1`. Therefore, since `cd` changes the working directory it is reasonable to add a rule for `pwd/1`. Unit `cd` could then be:

```
:- unit(cd(ARG, NEW_WD)).

check :- pwd(WD),
         fs(NEW_WD, ARG, WD).

pwd(NEW_WD).
```

Nevertheless this formulation is still incorrect. To see why, consider again the example above adapted to the new formulation of `cd` command:

```
?- fs :> user(foo, _ ) :> (item, cd(work, _) :> check).
```

Again, when solving `check/0`, the goal `pwd(WD)` is invoked in context `[cd(work,_), user(foo, 4), fs]` and since now unit `cd/2` is the topmost unit with a rule for this goal, it is this unit that is going to resolve it instead of unit `user`. In fact, what we would want to do is call `pwd/1` in the context obtained from the current, ignoring its head.

To solve this problem another operator called context switch is presented `C :< G` and stands for solving goal `G` in context `C`. This leads to the correct formulation of unit `cd`:

```
:- unit(cd(ARG, NEW_WD)).

check :- :< [_|C],
         C :< pwd(WD),
         fs(NEW_WD, ARG, WD).

pwd(NEW_WD).
```

Now `check/0` first queries for the current context (operator `:<`) and then solves `pwd/1` in the subcontext obtained from ignoring the head of the current context.

A.1.7 Supercontext

Using the immediate subcontext to solve a goal is so recurrent that an operator was proposed for that purpose. This operator is denoted by `:^ G` is called supercontext and behaves as if defined by the Prolog clause:

```
:^ G :- [_| C], C :< G.
```

Using this operator, the unit `cd` becomes:

```
:- unit(cd(ARG, NEW_WD)).

check :- :^ pwd(WD),
         fs(NEW_WD, ARG, WD).

pwd(NEW_WD).
```

A.1.8 Guided Context Traversal

Until now we just considered paths that were immediate subdirectories of the current one. But there is a variation of the command *cd* where no arguments are given. In this case it must change to the home directory of the user currently logged. How can we do this in our framework? If the argument of *cd* is empty (or nil as we are going to represent), then `pwd(WD)` must be solved in the greatest subcontext of the current one that contains unit `user` in its head.

Of course that we could do this by considering the suffix of the current context that contains unit `user` in his head and then, by a context switch, solve `pwd(WD)` in this

suffix. Again, since this technique is so widely used, there is an operator `U::G` called *guided context traversal*, for that.

Therefore, we could add the following rule in unit `cd`:

```
check :- ARG = nil,
         !,
         user(_, _) :: pwd(NEW_WD).
```

This operator behaves as if defined by the Prolog clause:

```
U :: G :- :< C, GC = [U|_], suffixchk(GC, C), GC :< G.
```

Where `suffixchk(SUFFIX, LIST)` is a deterministic predicate that succeeds if `SUFFIX` is a suffix of `LIST`.

As an illustration, consider the following sequence of *cd* commands:

```
?- fs :> user(foo, _) :> (item, cd(work, _) :>
                          (check, cd(nil, NEW_WD) :> check)).

NEW_WD = 4
```

as expected, 4 is the inode of `foo` home directory.

A.1.9 Calling Context

Suppose that besides regular user logins we also want to have ftp anonymous login. In this case just a subtree of the current filesystem is visible, i.e. the root of the filesystem is changed. To incorporate the *chroot* (*ch*ange *r*oot) command to our framework consider a new unit `chroot` defined as follows:

```
:- unit(chroot(INODE)).

root(INODE).
```

where `INODE` is the inode of the new root.

Of course that unit `fs` must reflect these changes, i.e. when asked for `fs(INODE, /, _)` it must answer according to the latest changes of root in the system. Moreover, if no change has occurred, it must have a default root of the system. The intended behaviour can be illustrated with the following goal:

Example 20 (Sequence of chroots)

```
?- fs :> (fs(I1, /, _) ,
```

```
                chroot(3) :> (fs(I2, /, _),
                              chroot(4) :> fs(I3, /, _))).

I1 = 1
I2 = 3
I3 = 4
```

From a carefull observation of the goal above we can see that `fs(I1, /, _)` is called in the context `[fs]` and resolved in the same context; `fs(I2, /, _)` is called in the context `[chroot(3), fs]` but resolved in the context `[fs]`; `fs(I3, /, _)` is called in the context `[chroot(4), chroot(3), fs]` but resolved in the context `[fs]`, i.e. all have different *calling context* but the same *current context.*

Therefore, we can say that the definition of `fs(I, /, _)` depends of the content of the calling context, i.e. from the latest `root` in the calling context. Since there is an operator to obtain the calling context denoted by `:> C` it is rather straightforward to define `fs(I, /, _)`:

```
fs(I, /, nil) :-
        :> C,
        C :< root(I).
```

In this rule we start by inspecting the calling context (`:> C`) and then resolve the goal `root(I)` in this context, by performing a context switch.

Finally, since `root/1` may not be defined in the context (this is what happens in `fs(I1, /, _)` of Example 20), there must be a default definition for it in unit `fs` (in our case is `root(1).`).

A.1.10 Lazy Call

Since evaluating a goal in the calling context is done quite often, another operator denoted by `:# Goal` is available. This operator behaves as if defined by the Prolog clause:

```
:# G :- :> C, C :< G.
```

Thefore using this operator unit `fs` can be defined by:

```
:- unit(fs).

% default root
root(1).
```

```
fs(INODE, /, nil) :-
        :# root(INODE).

fs(2, bin, 1).
fs(3, home, 1).
fs(4, foo, 3).
fs(5, bar, 3).
fs(6, baz, 3).
fs(7, work, 4).
fs(8, papers, 6).
```

A.2 Reference Manual

A.2.1 Introduction

In this section we present the Contextual Logic Programming directives, operators and predicates that are part of the GNU Prolog/CX implementation.

A.2.2 Directives

`unit/1`

Templates

```
unit(+unit_descriptor)
```

Description

`unit(Name)` declares unit `Name` if the argument is an atom, otherwise declares a unit whose name is the principal functor of `Name` and whose arguments are the arguments of `Name`.

The type `unit_descriptor` is either an atom or a compound term with the structure `functor(Var_1, ..., Var_n)` where `Var_1, ..., Var_n` are different variable names.

Errors

None.

Portability

CxLP directive.

A.2.3 Operators

Context Switch - :</2

Templates

```
:< (+unit_descriptor_list, +callable_term)
```

Description

`Context :< Goal` evaluates `Goal` in context `Context`, ie. totally bypassing the current context.

Errors

None.

Portability

CxLP operator.

Current Context - `:</1`

Templates

```
:< (-unit_descriptor_list)
```

Description

`:< Context` unifies `Context` with the current context.

Errors

None.

Portability

CxLP operator.

Calling Context - `:>/1`

Templates

```
:> (-unit_descriptor_list)
```

Description

`:> Context` unifies `Context` with the calling context.

Errors

None.

Portability

CxLP operator.

Context Extension - :>/2

Templates

```
:> (+unit_descriptor, +callable_term)
```

Description

`Unit :> Goal` extends the current context with unit `Unit` before attempting to reduce goal `Goal`. This operator behaves as if defined by the Prolog clause:

```
U :> G :- :< C, [U|C] :< G.
```

Errors

None.

Portability

CxLP operator.

Guided Context Traversal - ::/2

Templates

```
::(+unit_descriptor, +callable_term)
```

Description

`Unit :: Goal` behaves as if defined by the Prolog clause:

```
U :: G :- :< C, GC = [U|_], suffixchk(GC, C), GC :< G.
```

Where `suffixchk/2` is a deterministic predicate where `suffixchk(SUFFIX, LIST)` succeeds if `SUFFIX` is a suffix of `LIST`.

Errors

None.

Portability

CxLP operator.

Supercontext - `:^/1`

Templates

```
:^(+callable_term)
```

Description

`^ Goal` evaluates `Goal` in the context obtained by dropping the topmost unit of the current context. This operator behaves as if defined by the Prolog clause:

```
:^ G :- :< [_|C], C :< G.
```

Errors

None.

Portability

CxLP operator.

Lazy Call - `:#/1`

Templates

```
:#(+callable_term)
```

Description

`# Goal` evaluates `Goal` in the calling context. This operator behaves as if defined by the Prolog clause:

```
:# G :- :> C, C :< G.
```

Errors

None.

Portability

CxLP operator.

A.2.4 Utilities

`current_unit/2`

Templates

```
current_unit(?atom,?integer)
```

Description

`current_unit(Name, Arity)` succeeds if there is a unit `Name` with `Arity` arguments. This predicate is re-executable on backtracking and can be thus used to enumerate all the units.

Errors

None.

Portability

CxLP predicate.

Appendix B

Constraint Logic Programming

B.1 Introduction

Formally, a Constraint Satisfaction Problem (CSP) consists of a tuple $\langle V, D, C \rangle$ where V is a set of variables, D their domains and C the set of constraints to be satisfied, i.e. all CSP have the following abstract structure:

- there are variables to instantiate;
- the variables range over some domain;
- the solutions must satisfy a set of constraints.

The Constraint Logic Programming (CLP) scheme *CLP(X)* [JL87, JM94], has been proposed as a generalisation of Logic Programming to deal with constraints in some arbitrary domain. In the *CLP(X)* scheme, the parameter X refers the specialised domain of the constraints. More specifically, to the:

- domain of the variables
- language of constraints

In the following section we will briefly overview the domain and language of the main constraint domains.

B.2 Constraint Domains

B.2.1 Booleans: CLP(B)

- Domain: boolean (True/False).
- Language: equality ($=$) of well formed formulas on the usual boolean operators ($\neg, \wedge, \vee, \ldots$).

B.2.2 Pseudo–Booleans: CLP(PB)

- Domain: boolean variables (0/1).
- Language: a relation of the set $\{=, <, \leq, >, \geq\}$ applied to pseudo–boolean terms (i.e. arithmetic expressions with operators $\{+, -, .\}$).

B.2.3 Rationals/Reals: CLP(R)

- Domain: rational (real) variables.
- Language: a relation of the set $\{=, \neq, <, \leq, >, \geq\}$ applied to linear terms (i.e. arithmetic expressions built with operator $\{+\}$).

B.2.4 Finite Domains: CLP(FD)

- Domain: a finite set of (ordered) values, possible interpreted to some numerical domain.
- Language: quite varied.

Finite domains subsume booleans and pseudo–booleans. All CLP(B) and CLP(PB) problems can thus be used as finite domain problems, declaring variables to be in the domain $\{0, 1\}$.

B.3 Constraint Solvers

There are two types of constraint solvers in CLP:

- complete solvers: if it is guaranteed that there is an infer function that detects contradiction;
- incomplete solvers: if it is not guaranteed that the infer function always detects contradiction. In this case, the solver is augmented with a second constraint store, where all the constraints whose effects have not been fully assessed are maintained.

Regarding the properties of constraint solvers, they:

- must be incremental. The addition of a constraint must cause as little computation as possible. The later retraction of a constraint (upon backtracking) must also be as effortless as possible.
- must be efficient;
- must be sound;
- might not be complete.

Answers

- LP provides definite answers: the answers are the most general terms that make the query consistent with the program.
- CLP also provides definite answers if the solvers are complete. If the solver is incomplete, the answers are conditional: they represent terms that make the query consistent with the program.

B.3.1 Incomplete Constraint Solvers

Incomplete solvers are used in domains where no general algebraic theory exists (or is efficient). Rather than guaranteeing that a set of constraints is consistent, incomplete constraint solvers aim at *reducing* the possible values of the variables so as to prune the search for solutions. These solvers are based on *local propagation*, that can be briefly described as:

> if the domain of a variable is reduced by some constraint, all constraints involving that variable are *awaken* to check whether the domain of other variables can be further reduced. If that is true, the procedure recurs. Otherwise it stops, and is ready to handle further constraints.

Local propagation is usually much more efficient that algebraic methods, but requires the use of a complementary enumeration procedure.

If all that is required is one solution, then incomplete solvers are usually preferably. If one needs all solutions, then the trade–off is uncertain.

B.4 Finite Domains

There are three types of constraints in the finite domains:

- declaration of finite domain variables and their domains.
- general constraints on these variables. Such constraints are used actively to reduce the initial domains, before and during the enumeration.
- enumeration predicates. These predicates are used to find solutions. The reduced domains of some variables can contain elements that do not belong to a solution. If these values are assigned to the variables, other domains are eventually reduced to the empty set, this way detecting unsatisfiability.

B.4.1 Finite Domain Solvers

The finite domain solvers are based on local propagation and can be formalised as constraint networks where:

- variables are the nodes of the network;
- constraints are the arcs of the network.

B.4.2 Network Consistency

There are three types of network consistency:

- **Node consistency:** any value in the domain of a variable A that is not consistent with the constraints on that variable alone must be removed. Is quite inexpensive to maintain, but it has limited pruning capabilities.
- **Arc Consistency:** any value in the domain of a variable A that is not supported by a value in a variable B for which there is an arc A–B, must be removed. It is more expensive to maintain and, sometimes the extra effort does not pay off in terms of increased pruning.

- **Path Consistency:** any value in the domain of a variable A, which is either not supported by a value in variable B for which there is an arc A–B, or by a variable C for which there are arcs A–B and B–C, must be removed. It is still more expensive to maintain and quite often it does not increase domain pruning significantly.

Since node and arc consistency can leave values in the domain of a variable that cannot be part of a solution, they are not complete.

B.4.3 Constraint Propagation (CP) vs. Backtracking

The advantages (+) and disadvantages (-) of constraint propagations versus backtracking are:

- \- in the early stages of a program CP spends a large time inspecting values of the variables domains, possibly to verify that they are X–consistent.
- \+ in the later stages of the programs, CP has to instantiate less values for the variables, since their domains have been pruned by constraint propagation.
- \+ CP anticipates the detection of failures, and thus backtracking.
- \+ CP prevents irrelevant backtracking. By anticipating backtracing, it does not backtrack to the *wrong* choice points.

B.4.4 Constraint Propagation and Heuristics

There are two major procedures to speed up the execution of constraint solvers:

1. reducing the number of alternatives throughout constraint propagation;
2. *guessing* the right alternatives recurring to heuristics (which are often problem dependent).

Regarding finite domains, there are two choice points in the enumeration phase:

1. the variable to instantiate next;
2. the value to instantiate the variable with.

B.4.5 Advanced Techniques

So far, choice points were only considered in the enumeration phase. A conjunction of constraints was assumed in the constraint declaration phase to set up a unique constraint network. However some applications require the introduction of disjunctions in the constraint specification phase. The possibilities for circumventing this problem are:

- delay the choice as much as possible;
- derive common information.

B.4.6 Global Constraints

Constraint solvers based on local propagation do not detect global inconsistency over *larger* sets of constraints in the network. Maintaining k–consistency (consistency of all sets of k variables) is very expensive (path consistency maintains 3–consistency, etc). However, special and important cases like `alldifferent`, `cumulative`, `among`, `diffn` and `cycle`, might be handled separately.

B.4.7 Optimisation Constraints

In many applications one is interested not only in satisfying a set of constraints but also in optimising (minimising or maximising) some objective function.

B.5 Defeasible Constraints

In several applications to keep the declarative style of programming, it is necessary to consider soft constraints, i.e. constraints that might be relaxed to make the problem solvable. This was the motivation of Hierarchical Constraint Logic Programming (or HCLP for short), where constraints were assigned some preference weight.

The goal of a HCLP program is therefore to compute a solution (or solutions) that complies with:

- optimisation, i.e. satisfies as much constraints as possible;
- satisfaction, i.e. satisfies a sufficient number of constraints.

B.6 Conclusions

- So far, finite domains are the most successful domain of application of CLP, being more efficient than Operations Research approaches.
- Complete solvers, namely the ones for linear constraints over rationals, are less efficient than Operations Research packages for very large problems.
- Programming a CLP application is still an engineering task. One has to:
  - find the best constraint solving technique;
  - discover heuristics applicable to the problem;
  - introduce redundant constraints to help the (incomplete) constraint solver;
  - adapt, if possible, the solver to match the user needs.
- The need to experiment several techniques and possibly adapt the constraint solver favours the approach of *glass–box* constraint solvers rather than *back–box* solvers.
- There is room to improve not only the capabilities of constraint solvers, but also to take advantage of the flexibility of the CLP approach to tackle applications requiring mixed techniques.
- Many applications are naturally over constrained and there is much research to be done in defeasible constraint solving.
- Constraint solving over reals (intervals) shows good potential for engineering applications.

References

[Abr00] Salvador Abreu. A Logic-based Information System. In Enrico Pontelli and Vitor Santos-Costa, editors, *2nd International Workshop on Practical Aspects of Declarative Languages (PADL'2000)*, volume 1753 of *Lecture Notes in Computer Science*, pages 141–153, Boston, MA, USA, January 2000. Springer-Verlag.

[Abr01] Salvador Abreu. Isco: A practical language for heterogeneous information system construction. In *Proceedings of INAP'01*, Tokyo, Japan, October 2001. Prolog Association of Japan.

[Abr02] Salvador Abreu. Modeling Role-Based Access Control in ISCO. In LÃŋgia Maria Ribeiro and JosÃl' Marques dos Santos, editors, *The 8th International Conference of European University Information Systems*. FEUP EdiÃğÃţes, June 2002. ISBN 972-752-051-0.

[AD03a] Salvador Abreu and Daniel Diaz. Objective: In minimum context. In Catuscia Palamidessi, editor, *ICLP*, volume 2916 of *Lecture Notes in Computer Science*, pages 128–147. Springer, 2003.

[AD03b] Salvador Abreu and Daniel Diaz. Objective: in Minimum Context. In Catuscia Palamidessi, editor, *Logic Programming, 19th International Conference, ICLP 2003, Mumbai, India, December 9-13, 2003, Proceedings*, volume 2916 of *Lecture Notes in Computer Science*, pages 128–147. Springer-Verlag, 2003. ISBN 3-540-20642-6.

[ADN04] Salvador Abreu, Daniel Diaz, and Vitor Nogueira. Organizational information systems design and implementation with contextual constraint logic programming. In *IT Innovation in a Changing World – The 10th International Conference of European University Information Systems*, Ljubljana, Slovenia, June 2004.

[Aea07] A. Aggoun and et. al. Eclipse user manual release 5.10, November 2007.

[AKN86] H Aït-Kaci and R Nasr. Login: A logic programming language with built-in inheritance. *J. Log. Program.*, 3(3):185–215, 1986.

[All83] J .F. Allen. Maintaining knowledge about temporal intervals. *cacm*, 26(11):832–843, nov 1983.

[AM89] Martín Abadi and Zohar Manna. Temporal logic programming. *J. Symb. Comput.*, 8(3):277–295, 1989.

[AN05] Salvador Abreu and Vitor Nogueira. Using a Logic Programming Language with Persistence and Contexts. In Masanobu Umeda and Armin Wolf, editors, *Proceedings of the 16th International Conference on Applications of Declarative Programming and Knowledge Management (INAP 2005)*, Fukuoka, Japan, October 2005. Waseda University.

[AN06] Salvador Abreu and Vitor Nogueira. Towards structured contexts and modules. In Etalle and Truszczynski [ET06], pages 436–438.

[AR99] Tamas Abraham and John F. Roddick. Survey of spatio-temporal databases. *Geoinformatica*, 3(1):61–99, 1999.

[Ari86] Gad Ariav. A temporally oriented data model. *ACM Trans. Database Syst.*, 11(4):499–527, 1986.

[Ate] Atempo. Time navigator. http://www.atempo.com/.

[Aug01] Juan Carlos Augusto. The logical approach to temporal reasoning. *Artif. Intell. Rev.*, 16(4):301–333, 2001.

[BBJ98] Michael H. Böhlen, R. Busatto, and Christian S. Jensen. Point-versus interval-based temporal data models. In *ICDE '98: Proceedings of the Fourteenth International Conference on Data Engineering*, pages 192–200, Washington, DC, USA, 1998. IEEE Computer Society.

[BCC+97] F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla. The ciao prolog system. reference manual. Technical Report CLIP3/97.1, School of Computer Science, Technical University of Madrid (UPM), August 1997. Available from http://www.clip.dia.fi.upm.es/.

[BCLM98] Michele Bugliesi, Anna Ciampolini, Evelina Lamma, and Paola Mello. Optimizing modular logic languages. *ACM Comput. Surv.*, 30(3es):10, 1998.

[BK82] K. A. Bowen and R. A. Kowalski. Amalgamating language and metalanguage in logic programming. In K. L. Clark and S.-A. Tärnlund, editors, *Logic Programming*, pages 153–172. Academic Press, London, 1982.

[BK94] Anthony J. Bonner and Michael Kifer. An overview of transaction logic. In *Theoretical Computer Science (TCS)*, pages 133:205–265, 1994.

[BLM93] Michele Bugliesi, Evelina Lamma, and Paola Mello. Partial deduction for structured logic programming. *J. Log. Program.*, 16(1):89–122, 1993.

[BLM94] Michele Bugliesi, Evelina Lamma, and Paola Mello. Modularity in logic programming. *The Journal of Logic Programming*, 19 & 20:443–502, May 1994.

[BMPT94] Antonio Brogi, Paolo Mancarella, Dino Pedreschi, and Franco Turini. Modular logic programming. *ACM Trans. Program. Lang. Syst.*, 16(4):1361–1398, 1994.

[BMRT01] P. Baldan, P. Mancarella, A. Raffaetà, and F. Turini. MuTACLP: A language for temporal reasoning with multiple theories. Technical Report TR-01-22, Dipartimento di Informatica, Università di Pisa, 23 2001.

[BMRT02] Paolo Baldan, Paolo Mancarella, Alessandra Raffaetà, and Franco Turini. Mutaclp: A language for temporal reasoning with multiple theories. In Antonis C. Kakas and Fariba Sadri, editors, *Computational Logic: Logic Programming and Beyond*, volume 2408 of *Lecture Notes in Computer Science*, pages 1–40. Springer, 2002.

[Bro56] F. P. Brooks. *The Analytic Design of Automatic Data Processing Systems.* PhD thesis, Harvard University, May 1956.

[Brz98] Christoph Brzoska. Programming in metric temporal logic. *Theor. Comput. Sci.*, 202(1-2):55–125, 1998.

[BZ82] Jacov Ben-Zvi. *The time relational model.* PhD thesis, 1982.

[CC87] James Clifford and Albert Croker. The historical relational data model (hrdm) and algebra based on lifespans. In *Proceedings of the Third International Conference on Data Engineering*, pages 528–537, Washington, DC, USA, 1987. IEEE Computer Society.

[CC92] Patrick Cousot and Radhia Cousot. Abstract interpretation and application to logic programs. *J. Log. Program.*, 13(2&3):103–179, 1992.

[CCGT95] James Clifford, Albert Croker, Fabio Grandi, and Alexander Tuzhilin. On temporal grouping. In *Proceedings of the International Workshop on Temporal Databases*, pages 194–213, London, UK, 1995. Springer-Verlag.

[CCT94] James Clifford, Albert Croker, and Alexander Tuzhilin. On completeness of historical relational query languages. *ACM Trans. Database Syst.*, 19(1):64–116, 1994.

[Che80] Brian F. Chellas. *Modal Logic: An Introduction.* Cambridge University Press, Cambridge, 1980.

[Cho94] J. Chomicki. Temporal query languages: a survey. In D. M. Gabbay and H. J. Ohlbach, editors, *Temporal Logic: ICTL'94*, volume 827, pages 506–534. Springer-Verlag, 1994.

[CLM96] Anna Ciampolini, Evelina Lamma, and Paola Mello. An abstract interpretation framework for optimizing dynamic modular logic languages. *Inf. Process. Lett.*, 58(4):163–170, 1996.

[CM00] Luca Chittaro and Angelo Montanari. Temporal representation and reasoning in artificial intelligence: Issues and approaches. *Annals of Mathematics and Artificial Intelligence*, 28(1-4):47–106, 2000.

[Cor05] Oracle Corportation. Oracle database 10g workspace manager overview. Oracle White Paper, May 2005.

[CP03] Carlo Combi and Giuseppe Pozzi. Temporal conceptual modelling of workflows. In Il-Yeol Song, Stephen W. Liddle, Tok Wang Ling, and Peter Scheuermann, editors, *ER*, volume 2813 of *Lecture Notes in Computer Science*, pages 59–76. Springer, 2003.

[CP04] Carlo Combi and Giuseppe Pozzi. Architectures for a temporal workflow management system. In *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, pages 659–666, New York, NY, USA, 2004. ACM Press.

[CP06] Carlo Combi and Giuseppe Pozzi. Task scheduling for a temporal workflow management system. *Thirteenth International Symposium on Temporal Representation and Reasoning (TIME'06)*, 0:61–68, 2006.

[CT98] Jan Chomicki and David Toman. Temporal logic in information systems. In Jan Chomicki and Gunter Saake, editors, *Logics for Databases and Information Systems (the book grow out of the Dagstuhl Seminar 9529: Role of Logics in Information Systems, 1995)*, pages 31–70. Kluwer, 1998.

[Dat99] C. J. Date. *An introduction to database systems (7th ed.).* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[DBL07] *14th International Symposium on Temporal Representation and Reasoning (TIME 2007), 28-30 June 2007, Alicante, Spain.* IEEE Computer Society, 2007.

[DC01] Daniel Diaz and Philippe Codognet. Design and implementation of the gnu prolog system. *Journal of Functional and Logic Programming*, 2001(6), October 2001.

[DD02] Chris Date and Hugh Darwen. *Temporal Data and the Relational Model.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[DD05] Hugh Darwen and C. J. Date. An overview and analysis of proposals based on the tsql2 approach. http://www.dcs.warwick.ac.uk/~hugh/TTM/OnTSQL2.pdf, March 2005.

[DLM+92] Enrico Denti, Evelina Lamma, Paola Mello, Antonio Natali, and Andrea Omicini. Implementing contexts in Logic Programming. In Evelina Lamma and Paola Mello, editors, *3rd International Workshop on Extensions of Logic Programming (ELP'92)*, pages 145–170, Bologna, Italy, 26–28 1992. Tecnoprint Bologna.

[DMG94] M A Reynolds D M Gabbay, I M Hodkinson. *Temporal Logic: Mathematical foundations and computational aspects*, volume 1. Clarendon Press, Oxford, 1994.

[DMP91] Rina Dechter, Itay Meiri, and Judea Pearl. Temporal constraint networks. *Artif. Intell.*, 49(1-3):61–95, 1991.

[DNO92] Enrico Denti, Antonio Natali, and Andrea Omicini. Contexts as first-class objects: An implementation based on the SICStus Prolog system. In Stefania Costantini, editor, *7th Italian Conference on Logic Programming (GULP'92)*, pages 307–320, Tremezzo, Como, Italy, 17–19 1992. Città Studi, Milano, Italy.

[EMK+04] Andrew Eisenberg, Jim Melton, Krishna Kulkarni, Jan-Eike Michels, and Fred Zemke. Sql:2003 has been published. *SIGMOD Rec.*, 33(1):119–126, 2004.

[ET06] Sandro Etalle and Miroslaw Truszczynski, editors. *Logic Programming, 22nd International Conference, ICLP 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4079 of *Lecture Notes in Computer Science.* Springer, 2006.

[F. 06] F. Henderson and T. Conway and Z. Somogyi and D. Jeffery and P. Schachte and S. Taylor and C. Speirs and T. Dowd and R. Becket and

M. Brown. *The Mercury Language Reference Manual.* The University of Melbourne, 0.13.1 edition, 2006.

[FGV05] Michael Fisher, Dov Gabbay, and Lluis Vila. *Handbook of Temporal Reasoning in Artificial Intelligence (Foundations of Artificial Intelligence (Elsevier)).* Elsevier Science Inc., New York, NY, USA, 2005.

[Frü93] Thom W. Frühwirth. Temporal logic and annotated constraint logic programming. In Michael Fisher and Richard Owens, editors, *Executable Modal and Temporal Logics*, volume 897 of *Lecture Notes in Computer Science*, pages 58–68. Springer, 1993.

[Frü94a] T. Frühwirth. Temporal reasoning with constraint handling rules. Technical Report ECRC-94-5, European Computer-Industry Research Centre GmbH, ECRC Munich, Germany, 1994.

[Frü94b] Thom W. Frühwirth. Annotated constraint logic programming applied to temporal reasoning. In Manuel V. Hermenegildo and Jaan Penjam, editors, *PLILP*, volume 844 of *Lecture Notes in Computer Science*, pages 230–243. Springer, 1994.

[Frü96] Thom W. Frühwirth. Temporal annotated constraint logic programming. *J. Symb. Comput.*, 22(5/6):555–583, 1996.

[fS00] International Organization for Standardization. *ISO IEC 13211-2:2000: Information technology - Programming languages - Prolog - Part 2: Modules.* International Organization for Standardization, Geneva, Switzerland, 2000.

[fS03] International Organization for Standardization. *ISO/IEC 9075-*:2003. Information technology - Database languages - SQL.* International Organization for Standardization, Geneva, Switzerland, 2003.

[Gad88] Shashi K. Gadia. A homogeneous relational model and query languages for temporal databases. *ACM Trans. Database Syst.*, 13(4):418–448, 1988.

[Gal87] Antony Galton. The logic of occurrence. pages 169–196, 1987.

[Gal08] Antony Galton. Temporal logic. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy.* Spring 2008.

[Ger01] Manolis Gergatsoulis. Temporal and modal logic programming languages. In A. Kent and J. G. Williams, editors, *Encyclopedia of Microcomputers*, volume 27,Supplement 6, pages 393–408. Marcel Dekker, Inc, New York, 2001.

[GJ99] Heidi Gregersen and Christian S. Jensen. Temporal entity-relationship models-a survey. *IEEE Transactions on Knowledge and Data Engineering*, 11(3):464–497, 1999.

[GMR88] Laura Giordano, Alberto Martelli, and Gianfranco Rossi. Local definitions with static scope rules in logic programming. In *FGCS*, pages 389–396, 1988.

[GRP96] M. Gergatsoulis, P. Rondogiannis, and T. Panayiotopoulos. Disjunctive chronolog, 1996.

[GWW75] J.F. Fries G. Wiederhold and S. Weyl. Structured organization of clinical data bases. In *AFIPS National Computer Conference*, pages 479–485, 1975.

[GY88] Shashi K. Gadia and Chuen-Sing Yeung. A generalized model for a relational temporal database. In *SIGMOD '88: Proceedings of the 1988 ACM SIGMOD international conference on Management of data*, pages 251–259, New York, NY, USA, 1988. ACM Press.

[HF06] Rémy Haemmerlé and François Fages. Modules for prolog revisited. In Etalle and Truszczynski [ET06], pages 41–55.

[Hir97] Robin Hirsch. Expressive power and complexity in algebraic logic. *Journal of Logic and Computation*, 7(3):309–351, 1997.

[HM87] S. Hanks and D. McDermott. Default reasoning, nonmonotonic logics, and the frame problem. pages 390–395, 1987.

[Hry93] Tomas Hrycej. A temporal extension of prolog. *J. Log. Program.*, 15(1-2):113–145, 1993.

[HS91] Joseph Y. Halpern and Yoav Shoham. A propositional modal logic of time intervals. *J. ACM*, 38(4):935–962, 1991.

[IABC+95] Gad Ariav Ilsoo Ahn, Don Batory, James Clifford, Curtis E. Dyreson, Ramez Elmasri, Fabio Grandi, Christian S. Jensen, Wolfgang Käfer, Nick Kline, Krishna Kulkarni, T. Y. Cliff Leung, Nikos Lorentzos, John F. Roddick, Arie Segev, Michael D. Soo, and Suryanarayana M. Sripada. *The TSQL2 Temporal Query Language.* Kluwer Academic Publishers, 1995.

[IBM] IBM. Data propagator. http://www-306.ibm.com/software/data/integration/replication/.

[Int92] International Organization for Standardization. *ISO/IEC 9075:1992: Title: Information technology — Database languages — SQL.* 1992. Available in English only.

[Jen00] Christian S. Jensen. Temporal database management. Dr. Techn. Thesis, 2000.

[JK04] Peter Jonsson and Andrei Krokhin. Complexity classification in qualitative temporal constraint reasoning. *Artif. Intell.*, 160(1):35–51, 2004.

[JL87] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 111–119. ACM Press, 1987.

[JM94] Joxan Jaffar and Michael J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–581, 1994.

[JMR91] C. S. Jensen, L. Mark, and N. Roussopoulos. Incremental implementation model for relational databases with transaction time. *IEEE Transactions on Knowledge and Data Engineering*, 3(4):461–473, 1991.

[JMS79] Susan Jones, Peter Mason, and Ronald K. Stamper. Legol 2.0: A relational specification language for complex rules. *Inf. Syst.*, 4(4):293–305, 1979.

[Joh91] Johan Van Benthem. *The Logic of Time: A Model-Theoretic Investigation into the Varieties of Temporal Ontology and Temporal Discourse*, volume 156 of *Synthese Library*. Kluwer Academic Publishers, second edition, 1991.

[JS96] Christian S. Jensen and Richard Thomas Snodgrass. Semantics of time-varying information. *Inf. Syst.*, 21(4):311–352, 1996.

[JSS95] Christian S. Jensen, Richard T. Snodgrass, and Michael D. Soo. The tsql2 data model. In *The TSQL2 Temporal Query Language*, pages 153–238. 1995.

[Kim78] K. A. Kimball. The data system. Master's thesis, University of Pennsylvania, 1978.

[KJJ03] Andrei A. Krokhin, Peter Jeavons, and Peter Jonsson. Reasoning about temporal relations: The tractable subalgebras of allen's interval algebra. *J. ACM*, 50(5):591–640, 2003.

[Kou95] M. Koubarakis. Databases and temporal constraints: Semantics and complexity. In S. Clifford and A. Tuzhilin, editors, *Recent Advances in Temporal Databases*, pages 93–112, Zurich, Switzerland, sep 1995. Proceedings of the International Workshop on Temporal Databases, Springer Verlag.

[KS86] R. A. Kowalski and M. J. Sergot. A logic-based calculus of events. *New Generation Computing*, 4(1):67–95, 1986.

[KS97] Robert A. Kowalski and Fariba Sadri. Reconciling the event calculus with the situation calculus. *J. Log. Program.*, 31(1-3):39–58, 1997.

[KSW90] F. Kabanza, J-M Stevenne, and P. Wolper. Handling infinite temporal data. In *9th Annual ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, Nashville, TN, apr 1990.

[Lab03] The Intelligent Systems Laboratory. *Quintus Prolog User's Manual.* Swedish Institute of Computer Science, PO Box 1263, SE-164 29 Kista, Sweden, release 3.5 edition, December 2003.

[LEM88] Jr. Leslie Edwin Mckenzie. *An algebraic language for query and update of temporal databases.* PhD thesis, 1988. Director-Richard Snodgrass.

[LEMS91] Jr. L. Edwin McKenzie and Richard Thomas Snodgrass. Evaluation of relational algebras incorporating the time dimension in databases. *ACM Comput. Surv.*, 23(4):501–543, 1991.

[LFA07] Nuno Lopes, Cláudio Fernandes, and Salvador Abreu. Contextual Logic Programming for Ontology Representation and Querying. In Axel Polleres, David Pearce, Stijn Heymans, and Edna Ruckhaus, editors, *Proceedings of the ICLP'07 Workshop on Applications of Logic Programming to the Web, Semantic Web and Semantic Web Services (ALPSWS 2007)*, volume 287 of *CEUR Workshop Proceedings ISSN 1613-0073*, October 2007.

[LJ88] N. A. Lorentzos and R. G. Johnson. Extending relational algebra to manipulate temporal data. *Inf. Syst.*, 13(3):289–296, 1988.

[LM94] Evelina Lamma and Paola Mello. Modularity in logic programming. In Pascal Van Hentenryck, editor, *Logic Programming - Proceedings of the Eleventh International Conference on Logic Programming*, pages 15–17, Massachusetts Institute of Technology, 1994. The MIT Press.

[LMN92] E. Lamma, P. Mello, and A. Natali. An Extended Warren Abstract Machine for the Execution of Structured Logic Programs. *Journal of Logic Programming*, 14(3-4):187–222, 1992.

[LO96] Chuchang Liu and Mehmet Ali Orgun. Clocked Temporal Logic Programming. In *Proceedings of The 19th Australasian Computer Science Conference*, Melbourne, Australia, January 31-February 2 1996.

[Lor88] N. A. Lorentzos. *A Formal Extension of the Relational Model for the Representation of Generic Intervals.* PhD thesis, Birkbeck College, 1988.

[Lum] Lumigent. Log explorer for sql server. http://www.lumigent.com/products/le_sql.html.

[McC92] Francis G. McCabe. *Logic and objects.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[Mei91] Itay Meiri. Combining qualitative and quantitative constraints in temporal reasoning. In Thomas Dean and Kathleen McKeown, editors, *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 260–267, Menlo Park, California, 1991. AAAI Press.

[Mei96] Itay Meiri. Combining qualitative and quantitative constraints in temporal reasoning. *Artif. Intell.*, 87(1-2):343–385, 1996.

[MH69] John McCarthy and Patrick J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, 1969. reprinted in McC90.

[Mil86] Dale Miller. A theory of modules for logic programming. In *Symposium on Logic Programming*, pages 106–114, 1986.

[Mil89a] Dale Miller. A Logical Analysis of Modules in Logic Programming. *Journal of Logic Programming*, 6(2):79–108, 1989.

[Mil89b] Dale Miller. Lexical scoping as universal quantification. In *Sixth International Logic Programming Conference*, pages 268–283, Lisbon, Portugal, June 1989. MIT Press.

[Mil93] Dale Miller. A proposal for modules in lambda-prolog. In Roy Dyckhoff, editor, *ELP*, volume 798 of *Lecture Notes in Computer Science*, pages 206–221. Springer, 1993.

[MMZ06] Elisabetta De Maria, Angelo Montanari, and Marco Zantoni. An automaton-based approach to the verification of timed workflow schemas. In *TIME '06: Proceedings of the Thirteenth International Symposium on Temporal Representation and Reasoning (TIME'06)*, pages 87–94, Washington, DC, USA, 2006. IEEE Computer Society.

[MNR89] Paola Mello, Antonio Natali, and Cristina Ruggieri. Logic programming in a software engineering perspective. In *NACLP*, pages 441–458, 1989.

[MNRT00] Paolo Mancarella, Gianluca Nerbini, Alessandra Raffaetà, and Franco Turini. Mutaclp: A language for declarative gis analysis. In John W. Lloyd, Verónica Dahl, Ulrich Furbach, Manfred Kerber, Kung-Kiu Lau, Catuscia Palamidessi, Luís Moniz Pereira, Yehoshua Sagiv, and Peter J. Stuckey, editors, *Computational Logic*, volume 1861 of *Lecture Notes in Computer Science*, pages 1002–1016. Springer, 2000.

[Mou00] Paulo Moura. Logtalk 2.6 Documentation. Technical Report DMI 2000/1, University of Beira Interior, Portugal, July 2000.

[Mou03] Paulo Moura. *Logtalk - Design of an Object-Oriented Logic Programming Language.* PhD thesis, Department of Computer Science, University of Beira Interior, Portugal, September 2003.

[MP89] Luís Monteiro and António Porto. Contextual logic programming. In Maurizio Levi, Giorgio; Martelli, editor, *Proceedings of the 6th International Conference on Logic Programming (ICLP '89)*, pages 284–302, Lisbon, Portugal, June 1989. MIT Press.

[MP90] Luís Monteiro and António Porto. A transformational view of inheritance in logic programming. In *ICLP*, pages 481–494, 1990.

[MP93] Luís Monteiro and António Porto. A Language for Contextual Logic Programming. In K.R. Apt, J.W. de Bakker, and J.J.M.M. Rutten, editors, *Logic Programming Languages: Constraints, Functions and Objects*, pages 115–147. MIT Press, 1993.

[MRT97] Paolo Mancarella, Alessandra Raffaetà, and Franco Turini. Time in a multi-theory logical framework. In *TIME*, pages 62–70, 1997.

[MRT99] Paolo Mancarella, Alessandra Raffaetà, and Franco Turini. Knowledge representation with multiple logical theories and time. *J. Exp. Theor. Artif. Intell.*, 11(1):47–76, 1999.

[NA89] S. B. Navathe and R. Ahmed. A temporal relational model and a query language. *Inf. Sci.*, 49(1-3):147–175, 1989.

[NA06a] Vitor Nogueira and Salvador Abreu. Temporal contextual logic programming. In Francisco J. López Fraguas, editor, *Proceedings of the 15^{th} Workshop on Functional and (Constraint) Logic Programming (WFLP'06)*, Madrid, Spain, November 2006. Electronic Notes in Theoretical Computer Science.

[NA06b] Vitor Nogueira and Salvador Abreu. Towards temporal contextual logic programming. In Etalle and Truszczynski [ET06], pages 439–441.

[NA07a] Vitor Nogueira and Salvador Abreu. Integrating temporal annotations in a modular logic language. In Dietmar Seipel, Michael Hanus, Armin Wolf, and Joachim Baumeister, editors, *17th International Conference on Applications of Declarative Programming and Knowledge Management (INAP 2007)*, volume Technical Report 434, Würzburg, Germany, October 2007. Bayerische Julius-Maximilians-Universität Würzburg.

[NA07b] Vitor Nogueira and Salvador Abreu. Modularity and temporal reasoning: a logic programming approach. In *Time* [DBL07].

[NA07c] Vitor Nogueira and Salvador Abreu. Temporal Annotations for a Contextual Logic Programming Language. In José Neves, Manuel Santos, and José Machado, editors, *Progress in Artificial Intelligence, 13th Portuguese Conference on Artificial Intellige nce, EPIA 2007*, Universidade do Minho, 2007.

[NA07d] Vitor Nogueira and Salvador Abreu. Temporal contextual logic programming. *Electr. Notes Theor. Comput. Sci.*, 177:219–233, 2007.

[NAD03] Vitor Nogueira, Salvador Abreu, and Gabriel David. Using contextual logic programming for temporal reasoning. In Jaime Gómez Ernesto Pimentel, Nieves J. Brisaboa, editor, *Proceedings of the VIII Jornadas de Ingeniería del Software y Bases de Datos*, pages 479–489, Alicante, Spain, November 2003.

[NAD04] Vitor Beires Nogueira, Salvador Abreu, and Gabriel David. Towards temporal reasoning in constraint contextual logic programming. In *Proceedings of the 3rd International Workshop on Multiparadigm Constraint Programming Languages MultiCPL'04 associated to ICLP'04*, Saint–Malo, France, September 2004.

[NB94] Bernhard Nebel and Hans-Jürgen Bürckert. Reasoning about temporal relations: a maximal tractable subclass of allen's interval algebra. In *AAAI '94: Proceedings of the twelfth national conference on Artificial intelligence (vol. 1)*, pages 356–361, Menlo Park, CA, USA, 1994. American Association for Artificial Intelligence.

[NM88] Gopalan Nadathur and Dale Miller. An Overview of λProlog. In *Fifth International Logic Programming Conference*, pages 810–827, Seattle, August 1988. MIT Press.

[NO] Antonio Natali and Andrea Omicini. Objects with state in Contextual Logic Programming. In Maurice Bruynooghe and Jaan Penjam, editors, *Programming Language Implementation and Logic Programming*, volume 714 of *LNCS*, pages 220–234. Springer-Verlag. 5th International Symposium (PLILP'93), Tallinn, Estonia, 25–27.

[NO93] Antonio Natali and Andrea Omicini. Objects with state in CSM. In *2nd Compulog Network Area Meeting on Programming Languages joint with Workshop on Logic Languages*, pages 1–2, Pisa, Italy, 6–7 1993.

[O'K85] R. O'Keefe. Towards an algebra for constructing logic programs. In In J. Cohen and J. Conery, editors, *Proceedings of IEEE Symposium on Logic Programming*, pages 152–160. IEEE Computer Society Press, 1985.

[Org91] Mehmet Ali Orgun. *Intensional logic programming.* PhD thesis, Victoria, B.C., Canada, Canada, 1991.

[Org96] Mehmet A. Orgun. On temporal deductive databases. *Computational Intelligence*, 12:235–259, 1996.

[Pan95] M. Panayiotopoulos, T. and Gergatsoulis. A Prolog like temporal reasoning system. In M. H. Hamza, editor, *Proc. of 13th IASTED International Conference on APPLIED INFORMATICS*, pages 123–126, ICLS (Innsbruck), Austria, 1995.

[Pao] Paolo Terenziani. Reasoning about Time.

[Pin94] Javier Pinto. *Temporal Reasoning in the Situation Calculus.* PhD thesis, Department of Computer Science, University of Toronto, Toronto, Canada, January 1994.

[Por03] António Porto. An integrated information system powered by prolog. In Verónica Dahl and Philip Wadler, editors, *PADL*, volume 2562 of *Lecture Notes in Computer Science*, pages 92–109. Springer, 2003.

[Pri57] A. N. Prior. *Time and Modality.* The Clarendon Press, Oxford, 1957.

[Pri67] A. N. Prior. *Past, present and future.* The Clarendon Press, Oxford, 1967.

[Pri68] A. N. Prior. *Papers on Time and Tense.* The Clarendon Press, Oxford, 1968.

[Raf00] Alessandra Raffaetá. *Spatio-temporal knowlegde bases in constraint logic programming framework with multiple theories.* PhD thesis, Universitá Degli Studi di Pisa, Dipartamento di Informatica, Pisa, Italy, March 2000.

[Rei89] H. Reichgelt. A comparison of first order and modal logics of time. In P. Jackson, H. Reichgelt, and F. van Harmelen, editors, *Logic-Based Knowledge Representation*, pages 143–176. MIT Press, Cambridge, MA, 1989.

[RF00a] Alessandra Raffaetà and Thom Frühwirth. *Labelled deduction*, chapter Semantics for temporal annotated constraint logic programming, pages 215–243. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[RF00b] Alessandra Raffaetà and Thom Frühwirth. Two semantics for temporal annotated constraint logic programming. In M. Gergatsoulis and P. Rondogiannis, editors, *Intensional Programming II, Based on the Papers at ISLIP'99*, pages 78–92. World Scientific Singapore, 2000.

[RGP97] Panos Rondogiannis, Manolis Gergatsoulis, and Themis Panayiotopoulos. Cactus: A branching-time logic programming language. In *ECSQARU-FAPR*, pages 511–524, 1997.

[Rib93] Cristina Ribeiro. *Representation and Inference of Temporal Knowledge.* PhD thesis, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, 1993.

[RN03] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Pearson Education, 2003.

[RV05] H. Reichgelt and L. Vila. *Handbook of Temporal Reasoning in Artificial Intelligence (Foundations of Artificial Intelligence (Elsevier))*, chapter Temporal Qualification in Artificial Intelligence. In [FGV05], 2005.

[Sad87] Rubik Sadeghi. *A database query language for operations on historical data.* PhD thesis, 1987.

[Sar90a] N. L. Sarda. Algebra and query language for a historical data model. *Comput. J.*, 33(1):11–18, 1990.

[Sar90b] N. L. Sarda. Extensions to sql for historical databases. *IEEE Transactions on Knowledge and Data Engineering*, 2(2):220–230, 1990.

[SBJS97] Richard T. Snodgrass, Michael H. Böhlen, Christian S. Jensen, and Andreas Steiner. Transitioning temporal support in tsql2 to sql3. In *Temporal Databases, Dagstuhl*, pages 150–194, 1997.

[Sch98] Eddie Schwalb. *Temporal Reasoning With Constraints.* PhD thesis, University of California Irvine, June 1998.

[Sho88] Y. Shoham. Chronological ignorance: experiments in nonmonotonic temporal reasoning. *Artif. Intell.*, 36(3):279–331, 1988.

[SK91] Michael Stonebraker and Greg Kemnitz. The postgres next generation database management system. *Commun. ACM*, 34(10):78–92, 1991.

[SKD94] Eddie Schwalb, Kalev Kask, and Rina Dechter. Temporal reasoning with constraints on fluents and events. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, volume 2, pages 1067–1072, Seattle, Washington, USA, 1994. AAAI Press/MIT Press.

[SN04] Jörg Sander and Mario A. Nascimento, editors. *Spatio-Temporal Database Management, 2nd International Workshop STDBM'04, Toronto, Canada, August 30, 2004*, 2004.

[Sno] Richard Snodgrass. Tsql2 and sql3 interactions. http://www.cs.arizona.edu/~rts/sql3.html.

[Sno87] Richard T. Snodgrass. The temporal query language tquel. *ACM Trans. Database Syst.*, 12(2):247–298, 1987.

[Sno07] Richard T. Snodgrass. Towards a science of temporal databases. In *TIME* [DBL07], pages 6–7.

[SS87] Arie Segev and Arie Shoshani. Logical modeling of temporal data. In *SIGMOD '87: Proceedings of the 1987 ACM SIGMOD international conference on Management of data*, pages 454–466, New York, NY, USA, 1987. ACM.

[SSW+07] K. F. Sagonas, T. Swift, D. S. Warren, J. Freire, P. Rao, B. Cui, E. Johnson, L. de Castro, R. F. Marques, D. Saha, S. Dawnson, and M. Kifer. *The XSB System Version 3.1 Volume 1: Programmer's Manual*, August 2007.

[Ste05] Andreas Steiner. Timedb. http://www.TimeConsult.com/, 2005.

[Sub94] V. S. Subrahmanian. Amalgamating knowledge bases. *ACM Trans. Database Syst.*, 19(2):291–331, 1994.

[SV96] E. Schwalb and L. Vila. Logic programming with temporal constraints. In *TIME '96: Proceedings of the 3rd Workshop on Temporal Representation and Reasoning (TIME'96)*, page 51, Washington, DC, USA, 1996. IEEE Computer Society.

[SV98] Eddie Schwalb and Lluis Vila. Temporal constraints: A survey. *Constraints*, 3(2/3):129–149, 1998.

[TA86] Abdullah Uz Tansel and M. Erol Arkun. Hquel, a query language for historical relational databases. In *SSDBM'86: Proceedings of the 3rd international workshop on Statistical and scientific database management*, pages 135–142, Berkeley, CA, US, 1986. Lawrence Berkeley Laboratory.

[TC90] Alexander Tuzhilin and James Clifford. A temporal relational algebra as a basis for temporal relational completeness. In *Proceedings of the sixteenth international conference on Very large databases*, pages 13–23, San Francisco, CA, USA, 1990. Morgan Kaufmann Publishers Inc.

[TCG+93] Abdullah Uz Tansel, James Clifford, Shashi Gadia, Sushil Jajodia, Arie Segev, and Richard Snodgrass, editors. *Temporal databases: theory, design, and implementation.* Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1993.

[The07] The Intelligent Systems Laboratory. *SICStus Prolog User's Manual.* Swedish Institute of Computer Science, PO Box 1263, SE-164 29 Kista, Sweden, release 4.0.2 edition, November 2007.

[Tho91] P. M. Thompson. *A Temporal Data Model Based on Accounting Principles.* PhD thesis, Department of Computer Science, University of Calgary, Mar 1991.

[Vil82] Marc B. Vilain. A system for reasoning about time. In *AAAI*, pages 197–201, 1982.

[Vil94] Lluis Vila. A survey on temporal reasoning in artificial intelligence. *AI Communications*, 7(1):4–28, 1994.

[VK86] Marc Vilain and Henry Kautz. Constraint Propagation Algorithms for Temporal Reasoning. In *Proc. Fifth National Conference on Artificial Intelligence*, pages 377–382, Philadelphia, PA, USA, 1986.

[VKvB90] Marc Vilain, Henry Kautz, and Peter van Beek. Constraint propagation algorithms for temporal reasoning: a revised report. pages 373–381, 1990.

[War83] D.H.D. Warren. An abstract prolog instruction set. Technical Note 309, Artificial Intelligence Center, SRI International, Menlo Park CA, 1983.

[Wie97] J. Wielemaker. Swi-prolog reference manual, 1997.

[Wik08] Wikipedia. Time — wikipedia, the free encyclopedia, 2008. [Online; accessed 28-August-2008].

[WZZ05] Fusheng Wang, Carlo Zaniolo, and Xin Zhou. Temporal xml? sql strikes back! In *TIME '05: Proceedings of the 12th International Symposium on Temporal Representation and Reasoning (TIME'05)*, pages 47–55, Washington, DC, USA, 2005. IEEE Computer Society.

[YAP06] The YAP Prolog System. http://www.ncc.up.pt/~vsc/Yap. 2006.