

A Constraint Database System for Temporal Knowledge

Roman Gross and Robert Marti

Institut für Informationssysteme
ETH Zentrum, CH-8092 Zürich
Switzerland

(gross,marti)@inf.ethz.ch

Abstract

This paper describes how the technology of deductive constraint database systems and constraint query languages can be used to represent and reason with temporal knowledge. First, we summarize our approach to manipulating constraints over reals within deductive database systems. This approach is based on the compile-time rewriting of clauses which are not admissible. Then, we show how the timestamping of facts and rules in temporal databases can be mapped to constraints over reals. Subsequently, we present a more efficient and elegant approach which is based on a special temporal constraint solver.

Keywords: temporal databases, temporal reasoning, deductive databases, constraint query languages

1 Introduction

Current database management systems ultimately return a set of ground substitutions for the variables occurring in a query, that is, equations of the form $X = c$ where X is a variable and c is a constant. In other words, every answer corresponds to a tuple of constant values, typically of type integer, real or string. However, many real-world problems can not be solved by associating constant values to all of the variables e.g. because insufficient information is available in order to compute a precise answer. Instead, some variables may take part in constraints in the form of complex equations or inequalities, or they may even be completely free. Problems which naturally give rise to such “partial” answers which are subject to certain constraints include configuration tasks, circuit design and temporal reasoning.

In a similar vein, conventional database systems typically fail to answer queries if at least one of the subqueries has infinitely many solutions. Hence, the answer set cannot be enumerated as it would be in ordinary systems. Typical problems of this kind are

periodically recurring events such as the weekly departure of a particular flight.

<i>FlightNr</i>	<i>DepDay</i>	<i>DepTime</i>
‘SR100’	1994/12/10	13:05
‘SR100’	1994/12/17	13:05
⋮	⋮	⋮

However, the solutions may be subject to certain conditions which can be represented as a finite collection of constraints. The tuples in the example above have in common that the *DepDay* is equal to 1994/12/10 modulo 7 days.

<i>FlightNr</i>	<i>DepDay</i>	<i>DepTime</i>
‘SR100’	1994/12/10 mod 7 days	13:05

Jaffar et al. have successfully merged constraint solving techniques to deal with the above mentioned problems with logic programming [9]. In the last few years, many systems based on the approach of constraint logic programming (CLP) have been implemented, e.g. CLP(\mathcal{R}). This approach has been extended to temporal annotated CLP [6] to handle temporal constraints as investigated in [13]. A drawback of these systems is that they are main memory based and can only handle a limited amount of data (efficiently). Moreover, these approaches cannot handle concurrent access by several users adequately.

Therefore, [12] have proposed the concept of a constraint query language (CQL) in order to represent and handle constraints in deductive databases. CQLs are similar to CLP-languages in that they support the management of non-ground facts and non-allowed rules as well as non-ground answers. However, to the best of our knowledge, no complete and reasonably efficient implementation of a CQL exists with the exception of our own DeCoR¹ system ([8], see below).

During the same period, a lot of research has been done in the areas of temporal databases (see e.g. [17, 1]) and temporal reasoning (e.g. [13]). Temporal databases not only contain information about the

¹DeCoR stands for DEDuctive database system with CONstraints over Reals.

present state of the real world, but also the history leading up to this state and/or possible future developments. Most of these efforts are based on the relational model and SQL.

In this paper, we argue that the constraint handling technology of DeCoR can elegantly be applied to managing temporal information, providing the functionality associated with typical temporal database systems. Moreover, temporal information which is incomplete and/or potentially infinite can be represented by the system.

The paper is structured as follows: Section 2 gives an overview of DeCoR and how it deals with constraints. Section 3 sketches how temporal knowledge can be represented in the existing DeCoR system, using constraints over reals. Section 4 presents a more elegant approach which relies on building a special purpose constraint solver which can deal with constraints over both time points (instants) and time intervals.

2 Overview of DeCoR

The DeCoR system is a prototype of a deductive constraint database system implemented as a strongly coupled front-end (written in Prolog) running on top of an SQL-based commercial database product (Oracle). As a result of this architecture, the DeCoR system features full database functionality (i.e., concurrency control, logging and recovery, authorization and physical data independence).

A DeCoR database consists of a set of clauses (facts and rules) which are mapped to SQL base tables and views in a straightforward way. Rule bodies and queries are translated into SQL statements which can then be executed by the underlying relational database system (DBMS). The queries are evaluated in a bottom-up fashion in an attempt to minimize the calls to the DBMS. In addition to the components of typical deductive database systems, the DeCoR system also contains components to store and handle constraints.

2.1 Deductive Database with Constraints

In the following we assume familiarity with deductive database systems, as e.g. described in [5, 19, 15]. These systems usually handle equations and inequalities ($=, \neq, <, \leq, >, \geq$) on arithmetic terms (with $+, -, *$) only if the variables occurring in these expressions are ground and therefore the arithmetic expressions can be evaluated. DeCoR generalizes the treatment of arithmetic relations by allowing variables to be non-ground. Hence, these built-in predicates can be viewed as constraints on potential values for those variables.

Definition 2.1 A constraint is a relation $(t_1 \Theta t_2)$ where t_1, t_2 are arithmetic terms and the symbol Θ represents any of the symbols $(=, \neq, <, \leq, >, \geq)$.

Definition 2.2 A generalized clause is an implication of the form

$$p_0(\overline{X}_0) \leftarrow p_1(\overline{X}_1), \dots, p_k(\overline{X}_k), c_1(\overline{X}_1), \dots, c_m(\overline{X}_m).$$

where p_i are user defined predicates and c_j are constraints. The \overline{X}_i are vectors of variables. A generalized clause does not have to be range-restricted.

This definition follows that given by Kanellakis et al. [12] for a generalized fact.

Definition 2.3 A system of generalized clauses form a database with constraints, called a constraint database.

Formulas in the DeCoR system may contain conjunctions (\wedge), existential quantifiers (\exists) as well as negation (\neg). Disjunctions have to be represented by multiple clauses. Without loss of generality, we assume in the following that clauses and queries are in standard form, i.e. all arguments in atoms are variables and each variable occurs at most once in a user-defined predicate, c.f. [8, 10].

Example 2.1 The standard form of the user-defined clause “ $p(X, Y) \leftarrow q(X, 5), r(X, Y, Y)$.” is

$$p(X, Y) \leftarrow q(X_1, Z), r(X_2, Y_1, Y_2), Z = 5, \\ X = X_1, X_1 = X_2, Y = Y_1, Y_1 = Y_2.$$

2.2 Types of Constraints

In deductive database systems which demand allowedness, all variables become instantiated (ground) during bottom-up evaluation [5]. This is no longer guaranteed in constraint database systems because some variables might only be constrained by inequalities or even completely free. Therefore, two types of variables are distinguished according to their *groundness* which can be determined in a bottom-up fashion.

Definition 2.4 A variable X in the body of a clause is ground $\langle g \rangle$ if

- X is bound to a constant c .
- X can be bound to a term $f(Y_1, \dots, Y_n)$ by solving the constraints of the clause. (For this to be possible, the Y_i have to be ground.)
- X appears in a user-defined predicate at an argument position which is ground. (The groundness of the arguments is determined by the groundness patterns of the body predicates, see below.)

Otherwise the variable is non-ground $\langle n \rangle$.

The arguments in the head of a clause inherit the groundness information from the body. The groundness of the arguments in turn determines the groundness patterns of the predicates.

Example 2.2 The groundness pattern $\langle ggn \rangle$ for a predicate determines that the first, second and fourth arguments are ground whereas the third argument is non-ground.

Based on the groundness information of the variables, constraints can be separated into evaluable and non-evaluable ones.

Definition 2.5 A constraint C is evaluable if

- C is an equation which can be transformed into $X = f(Y_1, \dots, Y_n)$ where each Y_i is ground.
- C is an inequality not containing any non-ground variable.

Otherwise, the constraint is non-evaluable.

It can easily be seen that evaluable constraints correspond exactly to those allowed in “ordinary” deductive database systems. As shown e.g. in [5, 4] these constraints can be translated into selection conditions in relational algebra in a straightforward way. This is not the case with non-evaluable constraints which have to be manipulated separately.

2.3 Constraint Lifting Algorithm

The constraint database system DeCoR delays the evaluation of non-evaluable constraints until they become evaluable (if ever). For that purpose, the clauses are rewritten at compile-time in such a way that at run-time, only evaluable parts have to be dealt with [8]. This is achieved by propagating non-evaluable constraints into dependent clauses. This process is denoted as *constraint lifting (CL)*.

The constraint lifting algorithm (CLA) consists of the following 5 steps which are applied iteratively on each clause C in a bottom-up fashion on the reduced dependency tree.

1. Rewrite clause C into standard form. In doing so, all equality constraints become explicit.
2. For each literal in the body of C , lift the non-evaluable constraints occurring in its respective definitions into C .
3. Simplify (solve) the resulting conjunction of constraints.
4. Split the simplified constraints into evaluable and non-evaluable ones.
5. Fold the evaluable parts (user-defined predicates and evaluable constraints) into a unique auxiliary predicate which contains as arguments the variables of the non-evaluable part (non-evaluable constraints).

Step 3 of the CLA depends on a domain-specific constraint solver which has to meet some special requirements [7]. For example, it must be able to deal with variables for which it is only known whether or not they will become ground at run-time. The DeCoR system contains a solver for constraints over reals based on Kramer’s rule and Fourier elimination.

The following example shows the abilities of the CLA in the DeCoR system.

Example 2.3 All facts to the predicate tax_rate are supposed to be ground. They contain the lower and

upper bound of income ranges and the tax rate to be applied.

Min	Max	Rate
0	5 000	0
5 000	25 000	0.03
25 000	55 000	0.07

$$\begin{aligned}
 tax(Inc, Tax) \leftarrow & \\
 & tax_rate(Min, Max, Rate), \\
 & Min \leq Inc, Inc < Max, \\
 & Tax = Rate * Inc, Min < 10\,000.
 \end{aligned} \tag{1}$$

Steps 1 to 3 of the CLA do not affect the rewriting of this clause because tax_rate is a base predicate. Steps 4 and 5 result in

$$\begin{aligned}
 tax(Inc, Tax) \leftarrow & \\
 & tax_aux(Inc, Tax, Min, Max, Rate), \\
 & Min \leq Inc, Inc < Max, \\
 & Tax = Rate * Inc.
 \end{aligned} \tag{2}$$

$$\begin{aligned}
 tax_aux(_, _, Min, Max, Rate) \leftarrow & \\
 & tax_rate(Min, Max, Rate), Min < 10\,000.
 \end{aligned} \tag{3}$$

(2) contains the non-evaluable parts of the original clause and (3) the evaluable ones.

The query $?- tax(I, T)$ is also rewritten by the CLA. In step 2, the body of (2) is lifted into the query and replaces the predicate tax_rate . The resulting query

$$\begin{aligned}
 ?- tax_aux(I, T, Min, Max, Rate), \\
 Min \leq I, I < Max, T = Rate * I.
 \end{aligned}$$

does not have to be processed further and can be answered by

I	T	Constraints
	0	$0 \leq I, I < 5\,000$
		$5\,000 \leq I, I < 25\,000, T = 0.03 * I$

As a consequence of this approach, the run-time query evaluation mechanism of the DeCoR system only has to deal with the evaluable parts of the clauses and hence can be realized using well known techniques developed for deductive databases. In particular, the usual optimization techniques developed for standard bottom-up evaluation (e.g. Magic Template [14], Constraint Pushing [16]) can be combined with the CLA.

3 Managing Temporal Knowledge in DeCoR

Managing temporal knowledge can be considered as an application of the general-purpose constraint database system DeCoR. The DeCoR system provides the functionality to implement most of the

typical features of temporal databases. Similar to the constraint-based framework of [13], the temporal database should support time points (following [11], we will use the term instant) and time intervals as well as relations between objects of these types. As mentioned earlier and in contrast to our approach the system described in [13] is main-memory based and therefore it is limited to handle large amount of data.

In this paper we restrict ourselves to relations with only one temporal argument which can be considered as the valid-time of a tuple. Nevertheless, our ideas can easily be generalized to relations with multiple temporal arguments such as bitemporal relations.

In the following, familiarity with temporal databases as e.g. described in [17, 1] is assumed. Moreover, we attempt to adhere to the terminology introduced in [11]. In order to improve readability, a special notation for temporal terms is used: Instant variables (event variables) will be denoted by $\mathcal{E}_1, \mathcal{E}_2, \dots$ whereas interval variables will be denoted by $\mathcal{T}_1, \mathcal{T}_2, \dots$. The start (end) point of a time interval \mathcal{T} will be denoted by \mathcal{T}^s (\mathcal{T}^e). A concrete instant is represented as e.g. /1994/5/22~13:40:13.6. An instant such as /1994/6/15~00:00:00 will be abbreviated to /1994/6/15. Finally, time intervals are considered as closed at the lower bound and open at the upper one. The format for such an interval is [/1994/6/15 - /1994/6/23).

3.1 Temporal Datatypes and Relations

While the DeCoR system supports the datatypes string, integer and real, only constraints over reals can currently be solved. To apply constraint solving ability to temporal terms, an instant \mathcal{E} can be mapped to a value of type real. This is done by calculating the number of seconds between \mathcal{E} and a reference point, e.g. /1970/1/1.

Example 3.1 *The predicate rate shows the instant and the exchange rate between two currencies*

$$\begin{aligned} & \text{rate}(/1993/4/23 \sim 13:45:12.4, 'US\$', 'SFR', 1.14) \\ & \quad \updownarrow \\ & \text{rate}(7.251183e + 08, 'US\$', 'SFR', 1.14) \end{aligned}$$

A consequence of the above mapping is that the best precision is obtained for instants near the reference point. This is usually not a problem because most databases contain instants within a few decades only or do not need a temporal granularity smaller than microseconds.

In historical databases, each fact is usually time-stamped with a valid time interval which represents the knowledge that the fact is true at every instant within this interval. The distinction between a valid time interval and all the instants within the interval leads to two different representations of intervals in the DeCoR system. The *implicit* representation emphasizes the validity of the fact at every instant in the interval. The *explicit* representation emphasizes the

time interval and gives direct access to the bounds of the interval.

Definition 3.1 *The implicit representation of a fact $p(\overline{X})$ valid in the interval $[T^s - T^e]$ is*

$$p(\overline{X}, \mathcal{E}) \leftarrow T^s \leq \mathcal{E}, \mathcal{E} < T^e.$$

The explicit representation of the same fact is

$$p(\overline{X}, T^s, T^e).$$

The advantage of the implicit representation is that the temporal conjunction $(p(\overline{X}, \mathcal{E}), q(\overline{X}, \mathcal{E}))$, which requires the intersection of the valid times associated with p and q , can be directly performed by the constraint solver of the constraint database system. On the other hand, it is not possible to extract the bounds of the interval. Hence, it is not possible to calculate the duration of the interval. This drawback disappears if the intervals are represented explicitly. However, this representation requires that temporal relations have to be defined explicitly too. The advantages and disadvantages of these two representations are similar to those of instants respectively time intervals. (We refer to [1] for the discussion of time intervals versus instants and further references.)

In contrast to “ordinary” deductive database systems, the temporal relations can easily be represented in a constraint database system such as DeCoR. As a consequence, the explicit representation of time intervals is preferred.

The following three examples serve to convey the idea how temporal relations can be realized as clauses in a constraint database.

Example 3.2 *The interval $[T_3^s - T_3^e]$ is the (non-empty) temporal intersection of the intervals $[T_1^s - T_1^e]$ and $[T_2^s - T_2^e]$*

$$\begin{aligned} & / * \text{inter}(T_1^s, T_1^e, T_2^s, T_2^e, T_3^s, T_3^e). * / \\ & \text{inter}(T_1^s, T_1^e, T_2^s, T_2^e, T_1^s, T_1^e) \leftarrow T_1^s \geq T_2^s, T_1^e \leq T_2^e. \\ & \text{inter}(T_1^s, T_1^e, T_2^s, T_2^e, T_1^s, T_2^e) \leftarrow T_1^s \geq T_2^s, T_1^e > T_2^e. \\ & \text{inter}(T_1^s, T_1^e, T_2^s, T_2^e, T_2^s, T_1^e) \leftarrow T_1^s < T_2^s, T_1^e \leq T_2^e. \\ & \text{inter}(T_1^s, T_1^e, T_2^s, T_2^e, T_2^s, T_2^e) \leftarrow T_1^s < T_2^s, T_1^e > T_2^e. \end{aligned}$$

Example 3.3 *The relation $[T_1^s - T_1^e]$ before $[T_2^s - T_2^e]$ can be written as*

$$\text{before}(T_1^s, T_1^e, T_2^s, T_2^e) \leftarrow T_1^e < T_2^s.$$

Example 3.4 *The duration D of an interval \mathcal{T} is calculated in seconds and can be written as*

$$\text{duration}(T^s, T^e, D) \leftarrow D = T^e - T^s.$$

The temporal relations can be used as ordinary user-defined predicates. This can be seen at the following example adapted from [1].

Example 3.5 *Supposing a system contains the time-stamped relation $\text{works_in}(\text{EmpNo}, \text{DepNo}, T^s, T^e)$, the query: “In what period ($[T3s, T3e]$) did the person with the employee number 1354 and 245 work in*

the same department (*Dep*) and how many days (*D*) were this?" would be written as:

$$\begin{aligned} &? - \exists T1s, T1e, T2s, T2e, S \\ & \quad (works_in(1354, Dep, T1s, T1e), \\ & \quad works_in(245, Dep, T2s, T2e), \\ & \quad inter(T1s, T1e, T2s, T2e, T3s, T3e), \\ & \quad duration(T3s, T3e, S), \\ & \quad S = 86400 * D). \end{aligned}$$

The last constraint $S = 86400 * D$ is required because the predicate *duration* calculates the difference of *T3s* and *T3e* in seconds (*S*) and not in days (*D*) as demanded in the query.

If the predicate *works_in* contains ground facts only, all variables in the query will be ground and hence all constraints lifted from *inter* and *duration* turn out to be evaluable. However, this query will return a partially instantiated answer if there is a person for whom only the starting point of his or her employment in a department is known. Such an answer cannot be returned by conventional temporal database systems because they do not have a constraint component. This weakness of conventional system can not be removed even by introducing a value "forever".

3.2 Temporal Reduction (Coalescing)

As pointed out e.g. in [2, 3, 1], it is necessary for temporal complete systems to allow coalescing time intervals for value-equivalent facts (facts which are identical with the exception of their temporal arguments). They refer to this operation as *temporal reduction*.

Top-down constraint handling systems such as described in [6, 13] do not coalesce value-equivalent facts because they are tuple oriented. Hence, in contrast to our approach, these systems are not temporally complete [3]. As shown in [3] such systems are less expressive than temporally complete ones.

Example 3.6 *If a historical database contains the timestamped relation works_in*

Name	Dep	Start	End
'Mary'	'R&D'	1982/01/01	1988/01/01
'Mary'	'Eng'	1988/01/01	1995/01/01

it is necessary to coalesce the timestamps to generate the intended answer to the query "Who worked more than 10 years in the company".

Of course, we have to perform temporal reduction in our representation of a temporal database as well. Because the reduction operator is second order and because it is not integrated in the underlying DeCoR system, the reduction has to be programmed explicitly for every user-defined predicate.

In the following, the reduction scheme for a generic predicate $p(\bar{X}, T^s, T^e)$ is presented. The reduction requires two additional relations. $p_clos(\bar{X}, T^s, T^e)$ contains the transitive closure obtained by pairwise coalescing the intervals pertaining to value-equivalent

tuples. The relation $p_red(\bar{X}, T^s, T^e)$ contains the temporally reduced facts.

$$\begin{aligned} p_clos(\bar{X}, T^s, T^e) &\leftarrow p(\bar{X}, T^s, T^e). \\ p_clos(\bar{X}, T_1^s, T_2^e) &\leftarrow p(\bar{X}, T_1^s, T_1^e), p_clos(\bar{X}, T_2^s, T_2^e), \\ &T_1^s < T_2^s, T_2^s \leq T_1^e, T_1^e < T_2^e. \end{aligned}$$

$$\begin{aligned} p_red(\bar{X}, T^s, T^e) &\leftarrow p_clos(\bar{X}, T^s, T^e), \\ &\neg \exists T_1^s, T_1^e (p_clos(\bar{X}, T_1^s, T_1^e), T_1^s \leq T^s, T^e \leq T_1^e, \\ &\neg (T_1^s = T^s, T^e = T_1^e)). \end{aligned}$$

3.3 Assessment

The typical functionality associated with temporal database systems can be implemented as an application of the DeCoR system. Moreover, the possibility to manipulate partially instantiated facts and answers as supported by the DeCoR system can be exploited nicely.

Unfortunately, this approach has some serious drawbacks. First, it is not very efficient because the CLA has to lift many constraints. In particular, each temporal conjunction results in the lifting of four constraint parts. This overhead is not acceptable for such a frequent operation. Second, as seen in the examples above, it is clumsy to explicitly write two arguments for every time interval.

Both drawbacks are primarily due to the facts that time intervals are represented by two points and that the database system knows nothing about the special semantics of these two arguments as interval bounds.

4 A Temporal Extension of DeCoR

The drawbacks mentioned above disappear if temporal semantics are directly built into the underlying database system. This is done in the TDeCoR system which is an extension of the constraint database system DeCoR. The extensions consist of temporal datatypes, constraints relating temporal and non-temporal variables and the incorporation of the reduction algorithm. The two main goals are improving (1) the efficiency of the evaluation of temporal queries and (2) the readability of temporal clauses and queries.

4.1 Constraints over Temporal Domains

The DeCoR system is extended by the two datatypes *instant* and *interval*. Constants of one of these types are denoted as described at the beginning of Section 3. In contrast to the ChronoLog system [2, 1], there is no special syntax for temporal arguments. However, as shown below, the unification of temporal arguments has a special semantics. We mention in passing that this design decision supports the definition of multiple temporal arguments, e.g. to represent valid and transaction time.

\mathcal{T}_1 precedes \mathcal{T}_2
\mathcal{T}_1 follows \mathcal{T}_2
\mathcal{T}_1 contains \mathcal{T}_2
\mathcal{T}_1 equals \mathcal{T}_2
\mathcal{T}_1 overlaps \mathcal{T}_2

Table 1: interval \times interval constraints

\mathcal{T}_1 before \mathcal{T}_2	\mathcal{E} before \mathcal{T}
\mathcal{T}_1 starts \mathcal{T}_2	\mathcal{E} starts \mathcal{T}
\mathcal{T}_1 during \mathcal{T}_2	\mathcal{E} during \mathcal{T}
\mathcal{T}_1 ends \mathcal{T}_2	\mathcal{E} ends \mathcal{T}
\mathcal{T}_1 after \mathcal{T}_2	\mathcal{E} after \mathcal{T}

Table 2: interval \times interval or instant \times interval constraints

Similar to the constraints ($=, \neq, >, <, \geq, \leq$) between terms of type real currently supported in the DeCoR system, we introduce the temporal constraints shown in Tables 1 and 2 (as suggested in [13]). As seen in Table 2, some constraints are overloaded in the sense that they represent relations between two intervals as well as between an instant and an interval. For example, the constraint *starts* can not only be used to relate intervals with the same start point, but also to extract or set the start point of an interval.

In addition to the above constraints, we introduce relations to extract different properties of an instant (see Table 3).

$year(\mathcal{E}, I)$	
$month(\mathcal{E}, I)$	
$day(\mathcal{E}, I)$	I th day of the month
$weekday(\mathcal{E}, I)$	I th day of the week
$hour(\mathcal{E}, I)$	
$minute(\mathcal{E}, I)$	
$second(\mathcal{E}, I)$	

Table 3: instant \times integer constraints

Note that constraints are relations rather than functions and can therefore be used in a bidirectional way.

Example 4.1 *In the query “in which month is Peter born” the instant argument \mathcal{E} is the given argument for predicate month*

$?- \exists \mathcal{E} \text{ birthday}('Peter', \mathcal{E}), \text{month}(\mathcal{E}, \text{Month}).$

whereas in the rule “Paul has to present the progress of his work every Monday” the instant argument \mathcal{E} is not known:

$\text{duty}('Paul', 'present work', \mathcal{E}) \leftarrow \text{weekday}(\mathcal{E}, 1).$

As seen in the example above, it is possible to express periodical knowledge in the TDeCoR system. (Indeed, the constraints listed in Table 3 are somewhat similar to the modulo relations of [18].)

The constraint *duration* (interval \times integer \times unit) calculates the duration of an interval. The third argument which has to be given determines the granularity in which the duration is measured.

Example 4.2 *The constraint duration can be used to define the span between the two instants $\mathcal{E}_1, \mathcal{E}_2$ to be 6 month*

$\mathcal{E}_1 \text{ starts } \mathcal{T}, \text{duration}(\mathcal{T}, 6, \text{month}), \mathcal{E}_2 \text{ ends } \mathcal{T}$

The constraint *intersect*($\mathcal{T}_1, \mathcal{T}_2, \mathcal{T}_3$) determines the explicit intersection \mathcal{T}_3 of two time intervals \mathcal{T}_1 and \mathcal{T}_2 .

In addition to the above datatypes and constraints, special syntax expresses which parts of a formula have to be temporally coalesced. The TDeCoR system does not make an implicit reduction because in some cases, the user does not want to perform a reduction. Moreover the system is more efficient if the coalescing is performed only on demand. Following [1], the formula F which has to be coalesced must be enclosed by braces $\{ \}$. (If multiple temporal arguments are present the user has to specify which ones have to be reduced by supplying a second argument within the braces)

Example 4.3 *The query “Who worked in our company for more than 25 years” needs coalescing*

$?- \exists \mathcal{T}, Y (\{ (\exists Dep \text{ works_in}(\text{Emp}, Dep, \mathcal{T}), \mathcal{T} \}, \text{duration}(\mathcal{T}, Y, \text{year}), Y \geq 25).$

The reduction algorithm realized in the TDeCoR system is similar to the approach described in [2]. In addition, it is extended in a straightforward way to temporally reduce facts with multiple temporal arguments in each user specified direction. This is for example necessary to reduce bitemporal relations.

4.2 Handling of Temporal Constraints

The constraint lifting algorithm presented in Sect. 2.3 is not restricted to constraints over variables of type real. Indeed, similar to the CLP scheme [9], the CLA depends on a concrete domain. Ultimately, the constraint solver invoked in step 3 of the CLA only determines which constraints can be simplified and solved. The integration of the constraint solver of DeCoR with a temporal constraint solver will be investigated in the next section.

In order to apply the CLA on clauses with temporal constraints, it is necessary to define (1) the standard form of a temporal formula and (2) when a temporal constraint is evaluable.

Non-temporal formulas are rewritten into standard form by replacing the second and every further occurrence of a variable X by a new unique variable X_i and a constraint $X = X_i$ (see Example 2.1). However, if a

variable of type time occurs multiple times in a formula, this transformation is not so useful since it imposes the constraint that the two intervals be exactly the same.

In temporal databases, it is usually much more interesting to merely impose the constraint that time intervals associated with different facts *overlap* (i.e., have a non-empty intersection). As a result, multiple occurrences of variables denoting time intervals are related via the *intersect* constraint. (Note that this strategy corresponds to the notions of temporal join respectively temporal conjunction.)

Example 4.4 *The temporal formula*

$$\text{emp}(ENo, Name, T), \text{work_in}(ENo, -, T), \\ \text{salary}(ENo, Sal, T)$$

will be rewritten in standard form as

$$\text{emp}(ENo, Name, T), \text{work_in}(ENo_1, -, T_1), \\ \text{salary}(ENo_2, Sal, T_2), \\ ENo = ENo_1, ENo = ENo_2, \\ \text{intersect}(T, T_1, T_3), \text{intersect}(T_3, T_2, -)$$

As a consequence of this approach, the readability is improved. As a case in point, consider the query of Example 3.5 which will be written

$$\text{works_in}(1354, Dep, T), \\ \text{works_in}(245, Dep, T), \\ \text{duration}(T, D, day).$$

In order to define the evaluability of a temporal constraint, the definition of the groundness of a variable has to be extended. The terms ground or non-ground given in Def. 2.4 do not allow to adequately categorize variables bound to partially instantiated intervals.

Definition 4.1 *A partially instantiated interval is an interval with one bound ground and the other non-ground.*

Definition 4.2 *A variable X bound to the partially instantiated interval T is start-ground (s) if the start point of T is ground. If the end point of T is ground then the variable X is end-ground (e). A ground interval variable is both start-ground and end-ground.*

Example 4.5 *A variable bound to the partially instantiated interval $[/1994/8/23\sim 12:24:13.8 - \mathcal{E}]$ is start-ground whereas a variable bound to the interval $[\mathcal{E} - /1996/1/1]$ is end-ground.*

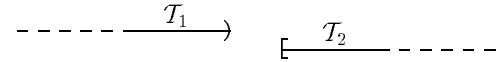
Table 4 shows which minimal groundness tags are required for every argument of a temporal constraint in order to guarantee its evaluability. (Note that the following partial order holds for groundness tags: $n < s, n < e, s < g, e < g$. Also, instant variables can only be ground or non-ground.)

Example 4.6 *The temporal constraint T_1 before T_2 can be evaluated if the end point of T_1 and the start*

<i>before</i>	$\langle es \rangle$
<i>after</i>	$\langle se \rangle$
<i>precedes</i>	$\langle en \rangle, \langle ns \rangle$
<i>follows</i>	$\langle sn \rangle, \langle ne \rangle$
<i>during, contains</i>	$\langle gg \rangle$
<i>equals</i>	$\langle gn \rangle, \langle se \rangle, \langle es \rangle, \langle ng \rangle$
<i>starts</i>	$\langle sn \rangle, \langle ns \rangle$
<i>ends</i>	$\langle en \rangle, \langle ne \rangle$
<i>overlaps</i>	$\langle eg \rangle$
<i>year, month...</i>	$\langle gn \rangle$
<i>duration</i>	$\langle gn \rangle, \langle sg \rangle, \langle eg \rangle$
<i>intersect</i>	$\langle ggn \rangle, \langle sge \rangle, \langle egs \rangle, \langle ngg \rangle, \langle gse \rangle, \\ \langle esg \rangle, \langle ges \rangle, \langle seg \rangle, \langle gng \rangle$

Table 4: Groundness patterns for evaluable temporal constraints

point T_2 are known. The following situation would evaluate to true.



4.3 Constraint Solving

For space reasons, the specification of a temporal constraint solver can not be given in this paper. However, a general specification for a solver in a constraint database system is given in [7]. Nevertheless, a temporal constraint solver can not simply be added to the existing non-temporal one. Instead, the two solvers have to collaborate by passing information about freshly bound variables between them. This is illustrated in example 4.7.

Example 4.7 *In the query “at which instant would John have worked twice as long in the sales department as in the engineering department?”*

$$?- \exists T_1, T_2, M_1, M_2 \\ (\text{works_in}('John', 'Eng', T_1), \\ \text{works_in}('John', 'Sal', T_2), \\ \text{duration}(T_1, M_1, month), \\ M_2 = 2 * M_1, \\ \text{duration}(T_2, M_2, month), \\ \mathcal{E} \text{ ends } T_2).$$

the temporal variable \mathcal{E} (the result we are looking for) depends on the non-temporal variable M_2 . M_2 depends on M_1 which in turn depends on the interval variable T_1 .

The first call of the temporal solver determines that the constraint “ $\text{duration}(T_1, M_1, month)$ ” is evaluable so that M_1 becomes ground. The temporal constraint “ $\text{duration}(T_2, M_2, month)$ ” remains non-evaluable because the first argument is suggested to be only start-ground and the second argument is non-ground. In the next step, the non-temporal solver determines “ $M_2 = 2 * M_1$ ” as evaluable and

therefore M_2 as ground. This enables the temporal constraint solver to identify the constraints “duration($\mathcal{T}_2, M_2, month$)” and “ \mathcal{E} ends \mathcal{T} ” as evaluable.

This example shows that it is necessary to iterate steps 3 and 4 of the constraint lifting algorithm. During this iteration ostensibly non-ground variables are determined to be ground because new constraints are identified to be evaluable. This may in turn lead to the solving of further constraints, and so on. The iteration terminates when no new non-ground variables can be determined as ground. If m non-temporal or event variables and n interval variables are present this is the case after at most $\max(m, 2 * n)$ steps.

This iteration takes place during the constraint lifting algorithm which on its own is executed at compile-time. At query-evaluation time no further manipulation of the temporal constraints is necessary.

5 Conclusions

In this paper, we have first shown how a deductive database system which handles generalized facts and non-allowed rules can be realized. Subsequently, we have demonstrated how its ability to manipulate constraints over reals can be used to represent the timestamping information typical of temporal databases. Finally, we have presented a more sophisticated approach to represent temporal knowledge which is based on the development and integration of a special purpose temporal constraint solver.

References

- [1] Michael Böhlen. *Managing Temporal Knowledge in Deductive Databases*. PhD thesis, ETH Zurich No. 10802, 1994.
- [2] Michael Böhlen and Robert Marti. Handling temporal knowledge in a deductive database system. In *Datenbanksysteme in Büro, Technik und Wissenschaft*, 1993.
- [3] Michael Böhlen and Robert Marti. On the completeness of temporal database query languages. In *Proc. 1st Int. Conf. on Temporal Logic*, July 1994.
- [4] Jan Burse. ProQuel: Using Prolog to implement a deductive database system. Technical Report TR 177, Departement Informatik ETH Zürich Switzerland, 1992.
- [5] Stefano Ceri, Georg Gottlob, and Letizia Tanca. *Logic Programming and Databases*. Surveys in Computer Science. Springer Verlag, 1990.
- [6] Thom Fruehwirth. Temporal logic and annotated constraint logic programming. In *IJCAI Workshop on Executable Temporal Logic*, 1993.
- [7] Roman Gross and Robert Marti. Compile-time constraint solving in a constraint database system. In *Workshop Constraints and Databases, Int. Logic Programming Symposium*, Ithaca, November 1994.
- [8] Roman Gross and Robert Marti. Handling constraints and generating intensional answers in a deductive database system. *Journal of Computers and Artificial Intelligence*, 13(2-3):233–256, 1994.
- [9] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Proc. of the 14th ACM Symposium on Principles of Programming Languages*, pages 111–119, January 1987.
- [10] Joxan Jaffar and Michael J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–582, 1994.
- [11] C. S. Jensen, J. Clifford, R. Elmasri, S. K. Gadia, P. Hayes, and S. Jajodia (eds). A glossary of temporal database concepts. *SIGMOD Record*, 23(1), March 1994.
- [12] Paris C. Kanellakis, Gabriel M. Kuper, and Peter Z. Revesz. Constraint query languages. In *Proc. 9th ACM Symp. on Principles of Database Systems (PODS)*, pages 299–313, Nashville, 1990.
- [13] Itay Meiri. Combining qualitative and quantitative constraints in temporal reasoning. In *AAAI*, pages 260–267, 1991.
- [14] Raghu Ramakrishnan. Magic Templates: A spellbinding approach to logic programs. In *Proc. Int. Conf. on Logic Programming*, pages 140–159, 1988.
- [15] Raghu Ramakrishnan, Divesh Srivastava, S. Sudarshan, and P. Seshadri. Implementation of the CORAL deductive database system. In *Proc. ACM Conf. on Management of Data (SIGMOD)*, 1993.
- [16] Peter J. Stuckey and S. Sudarshan. Compiling query constraints. In *Proc. ACM Symp. on Principles of Database Systems (PODS)*, 1994.
- [17] Adbullah Uz Tansel, James Clifford, Shashi K. Gadia, Sushil Hajodia, Arie Segev, and Richard Snodgras. *Temporal Databases: Theory, Design and Implementation*. Benjamin/Cummings Publishing Company, Inc., 1993.
- [18] David Toman, Jan Chomicki, and David S. Rogers. Datalog with integer periodicity constraints. In *Proc. Int. Logic Programming Symposium*, November 1994.
- [19] J. Vaghani, K. Ramamohanarao, David Kemp, Z. Somogyi, and Peter Stuckey. Design overview of the Aditi deductive database system. In *Proc. 7th Int. Conf. on Data Engineering*, pages 240–247, 1991.