UNIVERSITY OF CALIFORNIA
Irvine

Temporal Reasoning with Constraints

A dissertation submitted in partial satisfaction for the requirements for the degree of Doctor of Philosophy in Information and Computer Science

by

Edward Moshe Schwalb

Committee in Charge:
Professor Rina Dechter, Chair
Professor Mike Pazzani
Professor Dennis Kibler
Professor Yuval Shahar

UNIVERSITY OF CALIFORNIA
Irvine

Temporal Reasoning with Constraints

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy
in Information and Computer Science
by
Edward Moshe Schwalb

Committee in charge:
Professor Rina Dechter, Chair
Professor Dennis Kibler
Professor Michael Pazzani
Professor Yuval Shahar

1998

The dissertation of Edward Moshe Schealb is approved,
and is acceptable in quality and form for
publication on microfilm:

Committee Chair

University of California, Irvine
1998

To my parents,
Sara and Norbert Schwalb,

and to my wife,
Tammy.

Contents

List of Figures

List of Tables

Acknowledgements

My wonderful wife and best friend, Tammy, has helped me through my emotional roller coaster during the last three years. Without Tammy, my life would not be complete, and the completion of this dissertation would not be possible. The love, encouragement, and support from my parents, Sara and Norbert Schwalb, has made a world of difference. I am indebted to Professor Rina Dechter, my advisor, who introduced me to the topic of temporal constraint satisfaction, supported me through the long years in the Ph.D. program. In the early years, our intense collaboration has installed in me an effective writing style that will remain with me for years to come. Thanks also to Lluís Vila for the intense collaboration and friendship during the last three years. I would also like to thank my committee, Professor Dennis Kibler, Professor Mike Pazzani and Professor Yuval Shahar.

Curriculum Vitae

1991	Bsc. in Computer Science and Physics, Bar Ilan University.
1993	M.S. in Information and Computer Science, University of California, Irvine.
1998	Ph.D. in Information and Computer Science, University of California, Irvine.
	Dissertation: *Temporal Reasoing with Constraints.*

Abstract of the Dissertation

Temporal Reasoning with Constraints

by
Edward Moshe Schwalb
Doctor of Philosophy in Information and Computer Science
University of California, Irvine, 1998
Professor Rina Dechter, Chair

This dissertation is focused on representing and reasoning about temporal information. We design general temporal languages supported by specialized efficient inference procedures. The contribution is in combining the existing logic-based temporal reasoning languages with the existing temporal constraint models, and in designing new efficient inference algorithms for the combined languages. We explore a specific combination of Datalog, a polynomial fragment of logic programming, with Temporal Constraint Satisfaction Problems (TCSP). To render this combination meaningful, attention is given to the formal syntax, semantics and the inference algorithms employed. We address some historical challenges relevant to the introduction of time and constraints into logic programming.

The dissertation surveys and develops new and improved temporal constraint processing algorithms. When processing traditional Constraint Satisfaction Problems (CSP), path-consistency (PC) algorithms are polynomial. We demonstrate that when processing temporal constraints, PC is exponential, and thus does not scale up. To remedy this problem, two new polynomial algorithms are introduced: Upper Lower Tightening (ULT) and Loose Path Consistency (LPC). These algorithms are complete for a class of problems, called the STAR class. The empirical evaluation of these algorithms demonstrates a substantial performance improvement (up to six orders of magnitude) relative to other algorithms. We also demonstrate the existance of a phase transition for TCSPs.

Chapter 1

Introduction

It is hard to think of research areas in AI that do not involve reasoning about time. Medical diagnosis systems need to reason about time stamped medical histories and medical treatment plans involve scheduling the administration of drugs. Circuit analysis programs reason about the times that signals are sent and/or received. Schedulers need to ensure that the schedule meets a set of deadlines. These issues are addressed within the Temporal Reasoning subfield of Artificial Intelligence. While most other subfields of AI employ temporal terminology, the very goal of temporal reasoning is the development of a general theory of reasoning about time.

Time is important because it is associated with changes. In a world where no changes occur (e.g. no viruses infecting blood systems, no circuits moving electrical charges) the concept of time would become meaningless, and temporal reasoning would not be necessary. Consequently, languages for temporal reasoning should meet at least two requirements: They should allow describing what is true and what is false at various points in time, and they should enable expressing the laws governing the change of these truth values.

The majority of work on temporal reasoning languages is logic based. McCarthy and Hayes [69] introduced the situation calculus, a temporal formalism that, to this day, is the basis for many temporal representations. In this formalism, a situation is a snapshot of the world at a given moment, and actions map one situation to another. These actions define what is a lawful change by specifying a relationship (i.e. constraint) between the initial and final states. For example, let ON(A,B) be a proposition[1] which is *true* iff block A is on top of block B. Assume that initially, ON(A,B) is *true* and we execute the action of picking up A. This action changes the truth value of the proposition ON(A,B) because A is no longer on B[2]. The description of such an action must dictate that if ON(X,Y) is true before the action "pick up Y" is executed than ON(X,Y) no longer holds after the execution was completed.

[1]More precisely, ON(A,B) describes a set of states in which A is on B.

[2]The mapping is into a set of states in which A is *not* on B.

The situation calculus, *as described in the original paper*, makes strong commitments. The first is the modeling of discrete time, which precludes discussion about continuous change such as flow of water or electricity. A limitation of the situation calculus is the exclusion of concurrent actions. Finally, the most famous problem introduced by the situation calculus is the frame problem, which stems from the fact that the situation calculus requires describing explicitly everything that changes.

One of the strongest advocates of formal common sense reasoning has been Hayes [49]. He introduced the theory of histories which had a strong influence on temporal reasoning. In this theory, states can be associated with a time interval rather than a single time point. A justification for this approach was given in a form of a formalism describing the qualitative behavior of fluids which could be generalized to enable qualitative reasoning about general physical systems.

The interval-based view of time was further developed by Allen [5]. He identified 13 possible relations that could hold between two time intervals. The ontology he defined associates time intervals with either a property, an event or a process. A property is a proposition that is either *true* or *false*, namely "the pen is red". If property holds over an interval then it holds for all of its subintervals. If an event occurred over an interval, it did not't occur over any of its subintervals. For example, the event "John is going to the shop" is *true* only over the interval starting at the time John left and ending at the time John arrived at the shop. This statement is repeatable (i.e. can be repeated numerous times) but non-divisible. Process propositions are hybrid cases. For example, the statement "John is working" describes a process. If it is *true* for an interval, then it must be *true* for some subinterval, but does not need to be *true* for all subintervals (i.e. John could have had a break).

McDermott [70] began exploring the relationship between problem solving and theories of time. He proposed a theory of time and action to be used within a planning process. This theory takes the notion of a state as primitive, and associates it with a time point. His language distinguished between fact types and fact tokens where the token is an instantiation of the type. For example, "John is working" is a type while "John is working on May 2, 1990, from 8:00 to 17:00" is a token.

Subsequently, Shoham proposed a reified first order logic [98], which consolidated much of the prior work. The major contribution was addressing numerous technical issues in the qualification of time and formalization of a set of axioms that describe lawful change. This includes the realization that introducing time into logic requires modifying its syntax and semantics via a temporal qualification method, such as reification. Subsequently, Bacchus et al presented a non-reified first order logic [8] in which they invalidate the reasons that Shoham used to justify the reification of logic. Instead, a simple temporal qualification was used in which time is represented via temporal arguments.

Independently, Kowalski and Sergot proposed the Event Calculus [61], which is one of the first investigations into the difficulties of representing time and change within a logic programming framework. Finally, an integral logic was proposed [45]

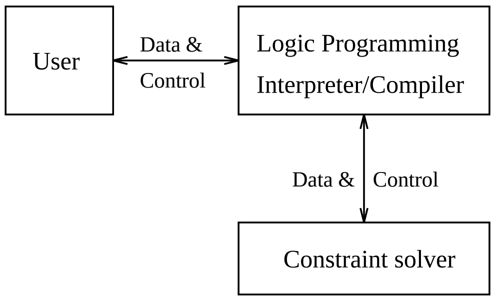


Figure 1.1: The logical structure of the temporal reasoner.

which focuses on the computational aspects of integrating over time intervals.

Parallel to the development of languages for temporal reasoning, various restricted frameworks for processing temporal relationships were introduced. These include the interval algebra [5], point algebra [110], Temporal Constraint Satisfaction Problems (TCSP) [24] and models combining quantitative and qualitative constraints [71, 55].

To extend the TCSP framework into a logic-based language there is a need to introduce constraints into logic. Constraint Logic Programming (CLP) is a field of research concerned with the syntactic, semantic and computational issues of introducing constraints into logic programming. The shortcoming of CLPs is the inability to perform temporal reasoning.

The constraint based reasoning paradigm, pursued in this thesis, combines logic-based *temporal* languages with *temporal* constraint models, as illustrated in Figure 1.1 The challenge is twofold: introduce *time* and *constraints* into logic programming. The user communicates with the system using a logic based language. Sentences in this languages are interpreted and executed using a reasoning algorithm (e.g. resolution). When this algorithm is executed, queries to the constraint solver are generated. These queries involve deciding consistency of a set of constraints or computing a set of feasible relations between constraint variables.

Our work is targeted at reasoning tasks commonly performed when the information is indefinite, incomplete or is naturally stated in terms of a set of constraints. Consider the following specific example of reasoning in a medical domain, where there are guidelines regarding the administration of drugs such as Digoxin, which is a brand name for digitalis, and potassium supplements. Digitalis is a well known medication for heart failure. Patients who have heart failure are often given diuretic drugs, such as Lasix, a brand name for Furosemide, to reduce their fluid load, usually within one hour if taken orally. However, Furosemide reduces the level of potassium in the blood, which can cause organ failure, such as heart arrytmia. To correct the depletion of potassium, potassium supplements are often administered. Unfortunately, digitalis toxicity may be potentiated by the potassium supplements and can potentially cause other types of heart arrhytmia. As a result, these two drugs (digitalis and potassium supplements) must not be taken together. However, taking them in sequence with sufficient time in between is often done for the same patients, since these patients are often using the diuretic drugs against the same heart failure. Representing constraints such as "*taking them in sequence with sufficient time in between*" and making inferences is often a critical aspect of medical reasoning, and is a central motivation for this thesis.

The contribution of this thesis is in combining existing logic-based temporal reasoning languages with the existing temporal constraint models. This is done with attention to CLP languages, which combine general (non-temporal) logic programming with general (non-temporal) constraint models. Specifically, we pursue research along two lines: (i) designing a logic programming based language (i.e. first component of the reasoner in Figure 1.1) and (ii) developing temporal constraint satisfaction algorithms to be used within the constraint solver (i.e. second component of the reasoner in Figure 1.1).

The languages we introduce are designed to enable temporal reasoning and address some historical challenges. These challenges include introducing time and constraints into logic programming. On the one hand, logic programming and CLP languages were not designed to represent time. On the other hand, temporal languages were not designed to accommodate constraints. We show that the standard resolution algorithm needs to be modified to accomplish some intended temporal inference. Once these modifications are made, the resulting algorithm is complete with respect to the intended temporal semantics.

Temporal Constraint Satisfaction Problems (TCSP) are used as the underlying constraint model. The variables in this model represent the times at which events occur and the constraints represent temporal relations between them.

The dissertation is organized as follows: The rest of this chapter introduces some relevant background. First, Constraint Satisfaction Problems (CSP) and Temporal CSPs are introduced. Subsequently, the logic programming framework is presented, and three relevant languages are briefly described: (i) Datalog, (ii) Datalog$_{nS}$ and Constraint Logic Programs (CLP).

Chapter 2 presents a short survey of results reported for Temporal Constraint Satisfaction Problems (TCSP). The three TCSP classes presented are: (i) Qualitative Point-Point constraints, (ii) Qualitative Point-Interval constraints, (iii) Metric Point-Point constraints, and (iv) combined metric and qualitative constraints. For each of these classes, the chapter surveys tractable classes and constraint processing techniques.

Chapter 3 presents new results that improve the state of the art in processing metric temporal constraints. Two algorithms are presented, Upper Lower Tightening

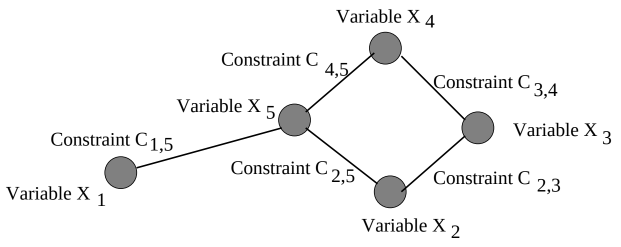


Figure 1.2: A sample constraint graph.

(ULT) and Loose Path Consistency (LPC). A new tractable class is identified. The empirical evaluation shows efficiency improvements of six orders of magnitude (i.e. by a factor of 10^6).

Chapters 4,5 present two new temporal languages, TCSP-Datalog and Token-Datalog, designed according to the constraint based reasoning paradigm. These languages are designed to address issues regarding the embedding of time and constraints into logic programming. The languages are supported by inference algorithms which are complete with respect to their temporal semantics.

Finally, in Chapter 6, we present some examples illustrating the applicability of the results presented in this work.

1.1 CSP Background

1.1.1 Discrete CSP

A (Binary) Constraint Satisfaction Problem (CSP) [65, 34, 22] consists of a finite set of **variables** $\{X_1, \ldots, X_n\}$, a **domain** of possible values D_i for each variable X_i, and a set of (binary) **constraints**. A **binary constraint** C_{ij} between variables X_i and X_j is a set of ordered pairs (x_i, x_j) where $x_i \in D_i$, $x_j \in D_j$, namely $C_{ij} \subseteq D_i \times D_j$. A binary CSP can be represented by a **labeled constraint graph** $G(V, E)$. Each vertex V represents a variable X_i and is labeled by the domain of this variable, D_i. Each edge e_{ij} in E between v_i, v_j represents the constraint C_{ij} and is labeled by the set of ordered pairs C_{ij} specifies. If $C_{ij} = D_i \times D_j$ then $e_{ij} \notin E$.

Example 1: An hypothetical constraint graph is illustrated in Figure 1.2. The variables are $X_1, \ldots, X_5$, and the constraints are $C_{1,5}, C_{2,5}, C_{2,3}, C_{3,4}, C_{4,5}$.

A **solution** is an n-tuple $(x_1, \ldots, x_n)$, representing an assignment of $x_i \in D_i$ to each variable X_i, such that $\forall i, j \ (x_i, x_j) \in C_{ij}$. A CSP is **consistent** if and only if it has at least one solution. Two CSPs are **equivalent** iff they have the same set of solutions. A value $x_i \in D_i$ is *feasible* iff there exists at least one solution in which $X_i = x_i$. A pair $(x_i, x_j) \in C_{ij}$ is *feasible* iff exists at least one solution in which $X_i = x_i$ and $X_j = x_j$. The *minimal domain* D_i^{min} contains only feasible values and

C_{12}	(1,3)(1,4)(2,4)(3,1)(4,1)(4,2)
C_{13}	(1,2)(1,4)(2,1)(2,3)(3,2)(3,4)(4,1)(4,3)
C_{14}	(1,2)(1,3)(2,1)(2,3)(2,4)(3,1)(3,2)(3,4)(4,2)(4,3)
C_{23}	(1,3)(1,4)(2,4)(3,1)(4,1)(4,2)
C_{24}	(1,2)(1,4)(2,1)(2,3)(3,2)(3,4)(4,1)(4,3)
C_{34}	(1,3)(1,4)(2,4)(3,1)(4,1)(4,2)

Table 1.1: The list of allowed pairs for every constraints of the 4-queen example

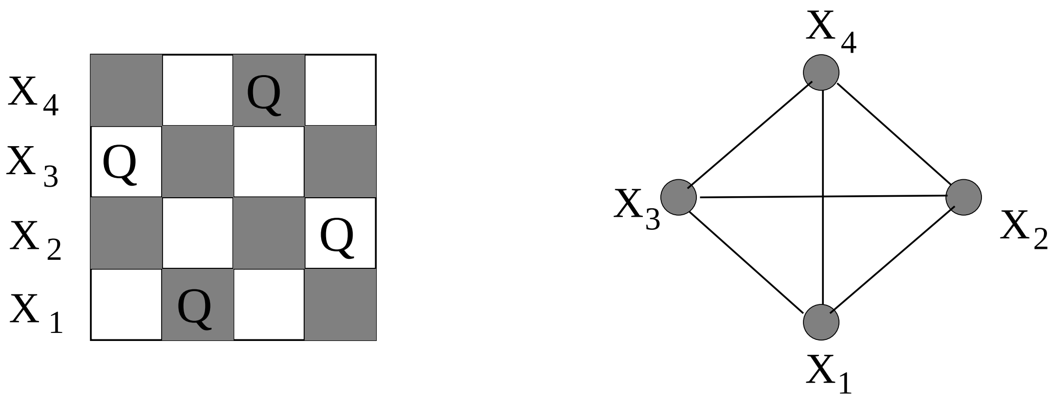


Figure 1.3: The 4-queen problem and constraint graph.

the *minimal constraint* C_{ij}^{min} contains only feasible pairs. A CSP is minimal iff all its constraints are minimal.

Example 2: As an example, consider the 4 queen problem (see Figure 1.3). The problem is to position 4 queens on a chess board of 4x4 such that no pair of queens is on the same row, column or diagonal. To formalize this problem as a CSP we could use 4 variables, $\mathcal{X} = \{X_1, X_2, X_3, X_4\}$ with the domain $D_1, \ldots, D_4 = \{1, 2, 3, 4\}$. The assignment $X_i = j$ means that the i-th queen is positioned in the j-th column. The complete set of constraints for the 4-queen example are given in table 1.1. The value assignment $X_1 = 1, X_2 = 2, X_3 = 3, X_4 = 4$ is not a solution because $(1, 2) \notin C_{12}$ [3]. The value assignment $X_1 = 2, X_2 = 4, X_3 = 1, X_4 = 3$ is a solution because $(2, 4) \in C_{12}$, $(2, 1) \in C_{13}$, $(2, 3) \in C_{14}$, ..., $(x_i, x_j) \in C_{ij}$.

Notational conventions We use $v = |V|$ as the number of variables and $e = |E|$ as the number of constraints.

Tasks The most common CSP tasks are deciding consistency (namely whether exists a solution), deciding whether a single value (or a pair of values) is feasible and finding a single (or all) solution.

[3]Only the list of allowed pairs is given.

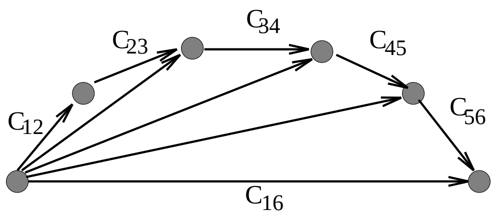


Figure 1.4: Enforcing path-consistency.

Local Consistency is commonly used to process CSPs [34, 26, 22].
2-consistency A CSP is 2-consistent, or **arc-consistent**, iff for every pair of variables X_i, X_j, for every value $x_i \in D_i$ exists a value $x_j \in D_j$ such that the pair (x_i, x_j) is in C_{ij}. Enforcing arc-consistency requires $O(d^2v^2)$ steps where d is the maximum domain size.
3-consistency For enforcing 3-consistency, two basic operations are required: constraint $\underline{\textit{intersection}}$ $\cap$ and $\underline{\textit{composition}}$ $\circ$. The intersection $\cap$ of two constraints is a set of ordered pairs of values appearing in both. The composition $C_{ij} = C_{ik} \circ C_{kj}$ is the set of pairs $C_{ij} \;=\; \{(x_i, x_j) \mid \exists x_k \;\; (x_i, x_k) \in C_{ik}, \; (x_k, x_j) \in C_{kj}\}$.

A CSP is *3-consistent* iff $\forall i, j, k \;\; C_{ij} \subseteq C_{ik} \circ C_{kj}$. A CSP is path-consistent if and only if the constraint induced along any path $X_{i_1}, \ldots, X_{i_m}$ is looser or equal to the direct constraint between X_{i_1} and X_{i_m}, namely iff

$$C_{i_1, i_k} \subseteq C_{i_1, i_2} \circ C_{i_2, i_3} \circ \cdots \circ C_{i_{m-1}, i_m}.$$

Theorem 1: *[65] A CSP is path-consistent iff it is 3-consistent.*

To understand the intuition behind this result, consider the constraint graph in which nodes and arcs are labeled by the corresponding domains and constraints. We would like to compute the constraints induced by the composition of $C_{12} \circ C_{23} \circ \cdots \circ C_{k-1,\, k}$ along a path from X_1 to X_k, (see Figure 1.4). This can be done by assigning C_{13} to be the result of the composition $C_{12} \circ C_{23}$. Similarly, we assign $C_{14} = C_{13} \circ C_{34}$, and so on until we obtain the constraint $C_{1,k} = C_{1,k-1} \circ C_{k-1,k}$. After path-consistency is enforced, we are guaranteed that $C_{1,k}$ is tighter or equal to the constraint induced along this path.

Due to theorem 9, path-consistency can be enforced by applying, for every constraint C_{ij} and every variable k where $k \neq i, \; k \neq j$, the operation $C_{ij} \leftarrow C_{ij} \cap (C_{ik} \circ C_{kj})$ until a fixed point is reached, namely until no changes are made. An algorithm for enforcing path-consistency, called PC, is given in Figure 1.5. For completeness we also present a weaker version called *Directional Path-Consistency* (DPC) which is more efficient but less effective [24, 71]. DPC enforces path-consistency relative to a specific variable ordering only.

The notions of 2-consistency and 3-consistency can be generalized. A CSP is locally k-consistent if every value assignment to $k-1$ variables, which is consistent

```
**Algorithm PC**
1. $Q \leftarrow \{(i,k,j) | (i < j)\ and\ (k \neq i,j)\}$
2. **while** $Q \neq \{\}$ **do**
3.     select and delete a path $(i,k,j)$ from $Q$
4.     **if** $C_{ij} \neq C_{ik} \circ C_{kj}$ **then**
5.         $C_{ij} \leftarrow C_{ij} \cap (C_{ik} \circ C_{kj})$
6.         **if** $C_{ij} = \{\}$ **then** exit (inconsistency)
7.         $Q \leftarrow Q \cup \{(i,j,k),(k,i,j) \mid 1 \leq k \leq n, i \neq k \neq j\}$
8.     **end-if**
9. **end-while**

**Algorithm DPC**
1. **for** $k \leftarrow n$ downto 1 by -1 **do**
2.     **for** $\forall i,j < k$ such that $(i,k),(k,j) \in E$ **do**
3.         **if** $C_{ij} \neq C_{ik} \circ C_{kj}$ **then**
4.             $E \leftarrow E \cup (i,j)$
5.             $C_{ij} \leftarrow C_{ij} \cap (C_{ik} \circ C_{kj})$
6.             **if** $C_{ij} = \{\}$ **then** exit (inconsistency)
7.         **end-if**
8.     **end-for**
9. **end-for**
```

Figure 1.5: Algorithms PC and DPC for CSP and TCSP alike.

with all the constraints on these $k-1$ variables, can be extended by assigning a value to every additional k-th variable such that all the constraints between the k-th variable and the previously assigned $k-1$ variables are satisfied.

1.1.2 Temporal CSP

Temporal CSP (TCSP) is a particular type of CSP where the variables represent the times at which events occur and the constraints represent a set of allowed temporal relations between them. A temporal constraint C_{ij} is of the form

$$(X_i\ r_1\ X_j)\ \vee\ \ldots\ \vee\ (X_i\ r_k\ X_j)$$

where X_i, X_j are temporal variables and $r_1, \ldots, r_k$ are a set of Basic Temporal Relations (`BTR`). A shorthand for this constraint is written in the form

$$X_i\ \ \{r_1, \ldots, r_k\}\ \ X_j \quad or \quad C_{ij} = \{r_1, \ldots, r_k\}.$$

This constraint is satisfied if at least one of the relations $r_1, \ldots, r_k$ holds between X_i, X_j. For example, let X_i, X_j be two intervals, and let r_1`= Before` and r_2`= After`. The corresponding temporal constraint is

$$\texttt{X}_i \texttt{ Before X}_j \quad \vee \quad \texttt{X}_i \texttt{ After X}_j$$

which can be rewritten using the shorthand

$$\texttt{X}_i \; \{\texttt{Before,After}\} \; \texttt{X}_j$$

or alternatively, $C_{ij} = \{\texttt{Before,After}\}$.

In some cases it is possible to introduce a special variable X_0 whose domain D_0 has a single element $D_0 = \{0\}$. A unary constraint $C_i = \{r_1, \ldots, r_k\}$ can then be described by a binary constraint C_{0i} containing the same set of BTRs $\{r_1, \ldots, r_k\}$. For example, let X_i be the i-th time point variable, then the unary constraint

$$X_i \in [10, 20] \cup [30, 40]$$

can be described using the binary constraint

$$X_i - X_0 \in [10, 20] \cup [30, 40]$$

because $X_0 = 0$.

Chapter 2 presents a short survey of the various types of TCSPs and the related literature. Chapter 3 demonstrates that the complexity of TCSPs stems from the disjunctive nature of its constraints. Even enforcing path-consistency is exponential. To cope with the disjunctions, two polynomial algorithms are introduced and shown to improve efficiency of processing TCSPs by orders of magnitude.

1.2 Logic Programming Background

Our approach for temporal reasoning is based on Logic Programming (LP). This is a class of languages in which information is described using "if-then" rules or assertions. Each rule is of the form `H :- B`$_1$`,...,B`$_n$ where `H` is the head of this rule and `B`$_1$`,...,B`$_n$ are atoms that constitute its body.

Example 3: Consider the statement "Medicine A should be administered if either symptom X or symptom Y are observed" which can be described by the following two-rule *propositional* logic program:

```
MedicineA :- SymptomX.
MedicineA :- SymptomY.
```

where the propositions `SymptomX` and `SymptomY` evaluate to *true* whenever symptoms X and Y are observed respectively, and the medicine should be administered whenever the proposition `MedicineA` is *true*.

First order logic programming is similar to propositional logic programming, except that propositions are replaced with predicates. A predicate $\mathtt{P(X_1,\ldots,X_n)}$ is a mapping of a tuple $X_1 = x_1, \ldots, X_n = x_n$ (i.e. value assignment to $X_1, \ldots, X_n$) to *true* or *false*.

Example 4: Let us introduce two variables, `Medicines, Symptoms`, taking values from a set of drugs and symptoms respectively. Let `MedicineA` be a drug in the domain of the variable `Medicines` and let `SymptomX` and `SymptomY` be two symptoms in the domain of the variable `Symptoms`. In addition, we use the variable `t` to represent time. We could extend the logic program from example 3 as follows:

```
Administrate(MedicineA,t+10) :- Observe(SymptomX,t).
Administrate(MedicineA,t+10) :- Observe(SymptomY,t).
```

where `Administrate(X,t)` evaluates to *true* iff medicine X was administered at time `t` and `Observe(Y,t)` evaluates to true iff symptom Y was observed at time `t`.

In Chapter 5, we show how to modify this program to describe statements such as "Medicine A should be administered at least 1 hour and at most 8 hours after symptom X was observed".

1.2.1 Datalog

Datalog [4, 53] is a tractable fragment of logic programming which we choose as a basis for the languages we design. Datalog is defined over a set of variables, $\mathcal{X} = X_1, \ldots, X_n$, which take values from their finite domains $\mathcal{D} = D_1, \ldots, D_n$. An *atom* is a formula $p(r_1, \ldots, r_k)$ where p is predicate over the terms $r_1, \ldots r_k$. A *term* r_i is either a variable or a value from its domain. In other words, functions are not allowed. The rest is similar to the above definition of logic programs. A *ground term* is a value (i.e. constant) and a *ground atom* is an atom whose terms are ground. A *fact* is a ground atom and a *database* is a finite set of facts. A *rule* is of the form $\mathtt{H\ :\!-\ B_1,\ldots,B_n}$. A *goal* is written in the form $\mathtt{:\!-\ B_1,\ldots,B_m}$. A *Datalog program* is a finite set of rules together with a database. Every program entails a set of facts.

Models. A model is a set of facts $\mathcal{F}=\{\mathtt{F_1,\ldots,F_n}\}$, satisfying the following condition: For every rule, if $\mathtt{B_1,\ldots,B_m}$ are all in $\mathcal{F}$ then `H` must also be in $\mathcal{F}$.

Queries. A query is a finite set of rules together with a goal formula. All variables in the rules and the goal are implicitly universally quantified. An answer to a query is the model of the program augmented with the query.

Theorem 2: *[4, 53] Every Datalog program Ψ has a unique minimal model M_Ψ containing all and only those atoms that are entailed by Ψ. M_Ψ can be computed in polynomial time.*

1.2.2 Datalog$_{nS}$

A possible approach for introducing time into Datalog was explored by Datalog$_{nS}$ [15, 16]. It extends Datalog with the *functional* sort that enables modeling discrete time.

Terms can be either data or functional terms. Data terms are either variables or constants as in Datalog. *Functional terms* (or arguments) are built from a distinguished functional constant '0', data constants, data variables, functional variables and function symbols. For example, the *functional term* '5' can be obtained by applying the function +1 five times on the constant '0', namely $5 = +1(+1(+1(+1(+1(0)))))$. Every functional term contains either 0 or a single occurrence of a functional variable. For example, if f is a function symbol and T is a functional variable then $f(T)$ and $f(T, a)$ are legal functional arguments but $f(T, T)$ and $f(T, 0)$ are not.

Predicates can be either functional or non-functional. Functional predicates take exactly one functional argument (assumed to be the first argument) and non-functional predicates take only data arguments.

Example 5: Consider the atom `Take(T, Medicine)` where the first term `T` is a functional term describing time and the second term, `Medicine`, is a data term. The program

```
Take(0,Medicine).
Take(T+8,Medicine) :- Take(T,Medicine).
```

is equivalent to the set of facts

```
Take(0,Medicine), Take(8,Medicine), Take(16,Medicine), ...
```

A *ground term* is a term which does not contain any variables. The *Herbrand universe* of a program Ψ, denoted $\mathcal{U}_\Psi$, is the set of *all possible ground terms* that are built on top of constants in Ψ. The *Herbrand base* of Ψ, denoted $\mathcal{B}_\Psi$, is the set of *all possible atoms* that can be constructed with predicates from Ψ taking arguments from $\mathcal{U}_\Psi$. An *interpretation* I is any subset of $\mathcal{B}_\Psi$. A *model* is an interpretation I which, for every rule, if $\mathtt{B}_1,\ldots,\mathtt{B}_m$ are all in $\mathcal{F}$ then `H` must also be in I.

In this work, we borrow the notion of *successor functions* and use them to represent infinite periodic relation such as those described by Datalog$_{nS}$.

1.2.3 Constraint Logic Programming

Constraint Logic Programming (CLP) began as a natural merger of two declarative paradigms: constraint solving and logic programming. This combination helps make CLP programs both expressive and flexible, and in some cases, more efficient than other kinds of programs. Though a relatively new field, a significant body of knowledge on the utility of introducing constraints into logic programming was accumulated in the Constraint Logic Programming (CLP) literature [106, 51].

A CLP program is composed of rules of the form

$$\texttt{H :- } \texttt{B}_1\texttt{,...,B}_n\texttt{,C}_1\texttt{,...,C}_m$$

where H, $\texttt{B}_1,\dots,\texttt{B}_n,\texttt{C}_1,\dots,\texttt{C}_m$ are all atoms, H is the head of the rule, $\texttt{B}_1,\dots,\texttt{B}_n$ are the non-constraint atoms in the body of the rule and $\texttt{C}_1,\dots,\texttt{C}_m$ are the constraint atoms in the body of the rule. Note that constraint atoms cannot appear in the head.

CLP programs differ from traditional logic programs in way the constraint atoms $\texttt{C}_1,\dots,\texttt{C}_m$ are processed. In standard logic programming, the constraint and non-constraint atoms are treated equally. In CLP, specialized techniques for deciding which constraints are entailed, called constraint propagation algorithms, may be applied to the constraint atoms only [106, 51]. While it may be possible to represent some CLP programs using general logic programs, without the CLP inference engine it is very difficult and time consuming to implement algorithms that compute the intended answers to queries [106, 51].

To illustrate some CLP concepts, consider defining the relation sumto(n, 1+2+$\cdots$+n), which maps every natural number n to the sum $\Sigma_{i=1}^{n}$. The following example CLP program, presented in [51], describes this relation.

```
sumto(0,0).
sumto(N,S) :- N≥1, N≤S, sumto(N-1, S-N).
```

To find out which values of n are mapped to sums which are at most 3, we use the query "S≤3, sumto(N,S)", which gives rise to three answers: (i) N=0, S=0, (ii) N=1,S=1, and (iii) N=2,S=3. The computation that arrives at the third answer (i.e. N=2, S=3), described by the set of subgoals generated at every step, is as follows:

Initially, the set of subgoals contains the query atoms:
Step 1: S≤3, sumto(N,S).

Subsequently, the query variables are mapped into new variables used to describe the constraints. This is done because CLP requires that the input variables do not participate in the constraint propagation.
Step 2: $\texttt{S}\leq\texttt{3, N=N}_1\texttt{, S=S}_1\texttt{, N}_1\geq\texttt{1, N}_1\leq\texttt{S}_1\texttt{,}$
$\texttt{sumto(N}_1\texttt{-1, S}_1\texttt{-N}_1\texttt{).}$

Subsequently, the rule sumto(N,S) :- N≥1, N≤S, sumto(N-1, S-N) is applied to derive a new set of variables and constraints:
Step 3: $\texttt{S}\leq\texttt{3, N=N}_1\texttt{, S=S}_1\texttt{, N}_1\geq\texttt{1, N}_1\leq\texttt{S}_1\texttt{,}$
$\texttt{N}_1\texttt{-1=N}_2\texttt{, S}_1\texttt{-N}_1\texttt{=S}_2\texttt{, N}_2\geq 1\texttt{, N}_2\leq\texttt{S}_2\texttt{,}$

```
          sumto(N₁-1, S₁-N₁).
```

The next iteration is performed by applying the same rules as in step 3.
The precondition for unifying with the fact `sumto(0,0)` is obtained.

```
Step 4:   S≤3, N=N₁, S=S₁, N₁ ≤S₁,
          N₁-1=N₂, S₁-N₁=S₂, N₂ ≥ 1, N₂ ≤S₂,
          N₂-1=0, S₂-N₂=0.
```

At this point, `sumto(N₂-1,S₂)` is unified with `sumto(0,0)`. The constraints in this final state are traced back to compute the answer `N=2, S=3`.

The operation of replacing $\mathtt{N}_i$ with $\mathtt{N}_{i-1}+1$ is not automatically performed by the standard logic programming inference engine. While it may be possible to write a standard logic program performing the above computation, this may not be easy. In general, implementing the CLP inference engine using standard logic programming may require significant effort, and in some cases, is not feasible [51]. This example illustrated the following key CLP features:

- Constraint atoms are used to specify the query.
- During execution, new variables and constraints are created.
- At every inference step, the collection of constraint atoms is tested as a whole for satisfiability.

Logic Programming (LP) is an instance of CLP in which all the constraints are explicit equality [51]. Among the CLP languages that received most attention are: CLP($\mathcal{R}$) [52] which is defined over the real numbers and describes linear arithmetic constraints. CHIP [29, 106] is defined over the booleans (i.e. true,false) and of bounded (i.e. not infinite) subsets of the integers. Prolog III [18, 17] is defined over the booleans, real numbers and strings. Second generation CHIP systems are CLP($\mathcal{FD}$) [28], Echinda [47], Flang [66], cc($\mathcal{FD}$). There are other CLP languages including LOGIN [2], LIFE [3] which compute over the domain of feature trees, BNR-Prolog [78] computes over boolean variables, Trilogy [112, 113], CAL [1], λ-Prolog [73] and its derivatives L_λ [72], ELF [80].

In some more detail, CLP($\Re$) introduces constraint variables whose domain is the set of real numbers. The constraints are linear inequations over these variables. Temporal constraints could be described by inequality constraints over the real numbers, as is done by Temporal CSPs, surveyed in Chapter 2 and analyzed in Chapter 3. The semantics of CLP($\Re$) is obtained by augmenting the standard LP semantics with the specific semantics of the constraint domain. An interpretation is composed of the standard LP interpretation and a value assignment to the constraint variables. A model is an interpretation which satisfies all the rules. The truth value of a non-constraint atom is determined according to the standard LP semantics. A constraint atom evaluates to *true* iff the constraint it specifies is satisfied.

CLP($\Re$) appears to be a natural candidate language for temporal reasoning, but it has some shortcomings. In chapters 4,5 we show that the use of CLP as a basis for temporal reasoning requires enhancing its syntax, semantics and proof procedure. Consider representing the statement "the signals a,b are always (i.e. for every time point) complementary, namely `a` $\leftrightarrow \neg$ `b`. Consider using `On(s,t)`, which evaluates to *true* iff the signal `s` is on at time point `t`. We could describe the above statement as follows:

```
On(a,t) :- On(¬b,t).            On(a,t) :- ¬On(¬a,t).
On(¬a,t) :- On(b,t).            On(b,t) :- ¬On(¬b,t).
```

where `¬a,¬b` denote the negation of a,b respectively and `¬On()` denotes stratified negation [4], which is a well known technique for introducing negation into logic programs. One problem is that the semantics of CLP does not enable inferring, from this program, that if the signal `a` was *on* during the interval $[t_1, t_2]$, and the signal `b` was *on* during the interval $[t_3, t_4]$, then the two intervals must be disjoint, namely these facts induce the constraint $(t_2 < t_3) \vee (t_4 < t_1)$.

1.3 Overview of Contributions

1.3.1 Processing Temporal Constraints

Processing temporal constraints involves taking, as input, a set of constraints over a set of variables, which typically describe the times events occur. For example, consider the following scenario: A large NAVY cargo must leave New York starting on March 7, go through Chicago and arrive at Los Angeles within 8-10 days. From New York to Chicago the delivery requires 1-2 days by *air or* 10-11 days on the *ground*. From Chicago to Los Angeles the delivery requires 3-4 days by air *or* 13-15 days on the ground. In addition, we know that an AIRFORCE cargo needs to be transported using the same terminal in Chicago as required for the NAVY's cargo transportation (i.e. the intervals of NAVY and AIRFORCE shipments should not overlap). The transportation of the AIRFORCE cargo should start between March 17 and March 20 and requires 3-5 days by air *or* 7-9 days on the ground.

The variables in this example are the cargo departure and arrival times. There are quantitative and qualitative constraints about the durations of the transport. The quantitative constraints specify *a set of intervals*, each describing how long each possible action requires (e.g. ground vs air transportation). Intervals are used instead of specific durations to accommodate some uncertainty in the duration. The qualitative constraints specify that, because the AIRFORCE and NAVY's cargo transportation need to use a common resource (i.e. the same terminal) the times at which they are scheduled to utilize this terminal must be disjoint.

Given the above constraints, we are interested in answering questions such as: "are the constraints satisfiable?", "can the NAVY cargo arrive in Los Angeles on March 13-14?", "when should the cargo arrive in Chicago ?", "how long may the NAVY cargo transportation take?". Most of these queries can be reduced to the task of deciding consistency of a Temporal CSP. When time is represented by (or isomorphic to the) integers[4], deciding consistency is in NP-complete [24, 71]. For qualitative networks, computing the tightest set of equivalent constraints (called the minimal network) is in NP-hard [44, 24]. In both qualitative and quantitative models, the source of complexity stems from allowing disjunctive relationships between pairs of variables, namely from considering several possible courses of action.

Chapter 2 surveys the state-of-the art in representing and processing temporal constraints. It is a self containing quick introduction into the field of temporal constraint satisfaction [83, 92, 91]. It covers algorithm complexity, algorithms and their empirical evaluation.

Chapter 3 presents efficient and effective polynomial *approximation* algorithms which are only guaranteed to correctly decide *inconsistency*, but they are not guaranteed to correctly decide *consistency* [85, 86, 87, 88]. We show that in contrast to discrete CSPs, where Path-Consistency (PC) algorithms are polynomial, when processing metric temporal constraints, algorithm PC may require time which grows exponentially in the size of the problem. This may result in an exponential blowup, leading to what we call *fragmentation*.

We address the *fragmentation problem* by presenting two algorithms called Upper-Lower-Tightening (ULT) and Loose-Path-Consistency (LPC). We demonstrate that these algorithms avoid fragmentation and are effective in detecting inconsistencies. We also discuss several variants of the main algorithms, called Directional ULT (DULT), Directional LPC (DLPC) and Partial LPC (PLPC).

We show that all these algorithms are complete for a class of problems, called STAR tractable class, in which the binary constraints are restricted to be singletons (but the unary constraints are not restricted). We demonstrate that this new tractable class is commonly encountered when scheduling events within non-contiguous time windows.

We address two questions *empirically*: (1) which of the algorithms presented is preferable for detecting inconsistencies, and (2) how effective are the proposed algorithms when used to improve backtrack search.

To answer the first question, we show that enforcing path-consistency may indeed be exponential in the number of intervals per constraint while ULT's execution time is almost constant. Nevertheless, ULT is able to detect inconsistency in about 70% of

[4]This is always the case in practice.

Method	Syntax
Reification	True(Residence(John,LA),I1). True(Residence(John,NY),I2). Before(I1,I2).
Arguments	Residence(John,LA,I1). Residence(John,NY,I2). Before(Residence(John,LA,I1),Residence(John,NY,I2)).
Tokens	Residence(John,LA,I1). Residence(John,NY,I2). Before(I1,I2).

Table 1.2: Temporal Qualification Methods.

the cases in which enforcing path-consistency does. Algorithm LPC further improves on ULT and, while being efficient, is capable of detecting almost all inconsistencies detected by PC.

To answer the second question, we apply the new algorithms in three ways: (1) in a preprocessing phase for reducing the fragmentation before initiating search, (2) in forward checking algorithm for reducing the fragmentation during the search and detecting dead-ends early, and (3) in an advice generator for dynamic variable ordering. We show that both ULT and LPC are preferred to PC and that LPC is the best algorithm overall through experiments with hard problems which lie in the transition region [79, 20]. We show that the performance of backtrack search can be improved by several orders of magnitude when using LPC for preprocessing, forward checking and dynamic ordering.

1.3.2 Temporal Constraint Logic Programming

In Chapters 4,5,6 we combine logic-based temporal reasoning languages with temporal constraint models, and design efficient inference algorithms for the combined languages [89, 90, 108, 109, 84]. We investigate a combination of the most restrictive deductive database language, Datalog Attention is given to the formal syntax, semantics and the inference algorithms employed. To introduce time and temporal constraints into logic programming, three components are required: (i) a temporal qualification method, (ii) a theory of time and temporal incidence and (iii) a temporal constraint domain.

Temporal qualification is the method in which non-temporal sentences are qualified with time. Consider describing the statement "John's residence was in LA before he moved to NY." We could use the facts

```
Residence(John,LA).   Residence(John,NY).
```

and introduce the temporal statement "`Residence(John,LA)` before `Residence(John,NY)`". There are three known ways to do so: (i) temporal reification, (ii) temporal arguments and (iii) time tokens. The use of these method is illustrated in Table 1.2.

Temporal incidence is the method by which the properties of the atoms in the language (which is logic-based) are described. For example, part of a temporal incidence theory is the homogeneity axiom which specifies that if `Holds(`$A \rightarrow B$`,` t_1, t_2`)` is *true* then the proposition $A \rightarrow B$ holds for every point inside the interval $[t_1, t_2]$.

Finally, the *temporal constraint domain* specifies the class of atomic constraints being used together with axioms describing their semantics. For example, we could use constraints such as $X < Y$ accompanied by the transitivity axiom $(X < Y) \wedge (Y < Z) \Rightarrow (X < Z)$.

In Chapter 4 we present two new languages called TCSP-Datalog and Token-Datalog, which are novel combinations of features inherited from their predecessors TCSP, Datalog [4, 53], Datalog$_{nS}$ [15, 16] and CLP [51]. These languages have well defined syntax and semantics and have the following properties:

1. The syntax is described using terminology and notions from traditional logic programs. The semantics are intuitive and are described in a declarative fashion (i.e. model-based).

2. Our approach builds on top of the framework established by the situation calculus, where fluents are used to represent what is true at every point in time. *Fluents* are *propositions having different truth values at different points in time.* This implies that our languages suffer from the famous frame (and ramification) problem in the same way that many other formalisms do.

3. We address temporal incidence by introducing a well defined theory of temporal incidence which enables representing both instantaneous and non-instantaneous events. It also can express the instantaneous and non-instantaneous holding fluents.

4. We use logic programming as the computational basis for making inferences. The *SLD-resolution* algorithm is modified by introducing *TCSP-resolution.* This allows making inferences which are correct with respect to the temporal semantics.

5. We use TCSPs for describing the constraints on the times at which events occur and the times that fluents are true, false or change values. Efficient algorithms for processing TCSPs are brought to bear.

6. Periodic relation can be defined using unary successor and predecessor functions. For example, to describe the period of one week we could write a rule stating that Sunday is followed by Monday, which is followed by Tuesday, . . ., Sunday is followed by Monday. In our language we write `Monday = successor(Sunday),` `Tuesday = successor(Monday), Wednesday = successor(Tuesday)` etc. In this example, the function "`successor`" described an *infinite periodic relation.*

Using this function we can express periodic occurrences such as"John is talking to his stock broker every Sunday between 10:00am and 11:00am".

7. Time tokens are used to improve the expressiveness and obtain numerous benefits, as described in Section 4.5. The increase in expressiveness implies an increase in complexity.

8. An inference algorithm is presented and its soundness and completeness is proved. This algorithm modifies standard SLD-resolution by introducing a new unification method capable of unifying constraints and ground token terms.

9. There is a wide range of applications that can benefit from the use of our temporal languages. In Chapter 6 we present four example domains.

Chapter 2

Temporal Constraint Satisfaction: State Of The Art

A *temporal constraint satisfaction problem* (TCSP) is a framework for representing and answering queries about events and the temporal relations between them. The work in the area of TCSP has progressed along three different lines: (i) identifying tractable classes, (ii) developing both efficient, exact algorithms for tractable classes and (iii) developing efficient approximation algorithms for processing problems which may or may not be tractable. In this chapter we survey results on three classes of TCSPs, namely *qualitative interval*, *qualitative point*, *metric*, and some of their combinations. Most of the techniques reported are based on two principles: enforcing local consistency and enhancing backtracking search.

2.1 Introduction

Constraint Satisfaction Problems (CSP) involves a set of constraints over a set of variables, where each variable has a set of possible values called its domain. Most queries of interest can be answered by deciding whether the set of constraints is satisfiable.

This paradigm is appropriate in domains where the information is indefinite, incomplete or is naturally stated in terms of constraints. Such domains are found in many areas of computer science including databases, computer aided design, software engineering, parallel computation, operational research and artificial intelligence.

Temporal CSP (TCSP) is a particular type of CSPs where the variables represent the times at which events occur and the constraints represent a set of allowed temporal relations between them. To illustrate the use of TCSPs, consider representing the constraints on the schedule of the following patient treatment plan. There are 3 exams, each followed within 12 hours by treatment session. Exams and treatments are completed within 4 hours and must be at least 8 hours apart. The exams require resources available on Jan 8-12, 20-21. The treatment session is performed by a therapist that is available 9:00am to 2:00pm on Tue and Thu.

Common queries are:

- Is there a schedule[1] ?
- Find a schedule.
- Find all schedules.
- What are the feasible times for an exam or a treatment?
- What are the feasible relations between two exams or treatments?
- What are the feasible relations between all exams and treatments?

Different TCSPs are defined depending on the time unit that variables represent (i.e. time points vs. time intervals) and the nature of the constraints (qualitative vs. metric). For example, qualitative constraints may specify that the treatment interval begins after the exam interval. Quantitative (metric) constraints may restrict the starting time of i-th exam, denoted t_i, to satisfy $t_i \in [8, 12] \cup [20, 21]$.

This chapter surveys the techniques for deciding TCSP satisfiability and answering TCSP queries. The classes of TCSPs surveyed are *qualitative point*, *qualitative interval*, *metric point*, and some of their combinations. Processing techniques were developed along three different lines of research: (i) identifying tractable classes and

[1]an assignment of starting and ending times for each exam and treatment such that all the constraints are satisfied

developing efficient exact algorithms for these classes, (ii) developing exact exponential search algorithms and (iii) developing efficient polynomial approximation algorithms. Most of the techniques developed are based on two principles: (i) enforcing path-consistency and (ii) enhancing naive backtracking search.

The chapter is organized as follows. Section 2.2 presents general definitions and techniques for TCSPs. Section 2.3 surveys qualitative point TCSPs, Section 2.4 surveys qualitative interval TCSPs, Section 2.5 surveys metric TCSPs and Section 2.6 surveys their combinations.

2.2 Temporal Constraint Satisfaction Problems (TCSP)

Although TCSPs are derived from CSPs which are described in the introduction, there are numerous fundamental differences enumerated below. A temporal constraint C_{ij} is of the form

$$(X_i \ r_1 \ X_j) \ \vee \ \ldots \ \vee \ (X_i \ r_k \ X_j)$$

where X_i, X_j are temporal variables and $r_1, \ldots, r_k$ are a set of Basic Temporal Relations (`BTR`). A shorthand for this constraint is written in the form

$$X_i \ \ \{r_1, \ldots, r_k\} \ \ X_j \quad \mathit{or} \quad C_{ij} = \{r_1, \ldots, r_k\}.$$

This constraint is satisfied if at least one of the relations $r_1, \ldots, r_k$ holds between X, Y. For example, let X, Y be two intervals, and let r_1`= Before` and r_2`= After`. The corresponding temporal constraint is

$$\texttt{X}_i \ \texttt{Before} \ \texttt{X}_j \quad \vee \quad \texttt{X}_i \ \texttt{After} \ \texttt{X}_j$$

which can be rewritten using the shorthand

$$\texttt{X}_i \ \{\texttt{Before,After}\} \ \texttt{X}_j$$

or alternatively, $C_{ij} = \{\texttt{Before,After}\}$.

Unary Constraints For Point TCSPs (qualitative and metric), it is common to introduce a special variable X_0 whose domain D_0 has a single element $D_0 = \{0\}$. A unary constraint $C_i = \{r_1, \ldots, r_k\}$ can then be described by a binary constraint C_{0i} containing the same set of `BTR`s $\{r_1, \ldots, r_k\}$. For example, let X_i be the i-th time point variable, then the unary constraint

$$X_i \in [10, 20]$$

can be described using the binary constraint

$$X_i - X_0 \in [10, 20]$$

because $X_0 = 0$. Qualitative Interval TCSPs cannot describe unary constraints.

Singleton Labelings A *singleton labeling* of a constraint C_{ij} is one of its BTRs. A *singleton labeling* of a TCSP N is a labeling of all the constraints in N.

Solutions A solution of a TCSP is a singleton labeling (rather than a variable instantiation) which is *consistent* . Even when a constraint C_{ij} is *not* specified in the input (because it is universal), every solution must specify some basic temporal relation that holds between X_i and X_j.

Feasible Relations The CSP notion of feasible values is replaced by the TCSP notion of feasible relations, because the solutions of TCSPs are singleton labelings rather than variable instantiations. A relation r between X_i, X_j is feasible iff exists at least one solution in which C_{ij} is labeled by r. The set of feasible relations between X_i and X_j constitutes the *minimal constraint* C_{ij}^{min}. When all the constraints are minimal the TCSP is said to be *minimal.*

Techniques were developed along three lines of research:

- Identifying *tractable subclasses* and developing specialized algorithms for these subclasses. They are characterized by two parameters: (i) properties of the constraint graph (e.g. degree, width) and (ii) the types of the constraints used.

- Developing *polynomial consistency-enforcing algorithms* that are sound but incomplete. When these algorithms detect an inconsistency the input constraints are guaranteed to be unsatisfiable. When these algorithms do not detect an inconsistency, the constraints are not guaranteed to be satisfiable (i.e. they might be unsatisfiable).

- Enhancing *search algorithms*. There are two well known methods: (i) backtracking search (ii) iterative refinement search (i.e. GSAT [93]). The former is guaranteed to terminate with the correct answer but does not scale up due to its exponential complexity. The latter scales up well but is not guaranteed to terminate with a solution.

For many tractable classes, enforcing path-consistency correctly decides consistency. Consequently, path-consistency algorithms became the most important family of approximation algorithms for TCSPs.

Operators Let R (and sometimes "?") denote the *universal constraint*, namely the constraint which contains all possible relations and thus poses no restrictions. The basic operations on temporal constraints (used to enforce path-consistency) are the following:

- *Complement* ($\neg$): $\neg C_{ij} = R - C_{ij}$.

- *Converse* ($\sim$): $^{\sim}(r_1, \ldots, r_k) = (^{\sim}r_1, \ldots, ^{\sim}r_k)$.
- *Intersection*: $S \cap T$ is the set intersection of the BTRs in S and T.
- *Composition*: $S \circ T$ is the disjunction of the individual composition of all atoms in T with all atoms in S, namely

$$T \circ S \;=\; (t_1, \ldots, t_p) \circ (s_1, \ldots, s_q) \;=\; ((t_1 \circ s_1), (t_1 \circ s_2), \cdots, (t_p \circ s_q)).$$

The above operations over BTRs are defined for each TCSP class separately.

Arc-Consistency Binary constraints relative to X_0 (recall that $X_0 = 0$) could be used to restrict the domains and a TCSP is arc-consistent iff $\forall i, j \; C_{ij} \subseteq C_{i0} \circ C_{0j}$ where $i \neq j$. Consequently, arc-consistency is a variant of path-consistency processing triangles that include X_0 only.

Search Methods To define the search space we distinguish between *complete* and *partial* singleton labelings. In a *complete* labeling each of the constraints consists of a single BTR while in a *partial* labeling some constraints may consist of disjunctions (or sets) of BTRs. Consistency of a complete labeling can be decided in $O(v^3)$ steps where v is the number of variables [27, 71].

The *search space* of a TCSP is defined over all possible *partial* singleton labelings. A naive backtracking search algorithm successively labels each constraint with one of its BTRs as long as the resulting partial labeling is consistent. Once inconsistency is detected, a dead-end is identified and the algorithm backtracks.

Figure 2.1 presents a backtracking algorithm, which reduces the number of future singleton assignments which lead to dead-ends, using *forward checking* as follows [63, 88]: Tighten the constraints by enforcing path-consistency. Thereafter, choose a disjunctive constraint and replace it with one of its BTRs. Enforce path-consistency again and replace another constraint in the tightened network by one of its BTRs. Repeat this process until either inconsistency is detected (by enforcing path-consistency) or all the constraints specify BTRs. When inconsistency is detected, declare a dead-end and backtrack by undoing the last BTR labeling. When all the constraints specify BTRs and the singleton labeling is path-consistent, terminate with this labeling as the solution.

The search algorithm chooses the next singleton assignment to a constraint non-deterministically. Thus the number of dead-ends encountered strongly depends on strategy for deciding on the ordering of the constraints to be labeled as well as the order BTRs are selected. For most tractable subclasses, however, enforcing path-consistency at step 2 (see Figure 2.1) is sufficient to guarantee that a solution can be found in a backtrack free manner, namely without encountering any dead-ends.

```
Backtracking
1.  $Depth \leftarrow 0$;
2.  Apply PC or DPC; this removes some redundant BTRs.
3.  if inconsistency was detected then
4.      if $Depth = 0$ then exit with failure.
5.      Undo the last BTR labeling.
6.      $Depth \leftarrow Depth - 1$; Go to step 8.
7.  if this is a singleton labeling (i.e. all constraints specify BTRs) then
            exit with the solution.
8.  Replace (non-deterministically) a disjunctive constraint by a single BTR.
9.  $Depth \leftarrow Depth + 1$; Go back to step 2.
```

Figure 2.1: The backtracking algorithm for TCSP.

2.3 Qualitative Point Constraints (PA)

Qualitative point TCSPs have been defined by Vilain and Kautz [111] and follow the classical view of time as a collection of instants related by qualitative relations such as $<$, $\leq$ or $\neq$.

A qualitative point TCSP is defined as follows:

Variables represent time points. The domains can be the naturals, integers, rationals or reals.

Relations The *basic relations*, namely the BTRs, are all possible order relations between two points, namely BTR $= \{<, =, >\}$. Three subalgebras have been studied:

name	abbrv	relations
basic point algebra	**BPA**	$<, =, >, ?$
convex point algebra	**CPA**	$\emptyset, <, =, >, \leq, \geq, ?$
point algebra	**PA**	$\emptyset, <, =, >, \leq, \geq, ?, \neq$

where '?' stands for the universal constraint.

2.3.1 Basic Point Algebra (BPA)

A **BPA** is a Point-Point TCSP whose constraints can specify four possible relations: $<, =, >$ and the universal constraint. It is either inconsistent or it must induce a strict partial order. If it is consistent then the non-universal input constraints are *minimal*. Thus, finding a solution is equivalent to finding a total order. This can be done using *topological sort* in $O(v+e)$ steps. Enforcing path-consistency also correctly decides consistency and computes the minimal constraints but requires $O(v^3)$ steps [111].

2.3.2 Convex Point Algebra (CPA)

A ConvexPoint Algebra (CPA) is a Point-Point TCSP whose constraints can specify six possible relations: $<, \leq, =, \geq, >$ and the universal constraint. The set of constraints can be represented as a weighted, directed graph using the following translation:

$$\begin{array}{lll} x_i = x_j & \textit{translates to} & x_i \xrightarrow{0} x_j \quad x_j \xrightarrow{0} x_i \\ x_i \leq x_j & \textit{translates to} & x_i \xrightarrow{+\infty} x_j \quad x_j \xrightarrow{0} x_i \\ x_i < x_j & \textit{translates to} & x_i \xrightarrow{+\infty} x_j \quad x_j \xrightarrow{-\epsilon} x_i \end{array}$$

Consequently, for the restricted case in which the relations $<, >$ are not allowed, finding a solution accounts for finding the *shortest-path* using Dijkstra's algorithm in $O(v^2)$ steps. Otherwise, if $<, >$ are allowed, we need to apply Floyd-Warshall *all-pairs shortest-paths* algorithm which is equivalent to enforcing path consistency in $O(v^3)$ steps [62, 19].

2.3.3 Point Algebra (PA)

Deciding Consistency

In *IxTeT* [41], a **PA** TCSP is translated into a *directional* graph whose edges are labeled by either $\leq$ or $\neq$, called $\leq$-$\neq$-graph. The translation is performed as follows:

$<$	*translates to*	$\xrightarrow{\leq}, \xrightarrow{\neq}$	?	*translates to*	no edge
$>$	*translates to*	$\xleftarrow{\leq}, \xleftarrow{\neq}$	$\leq$	*translates to*	$\xrightarrow{\leq}$
$=$	*translates to*	a single vertex after "collapsing" the two vertices	$\geq$	*translates to*	$\xleftarrow{\leq}$
			$\neq$	*translates to*	$\xleftarrow{\neq}$

where $\xleftarrow{rel}$ is a directional edge labeled by *rel*. The resulting graph, called a $\leq$-$\neq$-graph, has the following property [42] (not given as a theorem):

> "... A $\leq$-$\neq$-graph is consistent iff no pair of vertices connected by a $\neq$ edge are involved in a loop through $\leq$ edges."[2]

The algorithm for detecting consistency collapses every $\leq$-loop into a single vertex. If two collapsed vertices connected by a $\neq$-edge then the TCSP is inconsistent.

Identifying $\leq$-loops is done using standard algorithm for computing *strongly connected components* (SCC) [100]. Efficient algorithms for computing the SCC are based

[2]This claim was not presented as a theorem, and no proof was given in the original reference [42].

on two-way topological sort and require $O(v+e)$ steps [99]. $\{S_1, S_2, \ldots, S_q\}$ be the set of identified SCCs. The input **PA** TCSP is consistent iff the following TCSP is consistent:

- A variable X_i is introduced for each SCC S_i
- A constraint C_{ij} is given by $\forall i, j \in [1, q],\ C_{ij} \leftarrow \bigcap_{\substack{v \in S_i \\ w \in S_j}} C_{vw}$

This equivalent TCSP can be computed in $O(v)$ steps. Gerevini and Schubert follow the same approach ([40] theorems 2.8, subsection 3.1 and theorem 3.2).

Finding a Solution

Once $\leq$-loops are removed, a solution can be computed using topological sort in $O(v+e)$ steps.

Answering Queries

The feasible relations between all pairs of variables can be determined by computing the minimal constraint network. Path consistency can find the feasible relations for **BPA** and **CPA** but it is not complete for **PA**. Figure 2.2 shows a counter-example, commonly known as the *forbidden subgraph* [103].

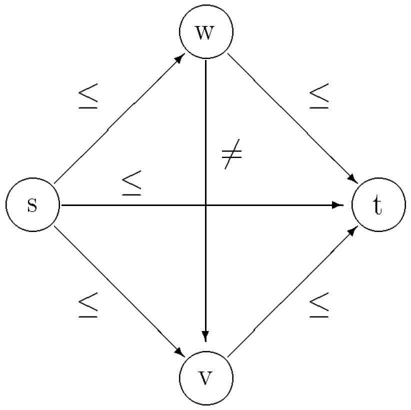


Figure 2.2: The unique non-minimal path-consistent **PA** TCSP.

For **PA** TCSPs, the minimal constraint network can be obtained by enforcing *4-consistency*. The algorithm is based on the following observation: The forbidden subgraph, illustrated in Figure 2.2, must be included in every **PA** TCSP which is path-consistent but not minimal [102][3]. This property dictates the following two step algorithm:

[3]A slight mistake in the proof is corrected in [39]

1. Enforce path consistency; this requires $O(v^3)$ steps.
2. Search systematically for the *forbidden subgraphs* and update the labels; this requires $O(e_{\neq}v^2)$ steps where $e_{\neq}$ is the number of $\neq$ constraints.

Although the worst case complexity of this algorithm is $O(v^4)$, it was observed empirically that the path consistency step, whose complexity is only $O(v^3)$, dominates the computation [101].

The 4-consistency algorithm based on the forbidden graph can be improved to process dynamic problems, where the variables and constraints are added and/or removed while several feasible queries are posted. The improvements are based on mantaining an internal representation that approximates a complete graph, allows efficient query answering and supports incremental update of temporal constraints.

Using an indexed spanning tree

IxTeT [42] builds and uses an internal representation based on (i) computing the *maximal weight spanning tree*, (ii) adding some residual edges between different branches of the tree, and (iii) labeling the nodes with an index that expedites query answering.

The indexed spanning tree is computed in $O(v+e)$ steps and allows efficient *retrieval* and *update*. Experimental results show that both retrieval and update are done in linear time [42]. Although the *IxTeT* system has clear practical interest, it will fail to compute the correct answer when the input TCSP includes the *forbidden subgraph* or when the $<$ relation is induced by $<$-paths. In other words, this algorithm is not sound.

Arranging time points into chains

TimeGraph II is intended (and optimized) to support natural language understanding where chain-like aggregates are dominant [40]. The internal data structure, called *timegraph*, is constructed to describe chains, and the algorithms for building and maintaining the timegraph are oriented to maximize the length of these chains. Building a *time-graph* involves three steps: (i) ranking the vertices, (ii) computing next-greater links, (iii) propagating $<$ through *forbidden graphs*.

In *TimeGraph II* the feasible relation between two events can be computed in $O(e+v)$ However, if the two events are in the same chain or related by $\neq$ this query can be answered in constant time.

2.3.4 Summary

Worst case bounds have been established for the tasks of deciding consistency, finding a solution and generating the minimal **BPA**, **CPA** and **PA** TCSP [101]. These results are difficult to contrast with the empirical evaluation of algorithms optimized to answer feasible relation queries for a restricted domain. *IxTeT* experiments show

	Task	Technique	Time Cost worst case	Time Cost average case
van Beek	Deciding Consistency Finding a solution	collapsing SCCs	$O(v+e)$	
	All feasible relations (Minimal Rep.)	PC+*forbidden graphs*	$O(\max(v^3, e_{\neq}v^2))$	$O(v^3)$
IxTeT	Feasible Relation		$O(v)$	
	Adding new Relation		$O(v)$	
TimeGraph II	Building timegraph	$O(e+v)$	$O(e)$	
	Feasible Relation	$O(e+v)$	$O(v)$	

Table 2.1: Results obtained for Point-Algebra TCSP.

that a structure based on an indexed maximal spanning tree can be used to answer both feasible relation queries and dynamic updating of constraints. *TimeGraph*'s major improvement upon *IxTeT* was in providing correct answers when the TCSP includes $<$-paths and *forbidden graphs*. The results are summarized in Table 2.1 above.

2.4 Qualitative Interval Constraints (IA)

Qualitative Interval TCSP, also called the Interval Algebra (IA), is a model for representing qualitative temporal information common in natural language [5].

Variables represent time intervals.

Domains are usually ordered pairs of integers or rationals.

Constraints are built upon a set of thirteen possible *basic* relations between a pair of intervals, namely

$$BTR = \left\{ \begin{array}{c} \textbf{before}, \textbf{after}, \textbf{meets}, \textbf{met\_by}, \\ \textbf{overlaps}, \textbf{overlaps\_by}, \textbf{during}, \textbf{contains}, \textbf{equals}, \\ \textbf{starts}, \textbf{started\_by}, \textbf{finishes}, \textbf{finished\_by} \end{array} \right\}.$$

Definite information IA `BTR`s are often called *definite* relations, and they can be described by conjunctions of **PA** relations. This is described in table 2.2, where X^-,X^+ are the beginning and end points of the interval X respectively.

Indefinite information To represent indefinite information, we use disjunctions of `BTR`s. As an example, the statement "John had his breakfast prior to going on a walk" is represented by specifying that the interval of John's breakfast is either `before` or

Table 2.2: The PA representation of the 13 IA relations.

Relation	PA representation	Inverse	PA representation
X `before` Y	$(X^+ < Y^-)$	X `after` Y	$(Y^+ < X^-)$
X `=` Y	$(X^- = Y^-) \wedge (X^+ = Y^+)$	X `=` Y	$(X^- = Y^-) \wedge (X^+ = Y^+)$
X `meets` Y	$(X^+ = Y^-)$	X `met-by` Y	$(X^- = Y^+)$
X `overlaps` Y	$(X^- < Y^-) \wedge (X^+ < Y^+)$ $\wedge\ (Y^- < X^+)$	X `overlaped-by` Y	$(X^- > Y^-) \wedge (X^+ > Y^+)$ $\wedge\ (Y^- > X^+)$
X `during` Y	$(Y^- < X^-) \wedge (X^+ < Y^+)$	X `contains` Y	$(X^- < Y^-) \wedge (Y^+ < X^+)$
X `starts` Y	$(X^- = Y^-) \wedge (X^+ < Y^+)$	X `started-by` Y	$(Y^- = X^-) \wedge (Y^+ < X^+)$
X `finishes` Y	$(X^- > Y^-) \wedge (X^+ = Y^+)$	X `finished-by` Y	$(Y^- > X^-) \wedge (Y^+ = X^+)$

`meets` the interval during which John was walking. This is denoted by

$$I_{Breakfast}\{before,\ meets\}I_{Walk}.$$

Such information is indefinite because it is not known whether John went for a walk immediately after breakfast or he was waiting in between. The 13 atomic relations are given in Table 2.3 and the composition table between these relation is given in Table 2.4. This table is used as follows: To compute the composition of the relation `o` with the relation `d`, we follow the row labeled by `o` up to the column labeled by `d`, and obtain the result `o d s` which means that it can be either `o` or `d` or `s`. The total number of possible indefinite relations is $2^{13} = 8192$.

Operators The converse $\tilde{}$, intersection $\cap$ and composition $\circ$ of `BTR`s are as follows: The converse of b, m, s, f, o, d is bi, mi, si, fi, oi, di respectively. The intersection of two non-identical `BTR`s is empty. The composition of every pair of `BTR`s is given by Table **??**, originally presented in [5]. Recall that the converse, intersection and composition of disjunctions of `BTR`s is defined in section 2.2.

Non-degenerate intervals An interesting feature of Interval Algebra is that one cannot coalesce the end points of an interval to obtain a zero length interval [5]. To illustrate the implications of this feature, consider the following conjunction of three constraints:

$$(X\ \{meets\}\ Y)\ \wedge\ (Y\ \{meets\}\ Z)\ \wedge\ (X\ \{meets\}\ Z)$$

The composition of $\{m\} \circ \{m\}$ is $\{b\}$, which has an empty intersection with $\{m\}$. Thus, the above conjunction of constraints is inconsistent.

Theorem 3: *[67] Deciding consistency (and computing a solution) of Interval TC-SPs is in NP-complete.*

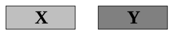
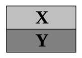
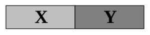
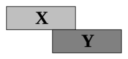
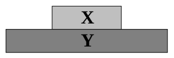
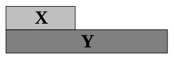
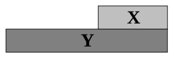
Relation	Symbol	Inverse	Example
X before Y	b	bi	
X equal Y	=	=	
X meets Y	m	mi	
X overlaps Y	o	oi	
X during Y	d	di	
X starts Y	s	si	
X finishes Y	f	fi	

Table 2.3: The 13 qualitative Interval-Interval relations and their inverses.

2.4.1 Tractable Classes

Pointisible TCSPs The first and most simple tractable class identified was the subclass containing the relations {`b`,{`b`,`m`},`=`,{`mi`,`bi`},`bi`} which can be represented by **PA** TCSPs, called Pointisible TCSPs [67]. Linear time algorithms for processing this class were developed [38, 30]. For this subclass, enforcing path-consistency decides consistency and enforcing 4-consistency computes the minimal constraints [102].

Macro Relations can be used to describe tractable classes. By shifting one of the four interval endpoints of the two intervals leaving the other three fixed, a partial order on the 13 relations is obtained [77]. This partial order was used to represent coarse temporal information through the notion of neighborhood [33]. Two relations are *conceptual neighbors* if they can be derived by shifting to the right one of the four endpoints leaving the other three fixed. A set of relations forms a *conceptual neighborhood* if each relation is a conceptual neighbor of at least one other relation in the set. It is convenient to consider the following macro

$$\begin{array}{ll}
\cap \ = \ \{\ m,\ mi,\ o,\ oi,\ s,\ si,\ f,\ fi,\ d,\ di,\ \equiv\} & \\
\alpha \ = \ \{m,\ o\}\ , & \alpha^{-1} \ = \ \{mi,\ oi\} \\
\subset \ = \ \{s,\ f,\ d\}\ , & \subset^{-1} \ = \ \{si,\ fi,\ di\} \\
\prec\!\cap \ = \ \{b, \cap\}\ , & \cap\!\succ \ = \ \{\cap,\ bi\} \\
\prec\!\cap\!\succ \ = \ \{b,\ \cap,\ bi\}\ , & \prec\!\succ \ = \ \{b,\ bi\}
\end{array}$$

Δ-classes The Δ-*notation* is used to describe subclasses of the Interval Algebra which

∘	b	a	m	mi	o	oi	d	di	=	s	si	f	fi
b	b	?	b	b o m d s	b	b o m d s	b o m d s	b	b	b	b	b o m d s	b
a	?	a	a d f oi mi	a	b d f oi mi	a	a d f oi mi	a	a	a	a	a	a
m	b	a oi mi di si	b	f fi =	b	o d s	o d s	b	m	m	m	d s o	b
mi	b o m di fi	a oi di	f fi =	a	d f oi	a	d f oi	a	mi	d f oi	a	mi	m
o	b	a oi di mi si	b	oi di si	b o m	b	o d s	b o m di fi	o	o	o di fi	d s o	d
oi	b o m di fi	a	o di fi	a	o d = oi di	o di fi	d f oi	a oi mi di si	oi	oi d f	oi a mi	oi	oi si
d	b	a	b	a	b o m d s	a d f oi mi	d	?	d	d	a d f oi mi	d	b o d
di	b o m di fi	a oi di mi si	o di fi	oi di si	o di fi	oi di si	o d = oi di	di	di	o di fi	di	di si oi	d
=	b	a	m	mi	o	oi	d	di	=	s	si	f	fi
s	b	a	b	mi	b o m	d f oi	d	b o m di fi	s	s	s si =	d	b o
si	b o m di fi	a	o di fi	mi	o di fi	o di fi	oi d f	di	si	s si =	si	oi	d
f	b	a	m	a	o d s	a oi mi	d	b oi mi di si	f	d	a oi mi	f	f =
fi	b	a oi mi di si	m	si oi di	o	oi di si	o d s	di	fi	o	di	f fi =	fi

Table 2.4: Transitivity table for Qualitative Interval Algebra.

Table 2.5: Tractable classes.

Class Name	Relations Used (Δ-class)	Reference
Interval Orders	$\{b,\ bi,\ \cap\}$	[32]
Interval Graphs	$\{-\!\!\!\prec\!\succ\!\!\!-,\ \cap\}$	[43, 31, 13, 57]
Circle (overlap) Graphs	$\{\ \{\alpha,\alpha^{-1},\equiv\},\ \{b,bi,\subset,\subset^{-1}\}\ \}$	[35, 14]
Interval Containment Graphs	$\{\ \{\subset,\subset^{-1}\},\ \{b,bi,\alpha,\alpha^{-1},\equiv\}\ \}$	[46]
Posets of dimension 2	$\{\ \subset,\ \subset^{-1},\ \{b,bi,\alpha,\alpha^{-1},\equiv\}\ \}$	[10, 9, 46]

are based on the macro relations described above. $\Delta = \{m_1,\ldots,m_k\}$ denotes the set of *macro relations* that form an algebra which is closed under converse, intersection and composition. An interesting result regarding the complexity of a very simple restricted subclass of the Interval TCSP is as follows:

Theorem 4: *[44] Deciding consistency of an Interval TCSP in which the relations are* $\Delta = \{b,\ bi,\ \cap, \prec\!\cap\!\succ\}$ *and* $\Delta = \{\prec\!\cap,\ \cap\!\succ,\ -\!\!\!\prec\!\succ\!\!\!-,\ \prec\!\cap\!\succ\}$ *is in NP-complete.*

Despite the fact that the restricted class described above is intractable, several Δ-based tractable subclasses were identified. Table 2.5, originally given in [44], describes a number of well-known recognition problems in graph theory and partially ordered sets which are restricted subclasses of the Interval Algebra. The notation $\{\{\mathtt{r}_1,\mathtt{r}_2\},\{\mathtt{r}_1,\mathtt{r}_4\}\}$ describes two macro relations, the first is the disjunction ofr_1 and r_2, while the second is a disjunction of r_3,r_4. A linear time algorithm for deciding consistency of $\Delta = \{b, bi, \cap, \prec\!\cap, \cap\!\succ, \prec\!\cap\!\succ\}$ is given in [44]. A cubic time algorithm for deciding consistency of $\Delta = \{b, bi, \cap, -\!\!\!\prec\!\succ\!\!\!-\}$ is given in [44]. Efficient algorithms for $\Delta = \{b, bi, \cap\}$ and $\Delta = \{-\!\!\!\prec\!\succ\!\!\!-, \cap\}$ can be found in [12, 11].

The Maximal Tractable Class

The unique maximal tractable subclass that includes *all* 13 relations was identified in [76] . Define three *atomic formulas*: $(X_i \leq X_j)$, $(X_i = X_j)$ and $(X_i \neq X_j)$, where X_i, X_j are point variables and $i < j$. A *ORD-Horn* clause is a disjunction of these *atomic formulas*, and a *ORD-Horn* formula is a conjunction of *ORD-Horn* clauses. The class of relations which can be described by *ORD-Horn* formulas, denoted $\mathcal{H}$, is closed under converse, intersection and composition [76].

Example 6: The ORD-Horn representation of the (pointisible) relation $X\{d,o,s\}Y$ is the formula

$$\begin{array}{ll} \{(X^- \leq X^+), & (X^- \neq X^+) \\ (Y^- \leq Y^+), & (Y^- \neq Y^+) \\ (X^- \leq Y^+), & (X^- \neq Y^+) \\ (Y^- \leq X^+), & (Y^- \neq X^+) \\ (X^+ \leq Y^+), & (X^+ \neq Y^+)\} \end{array}$$

where X^-, X^+ and Y^-, Y^+ denote the end points of the intervals X and Y respectively. The relation $(X^- \neq Y^- \;\vee\; X^+ \neq Y^+)$, the complement of $X\{=\}Y$, is in $\mathcal{H}$ but is not pointisible.

Theorem 5: *[76]*

- *A sub-algebra S is tractable iff the closure of S under the converse, intersection and composition operators is tractable.*
- *$\mathcal{H}$ is the unique maximal tractable subclass and*
- *enforcing path-consistency decides consistency for $\mathcal{H}$.*

Other Maximal Tractable Subclasses

Twelve maximal tractable subclasses that do not use all 13 relations were characterized [30]. Four of these can express *sequentability* of intervals, which cannot be described in the ORD-Horn subclass. The satisfiability algorithm, which is common to all these algebras, was shown to be linear. The definition of the classes and the algorithm rely on the notion of *maximal acyclic* relations.

Definition 1: An acyclic relation r is such that any cycle labeled by r is unsatisfiable. A maximal acyclic relation r is such that $\nexists r' \supset r$ where r' is acyclic.

The relation $<$ is acyclic but not maximal. The only maximal acyclic relations are {m,<,o,di,fi,s}, {m,<,o,di,fi,si}, {m,<,o,d,f,s}, {m,<,o,d,fi,s} and their respective converses.

Definition 2: [30] The subclass $A(r,s) = A_1(q) \cup A_2(r,q)$ where q can be any `BTR` except { m,mi}, r is an acyclic relation, $A_1(q) = \{r' \cup \{q, {}^{\sim}q\} | r' \in \mathcal{A}\}$, $A_2(r,q) = \{r' \cup \{=, q\} | r' \subseteq r\}$ and $\mathcal{A}$ is the interval algebra.

Theorem 6: *[30] Algorithm $\mathcal{SC}$ in Figure 2.3 decides consistency of $A(r,b)$ subalgebras and terminates in $O(v+e)$ steps where v is the number of variables and e is the number of constraints.*

This algorithm is very similar to that given in [102, 38] for **PA** TCSPs.

2.4.2 Techniques

The original constraint propagation algorithm Allen provides in [5] enforces path-consistency. This algorithm hasn't changed much over the years, and today it is still used as the major constraint propagation algorithm (for Interval TCSPs). A more sophisticated algorithm, which enforces 4-consistency, can be found in [101]. These algorithms are sound but incomplete for deciding consistency and for approximating the minimal constraints.

```
Algorithm $\mathcal{SC}$
1. Find all strongly connected components $C$ in $G'$;
2. for every arc $e$ in $G$ whose relation excludes equals do
3.     if $e$ connects two nodes in some $C$ then
4.         exit, the constraints are not satisfiable.
5.     endif
6. endfor
7. exit, the constraints are satisfiable.
```

Figure 2.3: The simple algorithm for deciding consistency of $A(r, q)$.

Hierarchical IA TCSPs

Reference intervals can be used to form clusters to reduce the space requirements and time complexity of enforcing path-consistency [5]. Clusters are formed by associating a set of intervals with one reference interval that subsumes them. Efficiency of constraint propagation is improved by enhancing path-consistency as follows: Path-consistency is applied within each cluster separately. Inter-cluster constraints, between a pair of variables X_i, X_j from different clusters, are computed by processing triangles in which X_k specifies a reference interval only. If the reference intervals are disjoint, then enforcing path-consistency within the clusters is sufficient to enforce path-consistency for the whole TCSP.

To improve efficiency of enforcing path-consistency on general TCSPs where there are no reference intervals (or they are not disjoint), reference intervals can be generated on-the-fly [56]. This reduces the number of triangles processed yet, if done correctly, computes a path-consistent TCSP.

Empirical Evaluation

Next, we survey results of three experiments reported in [63], aimed at evaluating the effectiveness of path-consistency for: (i) removing disjunctions, (ii) detecting inconsistencies, and (iii) prunning dead-ends in backtrack search.

Removing Disjunctions To analyze the ability of path-consistency to remove redundant disjunctions, its effectiveness was evaluated on randomly generated problems [63]. For consistent TCSPs, the average number of disjunctions removed by enforcing path-consistency was reported. For inconsistent TCSPs, the number of disjunctions after enforcing path-consistency is 0. Thus, it was suggested to measure the average number of disjunctions removed after the first iteration.

Problems of sizes 4-20 variables were analyzed. For each constraint, each of the 13 relations was included with probability 0.5. Thus, the average number of disjunction

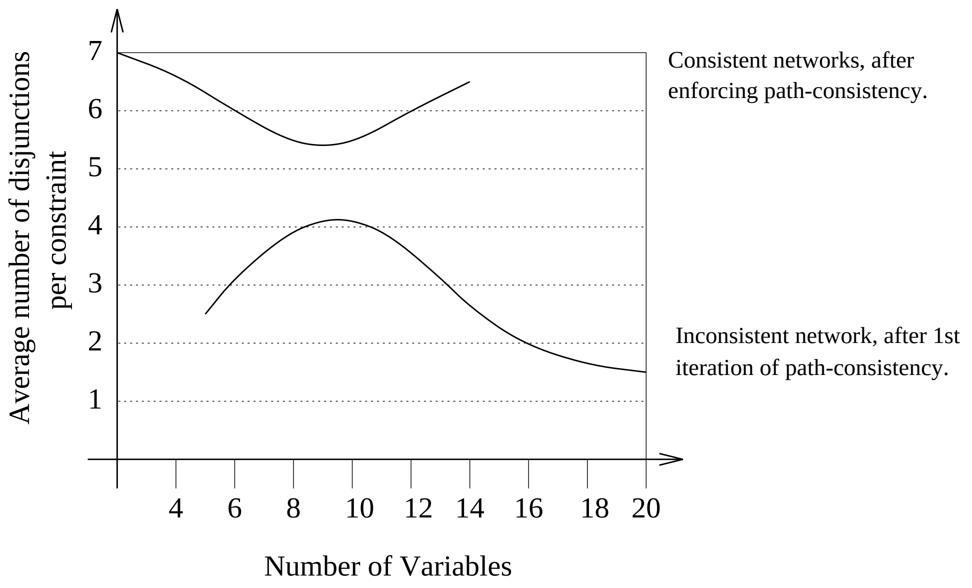


Figure 2.4: Removing disjunctions with path-consistency.

per constraint was 7.5 and the constraint graph was complete (i.e. for every pair of variables a constraint was specified).

The reproduced results form [63] are presented in figure 2.4. The average number of disjunctions generated was 7.5 (i.e. 50% of 13). For consistent TCSPs, after enforcing path-consistency the average number of disjunctions did not drop under 5.5. For inconsistent TCSPs, the average number of disjuncts after the first iteration of PC did not go above 4.5. Most inconsistencies were found in the first 3 iterations.

Similar results were reported by Ladkin and Reinefeld [64], van Beek and Manchak [104]. Nebel [75] considers instantiating labels by elements from a tractable relation set, and proved that this is completeness preserving. Furthermore, Nebel shows that his technique almost always terminates in polynomial time [75].

2.4.3 Summary

For qualitative Interval TCSPs, also called the Interval Algebra (IA), answering queries is intractable. Nevertheless, many relation based tractable classes exist and the unique maximal tractable class using *all* 13 relations was identified. The most common technique used for deciding consistency and computing feasible relations of the IA is enforcing path-consistency. For all the tractable classes surveyed, it correctly decides consistency. To compute a solution, backtrack search is used. Incorporating path-consistency as a forward checking procedure within backtrack search was shown to be very effective in pruning dead-ends.

2.5 Metric Point Constraints

Metric Point TCSP was introduced as a framework which extends constraint satisfaction to include continuous variables and allows processing temporal constraints [25].

Variables specify time points.

Domains represent an ordered and unbounded set of time points, and are usually the set of integers or rationals.

Constraints are built upon a single BTR, $X_j - X_i \in [a, b]$. A constraint C is expressed by a set of intervals

$$C \overset{\text{def}}{=} \{I_1, \ldots, I_n\} = \{[a_1, b_1], \ldots, [a_n, b_n]\}.$$

Having the following meaning: A unary constraint C_i restricts the domain of the variable X_i to the given set of intervals

$$C_i \overset{\text{def}}{=} (a_1 \le X_i \le b_1) \cup \ldots \cup (a_n \le X_i \le b_n).$$

where $\mapsto$ means "*translates to*". A binary constraint C_{ij} over X_i, X_j restricts the permissible values for the distance $X_j - X_i$; it represents the disjunction

$$C_{ij} \overset{\text{def}}{=} (a_1 \le X_j - X_i \le b_1) \cup \ldots \cup (a_n \le X_j - X_i \le b_n).$$

All intervals are assumed to be pairwise disjoint. All times can be specified relative to X_0 and thus each unary constraint C_i can be represented as a binary constraint C_{0i} (having the same interval representation).

Qualitative Point TCSPs, or PA networks, can be described using Metric Point TCSPs by mapping the qualitative point-point constraints into metric constraints, as described in table 2.6a [71, 55]. Similarly, metric TCSPs can be translated, with loss of information, into Qualitative TCSPs, as described in table 2.6b [71, 55].

Operators Let $T = \{I_1, \ldots, I_l\}$ and $S = \{J_1, \ldots, J_m\}$.

1. The *inverse* of $T = \{[a_1, b_1], \ldots, [a_k, b_k]\}$ is $\sim T = \{[-b_k, -a_k], \ldots, [-b_1, -a_1]\}$.
2. The *intersection* of T and S, denoted by $T \cap S$,
   admits only values that are allowed by both of them.
3. The *composition* of T and S, denoted by $T \circ S$, admits only values r for which there exists $t \in T$ and $s \in S$ such that $r = t + s$

Table 2.6: The mappings between Qualitative and Metric BTRs.

$$\begin{array}{lll} '<' & \mapsto & (-\infty, 0); \\ '\leq' & \mapsto & (-\infty, 0]; \\ '=' & \mapsto & [0, 0]; \\ '\geq' & \mapsto & [0, \infty); \\ '>' & \mapsto & (0, \infty); \end{array} \qquad [a,\ b] \mapsto \begin{cases} '<' & if \quad a \leq b < 0; \\ '\leq' & if \quad a \leq b \leq 0; \\ '=' & if \quad a = 0 = b; \\ '\geq' & if \quad 0 \leq a \leq b; \\ '>' & if \quad 0 < a \leq b; \end{cases}$$

(a) (b)

Solutions A solution is a consistent singleton labeling. A singleton labeling of a Metric TCSP is a selection of a single interval from each constraint. Consistency of a labeling can be decided by enforcing path-consistency in $O(v^3)$ where v is the number of variables. Note that when a constraint C_{ij} is not specified in the input, it is assumed to specify the single interval $[-\infty, \infty]$.

Theorem 7: *[25] Deciding consistency (and computing a solution) of a Metric Point TCSP is in NP-complete.*

2.5.1 Tractable Classes

There are two relation based tractable classes: Simple Temporal Problems (STP) and STP with inequation constraints (for continuous domains only). There is also a graph-based tractable class called series-parallel TCSPs.

Simple Temporal Problems (STP)

Simple Temporal Problems (STP) specify a single interval per constraint. An STP can be associated with a directed edge-weighted graph, G_d, called a *distance graph* (d-graph), having the same vertices as the constraint graph G; each edge $i \rightarrow j$ is labeled by a weight w_{ij} representing the constraint $X_j - X_i \leq w_{ij}$. An STP is consistent iff the corresponding d-graph G_d has no negative cycles and the minimal network of the STP corresponds to the *minimal distances* in G_d. Therefore, Floyed-Warshall's all-pairs shortest path algorithm enforces path-consistency and is complete for STPs [25]. Consequently, deciding consistency and computing the minimal network require $O(v^3)$ steps.

Single Intervals with Inequation constraints

The class of Simple Temporal Networks was further extended to include *disjunctions of inequations* (i.e. $x \neq y$). This extension is tractable if the domains are dense (i.e. rationals or reals) [58]. This class of constraints may be encountered when resolution is combined with variable elimination.

Example 7: [58] Consider the following set of constraints

$$X_3 \leq X_1, \quad X_5 < X_1, \quad X_1 \leq X_2, \quad X_4 \neq X_1$$

Eliminating X_1 (using standard elimination procedures) results in $X_3 \leq X_2,\ X_5 < X_2$ with the addition of disjunction $X_4 \neq X_3\ \vee\ X_4 \neq X_2$.

Deciding consistency can be done in $O(v^3 e)$ [58].

Minimal Constraints can be computed in $O(v^5)$ by enforcing 5-consistency [60].

Series Parallel

A TCSP is said to be *series-parallel* with respect to a pair of nodes, i and j, if it can be reduced to the edge (i,j) by repeated applications of the following *reduction* operation: select a node of degree 2 or less, remove it from the network, and connect its neighbors. This means that, once you have more than 2 domain constraints, the problem is not series-parallel. Deciding whether a TCSP is series-parallel requires $O(v)$ steps where v is the number of variables [25]. If the TCSP is series-parallel, deciding consistency can be done using algorithm DPC (recall section 2) in $O(nk)$ where k is the maximal number of intervals per constraint [25].

2.5.2 Techniques

The original path-consistency algorithm presented in [25] parallels the path-consistency algorithms used to process discrete CSPs and Qualitative TCSPs.

Complexity of Path-Consistency

When time is described by integer or rational numbers, algorithms PC and DPC terminate in $O(v^3R^3)$ and $O(v^3R^2)$ steps respectively, where R is the range of the constraints, namely the number of domain elements between the largest and smallest numbers specified [25]. However, as we show in Chapter 3, when the range R is very large or the domains are continuous, enforcing path-consistency is problematic and becomes impractical (exponential) [81, 88]. Consider the network presented in figure 2.5, having 3 variables, 3 constraints and 3 intervals per constraint. After enforcing path-consistency, two constraints remain unchanged in the path-consistent network while the third is broken into 10 subintervals. As this behavior is repeated over numerous triangles in the network, the number of intervals may become exponential.

2.5.3 Summary

Metric TCSPs provide a framework for describing *disjunctive* linear difference constraints. In general, answering queries is intractable. Four tractable classes were

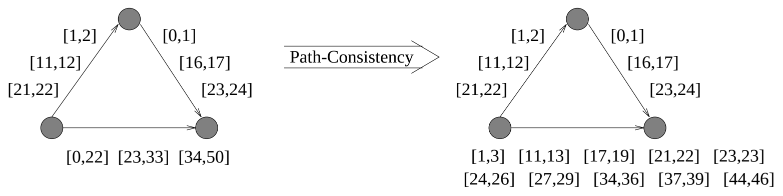


Figure 2.5: The fragmentation problem.

surveyed: Simple Temporal Problems (STP), STP with disjunctions of inequation constraints and series-parallel TCSP.

2.6 Combining Temporal Constraints

The qualitative and metric point and interval TCSPs were combined into a unified model [71]. This model is described using the same general concepts given in Section **??**.

2.6.1 Point Interval Qualitative Constraints

In the point-interval algebra, abbreviated **PIA** the variables represent either time points of time intervals. A new kind of constraints is introduced between a point variable and an interval variable. The new BTRs are { `before`, `starts`, `during`, `finishes`, `after`} and their inverses, as illustrated in Table 2.7. Table 2.8 represent the composition between four types of relations: (i) point-point, point-interval, interval-point and interval-interval. The composition between a Point-Point (PP) relation and another PP relation is given in the table T_{PPP} which is the Point Algebra (**PA**) given in Section 2.3. T_{III} is the **IA** transitivity table given in Section 2.4, Table 2.4. The composition between a PP relation and a Point-Interval (**PI**) relation is given in Table 2.9. The composition between PI relations and IP relations is given in Table 2.10. The rest of the tables can be derived similarly.

Although **PIA** is less expressive than **IA**, the former is also intractable.

Theorem 8: *[71] Deciding consistency of* **PIA** *TCSP is NP-complete.*

2.6.2 Point Algebra + Metric Domain Constraints

TCSPs resulting from augmenting PA with unary metric constraints are also intractable. The **PIA** subclasses investigated were either **CPA** or **PA** augmented with one of the following unary metric constraints [71]:

- **Discrete**: Specified by a finite set of values.

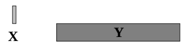
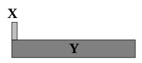
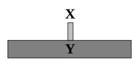
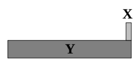
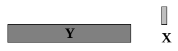
Relation	Symbol	Inverse	Example
X before Y	b	bi	
X starts Y	s	si	
X during Y	d	di	
X finishes Y	f	fi	
X after Y	a	ai	

Table 2.7: The 10 qualitative Point-Interval relations and their inverse.

	PP	PI	IP	II
PP	T_{PPP}	T_{PPI}		
PI			T_{PIP}	T_{PII}
IP	T_{IPP}	T_{IPI}		
II			T_{IIP}	T_{III}

Table 2.8: The combined transitivity tables.

T_{PPI}	b	s	d	e	a
$<$	b	b	$b\ s\ d$	$b\ s\ d$	?
$=$	b	s	d	e	a
$>$	?	$d\ e\ a$	$d\ e\ a$	a	a

Table 2.9: Composition of PP and PI relations.

T_{PIP}	ai	ei	di	si	bi
b	$<$	$<$	$<$	$<$	?
s	$<$	$<$	$<$	$=$	$>$
d	$<$	$<$	?	$>$	$>$
e	$<$	$=$	$<$	$>$	$>$
a	?	$<$	$<$	$>$	$>$

Table 2.10: Composition of PI and IP relations.

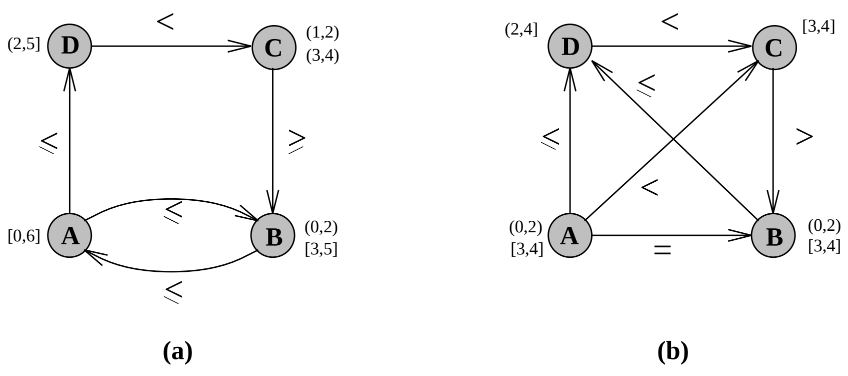


Figure 2.6: Augmented CPA networks (a) before path-consistency and (b) after path-consistency.

Table 2.11: Results on Combined TCSPs.

	Discrete	**Single interval**	**Multiple interval**
		Deciding consistency	
CPA	AC $O(ek)$	AC+PC $O(n^2)$	AC+PC $O(n^2k)$
PA	NP-Complete	AC+PC $O(en)$	NP-Complete
		Computing minimal constraints	
CPA	AC+PC $O(n^2k)$	AC+PC $O(n^2)$	AC+PC $O(n^2k)$
PA		AC+PC $O(en^2)$	

- **Single interval**: Specified by a single metric interval.
- **Multiple intervals**: Specified by a set of disjoint metric intervals.

Figure fig:a-cpaa presents a sample CPA augmented with multiple interval domains. There are four variables, A,B,C,D with the qualitative **PA** constraints A≤B, A≥B, A≤D, C>D, B≥C and metric domain constraints A∈[0,6], B∈(0,2)∪[3,5), C∈(1,2)∪(3,4) and D∈(2,5). Figure fig:a-cpab show the constraints obtained after enforcing path-consistency.

Table 2.11 summarize the complexity results and techniques, where AC denotes *Arc-Consistency*, PC denotes *Path-Consistency*, and k is the maximum number of intervals defining the domain [71].

2.6.3 Interval Algebra + Metric Constraints

TCSPs resulting from augmenting qualitative interval TCSPs with metric point constraints are intractable. Apart from backtracking search, there are two methods for deciding consistency and approximating the minimal constraints: (i) Enforcing path-consistency on a combined TCSP in $O(n^3R^3)$ steps where R is the range of the metric constraints (see Section 2.5) [71]. (ii) Iteratively enforcing path-consistency on the point metric and qualitative interval sub-TCSPs independently, and translating and propagating information between them in $O(n^5R^3)$ [55].

Chapter 3

Processing Metric Temporal Constraints

This chapter advances the state-of-the-art of temporal constraint processing by introducing new efficient processing algorithms. We demonstrate that even local consistency algorithms like path-consistency (PC) can be exponential on TCSPs due to the fragmentation problem. We present two new polynomial approximation algorithms, Upper-Lower Tightening (ULT) and Loose Path-Consistency (LPC), which are efficient yet effective in detecting inconsistencies and reducing fragmentation. We identify a new tractable class for which both ULT and LPC are complete. Our experiments on hard problems in the transition region show that LPC has the best effectiveness-efficiency tradeoff for processing TCSPs. When incorporated within backtrack search, LPC is capable of improving search performance by orders of magnitude.

3.1 Introduction

Problems involving temporal constraints arise in various areas including temporal databases [21], diagnosis [54], scheduling [82, 81], planning [70], common sense reasoning [98] and natural language understanding [7]. Among the formalisms for expressing and reasoning about temporal constraints are the interval algebra [5], point algebra [67], Temporal Constraint Satisfaction Problems (TCSP) [27] and models combining quantitative and qualitative constraints [71, 55].

The two main types of Temporal Constraint Networks can be characterized as qualitative [5, 67] and quantitative [27]. In the qualitative model, variables are time intervals or time points and the constraints are qualitative. In the quantitative model, variables represent time points and the constraints are metric. These two types have been combined into a single model [71, 55]. In this paper we build upon the model proposed by Meiri [71], in which variables are either points or intervals and there are three types of constraints: metric point-point and qualitative point-interval and interval-interval.

Answering queries in constraint processing reduces to the tasks of determining consistency, computing a consistent scenario and computing the minimal network. When time is represented by (or isomorphic to the) integers[1], deciding consistency is in NP-complete [27, 71]. For qualitative networks, computing the minimal network is in NP-hard [44, 27]. In both qualitative and quantitative models, complexity stems from disjunctive relationships between pairs of variables and occur in many applications.

Example 8: A large NAVY cargo must leave New York starting on March 7, go through Chicago and arrive at Los Angeles within 8-10 days. From New York to Chicago the delivery requires 1-2 days by *air or* 10-11 days on the *ground*. From Chicago to Los Angeles the delivery requires 3-4 days by air *or* 13-15 days on the ground. In addition, we know that an AIRFORCE cargo needs to be transported using the same terminal in Chicago as required for the NAVY's cargo transportation (i.e. the intervals of NAVY and AIRFORCE shipments should not overlap). The transportation of the AIRFORCE cargo should start between March 17 and March 20 and requires 3-5 days by air *or* 7-9 days on the ground.

Given the above constraints, we are interested in answering questions such as: "are the constraints satisfiable?", "can the NAVY cargo arrive in Los Angeles on March 13-14?", "when should the cargo arrive in Chicago ?", "how long may the NAVY cargo transportation take?". The first two queries reduce to deciding consistency and the third and fourth queries reduces to computing the minimal network.

Since answering such queries is inherently intractable, this paper focuses on the design of efficient and effective polynomial *approximation* algorithms for deciding consistency and computing the minimal network. The common approximation algorithm

[1]This is always the case in practice.

enforces path-consistency (PC) [27]. As we demonstrate, in contrast to discrete CSPs, enforcing path-consistency on quantitative TCSPs is exponential. This is because in the path-consistent quantitative TCSP intervals are broken into several smaller subintervals. This may result in an exponential blowup, leading to what we call fragmentation.

We present two algorithms for bounding fragmentation called Upper-Lower Tightening (ULT) and Loose Path-Consistency (LPC). We show that these algorithms avoid fragmentation and are effective in detecting inconsistencies. We also discuss five variants of the main algorithms, called ULT-2, Directional ULT (DULT), LPC-2, Directional LPC (DLPC) and Partial LPC (PLPC).

We address two questions empirically: (1) which of the algorithms presented is preferable for detecting inconsistencies, and (2) how effective are the proposed algorithms when used to improve backtrack search by preprocessing and (guiding the search) by forward checking.

To answer the first question, we show that enforcing path-consistency may indeed be exponential in the number of intervals per constraint while ULT's execution time is almost constant. Nevertheless, ULT is able to detect inconsistency in about 70% of the cases in which PC does. Algorithm LPC further improves on ULT; it is both efficient and capable of detecting almost all of the inconsistencies detected by PC.

To answer the second question, we apply the new algorithms in three ways: (1) in a preprocessing phase for reducing the fragmentation before initiating search, (2) in a forward-checking algorithm for reducing the fragmentation during the search and detecting dead-ends early, and (3) in an advice generator for dynamic variable ordering. Through experiments with hard problems which lie in the transition region (defined by [79, 20]), we show that both ULT and LPC are preferred to PC and that LPC is the best algorithm overall. We conclude that the performance of backtrack search can be improved by several orders of magnitude when using LPC for preprocessing, forward checking and dynamic variable ordering.

The organization of the paper is as follows. Section 2 summarizes the model of TCSPs and the known algorithms for processing them. Section 3 presents algorithm Upper Lower Tightening (ULT) and section 4 presents a new tractable class based on ULT. Section 5 presents Loose Path-Consistency (LPC). Section 6 extends the results of sections 3,4 and 5 to networks of combined qualitative and quantitative constraints. Section 7 presents backtracking algorithms and Section 8 provides an empirical evaluation.

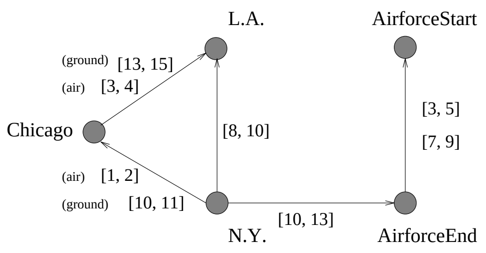


Figure 3.1: The constraint graph for the metric portion of the logistics problem.

3.2 Metric Temporal Constraint Networks

A quantitative TCSP involves a set of variables, $X_1, \ldots, X_n$, having *continuous* domains, each representing a time point. Each constraint C is a set of intervals

$$C \stackrel{\text{def}}{=} \{I_1, \ldots, I_n\} = \{[a_1, b_1], \ldots, [a_n, b_n]\}.$$

A unary constraint C_i restricts the domain of the variable X_i to the given set of intervals

$$C_i \stackrel{\text{def}}{=} (a_1 \le X_i \le b_1) \cup \ldots \cup (a_n \le X_i \le b_n).$$

A binary constraint C_{ij} over X_i, X_j restricts the permissible values for the distance $X_j - X_i$; it represents the disjunction

$$C_{ij} \stackrel{\text{def}}{=} (a_1 \le X_j - X_i \le b_l) \cup \ldots \cup (a_n \le X_j - X_i \le b_n).$$

All intervals are assumed to be open and pairwise disjoint.

Example 9: Consider the cargo example given in the introduction. Let the variables be:

$X_{N.Y.}$ = time point at which the NAVY cargo was shipped out of N.Y.,
$X_{Chicago}$ = time point the NAVY cargo arrived into and shipped out of CHICAGO
$X_{L.A.}$ = time point at which the cargo arrived into L.A.
$X_{AirforceStart}$ = time point at which the AIRFORCE shipment STARTS,
$X_{AirforceEnd}$ = time point at which the AIRFORCE shipment ENDS.

The *metric* constraints are:

$$\begin{aligned}
&X_{N.Y.} - X_0 \in [March\ 7th,\ March\ 7th] \\
&X_{Chicago} - X_{N.Y.} \in [1,2] \cup [10,11] \\
&X_{L.A.} - X_{Chicago} \in [3,4] \cup [13,15] \\
&X_{L.A.} - X_{N.Y.} \in [8,10] \\
&X_{AirforceEnd} - X_{AirforceBegin} \in [3,5] \cup [7,9] \\
&X_{AirforceBegin} - X_{N.Y.} \in [10,13]
\end{aligned}$$

Definition 3: [solution]
A tuple $X = (x_1, \ldots, x_n)$ is called a *solution* if the assignment $X_1 = x_1, \ldots, X_n = x_n$ satisfies all the constraints. The network is *consistent* iff at least one solution exists.

A quantitative TCSP can be represented by a *directed constraint graph*, where nodes represent variables and an edge $i \rightarrow j$ indicates that a constraint C_{ij} is specified. Every edge is labeled by the interval set as illustrated in Figure 3.1. A special time point X_0 is introduced to represent the "beginning of the world". All times can be specified relative to X_0 and thus each unary constraint C_i can be represented as a binary constraint C_{0i} (having the same interval representation). The constraint graph representing the logistics example is given in Figure 3.1.

The minimal network is useful for answering a variety of queries, as described below, because it describes explicitly all the implicit (induced) binary constraints.

Definition 4: [minimal network]
A value v_i and v_{ij} is a *feasible value* of X_i and $X_j - X_i$, respectively, if there exists a solution in which $X_i = v$ and $X_j - X_i = v_{ij}$ respectively. The *minimal domain* of a variable is the set of all *feasible values* of that variable. A *minimal constraint* C_{ij} between X_i and X_j is the set of feasible values for $X_j - X_i$. A network is minimal iff its domains and constraints are minimal.

3.2.1 Answering Queries

For completeness, we describe the set of queries that the quantitative TCSP model is designed to support. Consider the following sample queries:

1. Is the network consistent, and if so, what is a possible scenario ?
2. *Can* X_i occur 5 to 10 minutes after X_j ?
3. *Must* X_i occur 5 to 10 minutes after X_j ?
4. At what possible times can event X_i occur ?
5. Given the time at which event X_i occurred, when can X_j occur ?

These queries can be partitioned into two groups: those that can be reduced to the task of deciding consistency and those that require computing the minimal network.

Clearly, Query 1 requires testing the consistency of the TCSP. To answer Query 2, we add the constraint $X_j - X_i \in [5, 10]$ and test for consistency. If the resulting network is consistent the answer to the query is *yes*; otherwise it is *no*. Query 3, often referred to as entailment, can be answered by adding (to the network) the negation of the constraint, namely $X_j - X_i \in [-\infty, 5] \cup [10, \infty]$, and checking for inconsistency. If consistency was detected by computing a solution, that solution provides a counter example that shows how X_i can occur less than 5 minutes or more than 10 minutes after X_j.

Queries 4 and 5 can be processed in constant time by a simple table lookup, after the equivalent minimal network (recall Definition 2) has been computed. The event associated with X_i can occur at time t for every $t \in C_{0i}$, where C_{0i} is the constraint between X_0 and X_i in the minimal network. Given that X_i occurs at time t_1, event X_j can occur at time $t_2 \in C_{ij} - t_1$, where C_{ij} is the constraint between X_i and X_j in the minimal network.

3.2.2 Path-Consistency

Deciding whether a given network is consistent is in NP-complete [27] and deciding whether it is minimal is in NP-hard (which subsumes NP-complete). Therefore, it is common to use algorithms that detect some (but not all) inconsistencies and tighten the constraints to obtain an approximation of minimal constraints. Such algorithms enforce local k-consistency by ensuring that every subnetwork with k variables is minimal [23]. Here, we present path-consistency (3-consistency) for quantitative TCSPs. For qualitative TCSPs, 3,4-consistency algorithms are covered by [102].

Path-consistency is defined using the $\cap$ and the $\odot$ operations (see Figure 3.2:

Definition 5: [Operators]
Let $T = \{I_1, \ldots, I_l\}$ and $S = \{J_1, \ldots, J_m\}$ be two sets of intervals which can correspond to either unary or binary constraints.

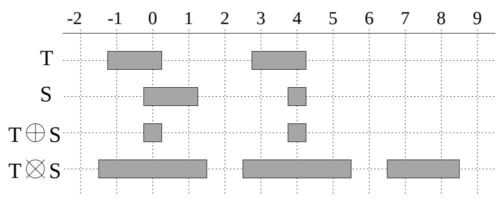
1. The *intersection* of T and S, denoted $T \cap S$, admits only values that are allowed by both of them.

2. The *composition* of T and S, denoted $T \odot S$, admits only values r for which there exists $t \in T$ and $s \in S$ such that $r = t + s$ (Figure 3.2).

The intuition behind enforcing path-consistency is as follows: We would like to compute the constraints induced by the composition of $C_{12} \odot C_{23} \odot \cdots \odot C_{k-1,k}$ along the path from X_1 to X_k. After path-consistency is enforced, we are guaranteed that $C_{1,k}$ is tighter than or equal to the constraint induced along this path.

Definition 6: A constraint C_{ij} is *path-consistent* iff $C_{ij} \subseteq \cap_{\forall k}(C_{ik} \odot C_{kj})$ and a network is *path-consistent* iff all its constraints are *path-consistent.*

Any arbitrary consistent quantitative TCSP with non-dense time domains (as is always the case in practice) can be converted into an equivalent *path-consistent* network by repeatedly applying the relaxation operation $C_{ij} \leftarrow C_{ij} \cap (C_{ik} \odot C_{kj})$ until a fixed point is reached. If the domains are dense, it is unclear under what conditions a fixed point can be reached in finite time. Figure 3.3 presents an algorithm for enforcing



$$\begin{aligned} T &= \{[-1.25, 0.25],\ [2.75, 4.25]\} \\ S &= \{[-0.25, 1.25],\ [3.77, 4.25]\} \\ T \cap S &= \{[-0.25, 0.25],\ [3.75, 4.25]\} \\ T \odot S &= \{[-1.50, 1.50],\ [2.50, 5.50],\ [6.50, 8.50]\} \end{aligned}$$

Figure 3.2: An illustration of the $\cap$ and the $\odot$ operations.

path-consistency. For completeness, we also describe a weaker yet more efficient version of path-consistency, called Directional Path Consistency (DPC), which is tied to a particular ordering of the variables [26].

Theorem 9: *[27]*
If time is not dense then algorithms PC and DPC terminate in $O(n^3R^3)$ and $O(n^3R^2)$ steps respectively, where n is the number of variables and R is the range of the constraints, i.e. the difference between the lowest and highest numbers specified in the input network.

Example 10: Consider a constraint $X_j - X_i \in [-1000, -990] \cup [-800, +800] \cup [990, 1000]$. The range R of this constraint is $[-1000, 1000]$. For such R the theorem 9 implies that PC might need to update the constraints thousands of times.

3.2.3 Fragmentation

In contrast to discrete CSPs, enforcing path-consistency on quantitative TCSPs is problematic when the range R is large or the domains are continuous [27, 81]. An upper bound on the number of intervals in $T \odot S$ is $|T| \cdot |S|$, where $|T|, |S|$ are the number of intervals in T and S respectively. As a result, the total number of intervals in the path-consistent network might be exponential in the number of intervals per constraint in the input network, yet bounded by R when integer domains are used.

Example 11: Consider the network presented in Figure 3.4, having 3 variables, 3 constraints and 3 intervals per constraint. After enforcing path-consistency, two constraints remain unchanged in the path-consistent network while the third is broken into 10 subintervals. As this behavior is repeated over numerous triangles in the network, the number of intervals may become exponential.

```
Algorithm PC
1. $Q \leftarrow \{(i,k,j) | (i < j)\ and\ (k \neq i,j)\}$
2. while $Q \neq \{\}$ do
3.       select and delete a path $(i,k,j)$ from $Q$
4.       if $C_{ij} \neq C_{ik} \odot C_{kj}$ then
5.             $C_{ij} \leftarrow C_{ij} \cap (C_{ik} \odot C_{kj})$
6.             if $C_{ij} = \{\}$ then exit (inconsistency)
7.             $Q \leftarrow Q \cup \{(i,j,k),(k,i,j) \mid 1 \leq k \leq n, i \neq k \neq j\}$
8.       end-if
9. end-while
```

```
Algorithm DPC
1. for $k \leftarrow n$ downto 1 by -1 do
2.       for $\forall i,j < k$ such that $(i,k),(k,j) \in E$ do
3.             if $C_{ij} \neq C_{ik} \odot C_{kj}$ then
4.                   $E \leftarrow E \cup (i,j)$
5.                   $C_{ij} \leftarrow C_{ij} \cap (C_{ik} \odot C_{kj})$
6.                   if $C_{ij} = \{\}$ then exit (inconsistency)
7.             end-if
8.       end-for
9. end-for
```

Figure 3.3: Algorithms PC and DPC.

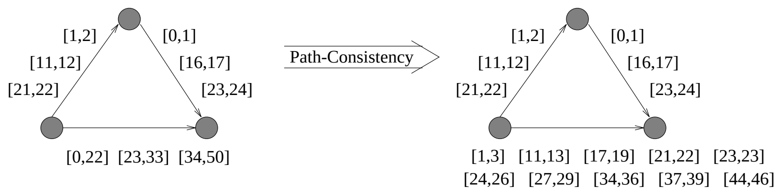


Figure 3.4: The fragmentation problem.

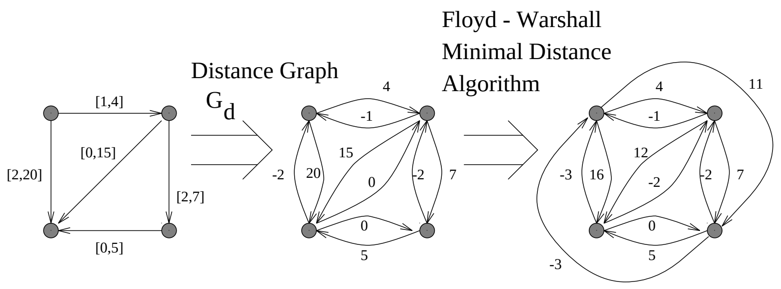


Figure 3.5: Processing an STP.

3.3 Upper Lower Tightening (ULT)

Enforcing path-consistency computes a tighter equivalent network that approximates the minimal network and is useful for answering a variety of queries. The problem with enforcing path-consistency is that the relaxation operation $C_{ij} \leftarrow C_{ij} \cap (C_{ik} \odot C_{kj})$ may increase the number of intervals in C_{ij}. Our idea is to compute looser constraints which consist of fewer intervals that subsume all the intervals of the path-induced constraint.

3.3.1 Simple Temporal Problems

Fragmentation does not occur when we enforce path-consistency on the special class of quantitative TCSPs called the *Simple Temporal Problem* (STP). In these networks, only a single interval is specified per constraint.

An STP can be associated with a directed edge-weighted graph, G_d, called a *distance graph* (d-graph), having the same vertices as the constraint graph G; each edge $i \rightarrow j$ is labeled by a weight w_{ij} representing the constraint $X_j - X_i \leq w_{ij}$, as illustrated in Figure 3.5. An STP is consistent iff the corresponding d-graph G_d has no negative cycles and the minimal network of the STP corresponds to the *minimal distances* in G_d. Therefore, an all-pairs shortest path procedure (Figure 3.5) is equivalent to enforcing path-consistency and is complete for STPs [27]. The focus of the rest of chapter is on two algorithms designed to bound the fragmentation.

Algorithm Upper-Lower Tightening (ULT)

```
1.  input: $N$
2.  $N''' \leftarrow N$
3.  repeat
4.      $N \leftarrow N'''$
5.      compute $N'$, $N''$, $N'''$.
6.  until   $\forall ij \ \ (low(C'''_{ij}) = low(C_{ij}))$ and $(high(C'''_{ij}) = high(C_{ij}))$
                                    ; which implies no change
            or  $\exists ij \ \ (high(C'''_{ij}) < low(C'''_{ij}))$
7.  if $\forall ij \ \ (high(C'''_{ij}) > low(C'''_{ij}))$     output: $N'''$
            otherwise                   output: "Inconsistent."
```

Figure 6: The Upper Lower Tightening (ULT) algorithm.

3.3.2 Avoiding Fragmentation

The algorithm for approximating path-consistency, called Upper Lower Tightening (ULT), utilizes the fact that an STP is tractable. The algorithm treats the extreme points of all the intervals associated with a single constraint as one big interval, yielding an STP, and then performs path-consistency on that STP. This process cannot increase the number of intervals per constraint. Finally, we intersect the resulting simple path-consistent minimal network with the input network.

Definition 7: [Upper Lower Tightening]
Let $C_{ij} = [I_1, \ldots, I_m]$ be the constraint over variables X_i, X_j and let $low(C_{ij}), high(C_{ij})$ be the lower and upper bounds of C_{ij}, respectively. We define N', N'', N''' as follows:

- N' is an STP derived from N by relaxing its constraints to $C'_{ij} = [low(C_{ij}),\ high(C_{ij})]$.
- N'' is the minimal network of N' (N' is an STP).
- N''' is the intersection of N'' and N, namely $C'''_{ij} = C''_{ij} \cap C_{ij}$.

Algorithm Upper Lower Tightening (ULT) is presented in Figure 6. The network N' is a relaxation of N. N'' is computed by applying the all-pairs shortest path algorithm on N'. Because N'' is equivalent to N', intersecting N'' with N results in a network that is equivalent to N.

Lemma 1: *Let N be the input to ULT and R be its output.*

1. *The networks N and R are equivalent.*
2. *Every iteration of ULT (except the last one) removes at least one interval.*

Proof: *Part 1: Let $Sol(N)$ denote the set of solutions of the network N, then $Sol(N) \subseteq Sol(N') = Sol(N'')$. This implies that $Sol(N) \cap Sol(N'') = Sol(N)$ and therefore $Sol(N''') = Sol(N)$. Part 2: Let N'_i, N''_i be the networks N', N'' at iteration i. If an interval is not removed at iteration i, $N''_i = N'_{i+1} = N''_{i+1}$, which implies no change.* □

Algorithm ULT computes looser networks than those resulting from enforcing full path-consistency. A complete comparison, is given in section 4 and is depicted in Figure 14.

Example 12: An example run of ULT on a sample problem instance is given in Figure 7. We start with N and compute $N'_{(1)}$, $N''_{(1)}$, $N'''_{(1)}$. Thereafter, we perform the second iteration in which we compute $N'_{(2)}$, $N''_{(2)}$, $N'''_{(2)}$ and finally, in the third iteration, there is no change. The first iteration removes two intervals, while the second iteration removes one. In addition, ULT computes an induced constraint C_{02}, which allows inferring a new implicit fact that was not specified explicitly in the input network.

Theorem 10: *Algorithm ULT terminates in $O(n^3ek + e^2k^2)$ steps where n is the number of variables, e is the number of edges, and k is the maximal number of intervals in each constraint.*

Proof: Because computing N' requires processing every interval in the network at most once, this computation requires $O(ek)$ steps. Computing N'' from N' can be done by applying the all-pairs shortest path algorithm (e.g. Floyd-Warshall) and thus requires $O(n^3)$ steps. Computing the intersection $T \cap S$ of two *sorted* constraints requires $O(|T| + |S|)$ steps, thus computing N''' from N'' requires $O(ek)$ steps. This means that each iteration requires $O(n^3 + ek)$ steps. The halting condition (Figure 6, line 6) implies that at every iteration at least one interval must be removed (Lemma 1). Therefore, at most $O(ek)$ iterations are performed yielding a total complexity of $O(n^3ek + e^2k^2)$ steps. □

To explain the difference between ULT and PC, we view every disjunctive constraint as a single interval with *holes*. The single interval specifies the upper and lower bounds of legal values while the holes specify intervals of illegal values.

Lemma 2: *Algorithms ULT and PC compute the same upper and lower bounds.*

Proof: The lower and upper bounds are modified using the $\cap$ and the $\odot$ operators. We observe that $low(C_{ik} \odot C_{kj}) = low(C_{ik}) + low(C_{kj})$ which is equal to the lower bound of $[low(C_{ik}), high(C_{ik})] \odot [low(C_{kj}), high(C_{kj})]$. A similar observation is made for the upper bound. Consequently, the lower and upper bounds of $C_{ij} \cap ([low(C_{ik}), high(C_{ik})] \odot [low(C_{kj}), high(C_{kj})])$ and $C_{ij} \cap (C_{ik} \odot C_{kj})$ are equal. Additional iterations performed by PC only enlarge the 'holes'. □

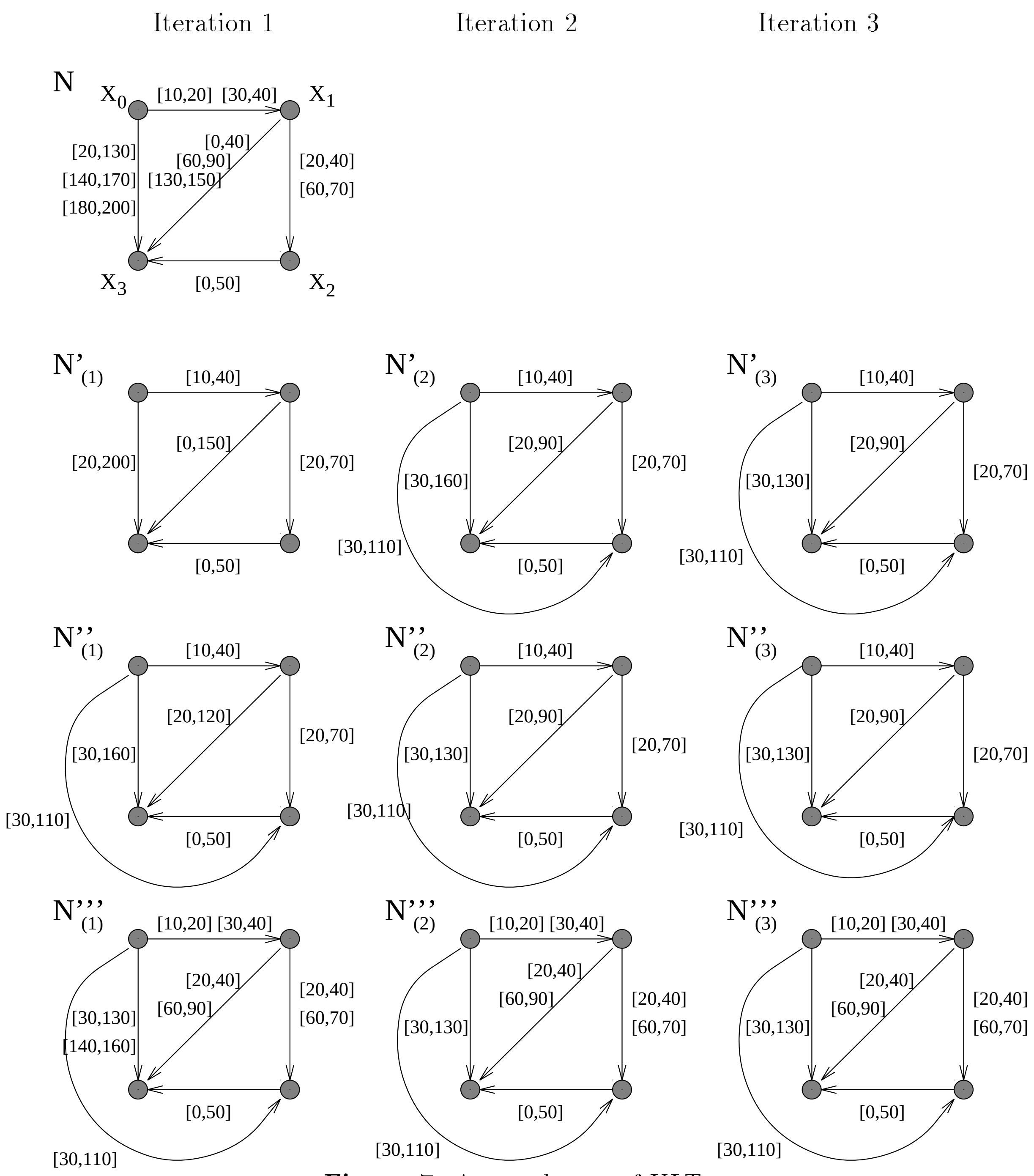


Figure 7: A sample run of ULT.

Algorithm ULT-2

```
1.  $Q \leftarrow \{(i,k,j) | (i < j) \ and \ (k \neq i,j)\}$
2.  **while** $Q \neq \{\}$ **do**
3.      select and delete a path $(i,k,j)$ from $Q$
4.      $T''_{ij} \leftarrow C_{ij} \cap ([low(C_{ik}), high(C_{ik})] \odot [low(C_{kj}), high(C_{kj})])$
5.      **if** $T'_{ij} = \{\}$ **then** exit (inconsistency)
6.      **if** $T''_{ij} \neq C_{ij}$ **then**
            $Q \leftarrow Q \ \cup \ \{(i,j,k),(k,i,j) \mid 1 \leq k \leq n, i \neq k \neq j\ \}$
7.      $C_{ij} \leftarrow T''_{ij}$
8.  **end-while**
```

Algorithm DULT

```
1.  **for** $k \leftarrow n$ downto 1 by -1 **do**
2.      **for** $\forall i,j < k$ such that $(i,k),(k,j) \in E$ **do**
3.      $T''_{ij} \leftarrow C_{ij} \cap ([low(C_{ik}), high(C_{ik})] \odot [low(C_{kj}), high(C_{kj})])$
4.          **if** $T''_{ij} = \{\}$ **then** exit (inconsistency)
5.          **if** $T''_{ij} \neq C_{ij}$ **then** $E \leftarrow E \cup (i,j)$
6.          $C_{ij} \leftarrow T''_{ij}$
7.      **end-for**
8.  **end-for**
```

Figure 8: Algorithms ULT-2 and DULT.

Thus, the difference between ULT and PC is the propagation of the holes. In contrast to PC, ULT is guaranteed to converge in $O(ek)$ iterations even if the interval boundaries are not rational numbers.

3.3.3 Variations of ULT

While an iteration of ULT is divided into three sequential stages that involve the whole network, algorithm PC uses simpler local operations over triplets of variables and admits parallel execution. We next present two variations on ULT, called ULT-2 and Directional ULT (DULT), which perform such local computations (see Figure 8). We use $low(C_{ij})$ and $high(C_{ij})$ to denote, respectively, the lowest lower bound and highest upper bound of the union of the intervals in C_{ij}.

Theorem 11: *Given a network N, let n be the number of variables, e the number of constraints and k the maximum number of intervals per constraint.*

1. *Algorithms ULT-2 and DULT terminate in $O(n^3k^2 + ek^3n)$, $O(n^3k^2)$ steps respectively and compute a network equivalent to their input network.*

2. *Algorithm ULT-2 computes a tighter network than DULT.*

Proof: Part 1. Algorithm ULT-2 initializes the queue with $O(n^3)$ triangles. A set of $O(n)$ triangles is added to Q (Figure 8, Alg ULT-2 line 6) only if at least one interval was removed from the network, and therefore at most $O(ekn)$ triangles are added. Since computing $T \odot S$ requires at most $O(k^2)$ steps, the total complexity for ULT-2 is $O(n^3k^2 + ek^3n)$. Algorithm DULT performs a single pass of $O(n^3)$ triangles and each triangle requires $O(k^2)$ steps.
Part 2. Every triangle that is considered in DULT is also considered in ULT-2 but not vice versa, thus DULT is weaker. □

Algorithm ULT can be extended to process discrete Constraint Satisfaction Problems (see Appendix A).

3.4 A Tractable Class

This section analyzes the class of quantitative TCSPs in which the binary constraints C_{ij} specify single intervals but the unary constraints C_{0i} may specify an arbitrary number of intervals. It subsumes the class of convex point algebra networks with holes in their domains [71], but it is not comparable to the class of STPs with disjunctions of inequations [59] over dense domains.

Definition 8: [STP Upper Bound]
The STP upper bound of a network N, denoted $N' = STP(N)$ is such that $C'_{ij} = [low(C_{ij}),\ high(C_{ij})]$.

Lemma 3: *[27] For a minimal STP with the constraints $C_{ij} = [L_{ij}, U_{ij}]$, the instantiation $X_i = L_{0i}$ is a solution.*

Lemma 4: *For every quantitative TCSP N, if $STP(N)$ is minimal then the instantiation $X_i = low(C_{0i})$ is a solution of N if all the binary constraints (i.e. $C_{ij},\ \forall i > 0, \forall j > 0$) specify a single interval.*

Proof: From Lemma 3 it follows that this instantiation is a solution of $STP(N)$. Thus, all the binary constraints are satisfied by this instantiation. Clearly, because $L_{0i} \in C_{0i} = C_i$, all the disjunctive constraints C_{0i} are also satisfied by this instantiation. □

Lemma 5: *Algorithm ULT computes a network N whose $STP(N)$ is minimal.*

Proof: Follows immediately from Definition 7. □
From Lemmas 4,5 we get:

Theorem 12: *Algorithm ULT correctly decides consistency of TCSPs in which $C_{ij},\ \forall i > 0,\ \forall j > 0,$ is specified by a single interval.*

This class of problems is frequently encountered. Consider, for example, scheduling tasks that use resources available in a set of time windows. The availability of resources constrains the times at which tasks can be accomplished and can be describe by unary constraints. For example, suppose we would like to transport cargo from the east coast to the west coast. To use an air carrier we need to consider resource availability constraints. These constraints need to be described by disjunctive unary constraints on the times that cargo loading and unloading can occur.

This class can be generalized using the notion of a *disjunctive constraint graph* $G(V, E)$ whose vertices V correspond to variables and edges E specify disjunctive constraints only.

Corollary 1: *ULT is complete for TCSPs whose disjunctive constraint graph is a* STAR*, namely a tree in which all edges are incident on a single node.*

Proof: Label the root of the STAR by X_0 and apply Theorem 4. $\square$

Moreover, even if the input TCSP is not a STAR, ULT may remove disjunctions and obtain a STAR network. In such cases, ULT is complete.

An important consequence of the above is that ULT reduces the search space by an exponential factor. Since the search algorithm need not consider all the disjunctive constraints connected to the node with the maximal degree, the search space is reduced by a factor of $O(k^{d(G)})$, where G is the *disjunctive constraint graph*, $d(G)$ be the maximal degree of G and k is the disjunction size, namely the number of intervals in each constraint. In Section 8.2.1 we show empirically that without preprocessing with ULT even tiny problems were computationally prohibitive and could not be solved in a reasonable amount of time.

Unfortunately, our analysis cannot be extended to networks whose disjunctive constraint graph is a general tree. Consider a triangle X_i, X_j, X_k with the constraint bounds $[L_{ij}, U_{ij}]$, $[L_{ik}, U_{ik}]$ and $[L_{kj}, U_{kj}]$ respectively. When STP(N) is minimal we are guaranteed that $L_{ij} \geq L_{ik} + L_{kj}$. Thus, instantiating $X_k = X_i + L_{ik}$ and $X_j = X_k + L_{kj}$ does not guarantee that $X_j - X_i \in [L_{ij}, U_{ij}]$.

3.4.1 ULT for discrete CSPs

The idea of ULT can be extended to approximate path-consistency in classical CSPs. While enforcing full path-consistency requires $O(n^3k^3)$ steps [74], approximating with a single iteration of ULT requires $O(n^3k^2)$, and using the complete ULT requires $O(n^3ek + e^2k^2)$. Using a single ULT iteration (weaker than ULT) may significantly reduce propagation time (compared to PC) when the domains are large.

A binary relation R_{ij} on X_i, X_j can be represented by a (0,1)-matrix with $|D_i|$ rows and $|D_j|$ columns by imposing an ordering on the domains. A zero entry at row

r and column s means that the pair consisting of the r-th element of D_i and the s-th element of D_j in not allowed.

Definition 9: (row convexity [101]) A (0,1)-matrix is *row convex* iff in each row all of the ones are consecutive, that is no two ones within a single row are separated by a zero in that same row. A constraint is *row convex* iff its matrix representation is *row convex* and the network is *row convex* iff all its constraints are *row convex.* A row convex relation can be represented by a set of k pairs of integers, (l_r, u_r), where l_r is the number of the first non-zero column and u_r is the number of the last non-zero column.

It was shown that enforcing path-consistency on *row convex* networks renders them globally consistent [101]. In Figure 9, we present algorithm ULT-CSP. The algorithm relaxes the network into a *row-convex* network, enforces path-consistency and intersects the resulting network with the original network, until there is no change.

Definition 10: Given an arbitrary matrix A, its *upper bound row convex* matrix is obtained by changing, for every row r, all the elements between column l_r and u_r, (e.g. $a_{r,l_r} \ldots a_{r,u_r}$) to ones. An upper bound row convex approximation of a binary constraint is obtained by computing an upper bound row convex of its matrix representation. The networks N', N'', N''' are defined as follows:

- N' is the *row convex upper bound* of N.
- N'' is the minimal network of N' (obtained by enforcing path-consistency).
- N''' is derived from N' and N'' by intersection.

Theorem 13: *Let N be the input to ULT-CSP and R be its output.*

1. *N and R are equivalent networks.*
2. *For row convex networks, ULT-CSP computes the minimal network in a single iteration.*
3. *Every iteration of algorithm ULT-CSP terminates in $O(n^3k^2)$ steps.*

Proof: Part 1: Let $Sol(N)$ denote the set of solutions of then $Sol(N) \subseteq Sol(N') = Sol(N'')$. This implies that $Sol(N) \cap Sol(N'') = Sol(N)$ and therefore $Sol(N''') = Sol(N)$. Part 2: Clearly, if the input network is row convex, then $N = N'$ and it is known that for row convex networks path-consistency is complete [102]. Part 3: Computing l_r, u_r for every row in every matrix requires $O(n^2k^2)$ steps and enforcing path-consistency on row convex networks requires $O(n^3k^2)$ steps. □

Algorithm ULT-CSP

```
1.  input: $N$
2.  $N''' \leftarrow N$
3.  repeat
4.      $N \leftarrow N'''$
5.      Compute $N'$ by computing the row-convex upper bound of $N$.
6.      Compute $N''$ by enforcing path consistency on $N'$.
7.      Compute $N'''$ by intersecting $N'$ and $N''$.
8.  until $N''' = N$.
9.  if  $N'''$ is consistent,   output: N"'.
                                output: "Inconsistent."
```

Figure 9: Algorithm ULT-CSP.

3.5 Loose Path-Consistency (LPC)

Now we present algorithm *Loose Path-Consistency (LPC)*, which is stronger than ULT and its variants, namely it generates tighter approximations to PC. The algorithm is based on the following loose intersection operator.

Definition 11: Let $T = \{I_1, I_2, \ldots, I_r\}$ and $S = \{J_1, J_2, \ldots, J_s\}$ be two constraints. The *loose intersection*, $T \triangleleft S$ consists of the intervals $\{I'_1, \ldots, I'_r\}$ such that $\forall i \;\; I'_i = [L_i, U_i]$ where $[L_i, U_i]$ are the lower and upper bounds of the intersection $I_i \cap S$.

It is easy to see that the number of intervals in C_{ij} is not increased by the operation $C_{ij} \leftarrow C_{ij} \triangleleft (C_{ik} \odot C_{kj})$. In addition, $\forall k \;\; C_{ij} \supseteq C_{ij} \triangleleft (C_{ik} \odot C_{kj}) \supseteq C_{ij} \cap (C_{ik} \odot C_{kj})$ and $T \triangleleft S \neq S \triangleleft T$.

Example 13: Let $T = \{[1,4],[10,15]\}$ and $S = \{[3,11],[14,19]\}$. Then $T \triangleleft S = \{[3,4],[10,15]\}$, $S \triangleleft T = \{[3,11],[14,15]\}$ while $S \cap T = \{[3,4],[10,11],[14,15]\}$.

According to Definition 2, a constraint C_{ij} is path-consistent iff $C_{ij} \subseteq \cap_{\forall k}(C_{ik} \odot C_{kj})$. By replacing the intersection operator $\cap$ with the loose intersection operator $\triangleleft$, we can bound the fragmentation. Algorithm LPC is presented in Figure 10. The network N' is a relaxation of N and therefore loosely intersecting N'' with N results in an equivalent network.

Example 14: In Figure 11 we show a trace of LPC on a sample quantitative TCSP. We start with N and compute $N'_{(1)}$, $N''_{(1)}$. Thereafter, we perform a second iteration in which we compute $N'_{(2)}$, $N''_{(2)}$. Finally, in the third iteration, there is no change. The first iteration removes 7 intervals while the second iteration removes a single interval. We see that LPC explicates an induced constraint C_{02}, which allows to infer

```
Algorithm Loose Path-Consistency (LPC)
1. input: $N$
2. $N'' \leftarrow N$
3. repeat
4.     $N \leftarrow N''$
5.     Compute $N'$ by assigning $T'_{ij} = \cap_{\forall k}(C_{ik} \odot C_{kj})$, for all $i, j$.
6.     Compute $N''$ by loosely intersecting $T''_{ij} = C_{ij} \triangleleft T'_{ij}$, for all $i, j$.
7. until     $\exists i, j \quad (T''_{ij} = \phi)$          ; inconsistency, or
             or $\forall i, j \quad |T''_{ij}| = |C_{ij}|$     ; no interval removed.
8. if  $\exists i, j \quad (T''_{ij} = \phi)$     then output "inconsistent."
                                    else output: $N''$.
```

Figure 10: The Loose Path-Consistency (LPC) algorithm.

a new implicit fact about the times that event X_2 can occur. Note that applying ULT on the same network will have no effect and applying PC on it results in the same network as results from applying LPC.

Lemma 6: *Let N be the input to LPC and R be its output.*

1. *The networks N and R are equivalent.*
2. *Every iteration of LPC (excluding the last) removes at least one interval from one of the constraints.*

Proof: Immediate. □

Theorem 14: *Algorithm LPC terminates in $O(n^3k^3e)$ steps where n is the number of variables, e is the number of constraints and k is the maximal number of intervals in each constraint.*

Proof: Computing N' requires processing every triangle in the network once, thus requires $O(n^3k^2)$ steps. Because in every iteration at least one interval is removed, there are at most ek iterations. The complexity is therefore $O(n^3k^3e)$. □

Algorithm LPC computes tighter networks than ULT. For detailed execution, see Figure 11. To clarify the differences among ULT, LPC and PC, we can view every disjunctive constraint as a single interval with *holes* (as in Section 3.2). The single interval specifies the upper and lower bounds of legal values, while the holes specify intervals of illegal values.

Lemma 7: *Algorithms ULT, LPC and PC compute the same upper and lower bounds.*

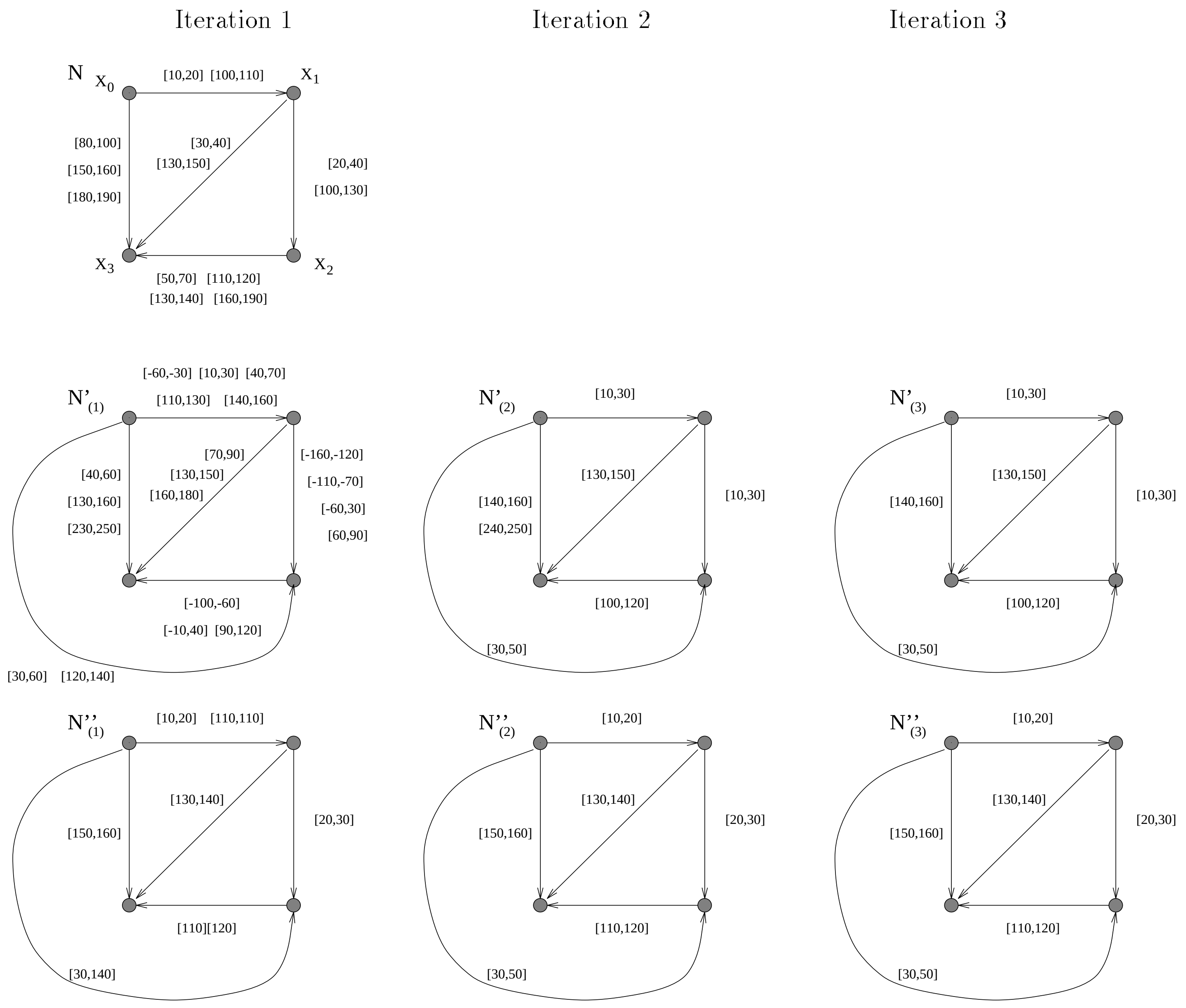


Figure 11: A sample run of LPC.

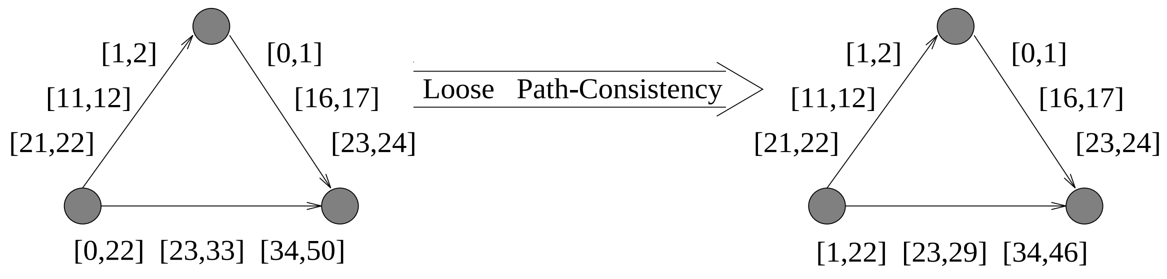


Figure 12: Solving the fragmentation problem.

Algorithm LPC-2

```
1.  $Q \leftarrow \{(i,k,j) | (i < j)\ and\ (k \neq i,j)\}$
2.  while $Q \neq \{\}$ do
3.      select and delete a path $(i,k,j)$ from $Q$
4.      $T'_{ij} \leftarrow C_{ij} \triangleleft (C_{ik} \odot C_{kj})$
5.      if $T'_{ij} = \{\}$ then exit (inconsistency)
6.      if $|T'_{ij}| < |C_{ij}|$ then
            $Q \leftarrow Q \cup \{(i,j,k),(k,i,j) \mid 1 \leq k \leq n, i \neq k \neq j\}$
7.      $C_{ij} \leftarrow T'_{ij}$
8.  end-while
```

Algorithm DLPC

```
1.  for $k \leftarrow n$ downto 1 by -1 do
2.      for $\forall i,j < k$ such that $(i,k),(k,j) \in E$ do
3.          $T'_{ij} \leftarrow C_{ij} \triangleleft (C_{ik} \odot C_{kj})$
4.          if $T'_{ij} = \{\}$ then exit (inconsistency)
5.          if $|T'_{ij}| < |C_{ij}|$ then $E \leftarrow E \cup (i,j)$
6.          $C_{ij} \leftarrow T'_{ij}$
7.      end-for
8.  end-for
```

Figure 13: Algorithms LPC-2 and DLPC.

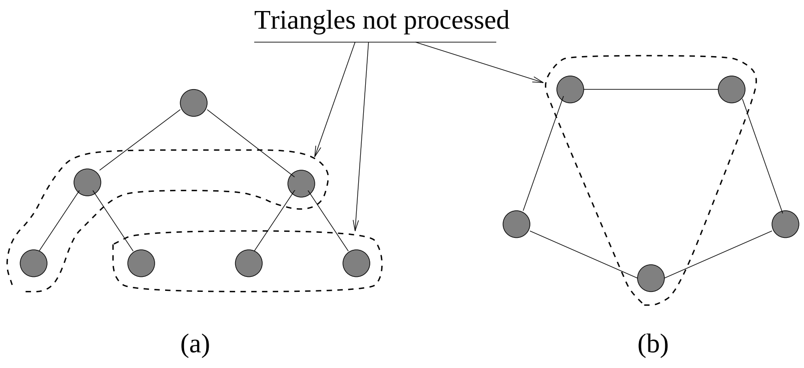


Figure 14: The utility of PLPC.

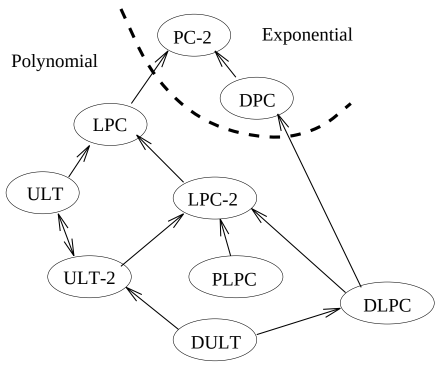


Figure 15: The partial order on the effectiveness.

Proof: Using the same arguments as in the proof of Lemma 2 we show that the lower and upper bounds of $C_{ij} \triangleleft (C_{ik} \odot C_{kj})$ and $C_{ij} \cap [low(C_{ik}), high(C_{ik})] \odot [low(C_{kj}), high(C_{kj})]$ are equal to the bounds of $C_{ij} \cap (C_{ik} \odot C_{kj})$. $\square$

Thus, the difference among ULT, LPC and PC is in their propagation of holes. Algorithm ULT does not change the holes. LPC may enlarge the holes, while PC may create additional holes.

3.5.1 Variations of LPC

We next present two variations on LPC which have the same structure as PC and DPC. These algorithms, presented in Figure 13, are called LPC-2 and Directional LPC (DLPC). They differ from PC and DPC only in using the loose intersection operator $\triangleleft$ instead of the strict intersection operator $\cap$.

Theorem 15: *Given a network N, let n be the number of variables, e be the number of constraints and k be the maximum number of intervals per constraint. Algorithms*

LPC-2 and DLPC terminate in $O(n^3k^2 + ek^3n))$, $O(n^3k^2)$ steps, respectively, and they compute TCSPs which are equivalent to their input.

Proof: Algorithm LPC-2 applies the operation $C_{ij} \leftarrow C_{ij} \triangleleft (C_{ik} \odot C_{kj})$ which does not change the set of solutions, and thus the resulting network is equivalent. Initially, the queue Q consists of $O(n^3)$ triangles. A set of $O(n)$ triangles is added to Q (LPC-2 line 6) only if at least one interval was removed from the network, and therefore at most $O(ekn)$ triangles are added. Since computing $T \odot S$ requires at most $O(k^2)$ steps the total complexity of LPC-2 is $O(n^3k^2 + ek^3n)$. Algorithm DLPC applies the operation $C_{ij} \leftarrow C_{ij} \triangleleft (C_{ik} \odot C_{kj})$ at most $O(n^3)$times. Each such operation does not change the set of solutions and requires $O(k^2)$ steps. Thus the overall complexity of DLPC is $O(n^3k^2)$. □

3.5.2 Partial LPC (PLPC)

To refine the tradeoff between effectiveness and efficiency, we suggest another variant for constraint propagation, called *Partial LPC (PLPC)*. We apply the relaxation operation $C_{ij} \leftarrow C_{ij} \triangleleft (C_{ik} \odot C_{kj})$ only in cases where C_{ij} and at least one of C_{ik} and C_{kj} is non-universal in the input network. Consider, for example, the tree network in Figure 14a and the circle network in Figure 14b. The dashed lines outline several triangles that are not processed.

3.5.3 Relative Effectiveness

The partial order on the effectiveness of all the algorithms presented in this chapter is shown in Figure 15. A directed edge from algorithm $\mathcal{A}_1$ to $\mathcal{A}_2$ indicates that $\mathcal{A}_2$ computes an equivalent network which is equal or tighter than $\mathcal{A}_1$ on an *instance by instance* basis. This means that $\mathcal{A}_2$ can detect inconsistencies that $\mathcal{A}_1$ cannot detect, but not vice versa. Note that algorithms PC and DPC are exponential.

3.6 Combining Quantitative and Qualitative Constraints

In this section, we present Meiri's extension [71] which combines qualitative and metric constraints over time points and intervals.

A combined qualitative and quantitative TCSP involves a set of variables and a set of binary constraints over pairs of variables. There are two types of variables, point variables and interval variables. The constraint C_{ij} between a pair of variables, X_i, X_j is described by specifying a set of allowed relations, namely

$$C_{ij} \overset{\text{def}}{=} (X_i\ r_1\ X_j) \lor \cdots \lor (X_i\ r_k\ X_j). \tag{3.1}$$

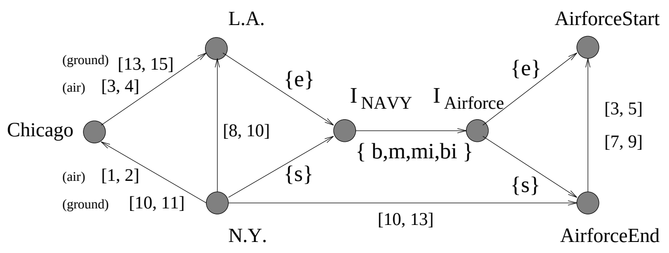


Figure 16: The complete constraint graph of the logistics problem.

There are three types of relations, or alternatively, disjunctive constraints:

1. A point-point constraint between two point variables X_i, X_j is *quantitative*[2] and has the form
$$X_j - X_i \in I_1 \cup \cdots \cup I_k$$
where $I_1, \ldots, I_k$ are intervals.

2. A point-interval constraint between a point variable and an interval variable, is *qualitative*, and is in the set { **before, starts, during, finishes, after** } abbreviated { b, s, d, f, a } respectively (see Section 2.6) [71].

3. An interval-interval constraint between two interval variables is *qualitative*, and is in the set
$$\left\{ \begin{array}{c} \textbf{before}, \textbf{after}, \textbf{meets}, \textbf{met}-\textbf{by}, \\ \textbf{overlaps}, \textbf{overlaps}-\textbf{by}, \textbf{during}, \textbf{contains}, \textbf{equals}, \\ \textbf{starts}, \textbf{started}-\textbf{by}, \textbf{finishes}, \textbf{finished}-\textbf{by} \end{array} \right\}$$
abbreviated { b,bi, m,mi, o,oi, d,di, =, s,si, f,fi } respectively (see Section 2.4) [5].

Example 15: Consider the cargo example of section 1. Let the variables be:

$X_{N.Y.}$ = time point at which the NAVY cargo was shipped out of N.Y.,
$X_{Chicago}$ = time point the NAVY cargo arrived into and shipped out of CHICAGO
$X_{L.A.}$ = time point at which the NAVY cargo arrived into L.A.
I_{NAVY} = transportation interval of the NAVY cargo.
$I_{Airforce}$ = transportation interval of the AIRFORCE cargo.
$X_{AirforceStart}$ = time point at which the AIRFORCE shipment *starts*,
$X_{AirforceEnd}$ = time point at which the AIRFORCE shipment *ends*.

[2]In Meiri [71] a distinction is made between qualitative and quantitative point-point constraints.

The constraints are:

$$\begin{array}{l}
X_{N.Y.} - X_0 \in [March7,\ March7] \\
X_{Chicago} - X_{N.Y.} \in [1,2] \cup [10,11] \\
X_{L.A.} - X_{Chicago} \in [3,4] \cup [13,15] \\
X_{L.A.} - X_{N.Y.} \in [8,10] \\
X_{N.Y.}\ \{starts\}\ I_{NAVY} \\
X_{L.A.}\ \{ends\}\ I_{NAVY} \\
X_{AirforceBegin}\ \{starts\}\ I_{Airforce} \\
X_{AirforceEnd}\ \{ends\}\ I_{Airforce} \\
X_{AirforceEnd} - X_{AirforceBegin} \in [3,5] \cup [7,9] \\
X_{AirforceBegin} - X_{N.Y.} \in [10,13] \\
I_{NAVY}\ \{before,\ meets,\ met\!-\!by,\ after\}\ I_{Airforce}
\end{array}$$

The last constraint means that I_{NAVY} and $I_{Airforce}$ are disjoint. The constraint graph representing this network is given in Figure 16.

3.6.1 Extending LPC for Combined Networks

For brevity we will describe the extension for LPC, but ULT can be extended using the same methodology. As defined in Section 2, the combined model involves three types of constraints: point-point (quantitative), point-interval (qualitative) and interval-interval (qualitative). Each node in a triangle can be either a point or an interval variable, resulting in $2^3 = 8$ types of triangles. We therefore modify the semantics of the $\triangleleft$ and the $\odot$ operators to accommodate all 8 types.

Let C_{ij}, C_{ik}, C_{kj} be the constraints on pairs X_i, X_j, X_i, X_k and X_j, X_k. For computing $T'_{ij} \leftarrow C_{ij} \triangleleft (C_{ik} \odot C_{kj})$, we use Meiri's tables, except when quantitative constraints are used. We consider the following cases:

Case 1: If X_i, X_j, X_k are interval variables then Allen's transitivity table [5] is used to compute $C_{ik} \odot C_{kj}$ and the $\triangleleft$ operator is interpreted as the usual intersection operator.

Case 2: If both X_i, X_j are interval variables and X_k is a point variable then Meiri's transitivity tables [71] are used to compute $C_{ik} \odot C_{kj}$ and the $\triangleleft$ operator is interpreted as the usual intersection.

Case 3: If exactly one of X_i, X_j is an interval variable and X_k is a point variable, then the quantitative point-point constraint, C_{ik} or C_{kj}, is translated into a qualitative point-point constraint (using $<, >, =$) and Meiri's transitivity tables [71] are used to compute $C_{ik} \odot C_{kj}$; the $\triangleleft$ operator is interpreted as the usual intersection.

Case 4: If X_i, X_j are point variables and X_k is an interval variable then $C_{ik} \odot C_{kj}$ is computed using Meiri's transitivity tables [71]. If $C_{ik} \odot C_{kj} \neq \{<, >\}$ then the resulting constraint is translated into a single interval and the $\triangleleft$ operator is interpreted as the $\cap$ operator in Definition 1. Otherwise, to avoid increasing the number of intervals in C_{ij}, we set $T'_{ij} \leftarrow C_{ij}$ (i.e. no change).

Case 5: If X_i, X_j, X_k are point variables, then the composition operation used is described by Definition 1 and the $\triangleleft$ operator is described in Definition 3.

With these new definitions of the operators $\odot$ and $\triangleleft$, we can apply algorithms LPC, LPC-2 and DLPC for processing combined networks.

3.7 General Backtracking

Algorithms ULT and LPC are useful for detecting inconsistencies and for explicating constraints, however they are not designed to find a consistent scenario. A brute-force algorithm for determining consistency or for computing consistent scenarios can decompose the network into separate simple subnetworks by selecting a single interval from each quantitative constraint and a single relation from a qualitative constraint [71, 27]. Each sub network can then be solved separately in polynomial time by enforcing path-consistency, and the solutions can be combined. Alternatively, a naive backtracking algorithm can successively select one interval or relation from each disjunctive constraint as long as the resulting network is consistent [71, 27]. Once inconsistency is detected, the algorithm backtracks. This algorithm can be improved by performing *forward checking* to reduce the number of possible future interval assignments during the labeling process.

Definition 12: [71] A *basic label* of an arc $i \rightarrow j$ is either a selection of a single interval from the interval set C_{ij} for quantitative constraints, or a selection of a single relation for qualitative constraints. A *singleton labeling* of N is a selection of a basic label for *all* the constraints in N and a *partial labeling* of N is a selection such that *some* constraints are assigned basic labels.

A singleton labeling of a combined network can be described by an STP [71]. Thus, deciding the consistency of a singleton labeling can be done in $O(n^3)$ steps, by enforcing path-consistency [71].

Lemma 8: *Algorithms ULT, ULT-2, DULT, LPC, LPC-2 and DLPC and their extension for processing combined networks decide consistency of a singleton labeling.*

Proof: When there are no disjunctions, the quantitative TCSP can be described by an STP, for which all of the above algorithms are complete. Enforcing path-consistency of a qualitative TCSP with no disjunctions is known to decide its consistency [5, 71]. □

We can apply backtrack search with forward checking in the space of partial labelings as follows: The algorithm chooses a disjunctive constraint and replaces

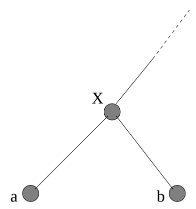


Figure 11: Sample portion of the search tree

it with a single interval (if metric) or a single relation (if qualitative) from that constraint. When the constraints are chosen in a dynamic order, the constraint with the smallest disjunction size is selected for labeling. Thereafter, the network can be tightened using ULT and LPC. Subsequently, the algorithm selects a new constraint from the tightened network, assigns it a label and tests consistency again. This is repeated until either inconsistency is detected (by ULT or LPC) or a consistent singleton labeling is found. When inconsistency is detected, a dead-end is declared and the algorithm backtracks by undoing the last constraint labeling.

Additional improvements we propose are (1) to avoid constraint propagation on any subnetwork that is already singly labeled (since it is already consistent) (2) to avoid using a stack for undoing the last constraint labeling[3], and instead, to reconstruct the previous partial labeling using the indexes of the labels; (3) to avoid instantiating constraints that were universal in the input network but became non-universal as a result of constraint propagation.

Algorithms ULT and LPC are also useful for preprocessing *before* initiating search. They reduce the number of disjuncts in the constraints, that is the number of intervals in quantitative constraints and the number of allowed relations in qualitative constraints. As a result, the branching factor of the search space is reduced. In addition to reducing the disjunction size, these algorithms render all the universal constraints non-universal. In contrast, using path-consistency algorithms for preprocessing increases the fragmentation and the branching factor.

3.7.1 Detailed Backtracking Algorithms

The key for scaling up the backtrack algorithm suggested above is effective memory management. In contrast to classical CSPs, when backtracking on TCSPs there is a need to store information about the complete network at each level of the search tree.

To illustrate the problem, consider part of the search tree shown in Figure 11, in which every node is a *partial labeling*. Suppose the search algorithm expands node "X", and thereafter visits the left child labeled by "a". Suppose that once node "a" was visited there is a need to backtrack due to inconsistency. The naive way to allow backtracking is to simply store the partial labeling of "X". Such an approach, how-

[3]In the stack there would be $O(n^2)$ entries of size $O(n^2)$ each - this was the major problem in [63].

ever, requires to store all partial labelings on the path from the root of the search tree to the current node, which may require $O(n^4)$ space[4]. Instead, we propose to construct the partial labeling of node "b" during search without storing or reconstructing "X". We store only the necessary information required for reconstruction of node "b", namely the index of the basic labels within every constraint. Note that applying LPC removes some intervals from the constraints and therefore such an indexing should be carefully handled. The saving obtained by this method is mostly apparent when the common parent of a and b is several levels up, closer to the root of the tree.

When a dead-end is encountered, we determine the source of the conflics as follows. Suppose the dead-ends occurred at the constraint C_{ij}, namely, instantiating C_{ij} with any of its intervals renders the networks inconsistent. Suppose the constraint instantiated before C_{ij} was C_{pq}. Then if the networks in which C_{pq} is made universal is inconsistent with every possible instantiation of C_{ij} then C_{pq} is clearly not responsible for the dead-end. In that case, we check the constraints instantiated before C_{pq} by making it universal and enforcing path-consistency, until we find a constraint for which path-consistency does not detect inconsistency.

The complete backtracking algorithm is presented in Figure 16. The function of $LabelNetwork$, shown in Figure 16, is to reconstruct the partial labeling based on the indexes. It receives as input the original network N (the root of the tree), the indexes of the basic labels to be selected from each constraint stored in the $Index$ matrix, and the last constraint which to be instantiated, C_{ij}. Two copies of the network are maintained: N is the original input network and N' is the partial labeling currently expanded; the ij-th constraint of N' is denoted by T'_{ij}.

In contrast to Ladkin and Reinefeld, we propose to perform limited propagation. As shown in Figure 15 lines 5-8, because every iteration of the "repeat" loop removes at least one atomic relation from T'_{ij} (otherwise no change) we perform at most $max(26n, 2nk)$ relaxation operations where n is the number of variables and k is the maximal number of intervals in a point-point constraint. In average, however, we perform much less than $2nk$.

Finally, the last improvement we propose is not to instantiate constraints that were initially universal. It is easy to see that every consistent labeling of all the non-universal constraints is also consistent with the universal constraints; as a result, unnecessary dead-ends can be avoided.

3.8 Empirical Evaluation

Our empirical evaluation is addressing two questions: (1) which of the polynomial approximation algorithms presented in this chapter is preferable for detecting inconsistencies, and (2) how effective are these algorithm when used to improve backtrack

[4]an entry for every constraint - $O(n^2)$ entries; each entry describes a complete network - $O(n^2)$ space each.

```
Input:    A network $N$ with $n$ variables.
Output:   A consistent *singleton labeling* of $N$ if consistent, or
          a notification that $N$ is inconsistent.
 1. $i \leftarrow 0$;  $j \leftarrow 1$
 2. repeat
 3.    $N' \leftarrow LabelNetwork(N, Index, i, j)$
 4.    $Index[i,j] \leftarrow 0$
 5.    repeat
 6.          $\forall k \in [j+1, n]$   $T'_{ik} \leftarrow T'_{ik} \triangleleft (T'_{ij} \odot T'_{jk})$
 7.          $\forall k \in [j+1, n]$   $T'_{kj} \leftarrow T'_{kj} \triangleleft (T'_{ki} \odot T'_{ij})$
 8.    until no change.
 9.    $Length[i,j] = |T'_{i,j}|$
10.    if $N'$ is inconsistent then let $i, j$ be the *previous* $i, j$ such that $i < j$.
11.    while $Index[i,j] \geq Length[i,j]$ and $j > 0$ do
12.          let $i, j$ be the *previous* $i, j$ such that $i < j$.
13.    if $j > 0$ then
14.          $Index[i,j] = Index[i,j] + 1$
15.          let $i, j$ be the next constraint to be instantiated such that $i < j$.
16. until $N'$ is a consistent *singleton labeling* or $j = 0$.
17. if   $N'$ is a consistent *singleton labeling*   then exit with $N'$ as the solution.
18.                                                 else exit with failure.
```

Figure 15: The general Backtracking algorithm.

```
LabelNetwork($N$, $Index$, $i, j$)
Input:    $N$, a network with $n$ variables,
          $Index$, the indexes of the labels,
          $i, j$, the current constraint not to be instantiated.
Output:   $N'$, a *partial labeling* of $N$.
1.    $\forall q \in [0, j-1]$,  $\forall p \in [0, q-1]$,  $T'_{pq} \leftarrow$ *the* $Index[p,q]$*-th label of* $C_{pq}$.
2.    $\forall q \in [0, i-1]$,  $T'_{jq} \leftarrow$ *the* $Index[j,q]$*-th label of* $C_{jq}$.
3.    return($N'$).
```

Figure 16: Reconstructing a partial labeling.

search via preprocessing, forward checking and dynamic variable ordering. Section 8.1 presents experiments addresing the first question by measuring the tradeoff between efficiency and effectiveness. Section 8.2 presents experiments addressing the second question.

The problems were generated with the following parameters: n and e are the number of variables and constraints respectively, and k is the number of intervals per quantitative point-point constraint. These quantitative constraints specify integers in the domain $[-R, R]$, and the tightness α of a constraint $T = \{I_1, \ldots, I_k\}$ is $(|I_1| + \cdots + |I_k|)/2R$ where $|I_i|$ is the size of I_i. We used uniform tightness for all constraints. The parameter β is the number of relations in every point-interval constraint and the parameter γ is the number of relations in every interval-interval constraint.

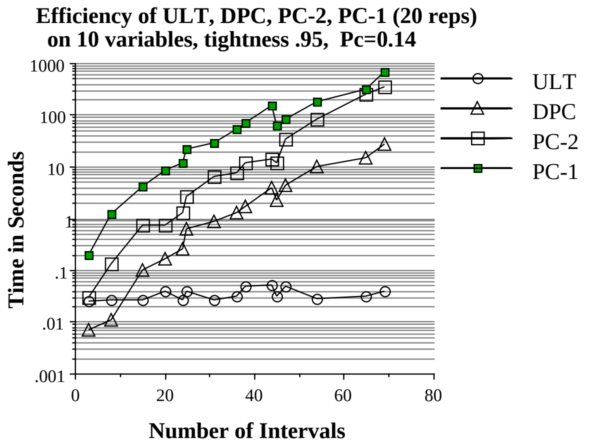
3.8.1 Comparing Constraint Propagation Algorithms

We evaluate the tradeoff between efficiency and effectiveness of ULT and LPC. Efficiency is measured by comparing the execution time. The effectiveness or accuracy of an algorithm $\mathcal{A}$ is the fraction of times $\mathcal{A}$ returns a correct consistency decision. Since comparing the correct answer by search is too time consuming, we propose to measure relative effectiveness instead. To define the notion of relative effectiveness, we rely on the observation that all the approximation algorithms described in this chapter are sound, namely when a problem is classified as inconsistent this classification is correct. Thus, two algorithms can differ only in the number of problems they incorrectly classify as consistent. We therefore define the relative effectiveness of two algorithms as the ratio between the number of inconsistencies detected by the algorithms evaluated. In all accuracy plots we use the strong algorithm as the reference point, namely it has 100% accuracy.

Path-Consistency versus ULT

In this subsection, we discuss two variants on PC: algorithms PC-1 and PC-2. By PC-1 we refer to the brute-force path-consistency algorithm presented in [27] and by PC-2 we refer to the algorithm presented in Figure 3.3a. We use PC as a collective name for both PC-1 and PC-2.

Figure 15 describes qualitatively the strength of the various algorithms. We next present a quantitative empirical comparison of algorithms PC-1, PC-2, DPC and ULT. In Figure 17 we show that both PC-1, PC-2 and DPC may be impractical even for small problems with 10 variables. We see that although ULT is orders of magnitude more efficient than PC-1 and PC-2, ULT is able to detect inconsistency in about 70% of the cases that PC-1, PC-2 and DPC detect inconsistencies. Subsequently, we measure the *relative* efficiency-effectiveness tradeoff for ULT and LPC.



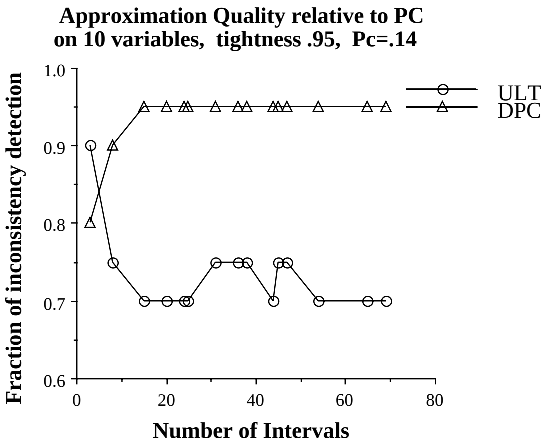


Figure 18: Execution times and quality of the approximations obtained by DPC and ULT relative to PC. Each point represents 20 runs on networks with 10 variables, 95

Comparing ULT, LPC, DLPC and PLPC

Here we measure the relative effectiveness tradeoffs of LPC, ULT, DLPC and PLPC. We test our algorithms on problems having 32 variables. The tightness of interval-interval constraints is 7 relations allowed out of 13, namely the tightness $\gamma = 7/13$; for point-interval constraints the tightness $\beta = 4/5$; and for point-point constraints the tightness is $\alpha = 0.45$.

The tradeoff between efficiency and effectiveness is presented in Table 3 and is plotted in Figure 18. Each table entry and data point represents the average of 200 instances. The columns of Table 3 labeled "Acc $< alg >$" specify the accuracy of algorithm $< alg >$ relative to LPC, namely the fraction of cases in which algorithm $< alg >$ detected inconsistency given that LPC did. The columns labeled "# Op $< alg >$" describe the number of revision operations made by algorithm $< alg >$. The basic revision operation of PC is $C_{ij} \leftarrow C_{ij} \cap (C_{ik} \odot C_{kj})$. The basic revision operation of LPC is $C_{ij} \leftarrow C_{ij} \triangleleft (C_{ik} \odot C_{kj})$, while the basic operation for ULT is described in Definition 3. Measuring the number of revision operations is machine and implementation independent, unlike execution time.

For networks with only point variables, having about 200 constraints, ULT detected 15% of the inconsistencies that LPC detected, while DLPC and PLPC detected 25% and 95% inconsistencies respectively. For the same benchmark, the execution time of ULT, DLPC, PLPC, LPC was 0.162, 0.259, 0.533, 0.623 seconds respectively. The general trends in Table 3 indicate that (1) ULT is clearly the most efficient algorithm and (2) PLPC is almost as effective as LPC in detecting inconsistencies.

Based on the results in Table 3 it is difficult to select a clear winner. We speculate that in applications where queries involve a small subset of the variables and efficiency is crucial (e.g. real-time applications, large databases), ULT will be preferable to LPC and its variants. However, on our benchmarks, LPC is by far superior to ULT. Based on experiments made so far, we cautiously conclude that PLPC seems to show the

# of Consts	*Acc of PLPC*	*Acc of DLPC*	*Acc of ULT*	*# Op. LPC*	*# Op. PLPC*	*# Op. DLPC*	*Time LPC*	*Time PLPC*	*Time DLPC*	*Time ULT*
32 *vars*, 100% *interval variables* (*pure qualitative*), 200 *reps.*										
250	100%	100%	100%	17K	13K	11K	0.621	0.467	0.417	0.621
300	100%	98%	100%	20K	17K	15K	0.748	0.632	0.551	0.748
350	100%	92%	100%	25K	22K	19K	0.886	0.807	0.689	0.886
400	100%	79%	100%	28K	27K	23K	1.001	0.970	0.807	1.001
450	100%	71%	100%	30K	30K	26K	1.056	1.056	0.907	1.056
496	100%	73%	100%	28K	28K	25K	0.971	0.971	0.885	0.971
32 *vars*, 50% *interval variables* (*mixed*), 200 *reps.*										
150	100%	100%	100%	13K	6K	5K	0.210	0.121	0.082	0.163
200	99%	98%	97%	18K	11K	8K	0.283	0.200	0.135	0.174
250	98%	93%	95%	23K	17K	11K	0.374	0.306	0.199	0.308
300	96%	63%	65%	26K	22K	15K	0.456	0.406	0.266	0.422
350	98%	32%	89%	27K	25K	20K	0.460	0.440	0.325	0.426
400	100%	46%	98%	24K	23K	20K	0.406	0.402	0.347	0.385
450	100%	86%	100%	20K	20K	19K	0.400	0.400	0.343	0.379
496	100%	100%	100%	16K	16K	16K	0.359	0.353	0.294	0.331
32 *vars*, 100% *point variables* (*pure quantitative*), 200 *reps.*										
150	98%	92%	90%	25K	12K	5K	0.546	0.400	0.165	0.132
200	99%	25%	15%	27K	17K	8K	0.623	0.533	0.259	0.162
250	100%	70%	45%	14K	11K	10K	0.380	0.350	0.315	0.181
300	100%	99%	77%	9K	8K	8K	0.287	0.275	0.270	0.164
350	100%	100%	94%	7K	7K	7K	0.244	0.241	0.235	0.126
400	100%	100%	100%	6K	6K	6K	0.211	0.212	0.204	0.105

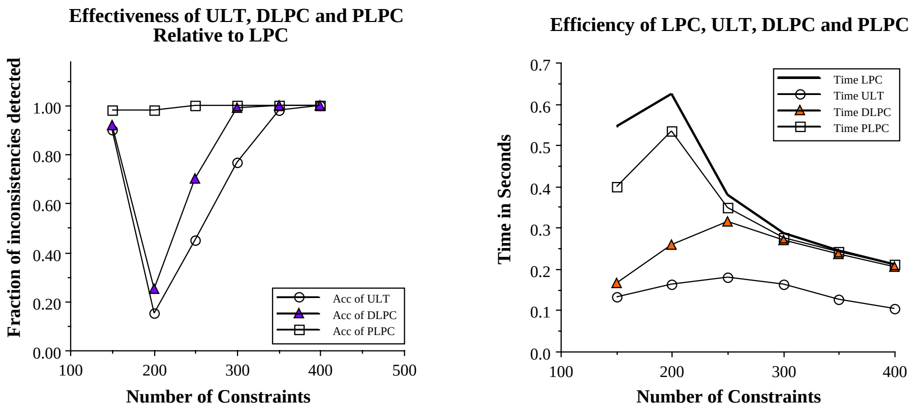
Table 3: Effectiveness and efficiency of LPC, DLPC, PLPC and ULT.

best overall efficiency-effectiveness tradeoff.

3.8.2 Backtracking

To improve backtrack search, our polynomial approximation algorithms can be used in three ways: (1) in preprocessing to reduce the number of disjuncts before initiating search, (2) to perform forward checking (within backtracking) for reduction of fragmentation and early detection of dead-ends, and (3) as an advice generator to determine the order of constraint labelings. For simplicity of exposition, we report results of experiments in which the same constraint propagation algorithm is used for preprocessing, forward-checking and dynamic variable variable ordering.

In selecting our benchmark problems, we drew on the recent observation that many classes of NP-complete problems have hard instances in a transition region



(a) (b)

Figure 18: Effectiveness and Efficiency of LPC, ULT, DLPC and PLPC (from Table 3).

[79, 20]. We therefore identified generation patterns that enable generating problems in the transition region and report the results obtained on those problems. Section 8.2.1 provides results on quantitative TCSPs and Section 8.2.2 provides results on qualitative networks.

Quantitative TCSPs

In general, constraint propagation algorithms are used as a preprocessing phase before backtracking in order to reduce the number of dead-ends encountered during search. When preprocessing with PC, problems become even harder to solve due to increased fragmentation. In contrast, preprocessing with ULT results in problems on which even naive backtracking is manageable (for small problems). This can be explained from the search space reduction argument mentioned at the end of Section 4.

We compare three backtrack search algorithms: "*Old-Backtrack+ULT*" which uses ULT as a preprocessing phase with no forward checking and static ordering; "*ULT-Backtrack+ULT*" and "*LPC-Backtrack+LPC*" which use ULT and LPC respectively for preprocessing, forward-checking and dynamic variable ordering.

The experiments reported in Figure 19 were conducted with networks of 10-20 variables, complete constraint graphs and 3 intervals in each constraint. Each point represents 500 runs. The region in which about half of the problems are satisfiable, is called the *transition region* [79, 20]. In Figures 19a and 19b we observe a phase-transition when varying the size of the network, while in Figures 19c and 19d we observe a similar phenomenon when varying the tightness of the constraints.

The experiments reported in Figure 20 were conducted with networks having 12 variables, 66 constraints (i.e. complete constraint graphs) and 3 intervals in each constraint. Each point represents 500 runs. ULT and LPC pruned dead-ends and improved search efficiency on our benchmarks by orders of magnitude. Specifically, averaged over 500 instances in the transition region (per point), Old-Backtrack+ULT is about 1000 times slower than ULT-Backtrack+ULT, which is about 1000 times slower than LPC-Backtrack+LPC. The latter encounters about 20 dead-ends on the peak (worst performance). As we depart from the transition region the execution times become smaller and the improvements are less significant.

Qualitative TCSPs

Here we present results obtained with backtracking on qualitative TCSPs. We show that (1) a transition region exists for qualitative networks and (2) for problems within this region PC [5] is completely ineffective. The backtracking algorithm is the algorithm used by Ladkin and Reinefeld [63]. In their implementation, they avoid enforcing path-consistency on any subnetwork that is already labeled (since it is already consistent).

The experiments reported in Figure 21 were conducted with networks having 12 variables, 66 constraints, and each point is averaged over 100 instances. We change the tightness of the constraints by changing γ. We measure the number of dead-ends (Figure 21a) and the fraction of cases in which enforcing path-consistency correctly decides consistency (Figure 21b).

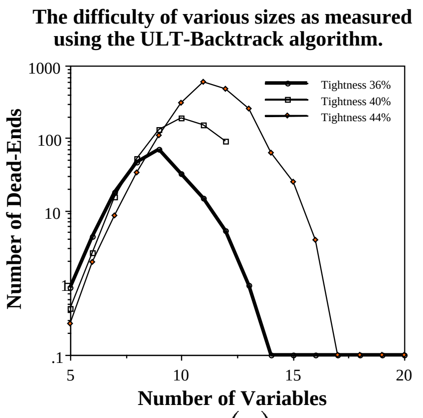
Figure 21a shows that qualitative networks exhibit a phase transition at $\gamma = 8/13$. The only difference between the experiments reported in this section and those reported in [63] is that the latter used a fixed $\gamma = 0.5$, namely in about half of the cases, six relations out of 13 interval relations were allowed and the other half, seven were allowed.

Our results agree with those reported in [63] in that for $\gamma = 0.5$ most of the generated problems were inconsistent. However, we see that for $\gamma = 9/13$, all the problems generated were consistent. For $\gamma = 6/13$, the problems were about two orders of magnitude easier than those at the peak (Figure 21a) because, in most of the cases, PC detected inconsistency before invoking backtracking search (Figure 21b).

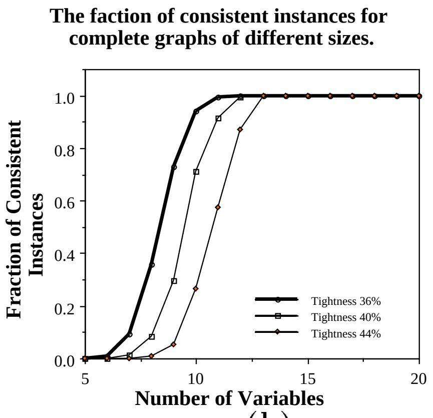
3.9 Conclusion

Temporal Constraint Satisfaction Problems (TCSP) provide a formal framework for reasoning about temporal information, which is derived from the framework of classical Constraint Satisfaction Problems (CSP). As in classical CSPs, the central task of deciding consistency is known to be NP-complete. To cope with intractability it is common to use polynomial approximation algorithms which enforce path-consistency.

In this chapter we demonstrated that, in contrast to classical CSPs, enforcing path-consistency on quantitative TCSPs is exponential due to the fragmentation problem. We controlled fragmentation using two new polynomial approximation algorithms, Upper lower Tightening (ULT) and Loose Path-Consistency (LPC). When evaluating

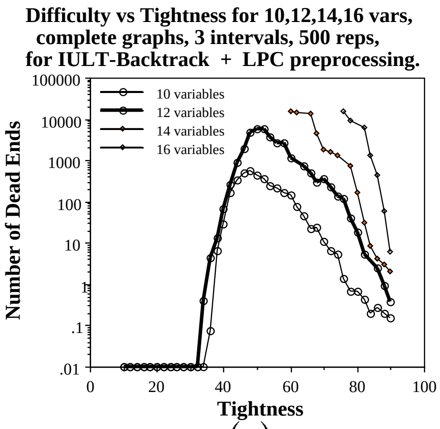


(a)

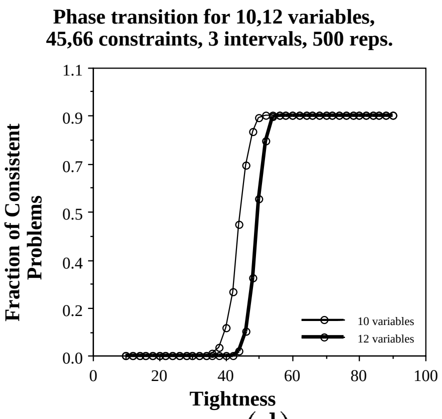


(b)

The difficulty as tightness is constant.



(c)



(d)

The difficulty as a function of tightness.

Figure 19

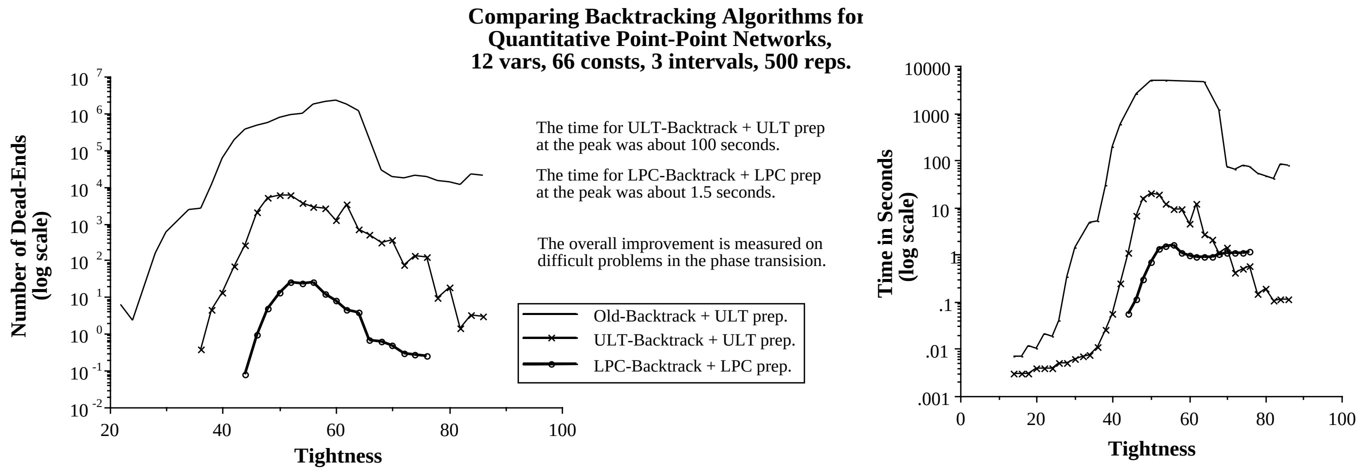
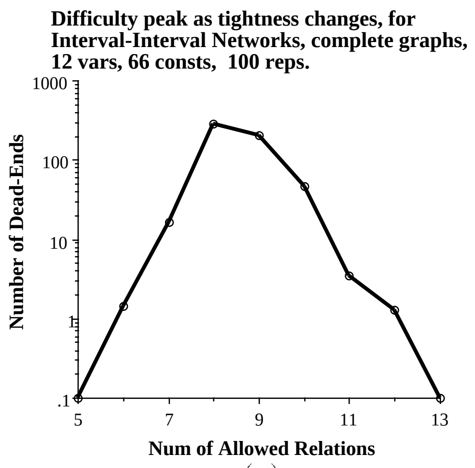
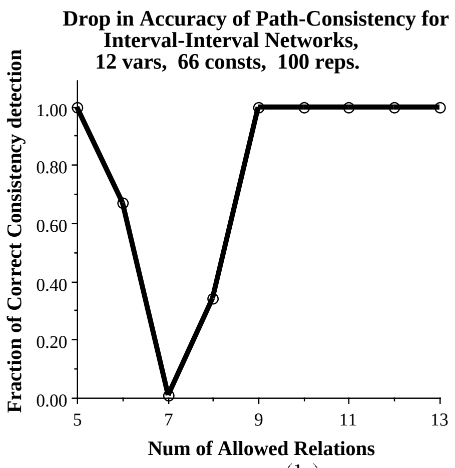


Figure 20: A comparison of three backtracking algorithms on quantitativeTCSPs.



(a)



(b)

Figure 21: The difficulty as a function of tightness for Qualitative Networks.

these algorithms, we addressed two questions empirically: (1) which of the algorithms presented is preferable for detecting consistency, and (2) how effective are they when incorporated within backtrack search.

To answer the first question, we measured the tradeoff between efficiency and effectiveness. Efficiency is measured by execution time while effectiveness is measured by counting the fraction of cases in which inconsistency was detected. Using some classes of randomly generated problems, we made two observations: (i) enforcing path-consistency may indeed be exponential in the number of intervals per constraint, and (2) ULT's execution time is almost constant in that number. Nevertheless, ULT is able to detect inconsistency in about 70% of the cases in which path-consistency does. The overall superior algorithm, LPC, is less efficient but more effective than ULT. It is also very effective relative to path-consistency.

To answer the second question, we applied the new algorithms in three ways: (1) in a preprocessing phase to reduce fragmentation before search, (2) as a forward checking algorithm for pruning the search and (3) as heuristic for dynamic variable ordering. We show that for relatively hard problems, which lie in the transition region [79, 20], incorporating ULT within backtracking search is preferred to incorporating path-consistency. Algorithm LPC is superior, in all three roles, as it improves the performance of backtracks search by several orders of magnitude.

Chapter 4

Temporal Constraint Logic Programming

4.1 Introduction

In this chapter we investigate temporal reasoning issues that arise when embedding the TCSP model within a more expressive temporal language. Most of the recent work on representing and reasoning about change is focused on logic based formalisms. These formalisms typically represent change as a transition from one state of the world to another. The language design we propose has the following properties:

- Syntactically, time is mentioned explicitly, either by including it as an argument or by tokenizing it. Consider the non-temporal statement "John visited Europe" which can be represented by the proposition `Visited(John,Europe)`, and consider representing the temporal statement "John visited Europe from Jan 12 to Jan 28".
  - When time is included as an argument, the above statement can be represented by adding two arguments to the predicate `Visited` resulting in the proposition `Visited(John,Europe, Jan12, Jan28)`.
  - When time is tokenized, by assigning every occurrence a unique ID, the occurrence `Visited(John,Europe)` is assigned the ID `k`, resulting in the proposition `Visited(John,Europe, k)`. The time of this occurrence can be specified by the temporal constraints `begin(k)=Jan12` and `end(k)=Jan28`.
- Semantically, we use the classic interpretation in which a formula is either *true* or *false*. The temporal semantics is augmenting the classic semantics with constraints ensuring that holding over an interval implies holding at all the time points in it and holding over its subintervals.
- The inference engine is based on classical resolution, with a few extensions that are required for implementing the temporal semantics, as explained in Chapter

5.

This chapter, as well as chapters 5,6, focuses on the design of formal temporal languages having clear syntax, well defined semantics and accompanied by efficient inference engine. We propose two languages based on combining Datalog with Temporal Constraint Satisfaction Problems (TCSP), called *TCSP-Datalog* and *Token-Datalog.*

One issue we address is called the *incompatibility* problem [89, 84]. When two incompatible propositions hold, they must hold on disjoint intervals. To illustrate the issues involved, we present three example domains: (i) formalizing medical treatment guidelines, (ii) verification of electronic circuits and (iii) warehouse database management.

Example 16: Consider medical regulation regarding the administration of Digoxin and potassium supplements. Digoxin is a Digitalis, which is a well known medication for heart failure. Potassium supplements are often used when patients who have heart failure are given diuretic drugs to reduce their fluid load. These two drugs must not be taken together, since potassium supplements render the Digoxin a very dangerous poison that can cause arrhytmia of the heart. However, taking them in sequence with sufficient time in between is OK, and is often done for the same patients, since they are often using the diuretic drugs against the same heart failure (e.g. Lasix).

Next we demonstrate how incompatibility is manifested in two application domains: electronic circuit verification and warehouse database management.

Example 17: Consider an electronic circuit processing n signals and assume the circuit can be described by a Finite State Automaton (FSA) with 2^n states, where n is the number of bits required to represent each individual state. Let `a,b` be two signals of the circuit and suppose that the circuit must satisfy the constraint `or(a,b)` at all times, namely at least one of `a,b` is *true* at all times. An approach based on temporal constraints defines a set of intervals S_1 during which `a` is known to be *false* and a set of intervals S_2 during which `b` is known to be *false*. The constraint `or(a,b)` is satisfied if and only if all the intervals in S_1 are pairwise disjoint from all intervals in S_2. This representation of the constraint `or(a,b)` does not require knowing the values of signals `a,b` at all time points. Moreover, it should be independent of the resolution of time representation and should be good for continuous time as well.

Example 18: Warehouse management systems are typically implemented using Relational Databases in which some of the attributes are temporal. Consider the following relation:

Part #	Description	Shelf	Start	End
10527	Connector A-B	3A	Jan 3	Feb 3
10527	Connector A-B	4A	Jan 17	Feb 15

This relation is inconsistent because part # 10527 cannot be simultaneously on shelves 3A and 4A. This inconsistency may result from a typing error or other sources. One way to handle such mistakes is to avoid specifying exact data. We would like to say that "we do not know when the connector was moved form shelf 3A to shelf 4A". To formalize this approach, we introduce two variables X and Y were X is the time point at which the connector was moved from shelf 3A and Y is the time at which it was moved onto shelf 4A.

Part #	Description	Shelf	Start	End
10527	Connector A-B	3A	Jan 3	X
10527	Connector A-B	4A	Y	Feb 15

Here, the temporal constraint induced by incompatibility is that $X \leq Y$, which means that the connector was taken off shelf 3A before it was put on shelf 4A. This constraint may be explicitly given in the input or can be deduced by other constraints. Here, consistency maintenance means (i) computing the implicit constraint $X \leq Y$ and (ii) ensuring that the constraint is always satisfied, no matter how X and Y change.

Other related issues we address in our language are *Homogeneity* and *Concatenability*. Homogeneity of a fluent holding means that when a fluent is true during an interval then it must be true during any subinterval and point within the interval [6, 36]. Concatenability of a fluent holding means that when a fluent is true on two consecutive time intervals it must be true on the time interval obtained by concatenating them. These issues are addressed by providing semantic tools within the language for describing axioms that ensure the intended inferences.

To illustrate the homogeneity issue, consider the following two statements having two distinct meanings: "John was alive between 1764 and 1821" and "John was working from 1784 to 1805". In the first statement the intended semantics implies that John was alive at every year between 1764 and 1821. In the second statement the intended semantics is that John might have had some vacations. Thus, there were some time intervals during which John was not actually working. In this work we will implement the first statement only.

To illustrate the concatenability problem, consider the statement "The power was on from 8:00am to 12:00pm", and "The power was on from 12:00pm to 6:00pm". Together, these statements implies that the power was on from 8:00am until 6:00pm. Now, consider the statement "Mary was pregnant from July to April, and she was also pregnant from "April to November". Do these statements imply that Mary was pregnant for 18 months? Is it possible to infer that Mary had two pregnancies?

4.1.1 Language Design Approach

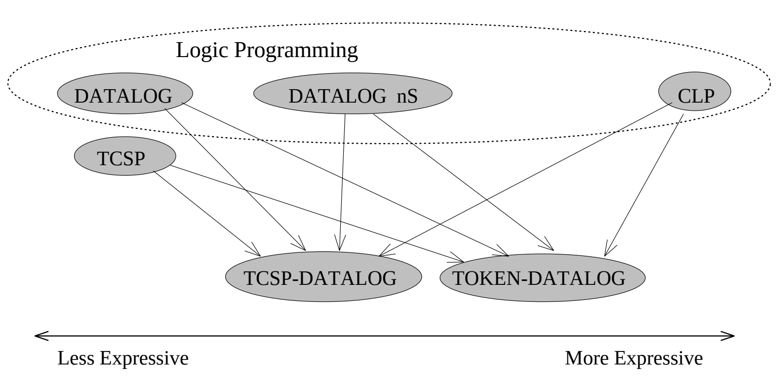


Figure 4.1: The language design approach.

Our goal is to design languages that equip a common-sense reasoner with the ability to express and infer desired temporal relationship in general (as illustrated in examples 1-3). Our starting point is TCSP on the one hand, and Datalog on the other hand. We also borrow techniques from Allen's temporal logic [6], Datalog$_{nS}$ [15, 16], first order Logic Programming (LP) and Constraint Logic Programming (CLP) [51] (see Figure 4.1). The properties borrowed from each of these languages are:

- LP is a general form of deductive databases. The techniques for deriving the fixed point semantics and the inference engine of LP are incorporated in our languages.
- Datalog is a fragment of LP which is computationally advantageous because its inference is polynomial. We use Datalog as the basis for the syntax and semantics, and extend it to accommodate temporal variables and temporal constraints.
- CLP is a general framework for embedding constraints into LP. Since we address similar issues, CLP is an important language to compare and contrast our approach with.
- Allen's temporal logic is based on the `Holds` predicate. The atom `Holds(P, I)` evaluates to *true* whenever the proposition `P` evaluates to *true* at the interval `I`. There are numerous types of propositions characterized by different holding properties. For example, proposition describing attributes of objects (e.g. color) are homogeneous, namely they hold at every point within the interval `I`. Our languages are designed with homogeneous propositions only.

- Datalog_{nS} is an extension of Datalog that allows performing a limited temporal reasoning over time points. This language is finite representable and decidable. This property is carried over when we introduced TCSPs into Datalog.

We present two extensions of Datalog that accommodate temporal constraints. The first, called TCSP-Datalog, uses temporal arguments to qualify propositions with time. The second, called Token-Datalog, uses token arguments as the temporal qualification method.

To illustrate the syntactic difference between TCSP-Datalog and Token-Datalog, consider representing the statement "John was in LA from Jan 15 to Feb 15". When using temporal arguments, this statement can be described with the proposition `Location(John, LA, Jan15, Feb15)`. Using token arguments, this statement can be described with the conjunction `Location(John, LA, k), begin(k)=Jan15, end(k)=Feb15`.

4.2 Temporal Qualification Methods

To specify the times at which logical statements are valid there is a need to incorporate time into logic using some temporal qualification method. Non temporal statements, such as "John is in LA", have no temporal qualification, namely the time during which they are valid is not described. Temporal statements, such as "John was in L.A. from Jan 1 until Feb 1", specify the time during which the statement is valid. There are three main temporal qualification methods. To illustrate each method and describe its properties, we consider representing the statement "John was located in L.A. before he moved to (i.e. located in) N.Y.". We will use the predicate "Location(P,X)" which evaluates to *true* iff person 'P' is at location 'X'.

We first illustrate the use of temporal reification [97], in which the above sample statement is described by the conjunction:

$$\texttt{True(Location(John, LA), } t_1\texttt{).} \quad \texttt{True(Location(John, NY), } t_2\texttt{).} \quad t_1 < t_2.$$

With this method, the predicate `Location(P,X)` is converted into a function and becomes an argument of the `True` predicate. This is computationally undesirable because introducing numerous new functions significantly increases the complexity of making inferences.

The next temporal qualification method is temporal arguments. In this method, the above sample statement is described by the conjunction:

$$\texttt{Location(John, LA, } t_1\texttt{).} \quad \texttt{Location(John, NY, } t_2\texttt{).} \quad t_1 < t_2.$$

The third temporal qualification method is token arguments [21, 107]. In this method, every proposition (e.g. Location(John. LA)) is associated with a unique ID, or a *token constant*. Every token constant `k` is associated with a time entity (e.g. point or interval) specifying the time during the statement whose ID is `k`. With this method, the above sample statement can be described by:

```
Location(John, LA, k1), Location(John, NY, k2), time(k1)<time(k2).
```

where `time(k)` is the time associated with the token `k` and k_1,k_2 are the unique ID of the propositions `Location(John, LA)` and `Location(John, LA)`. Notice that here, the propositions are expressed using atoms that are disjoint from those describing the temporal qualification. Because k_1,k_2 are the unique proposition IDs, the constraint `time(k1)<time(k2)` has the meaning "Location(John, LA) occurred before Location(John, NY)".

To further illustrate the token qualification method, consider representing the statement "John was located in LA from t_1 to t_2". Using tokens, this statement is described by the conjunction

```
Location(John, LA, k),  begin(k)=t1,  end(k)=t2.
```

where `k` is a token term, `Location(John, LA, k)` is the proposition `Location(John, LA)` qualified with `k`, `begin(k)`, `end(k)` are temporal variables and `begin(k)=`t_1, `end(k)=`t_2 are temporal constraints.

The tradeoffs between temporal arguments and temporal tokens are as follows:

1. Temporal arguments are simpler than temporal tokens. There is no need to introduce the new sort of tokens, and there are less semantical complications due to the exclusion of functions.
2. Temporal tokens are more expressive because they allow describing periodic occurrences. They also provide a better separation between the temporal and non-temporal components of sentences.

These tradeoffs are explained in the next chapter, section 5.4, where TCSP-Datalog and Token-Datalog are compared.

4.3 Temporal Incidence and other Technical Issues

Next we present a list of issues that we think should be addressed by any temporal reasoning language.

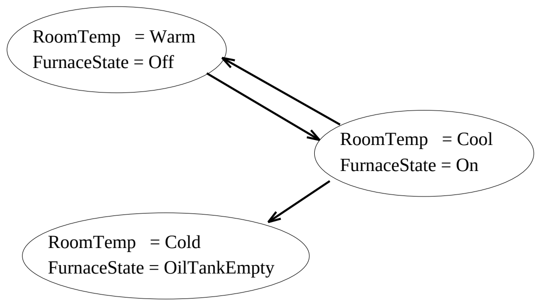


Figure 4.2: Maintaining the room temperature.

4.3.1 Embedding Time

Temporal reasoning involves reasoning about the change of some state of affairs, typically described by a set of variables whose values change with time. Such variables are called fluents.

Consider, for example, formalizing a system that maintains a constant room temperature using a furnace. We model the state of the room with a fluent `RoomTemperature` which takes the values {Warm, Cold}. We model the state of the furnace with the fluent `FurnaceState` which takes three values: {On, Off, OutOfOil}. As illustrated in Figure 4.2, the system comprising of room and furnace alternates between the states `(RoomTemp=Warm, FurnaceState=Off)` and `(RoomTemp=Cold, FurnaceState=On)`. After some time, the furnace runs out of gas (or oil), and we arrive at the state `(RoomTemp=Cold, FurnaceState=OutOfOil)`.

The goal of temporal reasoning is to reason about changes such as "the furnace runs out of gas". A temporal reasoning language must enable expressing why, how and when does the furnace change state. To achieve this goal, we present a list of specific issues through a variety of examples. For simplicity, we restrict the discussion to fluents taking the values *true* and *false* only (instead of general multi-valued fluents).

A. Instantaneous Events It is sometimes argued that it is possible to perform temporal reasoning using time intervals alone [6, 50]. However, modeling dynamic systems often involves some events that cannot be qualified by an interval. Some examples are "turn off the furnace" or "turn off the light", "shoot the gun", "start moving" and "sign a contract". Although these events may have an infinitely small yet non-zero interval length, it is often advantaguous to model the events as instantaneous. One reason is that the temporal resolution requirements for representing very short intervals may be too high, namely the duration of the interval may be smaller than the smallest unit of time we can represent with state-of-the-art data structures implemented on state-of-the-art computers.

B. Instantaneous Fluent Holding In many real life situations, it is crucial to know the truth value of a fluent at a certain time point, *at which an instantaneous event occurred*. For example, consider modeling a car accident in which John was hit, and assume the event `TheCarHitJohn` is modeled as instantaneous. Knowing the truth value of the fluent `TheLightIsRed` at the time point the event `TheCarHitJohn` occurred is crucial.

We also consider modeling *continuous change*, that requires representing fluents that hold at isolated time points. Consider, for example, the *speed* of a ball tossed upwards in the *Tossed Ball Scenario* (TBS) (see Figure 4.3).

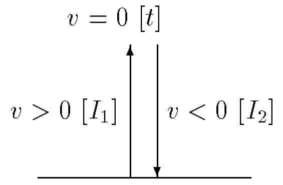


Figure 4.3: The Tossed Ball Scenario (TBS).

The ball moves up during an interval I_1 and down during an interval I_2. While the speed of the ball is non-zero during both I_1 and I_2, there must be an isolated time point `t` "in between" the two intervals where the speed of the ball is zero.

C. Non-Instantaneous Holding of Fluents Formalizing the properties of temporal incidence for non-instantaneous fluents is not a simple issue. There are two major classes of properties: (i) the holding over intervals and the holding at points inside these intervals and (ii) the holding of incompatible fluents.

Important properties of non-instantaneous holding statements that we incorporate in our languages are:

- *Homogeneity*: If a fluent is true during an interval then it must be true during any subinterval and point within the interval [6, 36].
- *Concatenability*: If a fluent is true on two consecutive time intervals it must be true on the time interval obtained by concatenating them [1].
- *Incompatibility*: If a fluent F is true during some interval, other fluents which together imply the negation of F cannot be all true during this interval.

To address these issues, we incorporate in our languages the following:

[1]There are different views for the meaning of *consecutive*.

- *Explicit Incompatibility*: Fluents come in pairs $F, \neg F$, where $\neg F$ is the name of a fluents which is the negation of F. The incompatibility between F and $\neg F$ is explicitly stated by rules augmenting the given theory expressed by the input set of rules.
- *Non-holding*: If $\neg F$ holds during a time interval I then there is no time point or interval within I at which F holds.
- *Disjointness*: If both F and $\neg F$ hold during the intervals I_1 and I_2 respectively, then I_1 and I_2 must be disjoint, namely the intersection $I_1 \cap I_2$ is empty.

The Dividing Instant Problem (DIP) Assume that we choose to represent both instantaneous and non-instantaneous holding of fluents. Consider using the fluent `TheLightIsOn` to model the state of the room. Let `TheLightIsOn` evaluate to *true* during I_1 and evaluate to false during I_2, where the end of I_1 equals the beginning of I_2. Let us use the event `SwitchOff` to model switching off the light. The problem at hand is deciding what is the truth value of the fluent `TheLightIsOn` at the time point that the event `SwitchOff` occurred (see figure 4.4) [48, 105, 6, 36].

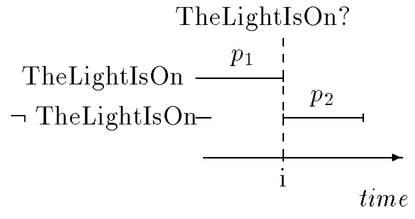


Figure 4.4: The Dividing Instant Problem.

This is a problem of maintaining logical consistency: If intervals are closed then TheLightIsOn and $\neg$TheLightIsOn are both *true* when `SwitchOff` occurred, which is inconsistent. If they are open we might have a "truth gap". The other two options are open/closed and closed/open intervals which are perceived to be artificial [6, 36].

4.3.2 Embedding Constraints

The primary benefit of embedding constraints within a logic programming framework, as is evident from reports in the CLP literature [51], is to enhance the efficiency of inference. A necessary hurdle to overcome before this can be achieved is to decide on the syntax and semantics of such an embedding. In the following we present some central issues.

Constraint Atoms in the Bodies of Rules

It is neccessary to determine when a constraint is entailed by a program. Consider the following program

$$\Psi \equiv \begin{array}{l}\mathtt{C_1.}\quad \mathtt{C_2.}\\ \mathtt{A\ :-\ C_3.}\end{array} \qquad \text{where} \quad \begin{array}{lcl} \mathtt{C_1} & \equiv & \mathtt{X_1\text{-}X_2} \in \mathtt{[2,5]} \\ \mathtt{C_2} & \equiv & \mathtt{X_2\text{-}X_3} \in \mathtt{[7,9]} \\ \mathtt{C_3} & \equiv & \mathtt{X_1\text{-}X_3} \in \mathtt{[5,20]} \end{array}$$

We need to decide when Ψ entails A. We design the semantics of our language such that Ψ entails `A` because C_3 is entailed by the conjunction of C_1 and C_2, regardless of the values of X_1, X_2 or X_3.

Constraint Atoms in the Heads of Rules

Consider describing the statement "*if both events E_1,E_2 occurred, E_2 occurred less than 120 time units after E_1*". We consider two ways of formalizing this statement:

1. We could use a variable `E` whose value is the time point at which the event occurred, and write the following short program:

   $\mathtt{E_2\text{-}E_1}\in\mathtt{[0,120]\ :-\ Occur(E_2),\ Occur(E_1).}$

   The bottom-up semantics of this rule is unclear because to perform bottom-up evaluation we need to assign values to both $\mathtt{E_1}$ and $\mathtt{E_2}$, namely to assign the exact times at which $\mathtt{E_1}$, $\mathtt{E_2}$ occur. For example, let us assume that $\mathtt{Occur(E_2)}$ and $\mathtt{Occur(E_1)}$ are *true* and we assigned $\mathtt{E_1} = 0$, $\mathtt{E_2} = 200$. Since $\mathtt{Occur(E_2)}$ and $\mathtt{Occur(E_1)}$ are *true*, the body of the rule evaluates to *true*, and we need to add $\mathtt{E_2\text{-}E_1} \in \mathtt{[0,120]}$ as a constraint fact. The result is inconsistency, because $\mathtt{E_2\text{-}E_1} = 200$ implying $\mathtt{E_2\text{-}E_1} \notin \mathtt{[0,120]}$,

   To avoid this problem, many constraint-based logic programming languages, *including CLP*, do not allow constraint atoms in the head. As a result, the statement "*if both events E_1,E_2 occurred, E_2 occurred less than 120 time units after E_1*" cannot be described in CLP.

2. We could map the non-constraint variables $\mathtt{E_1}$,$\mathtt{E_2}$ to constraint variables, denoted $\mathtt{time(E_1)}$,$\mathtt{time(E_2)}$, and write the following program:

   $\mathtt{time(E_2)\text{-}time(E_1)}\in\mathtt{[0,120]\ :-\ Occur(E_2),\ Occur(E_1).}$

   With this approach, the constraint variables $\mathtt{time(E_1)}$,$\mathtt{time(E_2)}$ remain non-ground (i.e. not fixed) even when both $\mathtt{E_1}$,$\mathtt{E_2}$ are ground. This enables fixing both $\mathtt{E_1}$,$\mathtt{E_2}$ without fixing neither $\mathtt{time(E_1)}$ nor $\mathtt{time(E_2)}$ and without introducing inconsistencies. Thus, bottom-up evaluation is well defined.

TCSP-Datalog incorporates the first option, allows constraint atoms in the heads of rules yet still has a well defined semantics. This is achieved by using a declarative model-based semantics in which interpretations consist of two partitions: constraint and non-constraint facts. The details are described below.

Token-Datalog incorporates the second option. It provides a much cleaner solution to the problem. The penalty is the increased complexity of making inferences.

Representing Constraint Facts

The representation of constraint facts depends on whether constraint atoms can appear in the heads of rules. When constraint atoms are not allowed to appear in the heads of rules (e.g. CLP), then facts are of the form `A:-C` where `A` is a non-constraint atom and `C` is a constraint atom. In our design, when constraint atoms are allowed to appear in the heads of rules, then constraint facts are empty-body rules containing a single constraint atom. TCSP-Datalog incorporates the first option, while Token-Datalog incorporates the second option.

Addressing these issues In our languages we use both time points and time interval as the basic temporal entities. This enables us to represent both instantaneous events and instantaneous holding by associating them with isolated time points. To address the DIP, the holding intervals are always open. To describe the holding over a closed interval, we use three holds statements: one non-instantaneous holding over the open interval and two instantaneous holding for the end-points of the interval. Holding over left-sided and right-sided open intervals is described with one non-instantaneous holds over the open interval and one instantaneous holds at the closed end-point.

By including the instantiation of constraint variables (i.e. value assignments) in the interpretation[2], we address the semantic problems associated with introducing constraint atoms into logic-programming framework. This enables describing the *negation* of a constraint atom using the complement of this constraint, and *allows introducing constraint atoms in the heads or rules as well as in their bodies, without restricting them to be positive or negative.*

4.4 The TCSP-Datalog Language

TCSP-Datalog extends the TCSP model to allow performing temporal reasoning tasks. We combine TCSPs with Datalog and analyze the emerging issues. The syntax of TCSP-Datalog is defined by the sorts (i.e. variable types), the terms, the predicates and rules. The semantics is given by defining interpretations and the satisfaction relation for deciding when an interpretation is a model. Using techniques from general LP we show that the intersection of all models is the unique minimal

[2]In some CLP languages this cannot be done.

model.

4.4.1 Syntax

Sorts are the types of arguments that predicates in the language can take. The formal name for these arguments is *terms*. There are two sorts of terms: data and temporal terms.

Terms are symbols that appear as arguments of predicates.

Data Terms are either data variables or data constants (i.e. no functions, as in Datalog).

Temporal Terms are either time points or intervals. A constant time point `t` and a constant interval `[a,b]` are temporal terms. If `I` is an interval temporal term then its end points are given by t_1, t_2 where $t_1\{\texttt{starts}\}I$, $t_2\{\texttt{ends}\}I$ are atoms in the language. No functions are allows.

Predicates are of three types: constraint, temporal and neutral predicates. Neutral predicates take only data terms. Constraint predicates, describing TCSP constraints, take exactly two temporal terms (i.e. binary constraints) and cannot take data terms. Temporal predicates take exactly one temporal term and may take data terms without restrictions.

Note that we restrict constraint predicates to take two temporal arguments and temporal predicates to take a single temporal argument for reasons of simplicity. Analyzing the more general cases is outside the scope of this work.

Atoms are built from predicates as usual. For the temporal constraints, we use $a=b$, $a\neq b$, $a<b$, $a\leq b$, $x-y\in[a,b]$ with the usual meaning as a short hand for `Eq(a,b)`, `Neq(a,b)`, `Less(a,b)`, `LessEq(a,b)`, `Difference(x,y,a,b)` respectively.

Example 19: The statement *"John was in LA between Jan 1 and Feb 12"* can be represented by the conjunction of the facts

$$\texttt{Location(John,LA, I), Jan1}\{\texttt{starts}\}\texttt{I, Fab12}\{\texttt{ends}\}\texttt{I.}$$

where `Location(X,Y,I)` is a predicate which evaluates to *true* iff person X was at location Y at every time point during the interval I, the symbols `John, LA` are data terms and `I` is an interval temporal term.

Similarly, the statement *"John was in LA for a duration of less than 10 days"* can be represented by the conjunction of the facts

`Location(John,LA, I), t`$_1$`{starts}I, t`$_2$`{ends}I. t`$_2$`-t`$_1$ $\leq$`10.`

Rules are of the form `H :- B`$_1$`,...,B`$_k$, where `H` and `B`$_i$ are arbitrary atoms. Rules must be *range restricted*, namely all data variables must appear in the body of some rule or equated to a variable in the body of some rule. Without this restriction, the fixed point semantics of TCSP-Datalog will not be well defined.

4.4.2 Predicate Buddies

Predicate buddies are introduced to enable both instantaneous and non-instantaneous holding. Galton proposed to address this technical problem using the `HoldsAt`,`HoldsOn` predicates [37] commonly used in temporal reification. `HoldsAt(F,t)` describes instantaneous holding and evaluates to *true* iff the fluent `F` evaluates to *true* at time *point* `t`. `HoldsOn(F,I)` describes non-instantaneous holding and evaluates to *true* iff the fluent `F` is *true* at every point in time within the *interval* `I`.

In TCSP-Datalog, we address this problem without using `HoldsAt` and `HoldsOn`. Atoms of the form `HoldsAt(F,t)` can be replaced by `F(t)` where `t` is a time point and the fluent name `F` is used as the predicate name. Both statements `HoldsAt(F,I)` and `HoldsOn(F,I)` should be written as `F(t)`,`F(I)` where `t` is a time point, `I` is an *open* time interval[3]. However, in logic, it is not possible to assign a predicate to two distinct signatures.

To overcome this problem, temporal predicates in TCSP-Datalog come in pairs. The predicate `P`$_{AT}$`(a`$_1$`,...,a`$_n$`,t)`, where `t` is a time point, has a pair, or *buddy* predicate `P`$_{ON}$`(a`$_1$`,...,a`$_n$`,I)`, where `I` is a time interval. We use the shortcut `P` whenever it is clear from the type of the temporal argument whether the meaning is instantaneous `P`$_{AT}$ or non-instantaneous `P`$_{ON}$.

4.4.3 Expressing Incompatibility

Incompatibility is usually implicit in the meaning the user gives to predicates. This meaning is usually too complicated to formalize due to syntactic restrictions as well as semantic problems.

Example 20: Consider expressing the fact that John cannot be in NY and LA simultaneously, unless NY and LA are two names of the same city. We can explicitly say that the atoms `Location(John, LA)` and `Location(John, NY)` are incompatible

[3]The interval is open to address the DIP. See section 4.4.6.

unless `LA=NY`. We could use a TCSP-Datalog rule of the form

I_1\{before,after,meets, met-by\}I_2 :-
    X$\neq$Y, Location$_{ON}$(John, X, I_1), Location$_{ON}$(John, Y, I_2).

Note that the above rule cannot be expressed in CLP languages because constraint atoms are not allowed in the heads of rules.

The limitation of the solution we presented for TCSP-Datalog is that we do not have a generic form for expressing incompatibility and we cannot infer incompatibility. Thus, we require that the user will enter a specific rule for every case incompatibility might arise.

In general, incompatibility can be expressed by rules of the form

$$\texttt{C :- } \texttt{A}_1\texttt{,A}_2\texttt{,...,A}_n$$

where $\texttt{A}_1\texttt{,A}_2\texttt{,...,A}_n$ are TCSP-Datalog atoms, $\texttt{C = C(t}_1\texttt{,t}_2\texttt{)}$ is a TCSP-Datalog constraint atoms expressing disjointness. The constraint C can be of three possible types:

1. If t_1, t_2 are two point temporal terms then the rule is of the form

   $$\texttt{t}_1 \neq \texttt{t}_2 \texttt{ :- A}_1\texttt{,A}_2\texttt{,...,A}_n$$

   where $\neq$ is a point-point constraint.

2. If $\texttt{t}_1$ is a time point and $\texttt{t}_2$ is a time interval then the rule is of the form

   $$\texttt{t}_1 \ \{\texttt{before,after}\} \ \texttt{t}_2 \texttt{ :- A}_1\texttt{,A}_2\texttt{,...,A}_n$$

   where $\{$`before,after`$\}$ is a point-interval constraint.

3. Otherwise, $\texttt{t}_1\texttt{,t}_2$ are both time intervals, then the rule is of the form

   $$\texttt{t}_1 \ \{\texttt{before,meets,met-by,after}\} \ \texttt{t}_2 \texttt{ :- A}_1\texttt{,A}_2\texttt{,...,A}_n$$

   where $\{$`before,after,meets,met-by`$\}$ is an interval-interval constraint.

Example 21: The above sample statement is represented by the TCSP-Datalog rule

```
I₁ { before,meets,met-by,after} I₂  :-
            X≠Y, Location(John, X, I₁), Location(John, Y, I₂)
```

where $\texttt{I}_1\texttt{,I}_2$ are interval variables and $\texttt{X,Y, I}_1\texttt{,I}_2$ are implicitly universally quantified (by the semantics of logic programming).

4.4.4 Standard Semantics

Once the syntax of the language is defined, its semantics can be either standard (i.e. non-temporal) or temporal. First, we introduce the classical definitions of semantics for non-temporal logic. This semantics, also called model based semantics, is described in terms of Herbrand interpretations and a satisfaction relation deciding which interpretations are models. For logic programs, a well known result states that the intersection of all models, also called the least Herbrand model, contains all the atoms that are entailed by the program [4, 53].

Definition 13: Given a TCSP-Datalog program Ψ, interpretations are defined in terms of the Herbrand universe of Ψ, denoted $\mathcal{U}_\Psi$ and the Herbrand base of Ψ, denoted $\mathcal{B}_\Psi$. A *ground term* is a term which does not contain any variables. The *Herbrand universe* of a program Ψ, denoted $\mathcal{U}_\Psi$, is the set of all possible ground terms that are built on top of constants in Ψ. The *Herbrand base* of Ψ, denoted $\mathcal{B}_\Psi$, is the set of all possible atoms that can be constructed with predicates from Ψ taking arguments from $\mathcal{U}_\Psi$. An interpretation $\mathcal{I}$ consists of the following:

1. A non-empty set $\mathcal{D}$ called the *domain*.
2. For each element in $\mathcal{U}_\Psi$, the assignment of an element in $\mathcal{D}$.
3. For each n-ary function in Ψ, an assignment from $\mathcal{D}^n$ to $\mathcal{D}$.
4. For each n-ary predicate in Ψ, an assignment from $\mathcal{D}^n$ to {*true,false*}.

Clearly, every interpretation is a subset of the Herbrand base.

Example 22: Consider representing the following guideline: "MedicineA is administered for the duration of 1-2 hours and MedicineB is administered for the duration of 2-3 hours. If both MedicineA and MedicineB need to be administered, then these events must be at least 7 hours and at most 9 hours apart." This statement can be expressed by the program Ψ =

```
t11{starts}I1.
t12{ends}I1.
t21{starts}I2.
t22{ends}I2.
t21-t12 ∈[7,9] :- AdministrateON( MedicineB, I2),
                  AdministrateON( MedicineA, I1).
t12- t11 ∈[1,2] :- AdministrateON( MedicineA, I1).
t22- t21 ∈[2,3] :- AdministrateON( MedicineB, I2).
```

The Herbrand universe $\mathcal{U}_\Psi$ is

$$\mathcal{U}_\Psi \ = \ \{MedicineA, MedicineB, I_1, I_2\} \cup \ \mathcal{Z}$$

where $MedicineA, MedicineB$ are data terms, I_1, I_2 are interval constraint terms and $\mathcal{Z}$[4] denotes the set of all integers, each of which is a point constraint term. One possible interpretation is as follows:

1. The domain $\mathcal{D} = \mathcal{U}_\Psi$.
2. Each element in $\mathcal{D} = \mathcal{U}_\Psi$ is mapped to itself.
3. There are two functions, `begin, end`, mapping interval constraint terms to point constraint terms at the end points of the intervals.
4. There are two predicates, the temporal predicate `Administrate` and the constraint predicate $\mathtt{t_2}$-$\mathtt{t_1} \in$`[a,b]`. The predicate `Administrate(X,I)` maps the ordered pairs `(MedicineA,[0,1])` and `(MedicineB,[5,6])` to *true* and all other pairs to *false*.

This sample interpretation is not a model because $\mathtt{begin(I_2)}$-$\mathtt{end(I_1)} \notin$`[7,9]`, and thus the unit clause in Ψ is not satisfied.

Models are defined in terms of substitutions and a satisfaction relation deciding which interpretations are models.

Definition 14: Given a TCSP-Datalog program Ψ, let $\mathcal{U}_\Psi$ be its Herbrand universe. A substitution θ is a set of ordered pairs $(v_1, u_1), \ldots, (v_i, u_i)$ where v_i is a variable occurring in Ψ and u_i is either a variable or any member of $\mathcal{U}_\Psi$. If all of u_i are constants, then θ is called a *ground substitution*. Let $\Psi\theta$ denote the result of applying θ on Ψ, namely substituting the variables v_i with u_i. In other words, if $\mathcal{L}_\Psi$ is the language defined over the constants in Ψ, then given an atom A, the atom $A\theta$ is in the language $\mathcal{L}_\Psi$ iff $A\theta \in B_\Psi$.

Given a ground substitution θ mapping all variables in Ψ to constants, the truth values of all ground atoms, which must be elements of B_Ψ, is well defined by condition 4 in definition 13. We say that a representation of an interpretation $\mathcal{I}$, alternative to the representation described in definition 13, is a subset of B_Ψ containing all and only those ground atoms that evaluate to *true* in $\mathcal{I}$. If the truth value assignment defined by $\mathcal{I}$, satisfies all clauses in $\Psi\theta$ (according to the standard meaning for connectives) for every possible ground substitution θ, then $\mathcal{I}$ is a model of Ψ.

Theorem 16: *[68] Let M_Ψ be the intersection of all models of Ψ. Then M_Ψ is a model and an atom A is in M_Ψ iff it is a logical consequence of Ψ.*

[4]The set $\mathcal{Z}$ is introduced automatically whenever all the numbers occurring in the constraints are integers.

4.4.5 Temporal Semantics

Temporal semantics is distinguished from standard (i.e. non-temporal) semantics in that predicates that take temporal arguments must satisfy some properties. In this chapter, we focus only on the requirement that holding over intervals implies holding over all sub-intervals and points within these intervals. In other words, $P(d_1, \ldots, d_n, I) \mapsto true$ for some interval I implies $P(d_1, \ldots, d_n, t) \mapsto true$ for every time point t within the interval I. This is the *homogeneous* property, which gives rise to the incompatibility issues described above.

Example 23: To continue with example 22, consider the atom `Administrate`$_{ON}$`( MedicineA, I`$_1$`)` and let `I`$_1$`=[0,2]`. According to the desired temporal semantics, if `Administrate`$_{ON}$`( MedicineA, [0,2])` evaluates to *true* then `Administrate`$_{ON}$`( MedicineA, [0,1])` must also evaluate to *true*. In contrast, according to the standard non-temporal semantics, the truth value of the latter is not constrained and can be either *true* or *false*. If, in addition, `MedicineA` and `MedicineB` cannot be given simultaneously, the user must add an appropriate incompatibility constraint. Because the holding over an interval implies the holding over point within the intervals and all subintervals, this incompatibility constraint implies that the intervals `I`$_1$ and `I`$_2$ are disjoint. Such incompatibility is not accounted for in the standard non-temporal semantics.

Next, we define the satisfaction relation of TCSP-Datalog, which modifies the standard satisfaction relation (described above) with the additional requirement that the non-instantaneous holding over intervals is homogeneous. We start by defining constraint entailment.

Definition 15: A set of constraints $\mathcal{C}$ entails a constraint C, denoted $\mathcal{C} \models C$, iff C specifies a pair of variables in $\mathcal{C}$ and C is satisfied by all solutions of $\mathcal{C}$, namely $\mathcal{C} \models C$ iff $\mathcal{C} \wedge \bar{C}$ is inconsistent where $\bar{C}$ is the complement of C.

Example 24: The constraint $C_1 = t_2 - t_1 \in [2,3]$ entails the constraint $C_2 = t_2 - t_1 \in [1,6]$ because the conjunction of C_1 with the complement $\bar{C_2} = t_2 - t_1 \in [-\infty, 0] \cup [7, \infty]$, is inconsistent.

The temporal semantics requires that, if a fact `P`$_{ON}$`( a`$_1$`,...,a`$_n$`,I)` is in a model M then this model must also contain numerous additional facts of the form `P`$_{ON}$`(a`$_1$`,...,a`$_n$`,I')`, `P`$_{AT}$`(a`$_1$`,...,a`$_n$`,t)`, for every interval `I'` is subsumed in `I` and for every time point `t` in `I`. The facts `P(a`$_1$`,...,a`$_n$`,t)` for every time point `t` in `I`.

TCSP-Datalog interpretations are standard (i.e. built from terms and facts in the usual way). The difference between the temporal and non-temporal semantics is the satisfaction relation.

Definition 16: An interpretation $\mathcal{I}$ is a model of Ψ under the *temporal semantics*, denoted $\mathcal{I} \models_{temporal} \Psi$, iff

1. $\mathcal{I}$ is a model of Ψ under the standard (i.e. non-temporal) semantics described above,
2. and, for every ground temporal atom $\texttt{A = P}_{ON}\texttt{(a}_1\texttt{,...,a}_n\texttt{, i)}$ in $\mathcal{I}$ where $\texttt{i}$ is an interval term and $\texttt{a}_1\texttt{,...,a}_n$ are data terms in $\mathcal{I}$, the following two conditions are satisfied:
   (a) if $A \in \mathcal{I}$ then
   $\forall\texttt{i', begin(i)}\leq\texttt{begin(i'), end(i')}\leq\texttt{end(i)} \longrightarrow \texttt{P}_{ON}\texttt{(a}_1\texttt{,...,a}_n\texttt{,i')}\in \mathcal{I}$ and $\forall\texttt{t, begin(i)}< t <\texttt{end(i)} \longrightarrow \texttt{P}_{AT}\texttt{(a}_1\texttt{,...,a}_n\texttt{, t)}\in \mathcal{I}$
   (b) if $A \notin \mathcal{I}$ then
   $\exists\texttt{i', begin(i)}\leq\texttt{begin(i'), end(i')}\leq\texttt{end(i)} \longrightarrow \texttt{P}_{ON}\texttt{(a}_1\texttt{,...,a}_n\texttt{, i')}\notin \mathcal{I}$ or $\exists\texttt{t, begin(i)}< t <\texttt{end(i)} \longrightarrow \texttt{P}_{AT}\texttt{(a}_1\texttt{,...,a}_n\texttt{, t)}\notin \mathcal{I}$

A ground atom A is entailed by a program Ψ iff A is in all models of Ψ.

The conditions 2a and 2b in Definition 16 are introduced to ensure that homogeneity holds. Condition 2a ensures that if the atom $\texttt{P}_{ON}\texttt{(a}_1\texttt{,...,a}_n\texttt{, i)}$ is in the set of facts, then for every time point `t` which is within the interval of `i`, the atom $\texttt{P}_{AT}\texttt{(a}_1\texttt{,...,a}_n\texttt{, t)}$ is in the set of facts. Similarly, condition 2b ensures that for every `i'` which is a subinterval of `i`, the atom $\texttt{P}_{ON}\texttt{(a}_1\texttt{,...,a}_n\texttt{, i')}$ is in the set of facts.

Example 25: Consider the statement "If Mary was in LA between Jan 1 and Feb 15 then John was with her", which can be represented by the program

$\texttt{Location}_{ON}\texttt{(Mary,LA, [Jan1,Feb20]).}$
$\texttt{Location}_{ON}\texttt{(John,LA, [Jan15,Feb15] :- Location}_{ON}\texttt{(Mary,LA, [Jan15,Feb15]}$

Consider two interpretations of this program:

1. $\texttt{Location}_{ON}\texttt{(Mary,LA, [Jan1,Feb20]).}$
2. $\texttt{Location}_{ON}\texttt{(Mary,LA,[Jan2,Feb15]), Location}_{ON}\texttt{(Mary,LA,[Jan3,Feb12]), ...}$
   $\texttt{Location}_{ON}\texttt{(John,LA,[Jan15,Feb15]), Location}_{ON}\texttt{(John,LA,[Jan16,Feb15]), ...}$

The second interpretation is a model under both standard and temporal semantics as well. The first interpretation is a model under the standard (non-temporal) semantics but it is not a model under the temporal semantics. Thus, every model which includes the fact `Location(Mary,LA, [Jan1,Feb20])` must also include the facts $\texttt{Location(Mary,LA, [t}_1\texttt{,t}_2\texttt{])}$ for every $\texttt{t}_1\texttt{,t}_2 \in\texttt{[1,20]}$ and $\texttt{t}_1 \leq\texttt{t}_2$. [5]

[5] Notice that there are no constraints in this interpretation because there are no constraints in the program.

Definition 17: The Buttom-Up fixed point semantics is defined as follows. Given an interpretation $\mathcal{I}$ of a program Ψ, let $\mathcal{C}$ be the set of constraint facts in $\mathcal{I}$ and let $\mathcal{F} = \mathcal{I} - \mathcal{C}$. We will use $\mathcal{I} =< \mathcal{F}, \mathcal{C} >$ as a shorthand for this relation. The operator $T_\Psi(\mathcal{I})$ is defined as

$$
\begin{aligned}
T_\Psi(\mathcal{I}) \;\; = \;\; & \texttt{the facts in } \mathcal{I} \cup \{ \texttt{H}_1\theta,\ldots,\texttt{H}_m\theta \mid \texttt{the rule r = H}_1,\ldots,\texttt{H}_m \texttt{ :- B}_1,\ldots,\texttt{B}_n \\
& \texttt{in } \Psi \texttt{ where } \mathcal{I} \models \texttt{B}_1\theta,\ldots,\texttt{B}_n\theta \texttt{ and the conjunction of the constraints} \\
& \texttt{in B}_1\theta,\ldots,\texttt{B}_n\theta\texttt{,H}_1\theta,\ldots,\texttt{H}_m\theta \texttt{ is satisfiable for the substitution } \theta \;\}.
\end{aligned}
$$

The successive iterations of $T_\Psi(\mathcal{I})$ are defined as follows:

$$
\begin{aligned}
T^1_\Psi &= \textit{the set of input facts} \\
T^{i+1}_\Psi &= T_\Psi(T^i_\Psi) \\
T^\omega_\Psi &= \textstyle\bigcup_{k\geq 0} T^i_\Psi \quad \equiv < \mathcal{F}^\omega_\Psi, \mathcal{C}^\omega_\Psi >
\end{aligned}
$$

A ground atom A is a logical consequence of Ψ iff $A \in T^\omega_\Psi$.

Theorem 17: *For every TCSP-Datalog program Ψ,*

1. *the intersection of all models is the unique minimal model, $M_\Psi^{temporal}$, under the temporal semantics;*
2. *An atom A is entailed by Ψ iff it is entailed by M_Ψ;*
3. *the minimal model $M_\Psi^{temporal}$ under the temporal semantics subsumes or equals the minimal model M_Ψ under the standard semantics;*
4. *A fact which is entailed under the standard semantics must also be entailed under the temporal semantics.*
5. *Buttom-Up evaluation computes $T^\omega_\Psi = M_\Psi^{temporal}$ where T^ω_Ψ is the pair $< \mathcal{F}^\omega_\Psi, \mathcal{C}^\omega_\Psi >$, $\mathcal{F}^\omega_\Psi$ is the intersection of all sets of non-constraint facts and $\mathcal{C}^\omega_\Psi$ is the constraint network containing all the constraint facts in $C^0_\Psi \cup \cdots \cup C^\omega_\Psi$.*

Proof:
Part 1: This property follows from the definition, as the intersection of all models is unique, and always contains everything entailed by the program.
Part 2: This property follows immediately from Part 1.
Part 3: This property follows immediately from Parts 1,2.

Part 4: We divide the atoms in $T_\Psi(\mathcal{I})$ into two partitions: $\mathcal{F}_\Psi(\mathcal{I})$, describing all non-constraint (also non-temporal) facts, and $\mathcal{C}_\Psi(\mathcal{I})$, describing all constraint facts (also temporal facts). We show that every model $M =< \mathcal{F}, \mathcal{C} >$ satisfies $\mathcal{F}^\omega_\Psi \subseteq \mathcal{F}$ and $\mathcal{C}^\omega_\Psi \models \mathcal{C}$, which implies that T^ω_Ψ is the unique intersection of all models.

According to definition 17, for every rule $r = (B \rightarrow H)$, every model $M =< \mathcal{F}, \mathcal{C} >$ satisfies $M \models B \rightarrow M \models H$. In addition, whenever $T^i_\Psi \not\models H$, if $T^{i+1}_\Psi \not\models B$ then $T^{i+1}_\Psi \not\models H$, namely $T^{i+1}_\Psi \models B \leftrightarrow T^{i+1}_\Psi \models H$. This generalizes to $T^\omega_\Psi \models B \leftrightarrow T^\omega_\Psi \models H$, which is stronger than $T^\omega_\Psi \models B \rightarrow T^\omega_\Psi \models H$. Thus, every fact in $\mathcal{F}^\omega_\Psi$ must be in $\mathcal{F}$.

The non-standard part of this proof involves the constraints. The model M satisfies a constraint C iff the conjunction $\mathcal{C} \wedge \bar{C}$ is inconsistent, where $\bar{C}$ is the complement of C. Since $T^\omega_\Psi \models B \leftrightarrow T^\omega_\Psi \models H$, $\mathcal{C}^\omega_\Psi$ is a conjunction of at least those constraints in $\mathcal{C}$. Thus, every constraint in $\mathcal{C}^\omega_\Psi$ is always tighter or equal to the corresponding constraint (on the same pair of variables) in $\mathcal{C}$, namely the conjunction $\mathcal{C} \wedge \bar{C}$ is always looser than the conjunction $\mathcal{C}^\omega_\Psi \wedge \bar{C}$. Consequently, if $\mathcal{C}^\omega_\Psi \not\models C$ then $\mathcal{C} \not\models C$, and the claim follows. $\square$

4.4.6 Addressing the issues from Section 4.3

We now show that the design of TCSP-Datalog enables addressing the issues presented in section 4.3.

Instantaneous Events
The use of predicate buddies enables representing events that occur either instantaneously or over intervals.

Instantaneous and Non-Instantaneous Holding
The use of predicate buddies enables representing both instantaneous and non-instantaneous holding. In the tossed ball example, we could specify that the speed of the balls changed from positive to negative, being zero at a single isolated instant time point t:

$$\texttt{BallSpeed}_{ON}\texttt{(Positive,I}_1\texttt{), BallSpeed}_{AT}\texttt{(Zero,t), BallSpeed}_{ON}\texttt{(Negative,I}_2\texttt{)}$$
$$\texttt{I}_1\texttt{\{meets\}I}_2\texttt{, t\{starts\} I}_2\texttt{, t\{ends\}I}_1\texttt{.}$$

where `meets,starts,ends` are the TCSP relations defined in previous chapters.

The Dividing Instant Problem (DIP)
Holding intervals are always open. Thus, the holding over an interval does not bear any implication on the holding over its end points. For example, the statement "the light was on during interval I" can have four possible meanings: the interval I is open, close, right-side open or left-side open. All four meanings can be expressed in TCSP-Datalog as follows:

- To describe an open interval statement we use $\texttt{Light}_{ON}\texttt{(On,I)}$.

- To describe a closed interval statement we use

$$\texttt{Light}_{AT}\texttt{(On,t}_1\texttt{), Light}_{ON}\texttt{(On,I), Light}_{AT}\texttt{(On,t}_2\texttt{), t}_1\{\texttt{starts}\}\texttt{I, t}_2\{\texttt{ends}\}\texttt{I.}$$

- To describe a right-side open interval statement we use

$$\texttt{Light}_{AT}\texttt{(On,t}_1\texttt{), Light}_{ON}\texttt{(On,I), t}_1\{\texttt{starts}\}\texttt{I.}$$

- To describe a left-side open interval statement we use

$$\texttt{Light}_{ON}\texttt{(On,I), Light}_{AT}\texttt{(On,t}_2\texttt{), t}_2\{\texttt{ends}\}\texttt{I.}$$

Representing Constraint Facts A constraint fact is a rule with an empty body (i.e. unit clause) containing a ground atom. For example, the atom

$$\texttt{t}_2\texttt{-t}_1 \in [3,7]\texttt{.}$$

is a constraint fact.

Constraint atoms in the bodies of rules TCSP-Datalog allows constraint atoms in the bodies of rules without restrictions. When a conjunction of constraints in the database (of facts) entails a constraint atom in the body of a rule, this atom is assigned the value "*true*". For example, consider the program

$$\begin{array}{l} \texttt{C}_1\texttt{.}\quad \texttt{C}_2\texttt{.} \\ \texttt{A :- C}_3\texttt{.} \end{array} \qquad \text{where} \qquad \begin{array}{lcl} \texttt{C}_1 & \equiv & \texttt{t}_1\texttt{-t}_2 \in \texttt{[2,5]} \\ \texttt{C}_2 & \equiv & \texttt{t}_2\texttt{-t}_3 \in \texttt{[7,9]} \\ \texttt{C}_3 & \equiv & \texttt{t}_1\texttt{-t}_3 \in \texttt{[5,20]} \end{array}$$

In TCSP-Datalog, Ψ entails `A` because C_3 is entailed by $C_1 \wedge C_2$, regardless of the values of $\texttt{t}_1\texttt{,t}_2$ `or` $\texttt{t}_3$. Such an inference cannot be made using standard logic programming nor using CLP languages.

Constraint atoms in the heads of rules TCSP-Datalog allows constraint atoms in heads of rules without restrictions. The statement "if the package will be shipped today, then it must be shipped before 4:00pm" can be expressed by the rule

```
t<4pm  :-  Ship(Package,t), Today(t).
```

This sentence cannot be expressed using CLP languages.

4.5 The Token-Datalog Language

The use of tokens dates back to the Time Map Management (TMM) system [21]. The intuition behind tokens is that they are the atomic data entities combining fluents with intervals or time points at which they holds. For example, a token `K` might comprise of a fluent `TheLightIsRed` and a time interval $[t_1, t_2]$ during which it was *true*.

In this chapter, we extend TCSP-Datalog by using tokens. The resulting language, called Token-Datalog, improved on TCSP-Datalog in a number of ways:

1. Periodic relation can now be defined using unary successor and predecessor functions. For example, to describe the period of one week we could write, without knowing the date, a rule stating that Sunday is followed by Monday, which is followed by Tuesday, ..., Saturday is followed by Monday. This infinite periodic relation cannot be described in TCSPs nor in TCSP-Datalog.

2. A clear separation between temporal and non-temporal atoms is achieved. In both languages the temporal terms must be constraint terms. In TCSP-Datalog, atoms can take both constraint and non-constraint terms, while in Token-Datalog they must be either constraint or non-constraint terms but not both, Token terms are used in both temporal and non-temporal atoms to create the links that are needed to preserve the meaning.

3. Queries about possible relations between objects can now be expressed and processed. For example, if `Love(John,Marry)`, `Married(John,Mary)` hold, then we can ask a query about the set of relations between Mary and John and obtain the answer `Love, Married`. This is not feasible in TCSP-Datalog nor in standard LP.

4.5.1 Syntax

There are three sorts of terms: data, token and temporal constraint terms.

Data Terms are either data variables or data constants (as in Datalog).

Token Terms are of two types: point or interval token terms. They are built from token constants, data constants, token variables, data variables and function symbols, as follows:

1. Every token constant symbol and token variable symbol is a token term.

2. If `f` is a function symbol, `k` is a token term, $\mathtt{d}_1,\ldots,\mathtt{d}_n$ are data terms then $\mathtt{f(k,d}_1,\ldots,\mathtt{d}_n\mathtt{)}$ is a token term of the same sort (i.e. point$\mapsto$point, interval$\mapsto$interval).

Example 26: Let `k` be a token variable, `scenarioA, scenarioB` be two data terms and let `next, prev` be two function names. The terms `k, next(k), next(k,scenarioA), prev(next(k,scenarioA),scenarioB)` are legal token terms but `scenarioA, next(k,k), prev(scenarioB)` are not legal token terms.

As we demonstrate below, functions are introduced in Token-Datalog to enable describing a generic succession relation that may be periodic.

Constraint Terms If `k` is a *point* token term, then `time(k)` is a constraint term. If `k` is an *interval* token, then `interval(k)`, `begin(k)` and `end(k)` are temporal constraint terms satisfying `begin(k){starts} interval(k)` and `end(k){ends}interval(K)`.

Example 27: The terms `interval(k)`, `begin(f(k))`, `end(f(f(k,`d_1`),`d_2`))` are legal temporal constraint terms but `interval(`d_1`)`, `interval(interval(k))`, `begin(interval(k` are not legal temporal constraint terms.

Note that the use of functions taking a single token argument is first introduced in Datalog$_{nS}$ [15, 16]. Here we adopt this idea to enable representing periodic patterns of occurrences. Recall that TCSP-Datalog does not allow functions of any kind.

Predicates are of three types: constraint, token and neutral predicates. Neutral predicates take only data terms. Constraint predicates represent TCSP constraints, take exactly two constraint terms (i.e. binary constraints) and cannot take data terms. Token predicates take exactly one token term and may take data terms without restrictions (i.e. cannot take constraint terms).

Token predicates come in pairs. The predicate $P(a_1,\ldots,a_n,t)$, where `t` is a point token, has a pair, or *buddy* predicate $P(a_1,\ldots,a_n,I)$, where `I` is an interval token.

In Token-Datalog it is possible to describe infinite periodic sequences into the past as well as into the future. This can be achieved by constraining a token k_0 to be either before or after the the token $f(k_0)$. For example, the terms k_0, `successor(`k_0`)` and `predecessor(`k_0`)` can be constrained such that

$$\texttt{time(predecessor(}k_0\texttt{))} < \texttt{time(}k_0\texttt{)} < \texttt{time(successor(}k_0\texttt{))}$$

Atoms are built from predicates as usual. For the temporal constraints, we use the TCSP constraints described in previous chapters. Specifically, we use `a=b`, `a≠b`, `a<b`, `a≤b`, $x - y \in [a, b]$ with the usual meaning as a short hand for `Eq(a,b)`, `Neq(a,b)`, `Less(a,b)`, `LessEq(a,b)`, `Difference(x,y,a,b)` respectively. Disjunctive temporal constraints are also allowed.

Example 28: The fact `time(K)-time(prev(K))=8` implies that the successor of `K` is a token which occurred before `K`. In contrast, `time(next(K))-time(K)=8` goes forward in time.

Rules are of the form $H_1, \ldots, H_m$ `:-` $B_1, \ldots, B_k$, where H_i and B_i are arbitrary atoms. Rules must be *range restricted*, namely all non-constraint (i.e. token or data) variables must appear in the body of some rule or equated to a variable in the body of some rule.

Example 29: To explicate the difference between Token-Datalog and TCSP-Datalog, consider representing the following guideline:"Treatment A requires a duration of 1-2 hours and Treatment B requires a duration of 2-3 hours. According to the guidelines, if both Treatment A and Treatment B are needed, then they must be given at least 7 hours and at most 9 hours apart." This statement can be represented by the program Ψ =

```
end(k₁)-begin(k₁)∈[1,2] :-  Administrate_ON( TreatmentA, k₁).
end(k₂)-begin(k₂)∈[2,3] :-  Administrate_ON( TreatmentB, k₂).
begin(k₂)-end(k₁)∈[7,9] :-  Administrate_ON( TreatmentB, k₂),
                            Administrate_ON( TreatmentA, k₁).
```

Traditional LP allows heads of rules to contain a single atom only. For express sentences that involve temporal constraints it is often move convenient and straitforward to introduce rules whose heads contains multiple atoms (both constraint and non-constraint atoms).

Example 30: Consider the following variation on the above guideline: "Medicine A should be administered every 7-9 hours". This statement could be represented by the rule

```
begin(next(k))-end(k)∈[7,9], Administrate_ON(MedicineA, next(k)) :-
            Administrate_ON(MedicineA, k).
```

where the head is a *conjunction* of atoms. The conjunction in the head present in the above rule can be described, on the propositional level, as

```
C,B :- A   ≡   (C∧B)∨¬A   ≡   (B∨¬A)∧(C∨¬A)   ≡   B :- A, C :- A.
```

where A,B,C are three simple propositions. Consequently, the sample Token-Datalog rule can be broken into the following two equivalent rules having a single atom in the head:

```
Administrate_ON(MedicineA, next(k)) :- Administrate_ON(MedicineA, k).
begin(next(k))-end(k)∈[7,9] :- Administrate_ON(MedicineA, k).
```

4.5.2 Predicate Buddies

The technical problem addressed by predicate buddies is the same as in TCSP-Datalog (see section 4.4.2). Atoms of the form `HoldsAt(F,t)` and `HoldsOn(F,i)` are replaced by the conjunction `F(k)`$\wedge$ `time(k)=t` where the fluent name `F` is used as the predicate name and the temporal term `t` is replaced by a token term `k`. The advantages of the token-argument temporal qualification method over the temporal argument qualification method used in TCSP-Datalog (described in the beginning of this section) are (i) enables expressing periodic relations, (ii) provides a better separation between temporal and non-temporal components, and (iii) improves expressiveness of the query language.

4.5.3 Expressing Incompatibility

The technical problem, described in section 4.4.3, involves describing which fluents cannot be *true* simultaneously. For example, a person cannot be in Europe and Australia simultaneously. This problem is addressed with the following Token-Datalog rules:

1. If k_1, k_2 are two point token terms then add the rule

   $$\mathtt{time(k_1)} \neq \mathtt{time(k_2)}\ \mathtt{:-}\ \mathtt{A_1,A_2,\ldots,A_n.}$$

2. If k_1 is a point token terms and k_2 is an interval token term then add the rule

   $$\mathtt{time(k_1)}\ \{\mathtt{before,after}\}\ \mathtt{interval(k_2)}\ \mathtt{:-}\ \mathtt{A_1,A_2,\ldots,A_n.}$$

3. Otherwise, use the rule

   $$\mathtt{interval(k_1)}\ \{\ \mathtt{before,meets,met\text{-}by,after}\}\ \mathtt{interval}(k_2)\ \ \mathtt{:-}\ \ \mathtt{A_1,A_2,\ldots,A_n.}$$

4.5.4 Standard Semantics

The standard semantics for Token-Datalog is as defined above for TCSP-Datalog. The only difference is that Token-Datalog introduces the additional sort of tokens, and thus the domain $\mathcal{D}$ is divided into three partitions: time constants (points and intervals), token constants and data constants. These constants enable defining three types of terms: temporal constraint terms, token terms and data terms.

4.5.5 Temporal Semantics

The functions `time, interval, begin, end` have predefined meaning. The function `time` maps point token terms into time points. The function `interval` maps interval token terms into time intervals. The functions `begin`,`end` map interval token terms into the end points of `interval(k)`, namely they satisfy the constraints

$$\texttt{begin(k)\{starts\} interval(k),} \qquad \texttt{end(k)\{ends\} interval(k).}$$

The domain $\mathcal{D}$ is divided into three partitions: (i) $\mathcal{T}$ is the temporal domain, (ii) $\mathcal{K}$ is the set of token constants and (iii) $\mathcal{D}'$ is the set of data constant. For every interpretation $\mathcal{I}$, ground token terms in $\mathcal{K}$ are constants, but their interpretation on the temporal domain $\mathcal{T}$ is not fixed, because every point token $\texttt{K}\in\mathcal{K}$ can be assigned a value $t \in \mathcal{T}$. A fact is a member of the Herbrand base (as usual).

The Token-Datalog satisfaction relation is similar to the TCSP-Datalog satisfaction relation. It is the conjunction of the standard satisfaction relation with the additional condition called the homogeneity property. The only difference between the TCSP-Datalog and the Token-Datalog definition is that interval terms are replaced by interval token terms.

Definition 18: An interpretation $\mathcal{I}$, partitioned into constraint facts $\mathcal{C}$ and non-constraint facts $\mathcal{F}$, is a model of Ψ under the *temporal semantics*, denoted $\mathcal{I} \models_{temporal} \Psi$, iff

1. $\mathcal{I}$ is a model of Ψ under the standard non-temporal semantics described above,
2. for every ground temporal atom $\texttt{A = P(a}_1\texttt{,...,a}_n\texttt{, k)}$ where $\texttt{k}$ is an interval token term and $\texttt{a}_1\texttt{,...,a}_n$ are data terms, the following two conditions are satisfied:

   (a) if $A \in \mathcal{I}$ then
   $\forall \texttt{k}_{interval}\texttt{,}\ \mathcal{C} \models \texttt{begin(k)}\leq\texttt{begin(k}_{interval}\texttt{)},\ \mathcal{C} \models \texttt{end(k)}\leq\texttt{end(k}_{interval}\texttt{)}$
   $\longrightarrow \texttt{P(k}_{interval}\texttt{, a}_1\texttt{,...,a}_n\texttt{)}\in \mathcal{I}$, and
   $\forall \texttt{k}_{point}\texttt{,}\quad \mathcal{C} \models \texttt{begin(k)<k}_{point}\texttt{<end(k)} \longrightarrow \texttt{P(k}_{point}\texttt{, a}_1\texttt{,...,a}_n\texttt{)}\in \mathcal{I}$

   (b) if $A \notin \mathcal{I}$ then
   $\exists \texttt{k}_{interval}\texttt{,}\ \mathcal{C} \models \texttt{begin(k)}\leq\texttt{begin(k}_{interval}\texttt{)},\ \mathcal{C} \models \texttt{end(k)}\leq\texttt{end(k}_{interval}\texttt{)}$
   $\longrightarrow \texttt{P}_{ON}\texttt{(a}_1\texttt{,...,a}_n\texttt{, k}_{interval}\texttt{)}\notin \mathcal{I}$, or
   $\exists \texttt{k}_{point}\texttt{,}\quad \mathcal{C} \models \texttt{begin(k)<time(k}_{point}\texttt{)<end(k)} \longrightarrow \texttt{P}_{AT}\texttt{(a}_1\texttt{,...,a}_n\texttt{, k}_{point}\texttt{)}\notin$
   $\mathcal{I}$

where $\mathcal{C}$ is the conjunction of constraint facts in $\mathcal{I}$, $\texttt{P}_{ON}\texttt{(a}_1\texttt{,...,a}_n\texttt{, k}_{interval}\texttt{)}$, $\texttt{P(k}_{point}\texttt{, a}_1\texttt{,...,a}_n\texttt{)}$ are buddies (recall syntax definition above).

The conditions 2a and 2b in Definition 18 are introduced to ensure that homogeneity holds. Condition 2a ensures that if the atom $\texttt{P}_{ON}\texttt{(a}_1\texttt{,...,a}_n\texttt{, k)}$ is in the set of facts, then for every point token $\texttt{k}_{point}$ which is within the interval of $\texttt{k}$, the atom $\texttt{P}_{AT}\texttt{(a}_1\texttt{,...,a}_n\texttt{, k}_{point}\texttt{)}$ is in the set of facts. Similarly, condition 2b ensures that for every $\texttt{k}_{interval}$ which is a subinterval of $\texttt{k}$, the atom $\texttt{P}_{ON}\texttt{(a}_1\texttt{,...,a}_n\texttt{, k}_{interval}\texttt{)}$ is in the set of facts.

Theorem 18: *For every Token-Datalog program* Ψ*,*

1. *the intersection of all models is the unique minimal model, $M_{\Psi}^{temporal}$, under the temporal semantics which contains everything that is entailed by Ψ;*

2. *the minimal model $M_{\Psi}^{temporal}$ under the temporal semantics subsumes or equals the minimal model M_{Ψ} under the standard semantics;*

3. *A fact which is entailed under the standard semantics must also be entailed under the temporal semantics.*

4. *Buttom-Up evaluation computes $T_{\Psi}^{\omega} = M_{\Psi}^{temporal}$ where T_{Ψ}^{ω} is the pair $< \mathcal{F}_{\Psi}^{\omega}, \mathcal{C}_{\Psi}^{\omega} >$, $\mathcal{F}_{\Psi}^{\omega}$ is the intersection of all sets of non-constraint facts and $\mathcal{C}_{\Psi}^{\omega}$ is the constraint network containing all the constraint facts in $C_{\Psi}^{0} \cup \cdots \cup C_{\Psi}^{\omega}$.*

Proof: The proof of Theorem 17 applies here. □

To answer queries, we could use the following corollary:

Corollary 2: *A non-constraint atom is a logical consequence of a program Ψ iff it is in $\mathcal{F}_{\Psi}^{\omega}$ and a (temporal) constraint atom is a consequence of Ψ iff it is entailed by $\mathcal{C}_{\Psi}^{\omega}$.*

Proof: From Theorem 18 it follow that, for every model $M =< \mathcal{F}, \mathcal{C} >$, $\mathcal{F}_{\Psi}^{\omega}$ is subsumed in $\mathcal{F}$ of every model, and $\mathcal{C}_{\Psi}^{\omega}$ entails $\mathcal{C}$. □

In other words, all the temporal constraints that are entailed by Ψ must also be entailed by temporal constraints in the unique minimal model M_{Ψ}.

Example 31: Consider the program Ψ =:

```
1.  Active(1,k0).
2.  Active(2,next(K)) :- Active(1,K)  .
3.  Active(1,next(K)) :- Active(2,K)  .
4.  interval(K){meets}interval(next(K)).
```

The *minimal model* of this program, M_{Ψ}, is:

```
Active(1,k0).
Active(2,next(k0)).        interval(k0){meets}interval(next(k0)).
Active(1,next(next(k0))).  interval(next(k0)){meets}interval(next(next(k0))).
                ⋮
```

The relation `interval(k0) {before} interval(next(next(k0)))` is not specified by Ψ yet it is included in the set of constraint facts Ψ entails because it is entailed by the conjunction `interval(k0) {meets} interval(next(k0))` $\wedge$ `interval(next(k0)) {meets} interval(next(next(k0)))`.

4.5.6 Addressing the issues from Section 4.3

Since Token-Datalog is more expressive than TCSP-Datalog which addresses all the issues of instantaneous events, instantaneous holding of fluents, non-instantaneous holding of fluents and the DIP, these issues are also addressed by Token-Datalog.

4.6 Tentative Summary

We presented two languages that reside within the logic programming paradigm, which are based on TCSP, Datalog, CLP and some of their variants. The glue connecting between TCSP (i.e. temporal constraint) and Datalog (i.e. logic programming) are the temporal qualification methods of Temporal Arguments and Temporal Tokens.

Our languages have well defined temporal domains, theory of temporal incidence, syntax and semantics. The semantics of these languages augments the standard semantics with a theory of temporal incidence which includes homogeneity. We analyzed the interaction between the temporal and non-temporal aspects of the language through issues such as incompatibility.

Our languages present the following desirable combination of features:

1. They incorporate a well defined theory of temporal incidence which enables representing both instantaneous and non-instantaneous events. It also allows expressing the instantaneous and non-instantaneous holding of both continuous and discrete fluents.

2. The logic programming framework provides a computational basis for making inferences. The inference algorithms are described in the next chapter.

3. The "Holds" predicates are used for the non-temporal component. These were introduced by [6] and refined by [37].

4. The well understood TCSP framework is used for the temporal component. Efficient algorithms such as those presented in Chapters 2 and 3 can be brought to bear, as we show in the next chapter.

5. The virtues of the languages described in this chapter are:

   (a) Periodic relation can be defined using unary successor and predecessor functions. For example, to describe the period of one week we could write, without knowing the date, a rule stating that Sunday is followed by Monday, which is followed by Tuesday, ..., Saturday is followed by Monday. This infinite periodic relation cannot be described in TCSPs nor in TCSP-Datalog.

(b) A clear separation between temporal and non-temporal atoms is achieved. In both languages the temporal terms must be constraint terms. In TCSP-Datalog, an atoms can take both constraint and non-constraint terms, while in Token-Datalog they must be either constraint or non-constraint terms but not both. Token terms are used in both temporal and non-temporal atoms to create the links needed to preserve the meaning.

Chapter 5

Making Temporal Inferences

In logic programming, inferences are performed through the computation of SLD-refutations. The elementary operation within an SLD-refutation is called SLD-derivation. It takes as input two (or more) expressions in a language $\mathcal{L}$, performs syntactic manipulations and computes a derived expression in the same language $\mathcal{L}$. A proof is an SLD-refutation, which is an SLD-derivation in which inconsistency is proved. The algorithm for computing SLD-refutations is called SLD-resolution. In this section, we show that standard SLD-resolution is incomplete with respect to the temporal semantics applied to our Datalog-based language (defined in the previous chapter) and propose a modification which renders it complete.

The shortcoming of standard resolution stems from (i) inadequate unification of constraints atoms (to be distinguished from regular atoms), and (ii) inadequate unification of ground terms which does not support homogeneity.

The first step in addressing these shortcomings involves designing a unification algorithm giving constraint atoms special treatment. Constraint C_1 can be unified with a constraint facts C_2 if $C_2 \models C_1$. The ability to make such a unification enables inferring what is considered standard constraint entailment in the CSP framework.

The second step involves unifying two ground terms. If $\mathtt{k}_1$ and $\mathtt{k}_2$ are two overlapping time entities, then we unify them into a third token, `k'`, which is the overlapping portion of the two. This enables reasoning about simultaneous occurrences.

5.1 SLD-Resolution

Although Datalog is propositional, augmenting it with temporal and token variables renders it non-propositional and raises the need for a generalized resolution inference engine. In this section, we introduce the standard resolution algorithm. This involves the standard definitions of *most general unifiers* and *derivations*.

Definition 19: Given a finite set S of atoms, we say that a substitution θ is a unifier if applying θ on all the elements of S results in identical atoms, namely $S\theta$

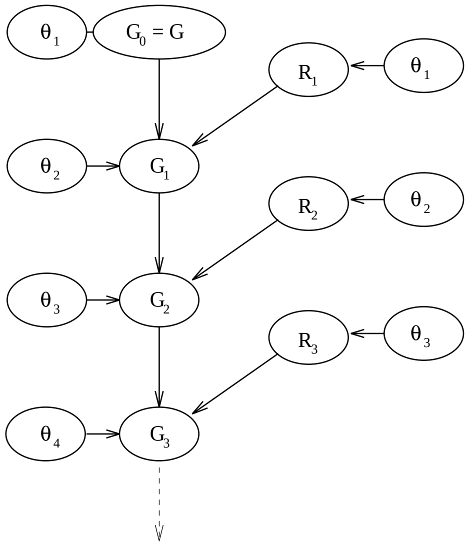


Figure 5.1: SLD-Derivation

is a singleton. A unifier θ is called a *most general unifier*, abbreviated *mgu*, iff for every σ which is a unifier of S, there exists a substitution γ such that applying σ is equivalent to applying $\theta\gamma$, namely $\sigma = \theta\gamma$.

Definition 20: Let G be the goal `:-` $\mathtt{A}_1$`,`...`,`$\mathtt{A}_m$`,`...`,`$\mathtt{A}_k$, and let R be a rule `H :-` $\mathtt{B}_1$`,`...`,`$\mathtt{B}_q$. Then G' is derived from, or the *resolvant* of G and R using the most general unifier θ if the following holds:

1. $\mathtt{A}_m$ is an atom, called the *selected* atom in G.

2. θ is a most general unifier of $\mathtt{A}_m$ and `H`.

3. G' is the goal `:-` $\mathtt{A}_1$`,`...`,`$\mathtt{A}_{m-1}$`,`$\mathtt{B}_1$`,`...`,`$\mathtt{B}_q$`,`$\mathtt{A}_m$`,`...`,`$\mathtt{A}_k$.

Definition 21: Given a program Ψ and a goal G, a *selected-derivation*, also called *SLD-derivation* of $\Psi \cup G$, consists of a sequence G, $G_1, G_2 \ldots$ of goals, a sequence $R_1, R_2, \ldots$ of rules, and a sequence $\theta_1, \theta_2 \ldots$ of most general unifiers such that G_{i+1} is the resolvant of $G_i\theta_{i+1}$ and $R_{i+1}\theta_{i+1}$, as illustrated in Figure 5.1. A *selected-refutation*, also called *SLD-refutation* or proof, is an *SLD-derivation* which includes (or terminates) with an empty clause.

Resolution algorithms are aimed at computing SLD-refutations. Because introducing time into logic requires syntactic and semantic modifications, standard resolution algorithms are no longer sufficient for temporal reasoning.

Example 32: Consider formalizing the statement "*The medicine needs to be taken every 8 hours during five days*". Let T be the only temporal variable used and let medicine(T) evaluate to *true* iff the medicine is taken at time point T. We could represent the example statement by the TCSP-Datalog program:

```
medicine(0).
medicine(T+8) :- medicine(T), T∈[0,120].
```

Using K as the only token variable and k_0 as the only token constant, an equivalent Token-Datalog program is as follows:

```
time(k₀)=0.
time(next(K))-time(K)=8.
medicine(k₀).
medicine(next(K)) :-  medicine(K), time(K)∈[0,120].
```

where next is a function mapping tokens to tokens. One of the features of this Token-Datalog formulation is that it is more expressive. The user can replace the deterministic temporal constraint time(next(K))-time(K)=8 with a non-deterministic constraint time(next(K))-time(K)∈[7,9], while staying within the Token-Datalog language.

Consider the query "*Is the medicine taken at time point 8 ?*" which can be formalized as follows:

```
time(k₁)=8.
:- medicine(k₁).
```

Using SLD-resolution, the evaluation of this query proceeds as follows: It begins by resolving the negation of the *query* literal ¬medicine(k_1) with the rule

```
medicine(next(K)) :-  medicine(K),
                      time(K)∈[0,120].
```

Because medicine(next(K)) is in the head (i.e. not negated), an attempt is made to unify it with ¬medicine(k_1). The unification succeeds with the substitution θ_1 = (next(K)/k_1) and the resolvant is

```
:- medicine(K), time(K)∈[0,120].
```

This resolvant consists of atoms that are still not ground, and K can be assigned values such that K≠next(k_1). This is undesirable because the relationship between the terms T_1 =next(K) and T_2 =K is lost. We will therefore resolve this issue by explicitly adding the substitution θ_2=(K/next^{-1}(k_1)) and deriving the resolvant:

```
:-  medicine(next⁻¹(k₁)),
    time(next⁻¹(k₁))∈[0,120].
```

The next inference step involves *temporal constraint propagation.* The fact `time(next(K))-time(K)=8` entails that $\texttt{time(k}_1\texttt{)-time(next}^{-1}\texttt{(k}_1\texttt{))=8}$. From the fact $\texttt{time(k}_1\texttt{)=8}$ we can infer that $\texttt{time(next}^{-1}\texttt{(k}_1\texttt{))=0}$ and that $\texttt{time(k}_0\texttt{)=time(next}^{-1}\texttt{(k}_1\texttt{))}$.

The problem is that in order to derive the last sentence we need to resolve again but we cannot unify the term $\texttt{next}^{-1}(k_1)$ with any other term, because it is already ground. To arrive at the correct answer and infer that the medicine is given at $\texttt{k}_1$, we need to substitute θ_3 = $\texttt{(next}^{-1}\texttt{(k}_1\texttt{)/k}_0\texttt{)}$. We call this non-standard operation of unifying (or substituting) two ground token terms, *token fusion.* After the fusion we obtain two new subgoals:

$$\texttt{:- medicine(k}_0\texttt{), time(k}_0\texttt{)}\in\texttt{[0,120].}$$

$\texttt{medicine(k}_0\texttt{)}$ is given as a fact. Because $\texttt{time(k}_0\texttt{)=0}$ is also a fact we know that the literal $\texttt{time(k}_0\texttt{)}\in\texttt{[0,120]}$ is entailed. This completes the proof for the query `:-` $\texttt{medicine(k}_1\texttt{)}$. The substitution used in answering this query is

$$\begin{aligned}\theta &= \theta_1\theta_2\theta_3\\ &= \texttt{(next(K)/k}_1\texttt{)(K/next}^{-1}\texttt{(k}_1\texttt{))(next}^{-1}\texttt{(k}_1\texttt{)/k}_0\texttt{)}\end{aligned}$$

where θ_3 is *token fusion.*

5.2 TCSP-Resolution

In the following we present a sample program and query for which there is no SLD-refutation for making the intended temporal inference. To address this problem, *temporal resolution* is introduced. It is a modification of the standard resolution algorithm in which the unification algorithm is modified to support temporal reasoning with constraints. The result is an inference algorithm capable of computing refutations according to the intended temporal semantics.

We start by illustrating why SLD-resolution is not sufficient for making inferences according to the temporal semantics.

Example 33: Consider the introductory example with Digoxin and Potassium Supplements. Assume that the drugs, A,B, need to be administered. The medical guidelines specify that if both Medicine A and B are administered, then B needs to be administered between 7 and 9 hours after A. The task at hand is: Observing that Medicine A was administered at time point 0 and Medicine B was administered at the 8 hour time point, determine whether the above guideline is satisfied. This treatment history is expressed below by two TCSP-Datalog facts. The guideline to be verified is expressed below as a TCSP-Datalog query.

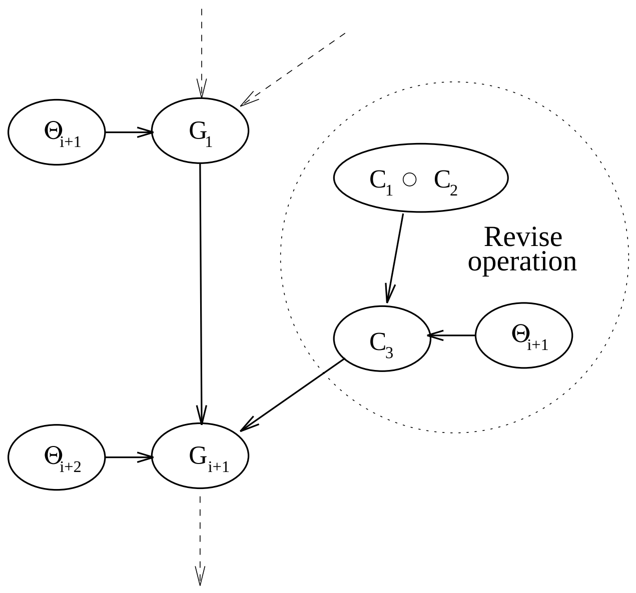


Figure 5.2: Simple TCSP-Derivation

```
Database:  Administrate(MedicineA, 0).
           Administrate(MedicineB, 8).
Query:     t₂-t₁ ∈[7,9], Administrate(MedicineB,t₂), Administrate(MedicineA,t₁).
```

where `Administrate(X,t)` evaluate to *true* iff medicine X was administered at time point t.

Although the guideline is clearly satisfied, there is no SLD refutation for this program. This is because we have $t_2 - t_1 = 8$ which cannot be related to $t_2 - t_1 \in [7, 9]$. To arrive at the answer 'Yes', we need to make a non-standard inference that the constraint query `t`$_2$`-t`$_1$ $\in$`[7,9]` is entailed by the constraint `t`$_2$`-t`$_1$`=8` which is induced by the database facts. This step is non-standard and is achieved by constraint inference, or by what we call constraint unification, as shown below.

As illustrated in Figures 5.2,5.3, this inference requires two steps: (i) inferring the constraint fact `Y-X=8` from the composition of `X=0, Y=8` and (ii) unifying the query constraint `Y-X`$\in$`[7,9]` with the constraint fact `Y-X=8`. The first step can be performed by the constraint propagation algorithms described in chapters 2 and 3. The second step is performed by the constraint unification algorithm described in this section. We call this two step derivation *TCSP-derivation.*

In summary, TCSP-Resolution differs from SLD-Resolution in that its derivation include one additional operation type, called constraint propagation.

Definition 22: In a *Simple TCSP-Derivation*, every derivation step is either a standard SLD-Derivation step or the operation `Revise(i,j,k)`, defined as follows: Let A_1, A_2, A_3 be three unit clauses specifying the constraint atoms $C(X_i, X_j), C(X_i, X_k), C(X_k, X_j)$ respectively. Then `Revise(i,j,k)` modifies A_1 by replacing its content with the

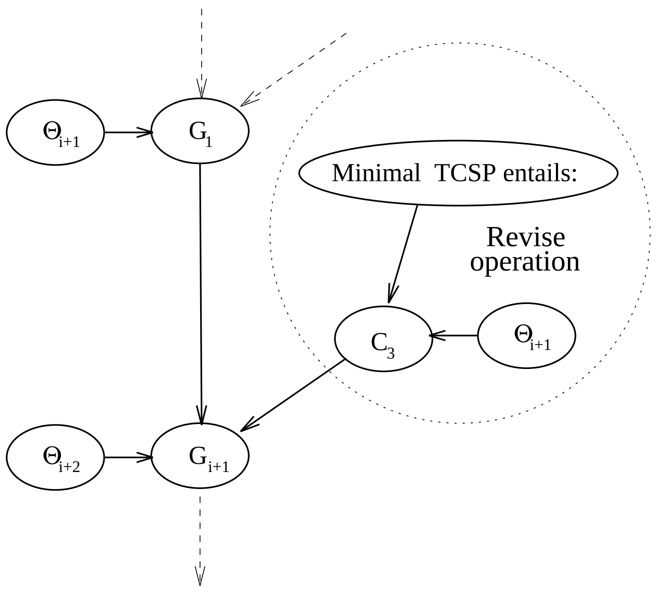


Figure 5.3: TCSP-Derivation (i.e. general - non-simple).

constraint resulting from the operation $C(X_i, X_j) \cap (C(X_i, X_k) \circ C(X_k, X_j))$ whose semantics is defined by TCSPs in chapters 2,3. A TCSP-derivation (i.e. general, non-simple) is a Simple TCSP-Derivation where the operation `Revise(i,j,k)` is replaced by the computation of the minimal TCSP. A *TCSP-Refutation* is a TCSP-derivation which contains either *an empty clause or an empty constraint.* A *Simple TCSP-Refutation* is a Simple TCSP-derivation which contains either *an empty clause or an empty constraint.*

In order to compute a TCSP-Derivation (and even a Simple TCSP-Derivation), there must be unit constraint clauses containing a constraint $C(X_i, X_j)$ for all i, j. The conjunction of these constraints together form the TCSP which is used to determine entailment. To guarantee the existence of unit clauses describing such constraints, we could add as a unit clause, for every i, j, the universal constraint over X_i, X_j.

Lemma 9:

1. *Computing an inference step for a Simple-TCSP-Derivation can be done in time which is linear in the length of the expressions unified.*

2. *Computing an inference step for a TCSP-Derivation is not tractable.*

Proof: Computing an inference step within the standard SLD-Derivation is linear in the length of the expressions. Simple TCSP-Derivations include steps which involve composition of two constraints, which requires a constant amount of work. General TCSP-Derivations, however, require computing the entailed constraint. This task is not tractable. □

Note, however, that once the minimal TCSP is available, deciding entailment can be done in constant time. Consequently, in some cases it may be beneficial to compute the minimal TCSP before we start computing the derivation steps. This minimal TCSP can be incrementally updated as new constraints are conjoined with it. Updating the minimal TCSP is tractable in case the TCSP falls within a tractable class (see Chapters 2,3).

5.2.1 TCSP-Datalog Constraint Unification Algorithm

A TCSP-Datalog constraint atom is a binary predicate of the form $C(t_1, t_2)$ where C is the temporal relation (i.e. constraint) and t_1, t_2 are point or interval constraint terms (i.e. constraint variables).

Algorithm UnifyConstraints for TCSP-Datalog is presented in Figure 5.4. Given a constraint atom A and a consistent TCSP $\mathcal{C}$, it computes a unifier θ such that $\mathcal{C} \models A\theta$ or $\mathcal{C} \models \bar{A}\theta$. This is achieved by trying to unify the pair of variables in A with some pair of variables in $\mathcal{C}$. This algorithm is based on the following definition:

Definition 23: A substitution θ is a unifier of $C_1(t_1, t_2)$ and $C_2(t_3, t_4)$ iff there $C_1(t_1\theta, t_2\theta) \models C_2(t_3\theta, t_4\theta)$. The atoms $C_1(t_1, t_2)$ and $C_2(t_3, t_4)$ are unifiable iff exists such a substitution.

Example 34: Consider the above sample guideline verification task and let $C_1 = t_2 - t_1 = 8$ and $C_2 = t_2 - t_1 \in [7, 9]$. The substitution $\theta = (t_1/t_1)(t_2/t_2)$ is a unifier. It is also the most general unifier.

Definition 24: Let $\mathcal{C}_U$ be the set of all unit constraint clauses containing temporal constraints.

Lemma 10: *Applying algorithm UnifyConstraints (see Figure 5.4) on a TCSP defined by $\mathcal{C}_U$ and a constraint atom A,*

1. *succeeds iff $\exists\theta \;\; \mathcal{C}_U \models A\theta \;\vee\; \mathcal{C}_U \models \bar{A}\theta$, and*
2. *computes θ such that $\mathcal{C}_U \models A\theta$ or $\mathcal{C}_U \models \bar{A}\theta$.*
3. *step 1 is intractable (we need an approximation) but steps 2-8 (see Figure 5.4) terminate in $O(nm)$ steps where n is the number of constraints in $\mathcal{C}_U$ and m is the maximum length of the terms unified.*

Proof: Follows immediately from the algorithm in Figure 5.4. □

Algorithm UnifyConstraints (Figure 5.4) deviates from the standard unification algorithms in that it unifies two atoms even if they differ in the constraint (i.e. predicate) they specify. Had we not unified different constraints on the same pair of variables it would have restricted the inference to entailing facts explicitly listed in the database only, thus not utilizing the ability of constraint propagation to infer implicit constraints.

```
Algorithm UnifyConstraints for TCSP-Datalog
Input: A constraint atom $A$ and a consistent TCSP $\mathcal{C}$.
Output: A unifier $\theta$ such that $\mathcal{C} \models A\theta$ or $\mathcal{C} \models \bar{A}\theta$ (if $\theta$ exists).
 1. Compute $\mathcal{C}_{min}$, the minimal TCSP of $\mathcal{C}$.
 2. For every constraint $C$ in $\mathcal{C}_{min}$ do
 3.     Define $C = C(t_1, t_2)$ and $A = A(t_3, t_4)$.
 4.     Let $\theta = (t_3/t_1)(t_4/t_2)$.
 5.     If $C\theta \models A\theta$ or $C\theta \models \bar{A}\theta$ (use TCSP techniques to test) then return $\theta$.
 6.     Else exit with failure.
 7.     End-if
 8. End-for
```

Figure 5.4: Unifying constraints in TCSP-Datalog.

5.2.2 Token-Datalog Constraint Unification Algorithm

The Token-Datalog constraint unification algorithm differs from its TCSP-Datalog counterpart in that it takes into account functions mapping from token-terms to constraint-terms.

Definition 25: A constraint atom is a binary predicate of the form

$$C(g_1(\mathtt{K}_1), g_2(\mathtt{K}_2))$$

where C is the temporal relation (i.e. constraint), $\mathtt{K}_1$,$\mathtt{K}_2$ are token terms, $g_1(\mathtt{K}_1)$ and $g_2(\mathtt{K}_2)$ are constraint terms (i.e. constraint variables) and g_1, g_2 are functions mapping token terms to points or interval constraint variables.

Example 35: Consider an atom `interval(next(K))` {`overlaps`}`interval(K)` which evaluates to *true* whenever the intervals associated with `K` and `next(K)` overlap. For this atom, $\mathtt{K}_1$`=next(K)`, $\mathtt{K}_2$`=K`, g_1=g_2`=interval` and $C = C(X, Y)$ is the relation `X{ overlaps} Y`.
The atom `begin(next(K))-end(K)=8` evaluates to *true* whenever the beginning of the interval associated with the token term `next(K)` is 8 time units after the end of the interval associated with the token term `K`. For this atom, $\mathtt{K}_1$ `= next(K)`, $\mathtt{K}_2$`=K`, g_1`=begin`, g_2`=end` and $C = C(X, Y)$ is the relation $X - Y \in [8, 8]$.

Algorithm UnifyConstraints is presented in Figure 5.5. Given a constraint atom A and a consistent TCSP $\mathcal{C}$, it computes a unifier θ such that $\mathcal{C} \models A\theta$ or $\mathcal{C} \models \bar{A}\theta$. This is achieved by trying to unify the pair of variables in A with some pair of variables in $\mathcal{C}$. In contrast to TCSP-Datalog, unifying tokens requires processing functions. Because the functions `time, begin, end` take a single token term, the unification task reduces to unifying their token arguments. This simplifies the unification process considerably.

Example 36: To illustrate the execution of algorithm UnifyConstraints, let $\mathcal{C}_U$ include the constraint C = `time(`$\texttt{k}_1$`)-time(`$\texttt{k}_0$`)=8` and the query A = `time(next(K))-time(K)`$\in$`[4,1`
Since $g_1 = g_3$ and $g_2 = g_4$ we try to unify `next(K), K` with $\texttt{k}_1$,$\texttt{k}_0$ respectively. The unification succeeds with the substitution θ = `(next(K)/`$\texttt{k}_1$`)(K/`$\texttt{k}_0$`)` which has the desired property that $C \models A\theta$ implying that $\mathcal{C}_U \models A\theta$ and that $\Psi \models A\theta$. Recall that the entailment is tested using the TCSP techniques described in chapters 2 and 3.

Lemma 11: *Applying algorithm UnifyConstraints on a TCSP defined by $\mathcal{C}_U$ and a constraint atom A,*

1. *succeeds iff $\exists\theta \;\; \mathcal{C}_U \models A\theta \;\vee\; \mathcal{C}_U \models \neg A\theta$, and*
2. *computes θ such that $\mathcal{C}_U \models A\theta$ or $\mathcal{C}_U \models \neg A\theta$.*
3. *step 1 is intractable (we need an approximation) but steps 2-9 terminate in $O(nm)$ steps where n is the number of constraints in $\mathcal{C}_U$ and m is the maximum length of the terms unified.*

Proof: Follows immediately from the algorithm in Figure 5.5. □

To see the problem solved here, recall that in example 32, without constraint unification we could not infer that `time(`$\texttt{k}_0$`)=time(`$\texttt{next}^{-1}$`(`$\texttt{k}_1$`))`. As is the case with TCSP-Datalog, algorithm UnifyConstraints for Token-Datalog (see Figure 5.5) deviates from the standard unification algorithms in that it unifies two atoms even if they differ in the constraint (i.e. predicate) they specify.

Example 37: Constraint unification includes an inference step and thus it deviates from standard unification. In this example, we show how it is possible to unify two atoms describing different predicates symbols, as long as the unified predicates are temporal constraints. Consider the two atom `interval(`$\texttt{k}_1$`){meets,starts, overlaps}interval(`$\texttt{k}_2$`)` and `interval(next(K)) {meets,overlaps}interval(K)`. The substitution θ = `(next(K)/`$\texttt{k}_1$`)(K/`$\texttt{k}_0$`)` unifies these two constraint atoms. To understand why such unification cannot be part of an SLD-refutation, we examine the predicate calculus representation of the two constraints being unified. The first constraint is equivalent to the disjunction

$$\begin{array}{l}\texttt{Meets(interval(k}_1\texttt{),interval(k}_2\texttt{))} \vee \\ \texttt{Starts(interval(k}_1\texttt{),interval(k}_2\texttt{))} \vee \\ \texttt{Overlaps(interval(k}_1\texttt{),interval(k}_2\texttt{))}\end{array}$$

while the second constraint is equivalent to the disjunction

$$\begin{array}{l}\texttt{Meets(interval(next(K)),interval(K))} \vee \\ \texttt{Overlaps(interval(next(K)),interval(K))} \quad .\end{array}$$

Because the sets of predicates are different, such a unification is an example of an inference step that may be included in a TCSP-refutation, but cannot be a step in an SLD-refutation.

```
Algorithm UnifyConstraints for Token-Datalog
Input: A constraint atom $A$ and a consistent TCSP $\mathcal{C}$.
Output: A unifier $\theta$ such that $\mathcal{C} \models A\theta$ or $\mathcal{C} \models \neg A\theta$ (if $\theta$ exists).
 1. Compute $\mathcal{C}_{min}$, the minimal TCSP of $\mathcal{C}$.
 2. For every constraint $C$ in $\mathcal{C}_{min}$ do
 3.     Define $C = C(g_1(K_1), g_2(K_2))$ and $A = A(g_3(K_3), g_4(K_4))$.
 4.     If $g_1 = g_3$ and $g_2 = g_4$ and
              $K_1, K_2$ are unifiable with $K_3, K_4$ respectively then
 5.         Assign $\theta = (K_3/K_1)(K_4/K_2)$.
 6.         If $C \models A\theta$ or $C \models \neg A\theta$ then return $\theta$.
 7.     End-if
 8. End-for
 9. Exit with failure.
```

Figure 5.5: Unifying constraints in Token-Datalog.

5.2.3 Problem: TCSP-Datalog Interval Fusion

Constraint unification is not sufficient to provide correct answers. To illustrate the problem that is still unresolved, consider the statements "Mary was located in Arrowhead from 1977 to 1986, when she moved to San Francisco. John was located in Arrowhead from 1980 to 1987, when he joined Mary in San Francisco." As we show below, in many cases standard resolution algorithms do not allow inferring that both Mary and John were in the same city simultaneously, namely that "both John and Marry were staying in Arrowhead simultaneously".

A possible TCSP-Datalog formalization of the above example is as follows:

```
Location(Marry, Arrowhead, I1).         1977{starts}I1.  1986{ends}I1.
Location(John, Arrowhead, I2).          1980{starts}I2.  1987{ends}I2.
```

Inferring that "both John and Marry were staying in Arrowhead simultaneously" accounts for deciding the entailment of the query

```
:- Location(John, Arrowhead, I), Location(Marry, Arrowhead, I)
```

Using standard resolution we cannot prove the entailment of this query because there is no interval I for which `Location(Marry,Arrowhead,I)` and `Location(John,Arrowhead,I)` are both in the database. To correctly decide entailment, we need to introduce the substitution θ=$(I_1/I)(I_2/I)$. Unfortunately, in standard resolution, this substitution enforces $I=I_1=I_2$. Clearly this is an undesirable side effect that disqualifies θ from being used in an SLD-refutation.

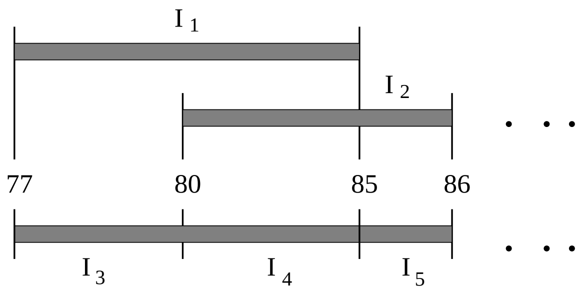


Figure 5.6: Interval fusion.

5.2.4 Solution: TCSP-Datalog Interval Fusion Algorithm

We introduce temporal fusion, which remedies this problem by removing the undesired side effect of applying the above θ. This is achieved by introducing the new intervals $\{\mathtt{I}_3,\mathtt{I}_4,\mathtt{I}_5\}$ which are the fusion of $\{\mathtt{I}_1,\mathtt{I}_2\}$. These intervals have the following end points (illustrated in figure 5.6):

$$\begin{array}{ll}\mathtt{1977\{starts\}I_3.} & \mathtt{1980\{ends\}I_3.}\\ \mathtt{1980\{starts\}I_4.} & \mathtt{1985\{ends\}I_4.}\\ \mathtt{1985\{starts\}I_5.} & \mathtt{1986\{ends\}I_5.}\end{array}$$

Definition 26: Given a program Ψ and its constraint facts $\mathcal{C}_U$, two constraint terms are *fusible* iff $\mathcal{C}_U$ entails that they overlap, namely there are three possible cases:

1. If $\mathtt{t}_1,\mathtt{t}_2$ are point constraint terms and $\mathcal{C}_U \models \mathtt{t}_1\mathtt{=}\mathtt{t}_2$ then $\mathtt{t}_1,\mathtt{t}_2$ are fusible.
2. If `t` is a point constraint term while `I` is an interval constraint term and $\mathcal{C}_U \models \mathtt{t\{during\}I}$, then `t` and `I` are fusible.
3. If both $\mathtt{I}_1,\mathtt{I}_2$ are interval constraint terms and

$$\mathcal{C}_U \models I_1 \left\{ \begin{array}{c} \mathtt{start},\ \mathtt{overlaps},\ \mathtt{during},\ \mathtt{equals},\ \mathtt{finishes}, \\ \mathtt{started\_by},\ \mathtt{overlapped\_by},\ \mathtt{contains},\ \mathtt{finished\_by} \end{array} \right\} I_2$$

then $\mathtt{I}_1,\mathtt{I}_2$ are fusible.

The key operation in temporal fusion is the processing of a set of linearly ordered points, or a set of intervals whose end points are linearly ordered. The fusion operator takes, as input, a set of constraint terms and computes, as output, a new set of constraint terms which is the fusion of the input terms.

Definition 27: Let $\mathcal{P}$ be a set of linearly ordered points, $\mathcal{I}$ be a set of intervals whose end points together with the point in $\mathcal{P}$ are *linearly ordered*, and let $E = \{e_1, \ldots, e_n\}$ be the linearly ordered set comprising of $\mathcal{P}$ and the end points of the intervals in $\mathcal{I}$. The fusion of the set $\mathcal{S} = \mathcal{P} \cup \mathcal{I}$, denoted $fusion(\mathcal{S})$, is a set of intervals $\mathtt{I}_1,\ldots,\mathtt{I}_n$ satisfying $\mathtt{I}_i\mathtt{\{meets\}}\mathtt{I}_{i+1}$ and $\mathtt{e}_i\mathtt{\{starts\}}\mathtt{I}_i$, $\mathtt{e}_i\mathtt{\{ends\}}\mathtt{I}_{i-1}$.

We modify the set of facts in the database to perform temporal fusion using algorithm TCSP-TemporalFusion, shown in Figure 5.7. The input consists of two sets of facts: (i) facts of the form `P(`$\mathtt{a}_1$`,...,`$\mathtt{a}_r$`,t)` where `t` is a point constraint term, and (ii) facts of the form `P(`$\mathtt{a}_1$`,...,`$\mathtt{a}_r$`,I)` where `I` is an interval constraint term. The first step requires testing whether $\mathcal{C}_U$[1] entails a linear order over all the point constraint terms and the end points of the interval constraint terms. The second step computes the fusion set $\mathtt{I}_1,\ldots,\mathtt{I}_n$ satisfying $\mathtt{I}_i\{\mathtt{meets}\}\mathtt{I}_{i+1}$ and $\mathtt{e}_i\{\mathtt{starts}\}\mathtt{I}_i$, $\mathtt{e}_i\{\mathtt{ends}\}\mathtt{I}_{i-1}$. Finally, the third step, generates the set of facts that are *true* for each of $\mathtt{I}_1,\ldots,\mathtt{I}_n$ and $\mathtt{e}_1,\ldots,\mathtt{e}_{n+1}$.

Example 38: To continue with the above example, the input to the temporal fusion algorithm is the following set of facts:

```
Located(Marry, Arrowhead, I1).          1977{starts}I1.  1986{ends}I1.
Located(John, Arrowhead, I2).           1980{starts}I2.  1987{ends}I2.
```

The output of the temporal fusion algorithm consists of the following set of facts:

```
Located(Marry, Arrowhead, I3).  Located(Marry, Arrowhead, I4).
Located(John, Arrowhead, I4).   Located(John, Arrowhead, I5).

1977{starts}I3.  1980{ends}I3.
1980{starts}I4.  1985{ends}I4.
1985{starts}I5.  1986{ends}I5.
```

Lemma 12: *Algorithm* `TCSP-Temporal-Fusion` *computes, in linear time, a set of facts which is equivalent to its input (with respect to the temporal semantics).*

Proof: The complexity is derived as follows. Step 2 can be implemented by topological sort, which requires $O(nc)$ steps where n is the number of variable and c is the number of constraints in $\mathcal{C}_U^{min}$. The loop from 5 to 8 terminates in $O(nm)$ where n is the number of end-points and m is the number of facts in the input database. Thus, the total complexity is $O(nc + nm)$.

Equivalence is achieved because (i) a fact is in the output if it is entailed by some input fact through homogeneity, and (ii) every input fact is mapped to at least one output fact. The first part of this claim follows immediately from the conditions in lines 6,7. The equivalence follows from the fact that $\mathtt{e}_1,\ldots,\mathtt{e}_{n+1}$ contain *all* the end-points. □

[1]This is the set of unit clauses containing constraint atoms.

```
Algorithm TCSP-Temporal-Fusion
Input:   A set of facts of the form P(a_1,...,a_r,t) where t is a point constraint term
         A set of facts of the form P(a_1,...,a_r,I) where I is an interval constraint term
         A consistent TCSP $\mathcal{C}_U$ over all the end-points and its equivalent minimal network $\mathcal{C}_U^{min}$.
Output:  An equivalent database $D$ such that each fact is mapped to a single input fact.
 1. Let e_1,...,e_{n+1} be all the end-points.
 2. Compute an order e_1,...,e_{n+1} such that $\forall C(e_i,e_j) \in \mathcal{C}_U^{min}$, (C(e_i,e_j) $\models$ e_i $\prec$ e_j) iff $i < j$
        (i.e. topological sort).
 3. if there is no linear order with the desired property then exit with failure.
 4. Initialize $D = \{\}$.
 5. for every $i$ do
 6.    Let I be the interval [e_i,e_{i+1}].
 7.    if there exists an input fact P(a_1,...,a_r,I) such that I{contains,started_by,finished_by}I_i
       then add the fact P(I,a_1,...,a_r) to the output database $D$.
 8.    if there exists an input fact P(a_1,...,a_r,t) such that t=e_i
       then add the fact P(e_i,a_1,...,a_r) to the output database $D$.
 9. end-for
10. return D.
```

Figure 5.7: Performing Temporal Fusion in TCSP-Datalog.

5.2.5 Problem: Token-Datalog Interval Fusion

A possible Token-Datalog formalization of the above example is as follows:

```
Located(Marry, Arrowhead, K1).          begin(K1)=1977.  end(K1)=1986.
Located(John, Arrowhead, K2).           begin(K2)=1980.  end(K2)=1987.
```

Inferring that "both John and Marry were staying in Arrowhead simultaneously" is equivalent to deciding the entailment of the query

```
:- GetMarried(John, K), BuyHouse(John, K)
```

5.2.6 Solution: Token-Datalog Interval Fusion Algorithm

The Token-Datalog fusion algorithm differs from the TCSP-Datalog fusion algorithm in that it fuses token terms.

Example 39: Similar to TCSP-Datalog, we introduce the interval tokens $\{K_3, K_4, K_5\}$ which are the fusion of $\{K_1, K_2\}$. These intervals have the following end points (illustrated in figure 5.6):

```
begin(K₃)=1977.  end(K₃)=1980.
begin(k₄)=1980.  end(K₄)=1985.
begin(K₅)=1985.  end(K₅)=1986.
```

Example 40: To continue with Example 32, the temporal constraints entail the constraint `time(next`$^{-1}$`(k`$_1$`))=time(k`$_0$`)`. This equality renders the ground terms `next`$^{-1}$`(k`$_1$`)` and `k`$_0$ fusible. Let `k'` be their fusion and assign `medicine(k')` to *true*. Then the substitution θ_3 `= (next`$^{-1}$`(k`$_1$`)/k')(k`$_0$`/k')` transforms

`medicine(k`$_0$`)` and
`:- medicine(next`$^{-1}$`(k`$_1$`)), time(next`$^{-1}$`(k`$_1$`)`$\in$`[0,120]).`

into

`medicine(k')` and `:-medicine(k'), time(k')`$\in$`[0,120].`

This enables resolving the two clauses together. Since we have derived the constraint `time(next`$^{-1}$`(k`$_1$`))=time(k')=0`, the atom `time(next`$^{-1}$`(k'))`$\in$`[0,120]` is entailed. As a result, the proof of the query is completed.

Lemma 13: *Let Ψ be a program whose $\mathcal{C}_U$[2] induces a linear order on the end points of* all *the token terms in Ψ, and let $\mathcal{K}$ be the set of token terms occurring in Ψ.*

1. *The truth value of the predicates in Ψ may change only at the end points of the token terms in $fusion(\mathcal{K})$ (see Example 38).*
2. *Computing $fusion(\mathcal{K})$ and the set of predicates that evaluate to true for each token in $fusion(\mathcal{K})$ requires $O(n(n+m))$ where n is the number of tokens in $\mathcal{K}$ and m is the total number of predicates.*

Proof: For Part 1 see the proof for TCSP-Datalog. The truth value of the non-constraint predicates that take token arguments does not change during the interval associated with the token argument. Thus, the truth value may change only at the end-points of the input tokens $\mathcal{K}$. Because every end-point of the tokens in $\mathcal{K}$ is an end-point of at least one (and at most two) tokens in $fusion(\mathcal{K})$, the thesis follows. $\square$

5.3 Completeness

In this section we show that the combination of the two algorithms, constraint unification and token-fusion, is complete for making inferences with Token-Datalog. Because TCSP-Datalog is strictly less expressive, this result is applicable to TCSP-Datalog

[2]All unit clauses containing constraint atoms (recall Definition 24).

as well. For clarity of presentation, we consider three classes of languages that have increasing levels of expressiveness and complexity:

- $\mathcal{L}_0$ is a subclass of Token-Datalog having no constraint atoms and all the token terms are ground.
- $\mathcal{L}_1$ extends $\mathcal{L}_0$ by allowing constraint atoms (token terms are ground).
- $\mathcal{L}_2$ is the general Token-Datalog which extends $\mathcal{L}_1$ by allowing non-ground token terms.

For all three $\mathcal{L}_0$, $\mathcal{L}_1$, $\mathcal{L}_2$ we use the combined TCSP as the temporal constraint component.
Let R be a generalized resolution algorithm, let R^c be R augmented with constraint unification, let R^k be R augmented with token fusion and let R^{kc} be R augmented with both. The algorithms R, R^c and R^{kc} are all non-deterministic in that, at every step, they may apply one of several possible operations. Algorithms R^c and R^k differ from R in that they can apply all the operations that R applies, with the addition of constraint unification and token fusion respectively. R^{kc} modifies line 4 in algorithm *UnifyConstraints* to include token fusion, namely the unification of $\mathtt{K}_1,\mathtt{K}_2$ with $\mathtt{K}_3,\mathtt{K}_4$ when $\mathtt{K}_1,\mathtt{K}_2,\mathtt{K}_3,\mathtt{K}_4$ are all ground token terms. Clearly, if R detects the inconsistency of a program Ψ then R^c, R^k and R^{kc} can also detect this inconsistency. This is because they have a larger set of operations to apply. We next formalize these algorithms.

Definition 28: An R-derivation is a standard SLD-derivation. An R^c-derivation is such that every step may be either standard SLD-derivation step or a *constraint unification* step. An R^k-derivation is such that every step may be either standard SLD-derivation step or a *token fusion* step. An R^{kc} derivation is such that every step may be either standard SLD-derivation step, *constraint unification* or *token fusion* step. An R-refutation is a standard SLD-refutation; R^c,R^k and R^{kc}-refutations are R^c,R^k and R^{kc}-derivations containing either an empty clause or an *empty constraint*.

Lemma 14: *If algorithm R is complete for $\mathcal{L}_0$ then algorithm R^c is sound and complete for $\mathcal{L}_1$.*

Proof: Assume R is complete for $\mathcal{L}_0$ and R^c is incomplete for $\mathcal{L}_1$. If R^c is incomplete for $\mathcal{L}_1$, then there exists an *inconsistent* program Ψ in $\mathcal{L}_1$ for which does not exist an R^c-refutation. To prove, by contradiction, that this assumption is *false*, we divide the atoms in Ψ into two partitions: constraint and non-constraint atoms. Let Ψ_c is the result of removing all constraint atoms from Ψ. Because $\Psi_c \in \mathcal{L}_0$, if Ψ_c is inconsistent then there exists an R-refutation for Ψ_c, and thus exists an R^c-refutation for Ψ_c.
Case 1: If the conjunction of the constraints described by *all* the constraint atoms in Ψ is satisfiable, then Ψ is consistent iff Ψ_c is consistent. This contradicts the assumption, concluding that R^c is complete for Case 1.

Case 2: If the conjunction of the constraints described by the *constraint facts* in Ψ is unsatisfiable, then their inconsistency is detected as follows. Given a TCSP-derivation λ, let $\mathcal{G}(\lambda) = G_1, \ldots, G_n$ be the set of all subgoals generated by λ (recall Figure 5.3). Before extending λ with another step, a TCSP N is generated from all the subgoals in $\mathcal{G}(\lambda)$ which contain constraints. Subsequently, the minimal network N_{min} is computed. A constraint C is entailed by N iff exists a substitution θ such that $N_{min}\theta \models C\theta$. From Lemma 11 we know that, if a constraint C is entailed by N_{min}, the desired substitution θ is guaranteed to be found. In particular, this claim is true in case θ enables inferring the empty constraint, which means that N_{min} is inconsistent, implying that N is inconsistent and thus Ψ is inconsistent. Adding the substitution θ to λ converts it into an R^c-refutation (for Ψ). This contradicts the assumption that no such refutation exists, concluding that R^c is complete for Case 2.
Case 3: In all other cases, Ψ is inconsistent due to the assertion of some constraint atom C. In this case, there exists a TCSP-derivation in which C is added as a *sub-goal*. In subsequent derivation steps, this subgoal, and all previous subgoals, are treated as *facts*. This satisfies the precondition of Case 2 above and enables constructing an R^c-refutation. This contradicts the assumption that no such refutation exists, concluding that R^c is complete for Case 3. $\square$

Lemma 15: *If algorithm R is complete for $\mathcal{L}_0$ then R^{kc} is complete and for every program $\Psi \in \mathcal{L}_2$.*

Proof: We claim that if R^{kc} terminates without identifying an inconsistency then the following holds: for every interval token term `k` and point token `k'`, if $\Psi \models$ `p(k)`$\wedge$ `(begin(k)`$<$ `time(k')`$<$ `end(k))` then $\Psi \models$`p(k')`. This claim is equivalent to the above lemma because it implies that the token-fusion algorithm enables to make all the inferences that should be made under the temporal semantics given in Chapter 4, Definition 18, having one condition: holding over a time interval implies holding at any time point within this interval.

To prove by contradiction, assume R^{kc} did not identify inconsistency and there exists a ground interval token term k in T^{ω}_{Ψ} such that $\Psi \models$ `p(k)`$\wedge$ `(begin(k)`$<$ `time(k')`$<$ `end(k))` and $\Psi \not\models$`p(k')`. This implies that exists a set of subgoals $\mathcal{G} = G_1, \ldots, G_n$ such that $\mathcal{G} \not\models p(k')$ yet $\mathcal{G} \models p(k)$. As shown in Lemma 13, described in Figure 5.7 (similar for both TCSP-Datalog and Token-Datalog) and illustrated in Example 40, R^{kc} is guarantees to find a substitution which unifies `k',k` with a ground token `k''`. This grounds the token terms and creates formula in $\mathcal{L}_1$, for which exists an R^c-refutation. Thus, inconsistency can be detected, leading to a contradiction and proving the intermediate claim. $\square$

Recall that without token fusion we might not find a ground token term `k'` that enables us to detect the inconsistency.

Theorem 19: *Algorithm R^{kc}, namely resolution augmented with constraint unification and token fusion, is complete for Token-Datalog.*

Proof: Follows from Lemmas 14,15. □

Consider algorithm Simple-R^{kc} which is equivalent to algorithm R^{kc} where Simple TCSP-Derivations are used instead of general TCSP-Derivations. Let Simple Token-Datalog be a fragment of Token-Datalog in which the constraints form a class of constraints for which enforcing path-consistency is complete for deciding entailment (see Chapters 2,3).

Theorem 20: *Algorithm Simple-R^{kc} is complete for Simple Token-Datalog.*

Proof: The proofs of Lemmas 14,15 rely on the completeness of constraint unification when deciding constraint entailment. In case entailment can be decided using path-consistency, entailment can be proved using a sequence of `Revise(i,j,k)` operations. This sequence can be converted into a Simple-TCSP-derivation which contains the empty constraint, namely a Simple-TCSP-refutation. □

Note, however, that Simple Token-Datalog may not be tractable because it allows describing periodic yet infinite sequences.

5.4 Comparing TCSP-Datalog and Token-Datalog

In this section, we explain the differences between TCSP-Datalog and Token-Datalog. Token-Datalog is more expressive than TCSP-Datalog, yet making inference in Token-Datalog is more complicated.

5.4.1 Syntax

There are two major syntactic differences between TCSP-Datalog and Token-Datalog: (i) the temporal qualification method and (ii) the use of functions.

TCSP-Datalog uses the temporal argument qualification method while Toke-Datalog uses the token argument qualification method. To illustrate the syntactic difference between these two temporal qualification methods, consider representing the statement "John was in LA from Jan 15 to Feb 15". When using temporal arguments, this statement can be described with the proposition

```
Location(John, LA, Jan15, Feb15).
```

Using token arguments, this statement can be described with the conjunction

```
Location(John, LA, k), begin(k)=Jan15, end(k)=Feb15.
```

TCSP-Datalog does not allow functions at all. Token-Datalog allows two types of functions: (i) mapping token terms to constraint terms, and (ii) mapping token terms to token terms.

In Token-Datalog, given a point token `k`, the function `time(k)` maps `k` into a constraint term representing a time point. Given an interval token `k`, the functions

```
interval(k),begin(k),end(k)
```

map `k` into constraint terms representing a time interval, the time point starting the interval and the time point ending the interval, respectively. In TCSP-Datalog, given an interval term `I`, its starting and ending points are described by point terms P_1,P_2 which are constrained to `I` with the constraint atoms:

$$P_1\{\texttt{starts}\}\ I, \quad P_2\{\texttt{ends}\}\ I.$$

Token-Datalog functions mapping token terms to token terms using *successor functions* have no parallel in TCSP-Datalog. This mapping does not require known the time point or interval associated with tokens. For example, the fact

```
time(K)-time(prev(K))=8
```

implies that the successor of `K` is a token which occurred before `K`. In contrast,

```
time(next(K))-time(K)=8
```

goes forward in time. Neither sequences are constrained to specific time points.

Successor functions can also be used in a more general form. For example, the statement "the medicine needs to be taken every 7-9 hours" can be represented by the program Ψ =

```
time(next(k))-time(k)∈[7,9].
Administrate( Medicine, next(k)) :- Administrate( Medicine, k).
```

where k_1,k_2 are point token terms, `Medicine` is a data constant and `take` is a predicate symbol.

5.4.2 Semantics

While a TCSP-Datalog maps every time point `t` to a set of ground atoms (i.e. facts) that evaluate to *true* at `t`, Token-Datalog maps every token term `k` to a set of ground atoms. To illustrate the difference, consider the statement "the party took place one week prior to the 200th independence day". In Token-Datalog, this statement can be represented by the conjunction

```
Occur(Party,k), time(independence200)-time(k)=7days.
```

where the token symbol `independence200` represents the time of the 200 independence day *without specifying the exact date.* Representing this statement in TCSP-

Datalog *requires specifying the exact point in time* that the 200th independence day occurred. If t be the required date then the equivalent TCSP-Datalog conjunction is

```
Occur(Party,t'), t'-t=7days.
```

5.4.3 Inference Algorithm

The unification algorithm for TCSP-Datalog and Token-Datalog are slightly different due to the syntactic differences between these languages. For example, consider the Token-Datalog database

Location(John, LA, k$_1$), begin(k$_1$)=Jan15, end(k$_1$)=Feb15,
Location(Mary, LA, k$_2$), begin(k$_2$)=Jan1, end(k$_2$)=Feb1.

According to the semantics of Token-Datalog, this conjunction entails the conjunction

```
Location(John,LA,k),Location(Mary,LA,k)
```

which means that both John and Mary were in LA at the same time. The grounding of `k` reveals that the time interval was between Jan15 and Feb1. As shown above, using standard resolution such inferences cannot be made. This problem was addressed by introducing Token-Fusion.

The equivalent TCSP-Datalog would be

```
Location(John, LA, Jan15, Feb15),   Location(Mary, LA, Jan1, Feb1)
```

and the equivalent query is

Location(John,LA,t$_1$,t$_2$),Location(Mary,LA,t$_1$,t$_2$)

where t$_1$=Jan15 and t$_2$=Feb1.

5.5 Relationship to CLP

5.5.1 Syntax

Similarities: It is possible to develop a CLP language using TCSP as a constraint domain. Our approach is more specific, as it focuses on the combination of Datalog with TCSP. In order to generalize our results and render the combination with general CLP useful for temporal reasoning, the issues presented in Chapter 4 need to be addressed and the inference algorithm need to be modified as described in Chapter 5.

Differences: While in our languages constraint atoms in the heads of rules are allowed, in CLP languages, constraint atoms in the heads of rules are not allowed. While constraint facts in our languages are unit clauses specifying a constraint atoms, a CLP fact is of the form `a :- c` where `c` is a constraint atom and `a` is a non-constraint atom.

5.5.2 Semantics

Similarities: The languages we propose as well as CLP languages have a fixed point semantics.

Differences: The temporal semantics of our languages cannot be implemented using existing CLP languages. Once the syntax of CLP is extended with the temporal qualification methods used by TCSP-Datalog and Token-Datalog, there is a need to enhance the semantics to accommodate (i) constraints in the heads of rules, (ii) homogeneity of holding. While many more issues may need to be addressed, solution to these issues were described in this work.

Inference: The major difference is the unification algorithm. To generalize our results to the CLP framework, there is a need to augment the existing resolution algorithms with the constraint unification and token fusion algorithms introduced in this work. These algorithms improve the state of the art in that the variables need not be ground in order for constraint propagation to occur.

5.5.3 Inference Algorithm

Similarities: Inferences in our languages as well as for CLP are made through a resolution algorithm.

Differences: In our languages, the resolution algorithm is modified to be consistent with the temporal semantics we defined. This new algorithm, which is capable of unifying constraints and fusing time entities, is called *temporal resolution*, which is based on TCSP-derivations and refutations. In contrast, because value assignment to CLP constraints variables are not included in the interpretation, the constraint solver in CLP is external to the resolution algorithm. Consequently, many constraint operations cannot be performed.

5.6 Conclusion

In this chapter we identified and resolved issues that arise when performing temporal reasoning using resolution. We focused on the problems that render resolution incomplete for TCSP-Datalog and Token-Datalog.

The two major difficulties are: (i) the need to unify ground temporal and token terms, and (ii) the need to unify constraints. To address these problems we introduced the *token fusion* and *constraint unification* methods. We have shown that augmenting the standard resolution algorithm with those methods renders it sufficient for making the intended inferences.

Our results render the logic programming paradigm a viable knowledge representation approach temporal reasoning. For example, our results could be used to extend a specific class of logic programs, Constraint Logic Programs (CLP), to make the inferences which intended under the temporal semantics by augmenting its inference engine with constraint unification and token fusion.

Chapter 6

Example Domains

Chapters 4 and 5 show that temporal reasoning requires extending the traditional inference engine, *resolution*, to allow unification of constraints and ground temporal terms. Our hypothesis is that, no matter which application domain is of interest, these extensions are likely to be needed whenever temporal reasoning is embedded within a logic programming framework. To support this hypothesis, we demonstrate the utility of modifying the inference algorithm when reasoning about knowledge in various domains using Token-Datalog. We extend the examples presented throughout this thesis for the domains of:

1. Formalizing medical guidelines and treatment plans. Such a formalization enables verifying whether a given treatment plan complies with the guidelines.

2. Formalizing electrical circuits and the intended specification. Such a formalization enables verifying whether a given circuit complies with the specifications. If the specifications are not met, bugs are identified by presenting counter examples.

3. Formalizing the flow of inventory within a warehouses and manufacturing organizations. The challenge is to make inferences even when the information about the time certain changes occurred is unknown.

4. Formalizing common sense reasoning is a difficult task which is still unresolved. Here we demonstrate how our languages can be used to describe the effects of actions in a fashion similar to traditional methods. The novelty is in the ability to answer queries about possible relation between objects. These queries could not be answered using standard first-order logic programs.

6.1 Treatment Plan Verification

A task common to many application domains is the analysis of data accumulated over time, leading to identification of past and present trends and to episodic decisions

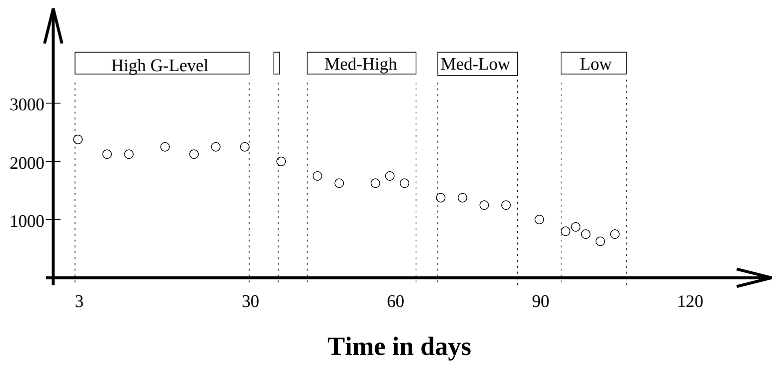


Figure 6.1: Temporal abstraction and interpolating missing values.

made on the basis of the previous and the current state of the world. An example of such a task in the medical domain is managing patients who are being treated with clinical guidelines [94]. An inherent requirement of such tasks is to accumulate and analyze patient data over time and to constantly revise an assessment of the patient's state.

These higher-level concepts can be used for summarizing large medical databases, for monitoring, for replanning therapy, for providing explanations to a user of a decision-support system, and as a basis for a more intelligent dialog between an automated decision-support system and a human health-care provider.

Physicians are often required to make diagnostic or therapeutic decisions based on numerous time-stamped patient data. These data may be overwhelming, particularly if the physicians' ability to *reason* with the data does not scale up to the amount of data collected. Most clinical data include a time stamp indicating when each datum was valid. Temporal abstraction of the data was proposed to deal with the large volume of clinical data [94, 95, 96]. Time oriented abstraction highlight meaningful trends and patterns, save the physician valuable time, and supports an intelligent decision-support patient-record system.

Temporal abstraction is the conversion of quantitative time-stamped data into propositions describing a qualitative state which persists over time intervals. Consider measuring Granulocyte counts and describing the results in a graph, as illustrated in Figure 6.1. The abstraction may specify when the Granulocyte count is under 2000, between 2000-2500, between 2500-3000, and greater than 3000. In Figure 6.1 four intervals were generated, describing time periods during which the count was at different qualitative levels. In addition, an isolated time point may be used describe data which should not contribute to generating the trend (e.g. because they might be errornous).

Each abstraction interval can be modeled as a token. The list of propositions that hold throughout the intervals associated with these tokens describe the qualitative state of the patient. In Token-Datalog, the holding of an abstraction "Granulocyte count is Low" at a point token $\mathtt{k}_1$ can be formalized using the atom $\mathtt{GLevel}_{AT}$(Low, `k`$_1$`)`. The holding over an interval token `k`$_2$ is represented by `GLevel`$_{ON}$`(Low, k`$_2$`)`. We perform abstraction using the atom `GCountLesserThan(2000, k`$_1$`)`, which is *true* if the Granulocyte count is less than 2000 at the time point `time(k`$_1$`)`. This abstraction can be defined by the rule:

```
GLevel_AT(Low, k_1) :- GCountLesserThan(2000, k_1).
```

To describe the holding of this abstraction over an interval we need yet another rule, which encodes the homogeneity axiom as follows:

```
GCountLesserThan_AT(2000, k_1) :-  GLevel_ON(Low, k_2),
                                   time(k_1) {during} interval(k_2).
```

This ensures that abstracting over the interval implies abstracting for every time point within the interval. Note that the homogeneous holding is different than homogeneous abstractions. While homogeneous holding is encoded within the semantics, the abstraction must be described explicitly using rules expressible in the language. In Figure 6.1, the decreasing length of these intervals indicates a trend. This trend could not be identified without having created the temporal abstraction such as `GLevel`$_{ON}$`(Low, k`$_2$`)`.

The problem here is that the above two rules do not enable inferring that if some points within an interval satisfy the abstraction `GCountLesserThan`$_{AT}$`(2000, k`$_1$`)`, then this is a license to infer the holding of the abstraction `GLevel`$_{ON}$`(Low, k`$_2$`)` over the whole interval. This is called the *truth value interpolation problem* [94, 95, 96].

To formalize the conditions under which the *truth value interpolation* can be made, we could formalize statements such as "if the abstraction held for n points where each pair is most two days apart then the point abstractions can be converted into an interval abstraction that subsumes all these points. In other words, if for $i \in 1, \ldots, n$ `GLevel`$_{AT}$`(Low, k`$_i$`)`, `time(k`$_i$`)-time(k`$_{i-1}$`)`$\leq$ `2` where `k`$_{i+1}$`=next(k`$_i$`)`, then we can infer that `GLevel`$_{ON}$`(Low, k`$_{all}$`)` where `k`$_{all}$ subsumes all the point tokens `k`$_1$`,...,k`$_n$, namely $\forall i$, `time(k`$_i$`) {during} interval(k`$_{all}$`)`. This inference can be performed in a bottom-up fashion, starting from pairs of tokens, `k`$_i$ and `next(k`$_i$`)`, with the following Token-Datalog rule:

```
GLevel_ON(Low, k_c),
time(next(k_i))=end(k_c),
time(k_i)=start(k_c)        :-  time(next(k_i))-time(k_i)≤2,
                                GLevel_AT(Low, k_i),
                                GLevel_AT(Low, next(k_i)).
```

and, an additional rule is necessary to specify that the truth value of the predicate `GLevel` is concatenatable:

```
GLevel_ON(X, k_c),
interval(k_p) starts interval(k_c),
interval(k_q) finishes interval(k_c)  :-  interval(p) meets interval(q),
                                          GLevel_ON(X, k_p),
                                          GLevel_ON(X, k_q).
```

where X stands for an arbitrary level (e.g. Low, Medium, High or VeryHigh). Similar rules can be used for the abstraction levels where the Granulocyte count is Medium between 2000,2500, or High between 2500,3000 or Very High if it is greater than 3000.

A typical reasoning task in this domain involves verifying whether the treatment history complies with the protocol under which the treatment is given. For example, the guideline might require that "Medicine A should be administered if the Granulocyte count increases beyond 2500, and Medicine B should be given when the count is greater than 3000, both no more than two days after the measurement was taken. These requirements could be described by the Token-Datalog rules:

```
Administer(MedicineA, k_2), time(k_2)-end(k_1)≤2 :- GLevel_ON(High, k_1)
Administer(MedicineA, k_2), time(k_2)-end(k_1)≤2 :- GLevel_ON(VeryHigh, k_1)
```

Note that the constraint `time(`k_2`)-end(`k_1`)` is not placed in the body. This is because the token k_2 need not be generated unless the atom in the body is *true*. Referring to k_2 in the body will create problems as the body cannot evaluate to *true* in case k_2 was not yet created.

Next, we illustrate the need to perform constraint unification and how it works. Assume we are given two Granulocyte counts measured at time point k_1,k_2 set to be one day apart, namely `time(`k_2`)-time(`k_1`)=1`. Could we infer that the this fact entails the query `:- time(`k_2`)-time(`k_1`)`$\leq$2? This inference is necessary to enable *interpolating*[1] the truth value of the `GLevel` predicate. Constraint unification enables unifying a query atom Q with a fact F if the constraint described by F, denoted C_F, entails the constraint defined by Q, denoted C_Q, namely $C_F \models C_Q$.

Next, we illustrate the need for token fusion and how it works. Assume that, in addition to the Granulocyte count parameter, we have parameter called *Bone Marrow Toxicity* level, we abbreviate as BMT. Consider a guideline stating that "In case both the Granulocyte and BMT level were found to be high, Medicine C must be administered at most 24 hours after this measurement. Suppose that the database contains two facts, `GLevel`$_{ON}$`(High,`k_1`)` and `BMTLevel`$_{ON}$`(High,`k_2`)` where `interval(`k_1`)=[10,16]` and `interval(`k_2`)=[15,17]`. To determine the need for administering Medicine C, there is a need to generate a new token k_3, where `interval(`k_3`)=[15,16]`, and infer both `GLevel`$_{ON}$`(High,`k_3`)` and `BMTLevel`$_{ON}$`(High,`k_3`)`.

[1]i.e. guessing the value at a time point which is between two other time points for which the truth value is known.

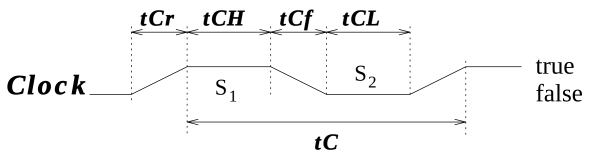


Figure 6.2: Clock timing specification.

6.2 VLSI Verification

The digital circuit verification task involves deciding whether a circuit description (in some procedural language or Finite State Automaton (FSA)) satisfies timing constraints (described in some declarative language).

Consider a clock circuit. The procedural Finite State Automaton (FSA) circuit description is as follows: There are two states, S_1, S_2 where S_1 is the initial state. There are two transitions, $S_1 \rightarrow S_2$ and $S_2 \rightarrow S_1$. The circuit has a single output signal. When the circuit is in state S_1 its output signal is *true*, and in state S_2 its output signal is *false*.

The timing constraints are as follows: There are four events, E_1, E_2, E_3, E_4, where event E_1 begins the transition $S_2 \rightarrow S_1$, event E_2 ends the transition from $S_2 \rightarrow S_1$, event E_3 begins the transition $S_1 \mapsto S_2$ and event E_4 ends the transition $S_1 \rightarrow S_2$. The time at which an event E occurs is denoted $time(E)$. The timing constraints are as follows: [2]$tCr = time(E_2) - time(E_1)$, [3]$tCH = time(E_3) - time(E_2)$, [4]$tCf = time(E_4) - time(E_3)$, [5]$tCL = time(E_1) - time(E_4)$ and $tC = tCr + tCH + tCf + tCL$ (see Figure 6.2).

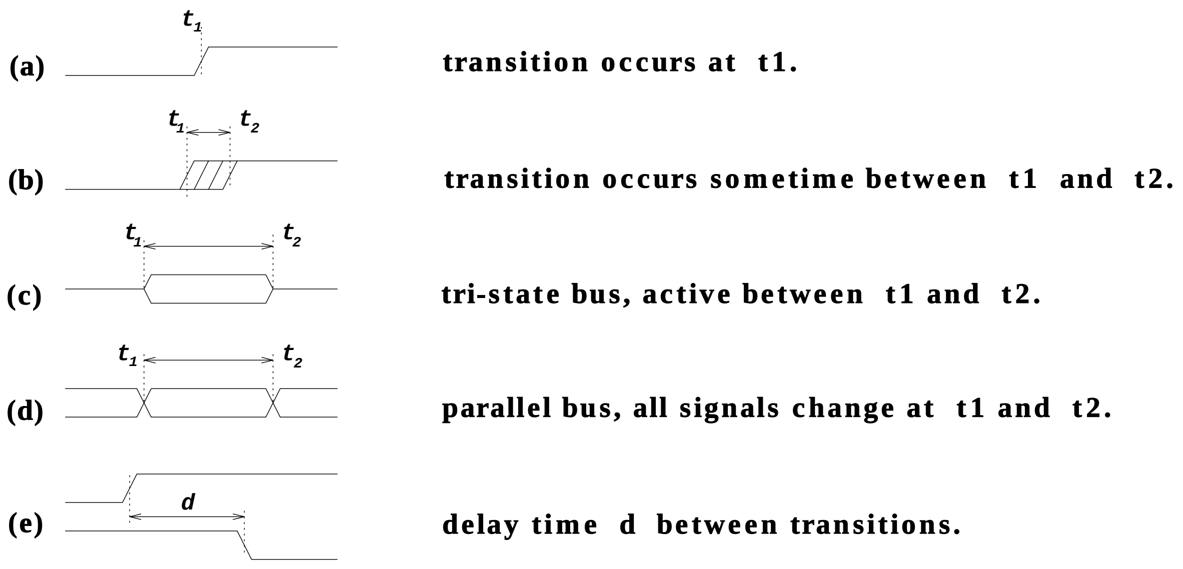
Timing specifications typically include the five types of transitions depicted in Figure 6.3:

(a) A signal transition that occurs at a time point v_1.
The signal changes value from *false* to *true*. This change requires some time. The transition point is defined as the middle time point between the transition beggining and end times.

(b) A simple transition that occurs within some interval $[t_1, t_2]$.
This specification is different from (a) in that a single time point is replaced by a time interval.

(c) An activation of a bus occurs during a time interval $[t_1, t_2]$.
A bus is a set of signals that collectively define an integer value (i.e. data). Busses typically have two states: active or passive. When a bus is active, it places data on the wires that physically construct the bus. When it is passive, it is *isolated* from the physical wires and has no impact on the values of

[2]tCr stands for "time clock rise", namely the time a low-to-high transision occurs.
[3]tCr stands for "time clock high".
[4]tCr stands for "time clock fall", namely the time a high-to-low transision occurs.
[5]tCr stands for "time clock low".



(a)
t_1
transition occurs at t1.
(b)
t_1
t_2
transition occurs sometime between t1 and t2.
(c)
t_1
t_2
tri-state bus, active between t1 and t2.
(d)
t_1
t_2
parallel bus, all signals change at t1 and t2.
(e)
d
delay time d between transitions.

Figure 6.3: The legend of timing specification.

signals. The transistion of a bus from passive to active state requires time. The time point v_1 is the middle time point between the beggining and end of this transition.

(d) A transition of all signals on an active bus at a time point t. When a bus is active, it may change the values of all its signals simultaneously. The transition point is defined as the middle between its beggining and end.

(e) A delay d (i.e. time difference) between two transitions of any kind.

Consider a simple microprocessor chip, such as Zilog Z80B, an eight-bit CPU, and a RAM chip, such as Toshiba 2015AP-90, a 2K static RAM[6]. We will focus on the timing specification of the data-write operation. Figure 6.4 shows the relevant ports of each chip, while Figures 6.5,6.6 provide the timing specification. In each of these figures the timing spec is given in terms of inequality constraints on the temporal variables. These constraints can be described by a restricted class of TCSPs calls Simple Temporal Problems (STP).

The specifications given in Figures 6.5,6.6 is described using 10 tokens, $\mathtt{k}_1$,...,$\mathtt{k}_1$0, and the following Token-Datalog constraint facts:

$$
\begin{array}{ll}
\texttt{(time(k}_2\texttt{)-time(k}_1\texttt{)} \in \texttt{[-}\infty\texttt{,90]).} & \texttt{(time(k}_3\texttt{)-time(k}_2\texttt{)}\in \texttt{[-}\infty\texttt{,130]).} \\
\texttt{(time(k}_5\texttt{)-time(k}_3\texttt{)}\in \texttt{[25,}\infty\texttt{]).} & \texttt{(time(k}_5\texttt{)-time(k}_4\texttt{)}\in \texttt{[70,}\infty\texttt{]).} \\
\texttt{(time(k}_7\texttt{)-time(k}_5\texttt{)}\in \texttt{[135,}\infty\texttt{]).} & \texttt{(time(k}_7\texttt{)-time(k}_6\texttt{)}\in \texttt{[-}\infty\texttt{,70]).} \\
\texttt{(time(k}_{10}\texttt{)-time(k}_8\texttt{)}\in \texttt{[-}\infty\texttt{,80]).} & \texttt{(time(k}_9\texttt{)-time(k}_8\texttt{)}\in \texttt{[-}\infty\texttt{,90]).}
\end{array}
$$

where 1 time unit is 10^{-9} seconds, also called *nano-seconds*.

[6]Modern chips have the same qualitative temporal characteristics as these obsolete chips

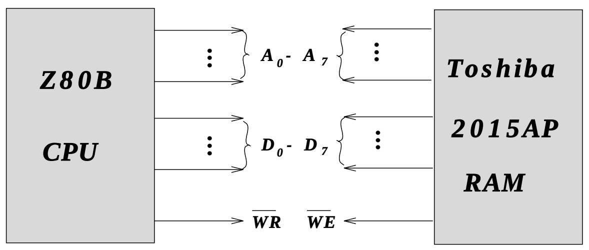
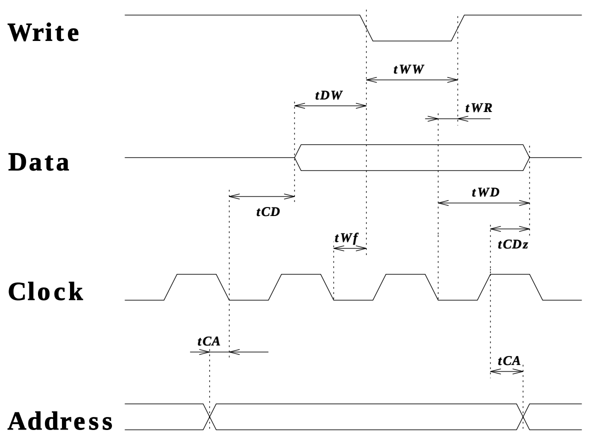
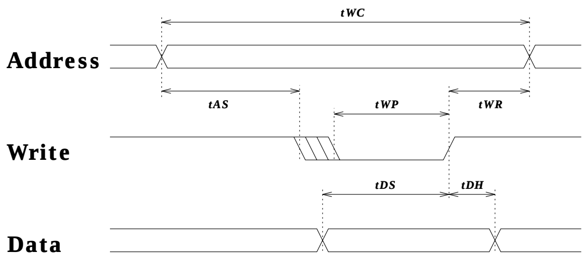


Figure 6.4: Connecting the Z80 CPU to the Toshiba 2015AP RAM.



$$\begin{array}{lcr}
tCA & \leq & 90 \\
tDW & \geq & 25 \\
tWf & \leq & 70 \\
tWW & \geq & 135 \\
tWr & \leq & 70 \\
tWd & \geq & 30 \\
tCDz & \leq & 80 \\
tCD & \leq & 130
\end{array}$$

Figure 6.5: Write cycle timing for Z80B CPU



$$\begin{array}{lcr}
tWC & \geq & 90 \\
tWR & \geq & 0 \\
tAS & \geq & 20 \\
tWP & \geq & 60 \\
tDS & \geq & 35 \\
tDH & \geq & 0
\end{array}$$

Figure 6.6: Write cycle timing for the Toshiba RAM

In the rest of this section we use `v,w` to denote two data terms indexing signal values, and `Eq`, `Neq` to denote the following propositional relations:

1. `Eq(v,w)` is satisfied iff v=w.
2. `Neq(v,w)` is satisfied iff v≠w.

Combinatorial constraints are represented as follows:

- `Same(a,b)` is satisfied iff for every time point, `a,b` have exactly the same value. It is expressed by the Token-Datalog rule

  ```
  interval(k1){disjoint}interval(k2) :-
            Neq(v1,v2), Value(a,v1,k1), Value(b,v2,k2)
  ```

- `Inv(a,b)` is satisfied iff for every time point, `a,b` have opposite values. It is expressed by the Token-Datalog rule

  ```
  interval(k1){disjoint}interval(k2) :-
            Eq(v1,v2), Value(a,v1,k1), Value(b,v2,k2)
  ```

- `MutEx(a,b)` is satisfied iff for every time point at most one of `a,b` is *true*. It is expressed by the Token-Datalog rule

  ```
  interval(k1){disjoint}interval(k2) :-
            Eq(v1,true), Eq(v2,true), Value(a,v1,k1), Value(b,v2,k2)
  ```

- `Or(a,b)` is satisfied iff for every time point at most one of `a,b` is *false*. It is expressed by the Token-Datalog rule

  ```
  interval(k1){disjoint}interval(k2) :-
            Eq(v1,false), Eq(v2,false), Value(a,v1,k1), Value(b,v2,k2)
  ```

Temporal dependence relationships between signals can also be described:

- `Implies(a,b,t1,t2)` is satisfied if signal `b` becomes *true* at least t_1 and at most t_2 seconds after signal `a` became *true*.
- `Eventually(a,b)` is satisfied if signal `b` becomes *true* at least once after signal `a` became *true*.

These are described as follows:

- `Implies(a,b,t1,t2)` is equivalent to the Token-Datalog rule

```
Value(b,true,f(k1)), begin(f(k1))-end(k1) ∈ [t1,t2]  :-
          Value(a,true,k2), Value(a,false,k1), end(k1)=begin(k2)
```

where `f` is a unary (successor) function mapping interval tokens to interval tokens.

- `Eventually(a,b)` is equivalent to `Implies(a,b,0,∞)`.

Next, we illustrate the need for constraint unification. Assume that the circuit at hand must react to a signal comming in at time $t = 0$ within 8 time units. Let `InSignal` be the input signal and `Response` be the output signal. Assume that, according to the lab tests, the circuit at hand satisfies Ψ_1 =

```
Value(Response, true, k2), start(k2)-start(k1)∈[2,6] :-
                                          Value(InSignal, true, k1)
```

The task is to verify whether the requirements, formalized as Ψ_2 =

```
Value(Response, true, k2), start(k2)-start(k1)∈[0,8] :-
                                          Value(InSignal, true, k1)
```

are satisfied. To test whether the required behavior encoded in Ψ_2 is entailed by the specifications encoded in Ψ_1, there is a need to test whether Ψ_1 entails Ψ_2. In this case, the answer is "Yes" because the tighter constraint `start(k2)-start(k1)∈[2,6]` entails the looser constraint `start(k2)-start(k1)∈[0,8]`. We can decide this entailment by unifying the two constraints and adding the tighter constraint as a fact.

Next, we illustrate the need for token fusion. Consider a circuit with three signals, `a,b,c` which must satisfy the constraint `Eq(c, a∧b)`. The behavior observed in practice is described the facts `Value(a, `*true*`, k1)`, `Value(b, `*true*`, k2)`, and `Value(c, `*true*`, k3)` where `interval(k1)=[2,7]`, `interval(k2)=[4,9]` and `interval(k3)=[4,8]`. By the closed world assumption, for every interval token `k` whose interval is not subsumed in `interval(k1)`, the atom `Value(a, `*true*`, k)` evaluates to *false*. Similarly, for every interval token `k` whose interval is not subsumed in `interval(k1)` or in `interval(k2)`, the atoms `Value(b, `*true*`, k)` and `Value(c, `*true*`, k)` evaluate to *false* respectively. As a result, the constraint `Eq(c, a∧b)` is not satisfied because during `interval(k)`, a=*false*, b=*true*, c=*true*. Detecting that the constraint is not satisfied requires identifying that during the interval [7,8] the constraint `Eq(c, a∧b)` is **not** satisfied. This can be done by generating a new token `k'` such that `interval(k')=[7,8]`. This operation is called *token fusion*. With this new token we can infer the facts `Value(a, `*false*`, k')`, `Value(b, `*true*`, k')`, and `Value(c, `*true*`, k')`, and thus `Eq(c, a∧b)` is not satisfied for `k'`.

6.3 Warehouse Management

We next demonstrate the ability of Token-Datalog to represent and reason about incompatibilities that exist in warehouse management systems, typically implemented using Temporal Relational DataBases (TRDB). The purpose of this example is to show that TRDB can be described within the Token-Datalog framework. This demonstrates the applicability of the languages to the wide range of domains that TRDB is applicable.
For example, consider the following typical relation:

Part #	Description	Shelf	Start	End
10527	Connector A-B	3A	Jan 3	Feb 3
10527	Connector A-B	4A	Jan 17	Feb 15

This relation is *inconsistent* because part # 10527 cannot be simultaneously on shelves 3A and 4A. This inconsistency may result from a typing error or other sources. One way to handle such mistakes is to avoid specifying exact data, we would like to say that "we do not know when the connector was moved form shelf 3A to shelf 4A". To formalize this approach, we introduce two variables X and Y were X is the time point at which the connector was moved from shelf 3A and Y is the time at which it was moved onto shelf 4A.

Part #	Description	Shelf	Start	End
10527	Connector A-B	3A	Jan 3	X
10527	Connector A-B	4A	Y	Feb 15

Here, the temporal constraint induced by incompatibility is that $X \leq Y$, which means that the connector was taken off shelf 3A before it was put on shelf 4A. This constraint may be explicitly given in the input or can be deduced by other constraints. Here, consistency maintanance means (i) computing the implicit constraint $X \leq Y$ and (ii) ensuring that the constraint is always satisfied, no matter how X and Y change.

Let D be above single relation TRDB to be represented in Token-Datalog. We assume that every tuple in D is qualified by a unique token constant called the tuple ID (i.e. identifier). To describe D, we introduce three special purpose predicates: `relation`, `member` and `attribute`. Data terms are diversified into *relation*, *attribute* and *generic* data terms. Let `r,a,v` be a relation, an attribute and a generic data term respectively and let `k` be a token term. The atom `Relation(r,a)` evaluates to *true* iff the scheme of the relation `r` in D contains the attribute `a`. The atom `Member(k,r)` evaluates to *true* iff the relation `r` in D has a tuple whose interval (or point) is equal to $\widehat{\text{interval}}$(k) (or `time(k)`). The atom `Attribute(k,a,v)` evaluates to *true* iff the attribute `a` of the tuple qualified by `k` is assigned the value `v`.

To represent the above relation, we could use the following set of Token-Datalog facts:

```
Relation(inventory, part_no).
Relation(inventory, description).
Relation(inventory, shelf).
Member(k₁, invernory)
Member(k₂, invernory)
Attribute(k₁, part_no, 10527)
Attribute(k₂, part_no, 10527)
Attribute(k₁, description, 'Connector A-B')
Attribute(k₂, description, 'Connector A-B')
Attribute(k₁, shelf, 3A)
Attribute(k₂, shelf, 4A)
begin(k₁) = Jan 3
end(k₂) = Feb 15
```

To identify the inconsistency of the input relation, we write the following Token-Datalog rules for every attribute `a` and for every a pair of incompatible values $\mathtt{v}_i$ and $\mathtt{v}_j$:

```
interval(kᵢ){disjoint}interval(kⱼ) :-  Attribute(kᵢ,a,vᵢ),
                                        Attribute(kⱼ,a,vⱼ),
                                        vᵢ ≠ vⱼ)
```

When assigning `interval(k₁)=[Jan3,X]` and `interval(k₂)=[Y,Feb15]` the constraint `interval(k₁){disjoint}interval(k₂)` induces the desired constraint $X \leq Y$.

6.4 Common Sense Reasoning

Consider the statement "John and Fred were roommates, but now, John owes Fred money, hates him and threatens to kill him. John unloaded his gun 10 minutes ago, but later he loaded it and now the gun is pointing at Fred". Consider the queries:

1. "What is the relationship between John and Fred now ?". The answer is "John owes money, hates and threatens Fred".

2. "What is the status of the gun now ?". The answer is "the gun is loaded and pointing at Fred."

3. "When did the gun get loaded ?". The answer is "John loaded the gun between 10 minutes ago and now.

A possible way to represent (in first order logic) the above temporal information is to use the following predicates: $Loaded(Gun, t)$, $PointedAt(X, Y, t)$, $Owe(X, Y, t)$, $Hate(X, Y, t)$ and $Threaten(X, Y, t)$ where t is a temporal qualification[7].

One problem that is addressed when using Token-Datalog is as follows. If we are to represent the queries described above in first order logic, we need a different set of predicates. This is because in first order logic it is not possible to quantify over predicate symbols, as seems to be required for representing the above three queries.

Another problem that is addressed when using Token-Datalog, originally identified by [61], is as follows. Consider the statement "John owes *someone* money". This could be represented by $\exists X \;\; Owes(John,\ X)$. In general, words such as "*someone* may require introducing an existential quantifier over an arbitrary number of variables. This is highly undesirable in environments such as logic programs because it translates to exhaustive search.

We next formalize this example in Token-Datalog. We have two classes of objects, People and Guns. Guns have two relevant properties: loaded/unloded and the direction at which they point. There are three interpersonal relations: roommate, owe, and threaten. In addition, the two actions specified are: John loading and unloading the gun. We therefore introduce two relations to represent people and guns, two relation to represent the actions of loading and unloading the gun, and three relations to specify the dynamics of relationship between John and Fred.

We use the same method introduce in the warehouse management example above (Section 6.3). To represent the relationship between John and Fred we use the database

```
Relation(Owe, Person1, Person2).
Relation(Threatens,  Person1, Person2).
Relation(RoomMate,  Person1, Person2).
Member(Owe, k1).  Attribute(Person1, John, k1).
Member(Threaten, k2).  Attribute(Person1, John, k2).  Attribute(Person2, Fred, k2).
Member(RoomMate, k3).  Attribute(Person1, John, k3).  Attribute(Person2, Fred, k3).
```

The first three lines describe facts defining the scheme of the relations used. The next three lines define the tuples in these relations and assigns values to their attributes.

[7]or a state, as in situation calculus.

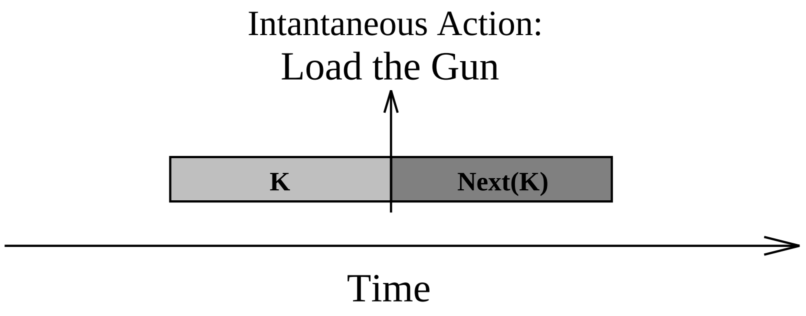


Figure 6.7: Formalizing an instantaneous action.

To describe the action of John loading the gun we assume this action is instantaneous. We use two interval tokens: `k` describes the interval that ends with the action and `next(k)` described the interval that immediately follows the action, as illustrated in Figure 6.7. Note that the lengths of these intervals need not be known. To formalize lawful change we use the following Token-Datalog program:

```
Relation(Load, Actor, Object).
Relation(Gun, Status).
interval(next(k1)) meets interval(k1).

Member(next(k1),Gun),  Attribute(next(k1),Status,Loaded)  :-
         Member(k1, Load), Attribute(k1, Actor, John), Attribute(k1, Object, Gun).
```

The first three facts define the relation `Load` which describes an action having two attributes: an actor the the object being acted upon. The second fact describes a relation for the object `Gun`, which has a single attribute, the gun's `status`. The third fact is required to ensure that the interval before the action took place meets the interval just after the action occurred (i.e. instantaneous action). The subsequent rule enforces that if John (the actor) loaded the gun (the object) then the gun becomes loaded immediately after the action took place.

Next, we formalize the three queries presented above. The first query, "What is the relationship between John and Fred?", requires to compute the set of relations that hold *now* between John and Fred:

```
:- Q(R).
Q(R) :-  Member(K, R),  Attribute(K, Person1, John),  Attribute(K, Person2, Fred),
         now {during} K.
```

where *now* is a point token representing the present time.

The the second query "What is the status of the Gun now?" is given by

```
:- Q(A,V).
Q(A,V)  :-   Member(K, Gun),   Attribute(K, A,V),  now {during} interval(K).
```

Finally, the third query "When was the gun loaded?" is given by

```
:- Q(begin(interval(K))).
Q(begin(interval(K))) :- Member(K, Load).
```

Chapter 7

Concluding Remarks

We have explored performing temporal reasoning with constraints. This work progressed along two lines of research: (i) improving the algorithms for processing Temporal Constraint Satisfaction Problems (TCSP), and (ii) identifying and resolving the problem that arise when embedding TCSPs within a logic programming language.

Along the first line of research, our results show that state-of-the-art algorithms could be improved by orders of magnitude. We designed two new algorithms called Upper Lower Tightening and Loose Path Consistency, whose efficiency was evaluated theoretically and empirically. We have also identified a new tractable class which is common in scheduling domains.

Along the second line of research, we have defined two new languages, TCSP-Datalog and Token-Datalog. TCSP-Datalog is the first step towards achieving the desired embedding of TCSP within logic programs. To further address some issues and improve the utility and flexibility of the language, Token-Datalog was designed. We identified and resolved several syntactic, semantic and inference algorithm issues regarding the embedding of time and constraints within logic programming framework.

Bibliography

[1] A. Aiba, K. Sakai, Y. Sato, D. Hawley, and R. Hasegawa. The constraint logic programming language cal. In *Proc. Intl. Conf. on Fifth Generation Computer Systems*, pages 263–276, 1988.

[2] H. Ait-Kaci and R. Nasr. Login: A logic programming language with built-in inheritence. *Journal of Logic Programming*, 3:185–215, 1986.

[3] H. Ait-Kaci and A. Podelski. Towards a meaning of life. *Journal of Logic Programming*, 16:195–234, 1986.

[4] C. A.K. and H. D. Horn clause queries and generalizations. *Journal of Logic Programming*, 2(1):1–16, 1985.

[5] J. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26:832–843, 1983.

[6] J. Allen. Towards a general theory of action and time. *Artificial Intelligence*, 23:123–154, 1984.

[7] J. Allen. *Natural Language Understanding*. Benjamin Cummings, 1987.

[8] F. Bacchus, J. Tenenberg, and J. Koomen. A non-reified temporal logic. In *Proc. KR'89*, pages 2–10, 1989.

[9] K. Baker, P.C.Fishburn, and F.S.Roberts. Partial orders of dimension 2. *Networks*, 2:11–28, 1972.

[10] B.Dushnik and E.W.Miller. Partially ordered sets. *Amer. J. Math*, 63:600–610, 1941.

[11] A. Belfer and M. Golumbic. The role of combinatorical structures in temporal reasoning. Technical report, IBM Research report, 1981.

[12] A. Belfer and M. Golumbic. A combinatorical approach to temporal reasoning. In *Proc. Information Technology IEEE*, pages 774–780, 1990.

[13] K. Booth and G. Leuker. Testing for the consecutive ones property, interval graphs, and planarity using pq-tree algorithms. *J. Comput. Sys. Sci.*, 13:335–379, 1976.

[14] A. Bouchet. Reducing prime graphs and recognizing circle graphs. *Combinatorics*, 7:243–254, 1987.

[15] J. Chomicki. Depth-bounded bottom-up evaluation of logic programs. *Journal of Logic Programming*, 18:68–81, 1995.

[16] J. Chomicki. Finite representation of infinite query answers. *ACM Transaction on Database Systems*, 1996.

[17] A. Colmeraner. An introduction to prolog iii. *Communications the ACM*, 33:69–90, 1990.

[18] A. Colmeraner. *Prolog II Reference and User Manual.* PrologIA, Marseilles, 1990.

[19] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms.* McGraw Hill, 1990.

[20] M. D. and B. Selman. Hard and easy distributions of sat problems. In *Proc. AAAI'92*, 1992.

[21] T. Dean and D. McDermott. Temporal data base management. *Artificial Intelligence*, 32:1987, 1987.

[22] R. Dechter. *Encyclopedia of Artificial Intelligence*, chapter Constraint Networks. John Wiley&Sons Inc., 2nd edition edition, 1992.

[23] R. Dechter. From local to global consistency. *Artificial Intelligence*, 55:87–107, 1992.

[24] R. Dechter, I.Meiri, and J.Pearl. Temporal constraint networks. In *Proc. KR'89*, pages 83–93, 1989.

[25] R. Dechter, I.Meiri, and J.Pearl. Temporal constraint networks. *Artificial Intelligence*, 49:61–95, 1991.

[26] R. Dechter and J.Pearl. Network-based heuristics for constraint satisfaction problems. *Artificial Intelligence*, 34:1–38, 1988.

[27] R. Dechter, I. Meiri, and J. Pearl. Temporal constraint networks. *Artificial Intelligence*, 49:61–95, 1991.

[28] D. Diaz and P. Codognet. A minimal extention of the wam for clp(fd). In *Proc. 10th Intl. Conf. on Logic Programming*, pages 774–790, 1993.

[29] M. Dincbas, P. vanHetenryck, H. Simonis, and A. Aggoun. The constraint logic programming language chip. In *Proc. 2nd Intl. Conf. on 5th Generation Computing Systems*, pages 249–264, 1988.

[30] T. Drakengren and P. Jonsson. Maximal tractable subclasses of allen's interval algebra: Preliminary report. In *Proc. AAAI'96*, pages 389–394. AAAI Press/MIT Press, 1996.

[31] D.R.Fulkerson and O.A.Gross. Incidence matrices and interval graphs. *Pacific J. Math.*, 15:835–855, 1965.

[32] P. Fishburn. Intransitive indifference with unequal indifference intervals. *J Math Psych.*, 7:144–149, 1970.

[33] C. Freksa. Temporal reasoning based on semi-intervals. *Artificial Intelligence*, 54:199–227, 1992.

[34] E. Freuder. A sufficient condition for backtracking-free search. *Journal of the ACM*, 29:24–32, 1982.

[35] C. Gabor, K. Supowit, and W. Hsu. Recognizing circle graphs in polynomial time. *J. ACM*, 36:435–473, 1989.

[36] A. Galton. A critical examination of Allen's theory of action and time. *Artificial Intelligence*, 42:159–188, 1990.

[37] A. Galton. Reified temporal theories and how to unreify them. In *Proc. IJCAI'91*, pages 1177–1182, 1991.

[38] A. Gerevini and L. Schubert. Efficient temporal reasoning through timegraphs. In *Proc. IJCAI'93*, pages 648–654, 1993.

[39] A. Gerevini and L. Schubert. On computing the minimal labels in time point algebra networks. Technical Report 9408-10, Insituto per la Ricerca Scientifica e Tecnologica, 1994.

[40] A. Gerevini and L. Schubert. Efficient algorithms for qualitative reasoning about time. *Artificial Intelligence*, 74(3):207–248, 1995.

[41] M. Ghallab and A. Mounir Alaoui. Managing efficiently temporal relations through indexed spanning trees. In *Proc. IJCAI'89*, pages 1297–1303, 1989.

[42] M. Ghallab and A. Mounir Alaoui. Managing efficiently temporal relations through indexed spanning trees. In *Proc. IJCAI'89*, pages 1297–1303, 1989.

[43] P. Gilmore and A.J.Hoffman. A characterization of comparability graphs and interval graphs. *Canadian J. Math*, 16:539–548, 1964.

[44] M. Golumbic and R. Shamir. Algorithms and complexity for reasoning about time. Technical Report 22-91, Rutgers, NewJersey, 1992.

[45] S. D. Goodwin and A. Trudel. Persistence in continuous first-order temporal logics. *International Journal of Expert Systems*, 3(3):249–265, 1991.

[46] M. Gulombic and E. Scheinerman. Containment graphs, posets and related classes of graphs. *Ann. N.Y. Acad. Sci*, 555:192–204, 1989.

[47] W. Habens, S. Sidebottom, and G. Sidebottom. A constraint logic programming shell. In *Proc. Pacific Rim Intl. Conf. on Artificial Intelligence*, 1992.

[48] C. Hamblin. Instants and intervals. In J. Fraser, editor, *The Study of Time*, pages 325–331. Springer-Verlag, 1972.

[49] P. Hayes. The naive physics manifesto. In J. Hobbs and R. Moore, editors, *Formal Theories of the Commonsense World.* Ablex, Norwood, New Jersey, 1985. Originally published in 1978.

[50] P. Hayes and J. Allen. Short time periods. In *Proc. IJCAI'87*, pages 981–983, 1987.

[51] J. Jaffar and J. Lassez. Constraint logic programming: A survey. *Journal of Logic Programming*, 19,20:503–581, 1994.

[52] J. Jaffar, S. Michaylov, P. Stuckey, and R. Yap. The clp(r) language and system. *ACM Transactions on Programming Languages*, 14(3):339–395, 1992.

[53] J.D.Ullman. volume 1. In *Principles of Database and Knowledge-Base Systems*, pages 97–121. Computer Science Press, Ithaca, N.Y., 1988.

[54] K. Kahn and G. Gorry. Mechanizing temporal knowledge. *Artificial Intelligence*, 9:87–108, 1977.

[55] H. Kautz and P. Ladkin. Integrating metric and qualitative temporal reasoning. In *Proc. AAAI'91*, pages 241–246, 1991.

[56] J. Koomen. The TIMELOGIC temporal reasoning system. Technical Report 231, Univ. of Rochester, Computer Science Dept., Nov. 1987. (revised March 1989).

[57] N. Korte and R. Möhring. An incremental linear time algorithm for recognizing interval graphs. *SIAM J. Comput.*, 18:68–81, 1989.

[58] M. Koubarakis. Dense time and temporal constraints with $\neq$. In *Proc. KR'92*, pages 24–35, 1992.

[59] M. Koubarakis. *Foundations of Temporal Constraint Databases.* PhD thesis, National Technical University of Athens, Athens, Greece, 1994.

[60] M. Koubarakis. From local to global consistency in temporal constraint networks. In *Proc. CP'95*, 1995.

[61] R. Kowalski and M. Sergot. A logic-based calculus of events. *New Generation Computing*, 3, 1986.

[62] P. Ladkin and R. D. Maddux. The algebra of constraint satisfaction problems and temporal reasoning. Technical report, Krestel Institute, 1988.

[63] P. Ladkin and A. Reinefeld. Effective solution of qualitative interval constraint problems. *Artificial Intelligence*, 57:105–124, 1992.

[64] P. B. Ladkin and A. Reinefeld. A symbolic approach to interval constraint problems. In J. Calmet and J. A. Campbell, editors, *Artificial Intelligence and Symbolic Mathematical Computing*, volume 737, pages 65–84. Springer-Verlag, Berlin, Heidelberg, New York, 1993.

[65] A. Mackworth. Consistency in networks of relations. *Artifical Intelligence*, 8(1):99–118, 1977.

[66] A. Mantsivoda. Flang and its implementation. In *Proc. Symp. on Programming Language Implementation and Logic Programming*, number 714 in Lecture Notes In Computer Science, pages 151–166. Springer-Verlag, 1993.

[67] H. K. Marc Vilain and P. van Beek. Constraint propagation algorithms for temporal reasoning: A revised report. In D. S. Weld and J. de Kleer, editors, *Readings on Qualitative Reasoning about Physical Systems*, pages 373–381. Morgan Kauffman, 1989.

[68] J. McArthur. *Foundations of Logic Programming.* Springer-Verlag, 1984.

[69] J. McCarthy and P. Hayes. Some philosophical problems from the standpoint of AI. *Machine Intelligence*, 4, 1969.

[70] D. McDermott. A temporal logic for reasoning about processes and plans. *Cognitive Science*, 6:101–155, 1982.

[71] I. Meiri. Combining qualitative and quantitative constraints in temporal reasoning. *Artificial Intelligence*, 87(1-2):295–342, 1996.

[72] D. Miller. A logic programming language with lambda-abstraction, function variables and simple unification. In *International Workshop on Extensions of Logic Programming*, number 475 in Lecture Notes in Compoter Science, pages 253–281, 1991.

[73] D. Miller and G. Nadathur. Higher order logic programming. In *Proc. of the 3rd International Conference on Logic Programming*, pages 448–462, 1988.

[74] R. Mohr and T. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28(2):225–233, 1986.

[75] B. Nebel. Solving hard qualitative temporal reasoning problems: Evaluating the efficiency of using the ORD-Horn class. *CONSTRAINTS*, 1(3):175–190, 1997.

[76] B. Nebel and H. Bürckert. Reasoning about temporal relations: A maximal tractable subclass of allen's interval algebra. In *Proc. AAAI94*, pages 356–361, 1994.

[77] K. Nökel. Convex relations between time intervals. In J. Retti and K. Leidlmair, editors, *5 Osterreichische Artificial-Intelligence-Tagung*, pages 298–302, 1989.

[78] W. Older and F. Benhamou. Programming in clp(bnr). In *Proc. Workshop on Principles and Practice of Constraint Programming*, pages 239–249, 1993.

[79] C. P. and K. B. Where the really hard problems are. In *Proc. IJCAI'91*, pages 163–169, 1991.

[80] F. Pfenning. Logic programming in the lf logical framework. In Huet and Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

[81] M. Poesio and R. Brachman. Metric constraints for maintaining appointments: Dates and repeated activities. In *Proc. AAAI'91*, pages 253–255, 1991.

[82] N. Sateh. *Look-Ahead techniques for Micro-opportunistic Job Shop Scheduling*. PhD thesis, Carnegie Mellon University, Department of Computer Science, August 1991.

[83] E. Schwalb. Aaai-96 tutorial: Processing temporal constraint networks. Technical report, UCI, 1996.

[84] E. Schwalb. A new unification method for temporal reasoning with constraints. In *Proc. AAAI'97*, 1997.

[85] E. Schwalb and R. Dechter. Coping with disjunctions in temporal constraint satisfaction problems. In *Proc. AAAI'93*, pages 127–132, 1993.

[86] E. Schwalb and R. Dechter. Processing temporal constraint networks. Technical report, UCI, 1995.

[87] E. Schwalb and R. Dechter. Processing temporal constraint networks. In *Proc. of the Intl. Workshop on Temporal Representation and Reasoning (TIME'96)*, 1996.

[88] E. Schwalb and R. Dechter. Processing temporal constraint networks. *Artificial Intelligence*, 93:29–61, 1997.

[89] E. Schwalb, K. Kask, and R. Dechter. Temporal reasoning with constraints on fluents and events. In *Proc. AAAI'94*, 1994.

[90] E. Schwalb and L. Vila. Deductive databases with temporal constraints. In *Proc. of the Intl. Workshop on Temporal Representation and Reasoning (TIME'96)*, 1996.

[91] E. Schwalb and L. Vila. *The Handbook of Time and Temporal Reasoning in Artificial Intelligence*, chapter 2.3. (in press), 1998.

[92] E. Schwalb and L. Vila. Temporal constraints: A survey. *the Constraints Journal*, 3(2/3), June 1998.

[93] B. Selman, H. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In *Proc. AAAI'92*, pages 440–446, 1992.

[94] Y. Shahar. *A Knowledge based method for temporal abstraction of clinical data.* PhD thesis, Dept. of Computer Science, Stanford University, October 1994.

[95] Y. Shahar. Knowledge-based temporal abstraction in clinical domains. *Artificial Intelligence in Medicine*, 8((3)):367–388, 1996.

[96] Y. Shahar. A framework for knowledge-based temporal abstraction. *Artificial Intelligence*, 90((1-2)):79–133, 1997.

[97] Y. Shoham. Temporal logics in AI: Semantical and ontological considerations. *Artificial Intelligence*, 33:89–104, 1987.

[98] Y. Shoham. *Reasoning about Change: Time and Causation from the Standpoint of Artificial Intelligence.* The MIT Press, 1988.

[99] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1:146–160, 1973.

[100] P. van Beek. Approximation algorithms for temporal reasoning. In *Proc. IJ-CAI'89*, pages 1291–1296, 1989.

[101] P. van Beek. *Exact and Approximate Reasoning about Qualitative Temporal Relations.* PhD thesis, Dept. of Computer Science, University of Alberta, August 1990.

[102] P. van Beek. Reasoning about qualitative temporal information. *Artificial Intelligence*, 58:297–326, 1992.

[103] P. van Beek and R. Cohen. Exact and approximate reasoning about temporal relations. *Computational Intelligence*, 6:132–144, 1990.

[104] P. van Beek and D. W. Manchak. The design and experimental analysis of algorithms for temporal reasoning. *Journal of AI Research*, 4:1–18, 1996.

[105] J. van Benthem. *The Logic of Time.* Kluwer Academic, Dordrecht, 1983.

[106] P. vanHentenrick. *Constraint Satisfaction in Logic Programming.* MIT Press, 1989.

[107] L. Vila and H. Reichgelt. The token reification approach to temporal reasoning. *Artificial Intelligence*, 83(1):59–74, May 1996.

[108] L. Vila and E. Schwalb. A theory of time and temporal incidence based on instants and periods. Technical Report 96-04, ICS Dept. UCI, 1996.

[109] L. Vila and E. Schwalb. A theory of time and temporal incidence based on instants and periods. In *Proc. of the Intl. Workshop on Temporal Representation and Reasoning (TIME'96)*, 1996.

[110] M. Vilain. A system for reasoning about time. In *Proc. AAAI'82*, pages 197–201, 1982.

[111] M. Vilain and H. Kautz. Constraint propagation algorithms for temporal reasoning. In *Proc. AAAI'86*, pages 377–382, 1986.

[112] P. Voda. The constraint language trilogy. Technical report, Complete Logic Systems, 1988.

[113] P. Voda. Proc. 5th intl. conf. on logic programming. In *Proc. IJCAI'85*, pages 580–589, 1988.