J. Symbolic Computation (1996) **11**, 1–000

Temporal Annotated Constraint Logic Programming

Thom Frühwirth

Ludwig-Maximilians-Universität (LMU), Institut für Informatik,

Oettingenstrasse 67, D-80538 Munich, Germany

email: fruehwir@informatik.uni-muenchen.de

www: http://www.pst.informatik.uni-muenchen.de/~fruehwir/

(Received 23 May 1996)

We introduce a family of logics and associated programming languages for representing and reasoning about time. The family is conceptually simple while allowing for different models of time. Formulas can be labeled with temporal information using annotations. In this way we avoid the proliferation of variables and quantifiers as encountered in first order approaches. Unlike temporal logic, both qualitative and quantitative (metric) temporal reasoning about definite and indefinite information with time points (instants) and time periods (temporal intervals) in different models of time are supported.

Our temporal annotated logic can be made an instance of annotated constraint logic, which is also presented in this paper. Given a logic in this framework, there is a systematic way to make a clausal fragment executable as a constraint logic program. We show this for the generic case and for the specific case of temporal annotated logic. In both cases we give an interpreter and a compiler that can be implemented in standard constraint logic programming languages.

Contents

1. Introduction

Our work [T. Frühwirth (1994), T. Frühwirth (1995)] aims at defining and implementing a family of temporal logics with the following characteristics:

© 1996 Academic Press Limited

- Conceptual simplicity
- Extension of first order logic
- Generalization of standard temporal logic
- Efficient execution of its clausal fragment

Our temporal logic should not deviate too much from the common approaches while avoiding the pitfalls of ad-hoc solutions and unifying seemingly exclusive but indispensable features. It should support

- qualitative and quantitative temporal reasoning
- definite and indefinite temporal information
- time points (instants) and time periods (temporal intervals)
- different models of time (linear, branching or circular, discrete or continuous, bounded or unbounded).

In this paper we are going to show how these ambitious goals can be tackled by relying on two concepts: Annotations and constraints. While annotations allow for conceptual simplicity, constraints enable an efficient implementation. We now present the basic principles behind our temporal logic and the associated programming language by relating it to other approaches.

1.1. Temporal Logics

There are two kinds of logic that have been used to express time-dependency of information: Modal logic and first order logic. Modal logic approaches capture naturally the relative position of formulae with respect to an implicit current time by talking about past, present and future. For example,

$$born(john) \rightarrow \mathsf{G}\ lives(john).$$[†]

where G is a temporal operator that makes the associated formula true for all future, means that if John is born now, he is alive in the future (from now on).

On the other hand, first order logic (FOL) approaches naturally support absolute positions of formulae along the time line by making time explicit. Usually, the logic will be reified, i.e. there are predicates that relate object formulas (that are terms in the logic) to temporal entities. E.g., "John was born in 1900" is expressed as

$$holds(born(john), 1900).$$

In an "unreified" logic, formulas have their usual status and the temporal information is included by adding extra arguments to the predicates and introducing additional predicates. E.g.,

$$born(john, 1900).$$

[†] Like in logic programming, predicate symbols start with lowercase letters.

FOL approaches suffer, however, from a proliferation of temporal variables and quantifiers. E.g.,

$born(john) \rightarrow \forall t(later(t, 1900) \rightarrow lives(john))$.

More on temporal logic and its applications can be found in [J. K. van Benthem (1983), A. Galton (Ed.) (1987)].

Temporal Annotated Logic. We propose a logic that lies inbetween the two approaches, while it keeps most of the advantages of both. We make time explicit but avoid the proliferation of temporal variables and quantifiers of the first order approach. We start from FOL and add time by "labeling" formulas with temporal information. The pieces of temporal information are given by *temporal annotations* which say at what time(s) the formula to which they are applied is valid:

$born(john)\ \underline{at\ t} \rightarrow lives(john)\ \underline{th\ [t, \infty]}$.
$born(john)\ \underline{at\ 1900}$.

where the annotations have been underlined for clarity. "*th*" stands for "throughout". Note that the formulas appearing in this paper are assumed to be universally closed at the outermost scope.

Temporal annotated logic can be regarded as a modal logic, where the annotations are seen as parameterized modal operators, e.g. $born(john)\ (at\ t)$. Likewise, it can be seen as reified FOL where annotated formulas correspond to binary relations between predicates and temporal information, e.g. $at(born(john), t)$.

Conceptually, our approach is simple. The underlying idea of devising a temporal logic that is conveniently executable is to separate the temporal from the nontemporal aspects. Annotations account for the special status of time in a natural way (unlike the unreified approach) and if we drop the annotations, we are left with ordinary FOL.

1.2. Temporal Logic Programming

One of the first temporal logic programming languages was Templog, a "temporal Prolog" [M. Abada, Z. Manna (1989)]. Templog implements a fragment of first order temporal logic (tense logic). Our example written in Templog is

```
□lives(john) <= born(john).†
```

Templog is implemented using a special "temporal SLD-resolution" strategy. This corresponds to a "direct" implementation approach which has the disadvantage that we have to start coding almost from scratch.

With the advent of constraints in logic programming (CLP)‡ [P. van Hentenryck (1991),

† In logic programs, variable names start with uppercase letters. Also, later read ":-" as "←", "," as conjunction and ";" as disjunction.

‡ In acronyms ending in "P", the letter will stand either for "program(s)" or for "programming" as required by the context.

T. Frühwirth et al. (1992), J. Jaffar, M. J. Maher (1994)], the implementation of temporal logic by mapping into constraint languages became possible. The idea is to translate the temporal logic into FOL by introducing temporal parameters as well as special relations and functions describing the structure of time.

As argued in [A. M. Frisch, R. B. Scherl (1991)], these special functions and relations can be regarded as constraints and the associated axioms as constraint theory. The advantage of this view is that there is a clear separation of the temporal aspects of the logic from the first order one: For the constraint theory, a special algorithm is used, while for the FOL part, standard deduction (e.g. SLD-resolution) suffices.

For example, the above Templog clause is expressed as

```
lives(john,S) :- S>=T, born(john,T).
```

where `S` and `T` stand for time points.

In [C. Brzoska (1993)], a powerful temporal logic (a tense logic extended by parameterized temporal operators) is translated into first order constraint logic. The resulting constraint theory is rather complex as it involves quantified variables and implication, whose treatment goes beyond standard CLP implementations. For example, to find out who (also) lives since John was born, one writes:

```
:- T=<S, current_date(S), born(john,T),
   for_all R ((T=<R,R=<S) implies lives(X,R)).†
```

Temporal Annotated Constraint Logic Programming (TACLP). The example in our language is simply

```
:- current_date(S), born(john) at T, lives(X) th [T,S].
```

Our temporal language, TACLP, is an instance of a more general framework, that of annotated constraint logic programs (ACLP). One advantage of ACLP is that they can be efficiently implemented by translation into existing CLP languages. ACLP is inspired by generalized annotated logic programs [M. Kifer, V.S. Subrahmanian (1992)].

In previous work [T. Frühwirth (1995), T. Frühwirth (1994)] the expressive power of TACLP was limited by the fact that only atoms could be annotated. In this paper we simplify this case and we investigate how non-atomic formulas can be executed. We present an interpreter for TACLP that is simpler than the one proposed in [T. Frühwirth (1994)] and for the first time show how to compile ACLP and TACLP.

Other Approaches. In TACLP and the above-mentioned languages predicates are time-dependent (flexible, extensional) and functions as well as variables are time–independent (rigid, intensional).

For completeness we mention another line of work in temporal programming languages with a rather different flavor. In languages like TEMPURA [B. Moszkowski (1986)], METATEM [M. Fisher, R. Owens (1992)] and TOKIO [S. Kono (1995)] variables are time-dependent. Execution in these languages tries to generate a model for the temporal formula at hand by stepping trough a sequence of successive states (valid throughout

† The actual syntax in [C. Brzoska (1993)] is somewhat different.

temporal intervals). Computation starts from what is known to hold in the initial state and proceeds in a bottom-up, forward-chaining way. This approach is advantageous for simulation of time-dependent processes and reactive systems.

In temporal constraint logic programming languages, one proceeds in a goal-driven, top-down, backward-chaining way along the causal relationships between time-dependent entities using deduction. There is no need to follow a temporal order or to restrict oneself to computations within single states. This approach is advantageous for reasoning and inquiring about time-dependent information.

For a survey of temporal and modal logic programming consider the overview paper of M. A. Orgun and W. Ma [M. A. Orgun, W. Ma (1994)], for recent trends the proceedings [M. Fisher, R. Owens (Eds.) (1995)].

1.3. Overview of the paper

A murder mystery example will be the frame for our paper. We will start with it (section 2) and end with it (section 6). In the meantime we will acquire the means to solve it by developing our temporal logic and their implementation. In the end, the murder case illustrates the diverse features of our temporal annotated logic. We will gradually build up our temporal logic starting from simple principles and notions (section 3). Our framework will not make any presuppositions on the ontology of time or denotation of formulas. We will specify the ontology of our choice by constraining the temporal annotations of the logic accordingly.

Temporal annotated logic as it is axiomatized is hard to implement. However, we can make our logic an instance of annotated constraint logic, a generalization of generalized annotated programs [M. Kifer, V.S. Subrahmanian (1992), S. M. Leach, J. J. Lu (1994)]. Their advantage is that their clausal fragment can be efficiently implemented in standard constraint logic programming languages. "Making an instance" means that we have to move to another axiomatization of our temporal logic in terms of annotation constraints.

In order to keep the paper comprehensible and self-contained we have to introduce annotated constraint logics next, in section 4. This section does not relate to temporal reasoning a priori, but it introduces the framework that will allow us to achieve much of our ambitious goals in temporal logic programming in the subsequent section 5. So this is also a paper about a powerful class of logics that can be made executable in a straightforward way. We show how their clausal fragment can be interpreted in standard constraint logic programming languages and how to compile annotations and the associated special inference rules away.

In section 5, we make temporal annotated logic an instance of this framework by providing the appropriate temporal constraint theory for annotations. In this way we are able to derive and specialize an interpreter and a compiler for temporal annotations from the one for annotated constraint logic.

At the end of each section we discuss related work.

Remark. The reader should keep in mind that at first section 3 and section 4 seem not much related. It is only in the next section, 5, that we put together what we have learned. To ease this process, section 5 shares the structure of sections 3 and 4. We could have started our paper with section 5, but then it would have been not clear how we came up with our family of temporal logics and why we can implement it efficiently in a straightforward way. By the way, a user of our temporal logic does not need to know

There is a workshop at the Plaza hotel.
(1) In the afternoon session, after the coffee break (3:00 - 3:25pm), there were four more talks, 25 minutes each - *time periods.*
Dr. Maringer gave the 3rd talk. The last talk was to be given by Prof. Lepov. But there was a murder.
(2) Prof. Lepov was found dead in his hotel room at 5:35pm - *time point.*
(3) The doctor said he was dead for one to one and a half hours - *duration and indefinite information.*
There are two suspects, Dr. Kosta and Dr. Maringer. They have alibis - *relates time periods.*
(4) Dr. Kosta took the last shuttle to the airport possible to reach the 5:10pm plane - *time point.*
(5) The shuttle from the hotel leaves every half hour between noon and 11pm - *recurrent (periodic) data.*
(6) It takes at least 50 minutes to get to the airport - *duration and indefinite information.*
(7) During the 2nd talk Dr. Maringer realized that he had forgotten to copy his 30 slides - *relates time periods.*
So he picked up the slides from his hotel room and copied them. It takes 5 minutes to get to the room, another 5 minutes to get to the copy room from there, and 5 more minutes to get back to the lecture hall - *durations.*
A copy takes half a minute - *repeated durations.*
(8) *Who murdered Prof. Lepov ?*

Figure 1. The Workshop Murder Mystery.

about its relationship to annotated constraint logic, but can just rely on the didactic presentation in section 3.

2. The Workshop Murder Mystery

We illustrate the expressiveness and conceptual simplicity of our approach with an example from [T. Frühwirth (1994)]. It involves reasoning about qualitative and quantitative (metric), complete and partial temporal information involving time periods, their duration, and time points. We will be able to solve the workshop murder mystery (see figure 1) at the end of this paper in section 6.

3. Temporal Annotated Logic

Our temporal logic is basically a FOL where formulas can (but must not) be labeled with temporal information. There will be three kinds of temporal annotations, all involving (sets of) time points. We then give some useful theorems characterizing the temporal annotations. At this point our logic does not make any presuppositions on the ontology of time or denotation of formulas. We can specify the ontology of choice by constraining the temporal annotations of the logic accordingly. We do so by introducing a partial order between time points. The temporal order allows for a notion of time periods (temporal intervals). Using time periods, we show how standard temporal logic can be embedded in our logic. At the end of the section we discuss related work.

3.1. Temporal Set Annotations

We start from standard FOL consisting of terms built from variables and function symbols with associated arities (including constants) applied to terms, atoms built from predicate symbols with associated arities (including propositions) applied to terms, and formulas built from atoms with the usual logical connectives. The axioms of FOL hold.

Definition 3.1. (at) *Let A, B be first order formulas and t be a time point*[†]*. The annotated formula A at t means that the formula A is true at time point t. In order to be able to deal with non-atomic formulas, we require that the at annotation distributes over all logical connectives*

($\neg at$) $\neg(A\ at\ t) \Leftrightarrow (\neg A)\ at\ t$
($\wedge at$) $(A \wedge B)\ at\ t \Leftrightarrow (A\ at\ t \wedge B\ at\ t)$

As needed, annotated formulas may represent events, states, properties, processes, actions and so on.

Example 3.1. (Mystery) In our workshop murder mystery, we can now express when Prof. Lepov was found dead and when Dr. Kosta boarded his plane.

```
found_dead('Lepov') at 5:35.                                % hint (2)
board_plane('Kosta') at 5:10.                               % hint (4)
```

We use discrete time of hours and minutes. In terms of implementation, we may think of "5:35" (5 hours and 35 minutes) as an abbreviation for "5*60+35".

Different from the standard first order approach to temporal logic, we will use annotations involving sets to capture quantified temporal variables. The idea is to see that quantification over a temporal variable intentionally defines a (possibly infinite) set of time points. The set approach itself is not new, already [D. McDermot (1982)] uses a similar construction.

(ZF) We therefore add the Zermelo-Fränkel axioms for set theory to our axioms.

We relate a formula to many time points by introducing two new temporal annotations, $th(roughout)$ and in.

Definition 3.2. (th) *Let I be a set of time points. If a formula A holds* throughout*, i.e. at* every *time point in a set I, then we write $A\ th\ I$. The first order definition of th annotated formulas is:*

(th) $A\ th\ I \Leftrightarrow \forall t\ (t \in I \rightarrow A\ at\ t)$

Example 3.2. (Mystery, contd.) The suspects have an alibi if they were on the shuttle, copying or giving a talk. Dr. Maringer was copying during the second talk.

```
alibi(X) th I :- on_shuttle(X) th I ; copying(X) th I ; talk(_,X,_) th I.

copying('Maringer') th I :- talk(2,Speaker,Title) th I. % hint (7)
```

[†] Note that we can use variables for sets of time points.

[†] We may think of a time point as denoting an indivisible, durationless instant or moment of time.
[†] The symbol "_" stands for a "anonymous" variable in CLP, one that is not referenced anywhere and can be ignored.

Definition 3.3. (*in*) *If a formula A holds at* some *time point(s) - but we do not know exactly when -* in *a set I we write A in I. This temporal annotation accounts for indefinite temporal information.*

$(in)\ A\ in\ I \Leftrightarrow \exists t\ (t \in I \wedge A\ at\ t)$

Example 3.3. (Mystery, contd.) Somebody is not the murderer if he has an alibi throughout the time the murder could have happened.

```
not_murder(X,Y) :- murdered(Y) in I, alibi(X) th I.     % Whodunnit? (8)
```

We may have slightly stronger definitions for temporal annotations that require sets to be nonempty or not to be singletons. We will later see that these variations characterize the relationship between time points and time periods.

3.2. Theorems about Temporal Annotations

The following theorems are helpful for familiarizing oneself with our temporal annotated logic. Many of these theorems can be found as axioms of temporal logics suggested in the literature (see related work). Moreover, we will use some of the theorems later in section 5 to derive a constraint theory and an interpreter for our temporal annotated logic.

Let t be a time point, I and J be sets of time points, A and B be formulas.

Atoms. These basic theorems appeared first in [T. Frühwirth (1995)]. If empty sets are allowed, we can express the basic propositions $true$ and $false$:

$(\{\}th)\ A\ th\ \{\} \Leftrightarrow true$
$(\{\}in)\ A\ in\ \{\} \Leftrightarrow false.$

If sets containing a single time point are allowed, the three temporal annotations coincide:

$(1th)\ A\ at\ t \Leftrightarrow A\ th\ \{t\}$
$(1in)\ A\ at\ t \Leftrightarrow A\ in\ \{t\}$

If A holds throughout a set, it also holds throughout all subsets of the set. Analogously, if A holds at some time in a set, it also holds at some time in all supersets:

$(\subseteq th)\ A\ th\ I \Leftrightarrow \forall J\ (J \subseteq I \rightarrow A\ th\ J)$
$(\subseteq in)\ A\ in\ I \Leftrightarrow \forall J\ (J \supseteq I \rightarrow A\ in\ J)$

Negation. The annotations in and th are dual with regard to classical negation:

$(\neg th)\ (\neg A\ th\ I) \Leftrightarrow \neg(A\ in\ I)$
$(\neg in)\ (\neg A\ in\ I) \Leftrightarrow \neg(A\ th\ I)$

The proofs that these logical equalities follow from the first order definitions of the temporal annotations are straightforward and analogous to those in [A. Galton (1990)] (also see subsection 3.5 on related work).

Conjunction and Disjunction. The *th* annotation distributes over conjunction and the *in* annotation over disjunction:

$(\wedge th)$ $(A \wedge B)\ th\ I \Leftrightarrow (A\ th\ I \wedge B\ th\ I)$
$(\vee in)$ $(A \vee B)\ in\ I \Leftrightarrow (A\ in\ I \vee B\ in\ I)$

The *in* annotation does not distribute over conjunction, the *th* annotation does not distribute over disjunction:

$(\wedge in)$ $(A \wedge B)\ in\ I \Rightarrow (A\ in\ I \wedge B\ in\ I)$
$(\vee th)$ $(A \vee B)\ th\ I \Leftarrow (A\ th\ I \vee B\ th\ I)$

Annotations for the same formula can be merged sometimes:

$(\cup in)$ $A\ in\ I \cup J \Leftrightarrow (A\ in\ I \vee A\ in\ J)$
$(\cup th)$ $A\ th\ I \cup J \Leftrightarrow (A\ th\ I \wedge A\ th\ J)$

The symmetric cases involving $\cap$ are only implications that are too weak to be interesting. The proofs of these theorems can be found in the appendix.

Example 3.4. (th) The properties of the *th* annotation are illustrated:

$monarchy\ th\ I \leftarrow king(Name)\ th\ I.$

means that there is a monarchy when there is a king.

$conflict\ th\ I \leftarrow king(Name1)\ th\ I \wedge king(Name2)\ th\ I \wedge Name1 \neq Name2.$

means that there is a conflict while there are two kings. Given the two annotated atoms

$king(hubert)\ th\ \{1717, ..., 1789\}.$
$king(kurt)\ th\ \{1787, ..., 1812\}.$

we can derive that

$monarchy\ th\ \{1717, ..., 1812\}.$ by theorem $(\cup th)$
$conflict\ th\ \{1787, 1788, 1789\}.$ by theorem $(\subseteq th)$

3.3. Temporal Order and Time Periods

We now consider an instance of temporal annotated logic which is more structured (time points can be ordered). With the order, time periods can be introduced as restricted sets of time points.

Let the time points be partially ordered by a relation " $\leq$ ", i.e. $r \leq t$ means that r is earlier than or the same as t. Let *lb* (*ub*) be the lower bound (upper bound) of the temporal time line or $-\infty$ (∞) in case of an unbounded temporal model. We require that every temporal variable, say t, is constrained to be within these bounds, $lb \leq t \leq ub$. Let 0 (zero) denote the current time.

Depending on the order chosen, time may be linear (one future) or branching (many possible futures), or circular (infinitely branching), discrete or continuous (dense), bounded or unbounded on either end (finitely or infinitely stretching into past or future).

Time periods[†] are stretches of time that have a duration. We can model time periods by the set of all time points that fall into the time period. Clearly, these sets are convex, i.e. for any two time points in the set, all the time points inbetween are also in the set.

In practice, convex sets are often represented by intervals, since intervals provide a compact, finite representation of these possibly infinite sets. We write the interval $[r, s]$ for the convex nonempty set $\{t \mid r \leq t \leq s\}$ if $r \leq s$. Intervals may be closed or open on either side. In an unbounded time model, $-\infty$ (∞) must not occur as right (left) end-point of a nonempty interval. The empty interval is represented by $[ub, lb]$.

Example 3.5. (Mystery, contd.) We can write down the time table of the workshop according to hint (1).

```
coffee_break th [3:00,3:25].                              % hint (1)
talk(1,'Hunon','Algebraic Semantics...') th [3:25,3:50].
...
```

As intervals represent sets, we can adopt from set theory relations (like equality and inclusion) and operations (like union and intersection) on intervals. These relations and operations can be efficiently implemented by comparison of and computation on the end-points of the intervals. However, intervals are not closed under union and complement. For dealing with time periods, the complement is not needed and for union it suffices if it is only defined when the result is an interval (i.e. another time period). Note that non-convex sets appearing in temporal annotations can be split into convex subsets using the theorems $(\cup in)$ and $(\cup th)$.

Example 3.6. (Indefiniteness) Often, time periods are defined by their end-points:

$lives\ th\ [T1, T2] \leftarrow born\ at\ T1, died\ at\ T2.$

We can deal with indefiniteness about birth and death in the following way:

$lives\ th\ [T1, T2) \leftarrow born\ in\ [T0, T1] \wedge died\ in\ [T2, T3].$
$lives\ in\ I \leftarrow born\ in\ I \vee died\ in\ I.$

Together with

$born\ at$ 1959.
$died\ in$ [1996, 2100].

[†] We use *(time) period* instead of *(temporal) interval* throughout, to avoid confusion with purely mathematical notion of interval.

Table 1. Temporal operators - in modal, annotated and first order logic

sometime in the past:	$\mathsf{P}\,A$	$\Leftrightarrow$	$A\ in\ [-\infty, 0)$	$\Leftrightarrow$	$\exists r\ (r < 0) \wedge A\ at\ r)$
always in the past:	$\mathsf{H}\,A$	$\Leftrightarrow$	$A\ th\ [-\infty, 0)$	$\Leftrightarrow$	$\forall r\ (r < 0 \rightarrow A\ at\ r)$
sometime in the future:	$\mathsf{F}\,A$	$\Leftrightarrow$	$A\ in\ (0, \infty]$	$\Leftrightarrow$	$\exists r\ (0 < r \wedge A\ at\ r)$
always in the future:	$\mathsf{G}\,A$	$\Leftrightarrow$	$A\ th\ (0, \infty]$	$\Leftrightarrow$	$\forall r\ (0 < r \rightarrow A\ at\ r)$
previous:	$\bullet A$	$\Leftrightarrow$	$A\ at\ -1$	$\Leftrightarrow$	$A\ at\ -1$
next:	$\circ A$	$\Leftrightarrow$	$A\ at\ +1$	$\Leftrightarrow$	$A\ at\ +1$

"1" is the time interval of unit length.

we can derive definite and indefinite information

$lives\ th\ [1959, 1996)$. by theorem $(1in)$ for `born`
$lives\ in\ [1996, 2100]$. by theorems $(1in)$ for `born` and $(\vee in)$

3.4. Relationship to Temporal Logic

In our temporal annotated logic with order we can embed the standard temporal logic resulting from introducing two new connectives, "since" (S) and "until" (U), to FOL. Time is strictly linear.

Definition 3.4. (Since, Until) *The definition of* S *and* U *in FOL is:*

$$\begin{array}{lllll} \text{since}: & A\ \mathsf{S}\ B & \Leftrightarrow & \exists r\ (r < 0 \wedge A\ at\ r \wedge \forall s\ (r < s < 0 \rightarrow B\ at\ s)) \\ \text{until}: & A\ \mathsf{U}\ B & \Leftrightarrow & \exists r\ (0 < r \wedge A\ at\ r \wedge \forall s\ (0 < s < r \rightarrow B\ at\ s)) \end{array}$$

where A and B are formulas and r, s and t are variables denoting time points.

The common temporal operators of tense logic can be defined in terms of the basic connectives S and U (table 1). When we embed tense logic in our temporal annotated logic (see also [T. Frühwirth (1995)]), we avoid the limited expressiveness of the modal approach (e.g. hard to talk about absolute time) and the proliferation of temporal variables, explicit quantification and complex constraints of the FOL approach. From table 1 it is also obvious that temporal annotated logic is really "between" the modal logic and FOL approach. Of course, S and U can also be defined in temporal annotated logic:

$$\begin{array}{lllll} \text{since:} & A\ \mathsf{S}\ B & \Leftrightarrow & \exists r\ (A\ in\ [r, 0) \wedge B\ th\ (r, 0)\ ) \\ \text{until:} & A\ \mathsf{U}\ B & \Leftrightarrow & \exists r\ (A\ in\ (0, r] \wedge B\ th\ (0, r)\ ) \end{array}$$

Note, however, that some nested temporal operators cannot be translated into temporal annotated logic without nesting annotations themselves and that we do not deal with nested annotations in this paper.

3.5. Related Work

In the literature, typically only nonempty intervals are considered as time periods. Moreover, in dense, linear time approaches, there are usually no singleton intervals. This implies that that each time period always has a proper sub-period. We call such restricted time periods "proper". For proper time periods the theorems (1) and ($\{\}$) are dropped and ($\subseteq$) applies to proper infinite subsets only.

Allen only considers time periods and no time points [J. F. Allen (1984)]. His axioms imply dense, linear time and proper time periods. His predicate $holds(P, I)$ (where P is a formula denoting a property) is equivalent to the annotated formula $P\ th\ I$. Hence all his axioms and theorems about *holds* correspond to theorems of our temporal annotated logic. As advocated in [A. Galton (1990)], and different from [J. F. Allen (1984)], we use classical negation. We have no correspondence to Allens *occur* predicate for events (which has been shown to prevent certain intuitive conclusions in [A. Galton (Ed.) (1987)]). Allens predicate $occurring(P, I)$ (where P is a formula denoting a process) corresponds to the annotated formula $P\ in\ I$.

Our temporal annotations also correspond to some of the predicates proposed in [A. Galton (1990)], which is a critical examination of Allens work. Galton provides for both time points and proper time periods. In particular, the predicate *holds-in(A,I)* can be mapped into $A\ in\ I$, *holds-on(A,I)* into $A\ th\ I$, and *holds-at(A,t)* into $A\ at\ t$, where A is restricted to be an atomic formula.

4. Annotated Constraint Logic

We introduce the general framework of annotated constraint logic. Our logic is basically standard FOL extended with constraints and annotations. There is a minimal constraint theory that axiomatizes lattice operations for annotations. For annotated formulas, special inference rules apply. Our definitions remove most of the restrictions on annotations in generalized annotated programs (GAP) [M. Kifer, V.S. Subrahmanian (1992)] and in annotated logic programs [S. M. Leach, J. J. Lu (1994)]. The flavor is also different: While in GAP annotations are truth values, we consider annotations as modal operators.

We show how a clausal fragment of our logic can be executed. The inference rules of annotated constraint logic are implemented by a generic interpreter that runs in any CLP language that can deal with the lattice constraints. Unlike the approach of [S. M. Leach, J. J. Lu (1994)], this results in a sound declarative implementation. Then we show how to compile annotated programs into standard CLP. Our compilation is computationally more feasible than the so-called "reductants" approach of M. Kifer and V.S. Subrahmanian. Related work is discussed in more detail at the end of this section.

4.1. Logics with Constraints and Annotations

Definition 4.1. *A* first order constraint logic *is a FOL with a distinguished class of predicates called* relational constraints *and a distinguished class of interpreted functions called* functional constraints. *A* constraint theory *is the set of all sentences involving only relational and functional constraints. Equality (=) as well as* true *and* false *are relational constraints.*

Next we add annotations to the constraint logic.

Definition 4.2. *There is a distinguished class of terms called* annotations. *The class of annotations forms an upper semilattice (every nonempty finite subset has a least upper bound). The partial ordering $\sqsubseteq$ is a relational constraint. The least upper bound operator $\sqcup$ is a functional constraint.*

The semilattice needs not be complete. If it exists, the maximal (resp. minimal) element of the lattice is denoted by top, $\top$, (resp. bottom, $\bot$).

Definition 4.3. *An* (first order) annotated constraint logic *is a first order constraint logic where formulas can be labeled with an annotation. We write the annotation immediately after the formula it labels.*

Unlike [M. Kifer, V.S. Subrahmanian (1992)] and [S. M. Leach, J. J. Lu (1994)], any formula can be annotated. Moreover we do not require the functions that occur in annotations to be total continuous or "effectively computable".

Annotated constraint logic includes a minimal constraint theory for the lattice operations on annotations and a minimal set of inference rules for annotated formulas.

4.1.1. Constraint Theory

The lattice operations $\sqsubseteq$ and $\sqcup$ can be axiomatized by a constraint theory.

Definition 4.4. (Lattice Order)

($\sqsubseteq$Reflexivity)	$\alpha \sqsubseteq \alpha$
($\sqsubseteq$Anti-Symmetry)	$\alpha \sqsubseteq \beta \wedge \beta \sqsubseteq \alpha \leftrightarrow \alpha = \beta$
($\sqsubseteq$Transitivity)	$\alpha \sqsubseteq \beta \wedge \beta \sqsubseteq \gamma \rightarrow \alpha \sqsubseteq \gamma$

If they exist,

($\sqsubseteq$Bottom)	$\bot \sqsubseteq \alpha$
($\sqsubseteq$Top)	$\alpha \sqsubseteq \top$

where α, β, γ are annotations.

Definition 4.5. (Least Upper Bound) *The definition of the least upper bound $\sqcup$ is in terms of $\sqsubseteq$:*

$$(\sqcup Def) \quad \alpha \sqsubseteq (\alpha \sqcup \beta) \wedge \beta \sqsubseteq (\alpha \sqcup \beta) \ \wedge\ \forall \gamma(\alpha \sqsubseteq \gamma \wedge \beta \sqsubseteq \gamma \rightarrow (\alpha \sqcup \beta) \sqsubseteq \gamma)$$

The definition of $\sqcup$ is not really constructive. It helps to keep these theorems in mind:

($\sqcup$Idempotency)	$\alpha \sqcup \alpha \ = \ \alpha$
($\sqcup$Commutativity)	$\alpha \sqcup \beta \ = \ \beta \sqcup \alpha$
($\sqcup$Associativity)	$\alpha \sqcup (\beta \sqcup \gamma) \ = \ (\alpha \sqcup \beta) \sqcup \gamma$

If they exist,

($\sqcup$Bottom)	$\bot \sqcup \alpha \ = \ \alpha$
($\sqcup$Top)	$\top \sqcup \alpha \ = \ \top$

Example 4.1. (Certainty) The lattice may be taken from real closed fields, with the set of the real numbers between 0 and 1 with the usual ordering $\leq$, and where the upper bound operator is the maximum function, maximal element 1, minimal element 0. The

functional constraints are max and min as well as $+$ and $-$, and relational constraints are $=, \leq$ and $<$. Infix notation may be used for relational constraints (e.g. $X \leq Y$) and functional constraints (e.g. $-X+Y$). We will use this lattice to model certainty.

4.1.2. Inference Rules

In addition to Modus Ponens

$$(\textit{Modus Ponens}) \quad \frac{B \ , \ (B \rightarrow A)}{A}$$

that applies to formulas A and B be they annotated or not, we add two finitary inference rules to our constraint logic which utilize the lattice structure of the annotations:

$$(\sqsubseteq) \quad \frac{A\ \alpha \ , \ \beta \sqsubseteq \alpha}{A\ \beta} \qquad (\sqcup) \quad \frac{A\ \alpha \ , \ A\ \beta}{A\ (\alpha \sqcup \beta)}$$

The $(\sqsubseteq)$ rule says that if a formula holds with some annotation, then it also holds with all annotations that are smaller according to the lattice. The $(\sqcup)$ rule says that if a formula holds with some annotation and the same formula holds with another annotation, then the formula also holds with the least upper bound of the annotations. This upward closure of annotations means that there is usually a single annotation that exactly represents all the annotations for which a formula holds[†].

Example 4.2. (Certainty, contd.) Consider formulas with numeric annotations representing degrees of certainty

$rain : 0.9 \leftrightarrow grass\_wet : 0.8.$
$rain : V \leftrightarrow clouds : V.$
$grass\_wet : 1.0.$
$clouds : 0.5.$

allow us to derive that it is likely to rain

$rain : max(0.9, 0.5).$

Variant 1. We can strengthen the $(\sqcup)$ inference rule to an equivalence because of the $(\sqsubseteq)$ inference rule.

Proof. Obvious, since applying rule $(\sqsubseteq)$ to the conclusion $A\ (\alpha \sqcup \beta)$ of rule $(\sqcup)$ allows us to derive the premises. $\square$

Variant 2. The two inference rules for annotations can be merged into one inference rule:

$$(\sqsubseteq \sqcup) \quad \frac{A\ \alpha \ , \ A\ \beta \ , \ \gamma \sqsubseteq \alpha \sqcup \beta}{A\ \gamma}$$

[†] Problems may arise if we take the closure of an infinite number of annotations - this issue is discussed at length in [M. Kifer, V.S. Subrahmanian (1992)].

Proof. This rule is a special case of the two inference rules produced by applying ($\sqsubseteq$) to the conclusion of ($\sqcup$). On the other hand rule ($\sqsubseteq \sqcup$) is as least as general, since taking $\alpha = \beta$ results in rule ($\sqsubseteq$) and taking $\gamma = \alpha \sqcup \beta$ results in rule ($\sqcup$). □

Variant 3. The three inference rules can be even merged into a single inference rule for annotated formulas provided a minimal element $\perp$ exists in the lattice and $A\perp$ is always *true*:

$$(A - \mathit{Resolution}) \quad \frac{A\ \alpha\ ,\ B\ ,\ (B \rightarrow A\ \beta),\ \gamma \sqsubseteq (\alpha \sqcup \beta)}{A\ \gamma}$$

For formulas without annotations, we have to keep the *(Modus Ponens)*. We will use this inference rule for implementation.

Proof. This rule is on one hand obtained by applying rule ($\sqsubseteq \sqcup$) to the conclusion of *(Modus Ponens)*, and on the other hand taking $\alpha = \perp$ and $\beta = \gamma$ results in *(Modus Ponens)* and taking $B = A\beta$ results in the inference rule ($\sqsubseteq \sqcup$). □

4.1.3. Axioms

The axioms that define the interplay of the logical connectives and the annotations come with the specific instance of the framework.

Example 4.3. (Certainty, contd.) We may have the following law:

$(A \wedge B)\ min(\alpha, \beta) \Leftrightarrow A\ \alpha \wedge B\ \beta.$

4.2. Interpreter

Our generic interpreter implements the annotation inference rules for a clausal fragment of annotated constraint logic in any CLP language that can deal with the lattice constraints at hand.

Definition 4.6. (ACLP) *An* ACLP *is a finite set of ACLP clauses. An* ACLP clause *is one of:*

$$A \leftarrow C_1 \wedge \ldots \wedge C_n \wedge B_1 \wedge \ldots \wedge B_m \quad (n, m \geq 0)$$
$$A\ \alpha \leftarrow C_1 \wedge \ldots \wedge C_n \wedge B_1 \wedge \ldots \wedge B_m \quad (n, m \geq 0)$$

where A is an atom (not a constraint), the C_j's are the relational constraints, the B_i's are arbitrary formulas built from the connectives $\neg$, $\wedge$ and $\vee$. α is an annotation. If the head A is annotated, the clause is called annotated. Any B_i or any subformulas of it may be annotated or not. If a formula is annotated, its subformulas may not be annotated since we dissallow nested annotations. The conclusion of the implication is called the head of the clause and the premise the body of the clause. If $n = m = 0$, then the body is empty which is the same as being "true".

The important restriction of ACLP clauses is that their heads have to be atoms. Even though arbitrary formulas may appear in the body, negation and disjunction are usually not complete in implementations. Together with the head restriction this enables efficient execution at the expense of expressiveness and completeness.

Logic programming languages are well suited for writing interpreters as they can treat programs as data [L. Sterling, E. Shapiro (1994)]. The object program is reified, i.e. the predicates are represented by functions in the interpreter. The clauses of the standard interpreter are part of the Prolog folklore. They handle relational constraints, negation, conjunction and disjunction.

The unary predicate $\mathsf{prove}(A)$ is true if and only if the formula A is true at the object level. ACLP clauses of the object program, say $A \leftarrow B$, where B is a formula, are represented at the meta level as the binary predicate $\mathsf{clause}(A, B)$.

$\mathsf{prove}(A) \leftarrow \mathsf{constraint}(A) \wedge A.$
$\mathsf{prove}(\neg A) \leftarrow \neg\mathsf{prove}(A).$
$\mathsf{prove}(A \wedge B) \leftarrow \mathsf{prove}(A) \wedge \mathsf{prove}(B).$
$\mathsf{prove}(A \vee B) \leftarrow \mathsf{prove}(A) \vee \mathsf{prove}(B).$

Note that the formulas A and B may be constraints, or be annotated or not. Further note that according to the interpreter clause, $\neg A$ is proven if A cannot be proven. This kind of negation usually used in CLP languages is termed negation as failure. It is only sound if no variable of the formula A is further constrained during the execution inside the negation. Furthermore, the disjunction used in CLP means to prove either A or B at a time. These limitations will show up again when implementing TACLP as an instance of this interpreter.

4.2.1. Inference Rules

The most important clause of the standard interpreter implements *(Modus Ponens)* for non-annotated clauses:

(Modus Ponens) $\mathsf{prove}(A) \leftarrow \mathsf{non\_annotated}(A) \wedge \mathsf{clause}(A, B) \wedge \mathsf{prove}(B).$

We next make sure that $A\bot$ is always true:

(Bottom) $\mathsf{prove}(A\ \bot).$

The annotation inference rule *(A-Resolution)* can be put into clausal form easily.

$\mathsf{prove}(A\ \gamma) \leftarrow \gamma \sqsubseteq (\alpha \sqcup \beta) \wedge \mathsf{clause}(A\ \alpha, B) \wedge \mathsf{prove}(B) \wedge \mathsf{prove}(A\ \beta).$

The recursive call $\mathsf{prove}(A\ \beta)$ may produce an annotation β equal to α and thus cause nontermination. We can avoid this by making sure that β is never the same as α. Moreover, if it is greater, we could have produced β in the first place instead of the smaller α. In this way, many nonterminating and redundant computations are avoided.

(A-Resolution)
$\mathsf{prove}(A\ \gamma) \leftarrow \gamma \sqsubseteq (\alpha \sqcup \beta) \wedge \mathsf{clause}(A\ \alpha, B) \wedge \mathsf{prove}(B) \wedge \neg(\alpha \sqsubseteq \beta) \wedge \mathsf{prove}(A\ \beta).$

For more complex improvements see also the interpreter in [T. Frühwirth (1994)] which does not rely on the existence of a minimal annotation $\bot$.

Instances of ACLP can further specialize the interpreter clauses that explicitly deal with annotations. This is important especially for clause *(A-Resolution)*, because the recursion can be regarded as a potential efficiency "bottleneck" of the implementation.

Example 4.4. (Certainty, contd.) The interpreter clause *(A-Resolution)* specialized for certainty is

$$\mathsf{prove}(A:\gamma) \leftarrow \gamma \leq max(\alpha,\beta) \wedge \mathsf{clause}(A:\alpha,B) \wedge \mathsf{prove}(B) \wedge \neg(\alpha \leq \beta) \wedge \mathsf{prove}(A:\beta).$$

The recursion can even be simplified away, since the maximum of two number is on of the two numbers and since $max(\alpha,\beta)$ always results in α if $\neg(\alpha \leq \beta)$:

$$\mathsf{prove}(A:\gamma) \leftarrow \gamma \leq \alpha \wedge \mathsf{clause}(A:\alpha,B) \wedge \mathsf{prove}(B).$$

Without recursion, there is no need for the interpreter clause *(Bottom)* either.

4.2.2. Axioms

Finally we add additional interpreter clauses for the additional axioms of the annotated logic at hand. This is straightforward if the axioms and theorems are ACLP clauses.

Example 4.5. (Certainty, contd.) We would add:

$$\mathsf{prove}((A \wedge B)\ min(\alpha,\beta)) \leftarrow \mathsf{prove}(A\ \alpha) \wedge \mathsf{prove}(B\ \beta).$$

We will see specialization and additional clauses when implementing TACLP.

4.3. Compiler

We can implement ACLP by translation into CLP. We define a compilation function comp that translates an annotated formula into its CLP form.

The essential change is that annotated atoms are "unreified"

$$\mathsf{comp}(p(t_1,\ldots,t_n)\ \alpha) \longrightarrow p(t_1,\ldots,t_n,\alpha)$$

while non-annotated atoms remain unchanged

$$\mathsf{comp}(p(t_1,\ldots,t_n)) \longrightarrow p(t_1,\ldots,t_n)$$

We can basically can read off the other rules of the translation function comp either from the interpreter by looking at its clauses or directly from the axioms of the instance of annotated constraint logic at hand.

The standard clauses of the interpreter result in translation rules that push comp inwards.

$$\mathsf{comp}(A) \longrightarrow A \quad \text{if } A \text{ is a constraint}$$

$\mathsf{comp}(\neg A) \longrightarrow \neg\mathsf{comp}(A)$
$\mathsf{comp}(A \wedge B) \longrightarrow \mathsf{comp}(A) \wedge \mathsf{comp}(B)$
$\mathsf{comp}(A \vee B) \longrightarrow \mathsf{comp}(A) \vee \mathsf{comp}(B)$

4.3.1. Inference Rules

For each annotated predicate symbol p with arity n in the program we add the clause (with empty body) resulting from

(Bottom) $\mathsf{comp}(p(X_1, ...X_n)\ \bot)$

where the X_i are distinct variables.

Either the interpreter clauses containing clause or the special inference rules themselves tell us how to translate the program clauses as a whole:

(*Modus Ponens*) $\mathsf{comp}(A) \leftarrow \mathsf{comp}(B)$

for each clause $A \leftarrow B$ where A is not annotated.

(*A-Resolution*) $\mathsf{comp}(A\ \gamma) \leftarrow \gamma \sqsubseteq (\alpha \sqcup \beta) \wedge \mathsf{comp}(B) \wedge \neg(\alpha \sqsubseteq \beta) \wedge \mathsf{comp}(A\ \beta)$

for each clause $A\alpha \leftarrow B$.

Summarizing, the translation proceeds clause-wise. What comp does is reifying annotated atoms, adding the *Bottom* clause for each annotated predicate, and adding special constraints to clauses for annotated predicates. Formulas without annotations are left unchanged. Note that the translation from ACLP to CLP can at most double the number of clauses.

Example 4.6. (Certainty, contd.) Taking the optimization of the interpreter into account, the result of compilation into CLP is:

$rain(R) \leftarrow R \leq 0.9 \wedge grass\_wet(0.8).$
$rain(R) \leftarrow R \leq V \wedge clouds(V).$
$grass\_wet(G) \leftarrow G \leq 1.0.$
$clouds(C) \leftarrow C \leq 0.5.$

4.3.2. Axioms

In a similar way, the translation rules for additional axioms and theorems are produced either directly or from the interpreter.

Example 4.7. (Certainty, contd.) We would obtain

$\mathsf{comp}((A \wedge B)\ min(\alpha, \beta)) \longrightarrow \mathsf{comp}(A\ \alpha) \wedge \mathsf{comp}(B\ \beta).$

4.4. Related Work

In [M. Kifer, V.S. Subrahmanian (1992)], additional clauses are derived from existing clauses, the so-called "reductants", to implement the GAP language. While reductants achieve the same as our closure clause, they result in a combinatorial explosion of the number of clauses in the program. Kifer and Subrahmanian write "The need to use reductants ... is another major obstacle [for implementation]. ... one my generate an infinite number of them out of a finite set of program clauses".

Recently, "ca-resolution" for annotated logic programs was proposed and implemented in C [S. M. Leach, J. J. Lu (1994)]. The idea is to compute dynamically and incrementally the reduction (resulting in the reductants in [M. Kifer, V.S. Subrahmanian (1992)]) by collecting partial answers. Operationally this is similar to our approach which relies on recursion to collect the partial answers [J. J. Lu, T. Frühwirth (1994)]. However, in [S. M. Leach, J. J. Lu (1994)] the class of programs considered is smaller and the intermediate stages of a reduction are not sound with respect to the standard CLP semantics.

5. Temporal Annotated Constraint Logic Programming

In this section we make temporal annotated logic (section 3) an instance of annotated constraint logic (section 4). Through this embedding, a clausal fragment of our temporal logic can be executed efficiently in a standard CLP language. The results of this chapter are summarized in tables 2 and 3.

5.1. Constraint Theory: Temporal Set Annotations

The embedding requires that the temporal annotation theorems (subsection 3.2) have to be consequences of the inference rules for annotated constraint logic (subsection 4.1.2) using an appropriate specialization of the constraint theory for the lattice operations (subsection 4.1.2). In other words, we want to reflect our theorems into axioms of the constraint theory (justified by the annotation inference rules), where reasoning is just performed on annotations and ergo easier.

The temporal set constraint theory includes the standard lattice axioms of the generic constraint theory (subsection 4.1.1) and the set theory axioms (ZF) of temporal annotated logic (subsection 3.1). The constraint theory for temporal set annotations is specialized by further axioms that we will derive now.

Lattice Order. From the annotation inference rule ($\sqsubseteq$) (subsection 4.1.2) we immediately obtain:

$$\beta \sqsubseteq \alpha \;\Rightarrow\; (A\ \alpha\ \Rightarrow A\ \beta)$$

To push temporal theorems of the form $(A\ \alpha\ \Rightarrow A\ \beta)$ into the constraint $\sqsubseteq$ we will try to use $(A\ \alpha\ \Rightarrow A\ \beta)$ as a "definition" for $\sqsubseteq$.

The theorems ($\{\}$), (1) and ($\subseteq$) (subsection 3.2) induce an equivalence class and the partial order for temporal annotations. The result is the following axiomatization:

$$\begin{array}{lllll}
(th\bot) & th\ \{\} = \bot & & \\
(in\top) & in\ \{\} = \top & & \\
(atth) & at\ t = th\ \{t\} & & \\
(atin) & at\ t = in\ \{t\} & & \\
(th\sqsubseteq) & th\ I \sqsubseteq th\ J & \Leftrightarrow & I \subseteq J \\
(in\sqsubseteq) & in\ I \sqsubseteq in\ J & \Leftrightarrow & J \subseteq I
\end{array}$$

Note that *at* annotations are incomparable. It is instructive to arrange the axioms in the following chain (assuming $I \supseteq J \supseteq \{t\}$):

$$\bot = th\,\{\} \sqsubseteq in\,I \sqsubseteq in\,J \sqsubseteq in\,\{t\} = at\,t = th\,\{t\} \sqsubseteq th\,J \sqsubseteq th\,I \sqsubseteq in\,\{\} = \top$$

Least upper bound. A useful theorem about $\sqcup$ can be derived using its general definition ($\sqcup$Def) (subsection 4.1.1) and the above axioms (see also the temporal theorem ($\cup th$))

$$(th\sqcup) \quad th\ I \sqcup th\ J = th\ (I \cup J)$$

The dual theorem $in\ I \sqcup in\ J = in\ (I \cap J)$ can also be derived in this way. However, it contradicts the first order definition of the *in* annotation in temporal annotated logic when used in the inference rule ($\sqcup$).

Example 5.1. (LUB in) According to the lattice

$$in\ \{1,2\} \sqcup in\ \{2,3\} = in\ \{1,2\} \cap \{2,3\} = in\ \{2\} = at\ 2$$

But

$$A\ in\ \{1,2\} \wedge A\ in\ \{2,3\} \Rightarrow A\ at\ 2$$

does not hold in temporal annotated logic, since $A\ at\ 1$ and $A\ at\ 3$ may hold. In other words, the least upper bound produced by the temporal annotation lattice is too large.

We can fix this problem by introducing additional annotations into the lattice. It suffices to allow for unevaluated least upper bound expressions.

Example 5.2. (LUB in, contd.) The right least upper bound of $in\ \{1,2\}$ and $in\ \{2,3\}$ is just the annotation $in\ \{1,2\} \sqcup in\ \{2,3\}$. This lattice element is smaller than $in\ \{2\}$, because both $in\ \{1,2\}$ and $in\ \{2,3\}$ are smaller than $in\ \{2\}$. $in\ \{1,3\}$, $in\ \{1\}$ and $in\ \{3\}$ are not comparable with $in\ \{1,2\} \sqcup in\ \{2,3\}$.

In practice, there is no need to compute the least upper bound of two *in* annotations, since it does not allow for more conclusions than with the original annotations. Therefore we can always replace $A\ in\ I \sqcup in\ J$ by the equivalent $A\ in\ I \wedge A\ in\ J$ (by variant 1 of the inference rule ($\sqcup$) in subsection 4.1.2).

5.2. Constraint Theory: Temporal Order and Time Periods

Now we consider a more common and richer temporal constraint theory that also reflects the structure of time: We further specialize the constraint theory to time periods and make use of the partial order that comes with them.

Let t, s_1, s_2, r_1, r_2 be time points within the bounds lb and ub, and $[s_1, s_2]$ and $[r_1, r_2]$ be time periods. First of all, the constraint theory includes the axioms defining the order on time points, $\leq$. The lattice constraint theory for temporal annotations is further specialized.

Lattice Order.

$$\begin{array}{lllll}
(th\bot) & th\ [ub, lb] = \bot & & \\
(in\top) & in\ [ub, lb] = \top & & \\
(atth) & at\ t = th\ [t, t] & & \\
(atin) & at\ t = in\ [t, t] & & \\
(th \sqsubseteq) & th\ [s_1, s_2] \sqsubseteq th\ [r_1, r_2] & \Leftrightarrow & r_1 \leq s_1 \wedge s_2 \leq r_2 \\
(in \sqsubseteq) & in\ [r_1, r_2] \sqsubseteq in\ [s_1, s_2] & \Leftrightarrow & r_1 \leq s_1 \wedge s_2 \leq r_2
\end{array}$$

An interesting consequence is

$$(ti \sqsubseteq) \quad in\ [s_1, s_2] \sqsubseteq th\ [r_1, r_2] \quad \Leftrightarrow \quad s_1 \leq r_2 \wedge r_1 \leq s_2$$

i.e. A formula annotated by *in* holds in any time period that overlaps with a time period where the formula holds throughout.

Least Upper Bound. In practice, we only need to consider the least upper bound for time periods that produce another different single time period. Therefore we can restrict ourselves to *th* annotations with overlapping time periods that do not include one another. This is the only case where "new" information (a bigger time period) is produced.Without loss of generality thanks to commutativity, we can also require that the the time period $[s_1, s_2]$ is the one that starts before the time period $[r_1, r_2]$, i.e. that $s_1 \leq r_1 \wedge r_1 \leq s_2 \wedge s_2 \leq r_2$. We are left with a useful theorem:

$$(th\sqcup) \quad th\ [s_1, s_2] \sqcup th\ [r_1, r_2] = th\ [s_1, r_2] \quad \Leftrightarrow \quad s_1 \leq r_1 \wedge r_1 \leq s_2 \wedge s_2 \leq r_2$$

Example 5.3. (LUB) The annotation $th\ [1, 3] \sqcup th\ [2, 5]$ is the same as $th\ [1, 5]$ and greater than $th\ [2, 3]$, while $in\ [1, 3] \sqcup in\ [2, 5]$ is smaller than $in\ [2, 3]$ but also greater than $in\ [1, 5]$.

This constraint theory for time periods in temporal annotated logic improves on the more complex one to be found in [T. Frühwirth (1995)]. Table 2 gives an overview of the generic and specific lattice constraint theories we have developed and the associated theorems of temporal annotated logic.

Complexity. Note that all the lattice constraints on time periods can be reduced to conjunctions of order constraints between time points. Such constraints can be solved in $O(n^3)$ time complexity, where n is the number of time point variables, by applying a path consistency algorithm [I. Meiri (1991)].

Actually, the complexity result applies to a larger interesting class of constraints, those involving distance of time points or duration of intervals. We can admit constraints that reduce to the normal form

$$s_1 + d \leq s_2 \quad \text{or} \quad s_1 \leq s_2 + d$$

where d is a temporal constant. This means that in the annotations we can use expressions with duration, e.g. $at\ t + d$ and $th\ [t, t + d]$. Such constraints naturally appear in many temporal reasoning applications, e.g. scheduling.

Example 5.4. (Mystery, contd.) With durations we are able to compute in what time period Prof. Lepov was murdered and when Dr. Kosta was on the shuttle.

```
murdered(X) in [T1-1:30,T2-1:00] :-                     % hint (3)
        found_dead(X) in [T1,T2].

% Dr. Kosta's Alibi

   shuttle at 0:00.                                     % hint (5)
   shuttle at T+30 :-
        (in [0:00,11:00]) =< (at T), shuttle at T.

on_shuttle(X) th [T1,T2] :-                             % hint (6)
        T2 = T1+50, shuttle at T1, board_plane(X) in [T2,T2+50].
```

Note that the causal relationships expressed through head and body of the clauses can reason into the past (clause for hint (3)), into the future (hint (5)) or both (hint (6)), as is convenient.

5.3. TACLP Interpreter

In the programming language, we implement the case of single time periods. This does not result in a loss of generality except for *in* annotations that appear in the head of clause as the consideration below shows. To keep the presentation simple we assume also that *at* annotations are rewritten into the equivalent *th* annotations using (*atth*).

Remember that formulas labeled by annotations with non-convex sets can be rewritten by thereoms ($\cup th$) and ($\cup in$) involving only convex sets (i.e. time periods). Conjunctions coming from rewritten *th* annotations now appearing in the head of a clause can be rewritten into clauses whose head have one of the conjuncts each. With disjunctions from *in* annotations this is not possible and so the ACLP clausal form would be violated.

The computational disadvantage of this approach is that some unnecessary choices (between the rewritten clauses) and repeated work (through the conjunctions in the bodies) may be introduced. However, in many applications, the number of conjuncts is rather small. The advantage is that the constraint theory is the simple one from subsection 5.2: Only for overlapping *th* annotations, the least upper bound has to be computed.

Table 2. The annotation lattice constraint theories at a glance

Annotated Constraint Logic, Sections 4 and 4.1.1	
($\sqsubseteq$Reflexivity)	$\alpha \sqsubseteq \alpha$
($\sqsubseteq$Anti-Symmetry)	$\alpha \sqsubseteq \beta \wedge \beta \sqsubseteq \alpha \leftrightarrow \alpha = \beta$
($\sqsubseteq$Transitivity)	$\alpha \sqsubseteq \beta \wedge \beta \sqsubseteq \gamma \rightarrow \alpha \sqsubseteq \gamma$
($\sqsubseteq$Bottom)	$\bot \sqsubseteq \alpha$
($\sqsubseteq$Top)	$\alpha \sqsubseteq \top$
($\sqcup$Def)	$\alpha \sqsubseteq (\alpha \sqcup \beta) \wedge \beta \sqsubseteq (\alpha \sqcup \beta) \ \wedge \ \forall\gamma(\alpha \sqsubseteq \gamma \wedge \beta \sqsubseteq \gamma \rightarrow (\alpha \sqcup \beta) \sqsubseteq \gamma)$
($\sqcup$Idempotency)	$\alpha \sqcup \alpha \ = \ \alpha$
($\sqcup$Commutativity)	$\alpha \sqcup \beta \ = \ \beta \sqcup \alpha$
($\sqcup$Associativity)	$\alpha \sqcup (\beta \sqcup \gamma) \ = \ (\alpha \sqcup \beta) \sqcup \gamma$
($\sqcup$Bottom)	$\bot \sqcup \alpha \ = \ \alpha$
($\sqcup$Top)	$\top \sqcup \alpha \ = \ \top$
Relevant Temporal Annotation Theorems 3.2	
($\{\}th$)	$A\ th\ \{\} \Leftrightarrow true$
($\{\}in$)	$A\ in\ \{\} \Leftrightarrow false$
($1th$)	$A\ at\ t \Leftrightarrow A\ th\ \{t\}$
($1in$)	$A\ at\ t \Leftrightarrow A\ in\ \{t\}$
($\subseteq th$)	$A\ th\ I \Leftrightarrow \forall J\ (J \subseteq I \rightarrow A\ th\ J)$
($\subseteq in$)	$A\ in\ I \Leftrightarrow \forall J\ (J \supseteq I \rightarrow A\ in\ J)$
($\cup th$)	$A\ th\ I \cup J \Leftrightarrow (A\ th\ I \wedge A\ th\ J)$
Definition by Inference Rules of Annotated Constraint Logic 4.1.2	
($\sqsubseteq$)	$\beta \sqsubseteq \alpha \ \Rightarrow \ (A\ \alpha \ \Rightarrow A\ \beta)$
($\sqcup$)	$\gamma \ = \ \beta \sqcup \alpha \ \Rightarrow \ (A\ \alpha \ \wedge A\ \beta \Leftrightarrow A\ \gamma)$
Temporal Set Annotations 5.1	
(ZF)	Zermelo-Fränkel axioms for set theory
($th\bot$)	$th\ \{\} \ = \ \bot$
($in\top$)	$in\ \{\} \ = \ \top$
($atth$)	$at\ t \ = \ th\ \{t\}$
($atin$)	$at\ t \ = \ in\ \{t\}$
($th \sqsubseteq$)	$th\ I \ \sqsubseteq \ th\ J \ \Leftrightarrow \ I \subseteq J$
($in \sqsubseteq$)	$in\ I \ \sqsubseteq \ in\ J \ \Leftrightarrow \ J \subseteq I$
($th\sqcup$)	$th\ I \ \sqcup th\ J \ = \ th\ (I \cup J)$
Time Period Annotations 5.2	
($\leq$)	Axioms defining the order on time points
($th\bot$)	$th\ [ub, lb] \ = \ \bot$
($in\top$)	$in\ [ub, lb] \ = \ \top$
($atth$)	$at\ t \ = \ th\ [t, t]$
($atin$)	$at\ t \ = \ in\ [t, t]$
($th \sqsubseteq$)	$th\ [s_1, s_2] \ \sqsubseteq \ th\ [r_1, r_2] \ \Leftrightarrow \ r_1 \leq s_1 \wedge s_2 \leq r_2$
($in \sqsubseteq$)	$in\ [r_1, r_2] \ \sqsubseteq \ in\ [s_1, s_2] \ \Leftrightarrow \ r_1 \leq s_1 \wedge s_2 \leq r_2$
($th\sqcup$)	$th\ [s_1, s_2] \sqcup th\ [r_1, r_2] = th\ [s_1, r_2] \ \Leftrightarrow \ s_1 \leq r_1 \wedge r_1 \leq s_2 \wedge s_2 \leq r_2$

The generic interpreter for ACLP can be specialized to TACLP in a straightforward way. The standard interpreter clauses (including *(Modus Ponens)* are the same as in the generic case.

5.3.1. Inference Rules

The interpreter clause (*Bottom*) for annotated constraint logic is specialized to one for temporal annotated logic

$\mathsf{prove}(A\ th\ [ub, lb]))$.

The interpreter clause *(A-Resolution)* can be specialized into three instances reflecting the temporal annotation lattice constraint theory. For th annotations we obtain

$$\begin{aligned}&\mathsf{prove}(A\ th\ [t_1, t_2]) \leftarrow\\ &\quad th\ [t_1, t_2] \sqsubseteq (th\ [s_1, s_2] \sqcup th\ [r_1, r_2]) \wedge \mathsf{clause}(A\ th\ [s_1, s_2], B) \wedge\\ &\quad \mathsf{prove}(B) \wedge \neg(th\ [s_1, s_2] \sqsubseteq th\ [r_1, r_2]) \wedge \mathsf{prove}(A\ th\ [r_1, r_2])).\end{aligned}$$

which can be rewritten using ($th\sqcup$) and ($th \sqsubseteq$) from the temporal constraint theory. The negation $\neg(th\ [s_1, s_2] \sqsubseteq th\ [r_1, r_2])$ becomes redundant then. However, if we just want to use ($th\sqcup$) alone, the (*Bottom*) interpreter clause is no longer applicable to terminate the recursion via $\mathsf{prove}(A\ th\ [r_1, r_2]))$. We therefore introduce an explicit disjunction that allows the time period $[r_1, r_2]$ to coincide with $[s_1, s_2]$.

$$\begin{aligned}&(th \sqsubseteq)\ \ \mathsf{prove}(A\ th\ [t_1, t_2]) \leftarrow\\ &\quad (s_1 \leq t_1 \wedge t_2 \leq r_2) \wedge (s_1 \leq r_1 \wedge r_1 \leq s_2 \wedge s_2 \leq r_2) \wedge \mathsf{clause}(A\ th\ [s_1, s_2], B) \wedge\\ &\quad \mathsf{prove}(B) \wedge (r_1 = s_1 \wedge r_2 = s_2 \vee \mathsf{prove}(A\ th\ [r_1, r_2]))).\end{aligned}$$

For the in annotations, the recursive call in the interpreter clause can be simplified away, since there is no need to compute the least upper bound for in annotations at all.

$(in \sqsubseteq)$ $\mathsf{prove}(A\ in\ [t_1, t_2]) \leftarrow (t_1 \leq s_1 \wedge s_2 \leq t_2) \wedge \mathsf{clause}(A\ in\ [s_1, s_2], B) \wedge \mathsf{prove}(B)$.
$(ti \sqsubseteq)$ $\mathsf{prove}(A\ in\ [t_1, t_2]) \leftarrow (t_1 \leq s_2 \wedge s_1 \leq t_2) \wedge \mathsf{clause}(A\ th\ [s_1, s_2], B) \wedge \mathsf{prove}(B)$.

5.3.2. Axioms

The theorems of our temporal annotated logic about non-atomic annotated formulas (subsection 3.2) have to be axioms for temporal annotated constraint logic and thus introduce additional interpreter clauses. We try to reduce every non-atomic or negative annotated formula to an equivalent formula where only atoms are annotated, since only those can be resolved with a TACLP clause (whose head is restricted to be an atom).
Negation. The interpreter clauses for annotated negation are:

$\mathsf{prove}((\neg A)\ th\ I) \leftarrow \neg\mathsf{prove}(A\ in\ I)$.
$\mathsf{prove}((\neg A)\ in\ I) \leftarrow \neg\mathsf{prove}(A\ th\ I)$.

Disjunction and Conjunction. The interpreter clauses derived from the theorems

$(\wedge th)$ and $(\vee in)$ in subsection 3.2 are obvious, too.

$$\mathsf{prove}((A \wedge B)\ th\ I) \leftarrow \mathsf{prove}(A\ th\ I) \wedge \mathsf{prove}(B\ th\ I)$$
$$\mathsf{prove}((A \vee B)\ in\ I) \leftarrow \mathsf{prove}(A\ in\ I) \vee \mathsf{prove}(B\ in\ I)$$

The theorem $(\wedge in)$ does not give us an equivalence that could be used to define an interpreter clause, intuitively on the right hand side the information that A and B happen at the *same* time is lost. Since there is no such equivalence this also means that we cannot express that two predicates hold at the same unknown time in the first place, because this would result in a non-atomic clause head, i.e. $A \wedge B\ in\ I$. Because of this limitation, it suffices to implement $A \wedge B\ in\ I$ according to its first order definition (in) (subsection 3.1) where at is replaced by th.

THEOREM 5.1. $(\wedge inth)$ *The corresponding theorem is*

$$(\wedge inth) \quad (A \wedge B)\ in\ I\ \Leftrightarrow \exists J\ (in\ I \sqsubseteq th\ J\ \wedge A\ th\ J \wedge B\ th\ J).$$

The proof can be found in the appendix. The interpreter clause thus is

$$\mathsf{prove}((A \wedge B)\ in\ [t_1, t_2]) \leftarrow (t_1 \leq s_2 \wedge s_1 \leq t_2) \wedge \mathsf{prove}(A\ th\ [s_1, s_2]) \wedge \mathsf{prove}(B\ th\ [s_1, s_2])$$

Similarly, the theorem $(\vee th)$ is not an equivalence. We cannot express that either A or B holds throughout some time period, since $(A \vee B\ th\ I)$ cannot be rewritten and is not allowed in the head of a TACLP clause. Because of this limitation, given $(A \vee B\ th\ I)$, at each time point in I, at least one of A or B *definitely* has to hold. In this case, theorem $(\vee th)$ is sufficient.

$$\mathsf{prove}((A \vee B)\ th\ I) \leftarrow \mathsf{prove}(A\ th\ I) \vee \mathsf{prove}(B\ th\ I)$$

INDEFINITENESS. In the treatment of non-atomic annotated formulas we are confronted with the limitations of reasoning with indefinite information in logic programming. This indefiniteness occurs either as annotated disjunctive formula or as *in* annotated formula. Of these formulas, only *in* annotated atoms can be the head of a TACLP clause. Still, this means that unlike in CLP languages, a limited kind of disjunction (all disjuncts involve the same predicate) is available in TACLP clause heads. We call predicates that appear with an *in* annotation in the head of a clause *indefinite predicates.*

Negation and disjunction can appear in the body of a clause, but as implemented in CLP languages they are in general too weak to handle indefinite information, even if the information is expressed as *in* annotated formula. The following examples illustrate this limitation.

EXAMPLE 5.5. (INDEFINITENESS, CONTD.) Given the clause

```
born in [1964,1965].
```

the query `:-prove(born at 1965)` fails and the query `:-prove((not born) at 1965))` succeeds even though we cannot tell if `born at 1965` holds or not.

The problem is that by negation as failure we succeed with a query if we cannot prove the negated query. A practical solution to the problem is to simply dissallow that indefinite predicates appear inside negated formulas.

Example 5.6. (Indefiniteness, contd.) The limitation of disjunctive queries about indefinite predicates shows up in the query

```
:- prove((born at 1964 ; born at 1965)).
```

fails, since neither disjunct holds on its own, even though the disjunction does hold. The logically equivalent query `:- prove(born in [1964,1965])` correctly succeeds. In practice, it will be unlikely that the first form of the query is used instead of the more compact and natural second one.

5.4. TACLP Compiler

We define a compilation function comp that translates a TACLP clause into its CLP form. The compilation of predicates and non-annotated formulas is the same as in the generic case.

5.4.1. Inference Rules

For each predicate symbol p with arity n in the program we add a clause

$\mathsf{comp}(p(X_1, ...X_n)\ th\ [ub, lb])$

where X_i are distinct variables.

The compiler for the inference rules can be obtained from the TACLP interpreter clauses in the same way as in the generic case for ACLP clauses.

$$(th \sqsubseteq)\ \ \mathsf{comp}(A\ th\ [t_1, t_2]) \leftarrow \\ (s_1 \leq t_1 \wedge t_2 \leq r_2) \wedge (s_1 \leq r_1 \wedge r_1 \leq s_2 \wedge s_2 \leq r_2) \wedge \\ \mathsf{comp}(B) \wedge (r_1 = s_1 \wedge r_2 = s_2 \vee \mathsf{comp}(A\ th\ [r_1, r_2])))$$

for each clause of the form $(A\ th\ [s_1, s_2] \leftarrow B)$.

$$(in \sqsubseteq)\ \mathsf{comp}(A\ in\ [t_1, t_2]) \leftarrow (t_1 \leq s_1 \wedge s_2 \leq t_2) \wedge \mathsf{comp}(B)$$

for each clause of the form $(A\ in\ [s_1, s_2] \leftarrow B)$.

$$(ti \sqsubseteq)\ \mathsf{comp}(A\ in\ [t_1, t_2]) \leftarrow (t_1 \leq s_2 \wedge s_1 \leq t_2) \wedge \mathsf{comp}(B)$$

for each clause of the form $(A\ th\ [s_1, s_2] \leftarrow B)$.

Program clauses whose head is not annotated are translated as in the generic case.

In the worst case (only *th* annotated predicates defined by a single clause each), for each TACLP clause three CLP clauses are produced by the compilation.

5.4.2. Axioms

The compilation rules for the axioms can be obtained from the interpreter clauses by replacing `prove` with `comp` and $\leftarrow$ with $\longrightarrow$ (see table 3).

5.5. Related Work

An efficient optimized interpreter for TACLP where only atomic formulas can be annotated has been implemented in the constraint logic programming platform ECLiPSe [M. Meier, J. Schimpf et al. (1994)] and is described in [T. Frühwirth (1994)].

In [M. Kifer, V.S. Subrahmanian (1992)], Templog and an interval based temporal logic are translated into GAP. The annotations used correspond to our th annotations. It is also shown that the temporal logic of Shoham can be encoded in GAP.

In [C. Brzoska (1993)], a powerful temporal logic named MTL (tense logic extended by parameterized temporal operators) is translated into first order constraint logic. The resulting constraint theory is rather complex as it involves quantified variables and implication, whose treatment goes beyond standard CLP implementations. In Brzoskas programming language, which subsumes Templog, temporal operators can be nested, but indefiniteness in the heads of clauses is dissallowed [C. Brzoska (1993)]. In our implementation, we can allow indefiniteness even in the head of a clause - but it cannot be queried negatively. The main conceptual difference to our approach is that Brzoska implements this FOL itself, while we derive a constraint-based implementation of annotations from the FOL definitions.

6. The Workshop Murder Mystery Solved

We can now express the complete murder mystery as TACLP (fig. 2). The program should be self-explanatory. The idea is that the murderer is a person who is involved in the case and does not have an alibi during the time Prof. Lepov was murdered.

The query `:- prove(murder(X,Y))` yields two answers `X = 'Lepov', Y = 'Lepov'` and `X = 'Maringer', Y = 'Lepov'`. The first one means that Prof. Lepov could have committed suicide. This unexpected solution is found because Prof. Lepov does not have an alibi for the time of his death. Dr. Maringer could be the murderer, because his alibi does not hold. Analysis of the failure of `alibi('Maringer') th I` reveals that Maringer gave a wrong alibi, because the copying would have taken 30 minutes, so it cannot have happened during a talk of 25 minutes. Dr. Kostas alibi holds.

7. Conclusions

We have defined a temporal annotated logic allowing for various models of time and various temporal operators for both time points (instants) and time periods (temporal intervals). Temporal annotated formulas avoid the proliferation of temporal variables and quantifiers of the first order approach while making temporal information explicit. In TACLP, we can reason about qualitative and quantitative (metric), definite and indefinite information about the absolute and relative location of literals annotated with time points and time periods along the time line.

We have introduced the general framework of annotated constraint logic. Given a logic in this framework, there is a systematic way to make a clausal fragment executable as a

Table 3. The ACLP and TACLP implementations at a glance

	Annotated Constraint Logic - Inference Rules	Sections 4 and 4.1.2
(Modus Ponens)	$\frac{B\ ,\ (B \rightarrow A)}{A}$	
(Bottom)	$A\ \bot$	
(A-Resolution)	$\frac{A\ \alpha\ ,\ B\ ,\ (B \rightarrow A\ \beta),\ \gamma \sqsubseteq (\alpha \sqcup \beta)}{A\ \gamma}$	
	Interpreter	4.2
Standard	$\mathsf{prove}(A) \leftarrow \mathsf{constraint}(A) \wedge A.$	
	$\mathsf{prove}(\neg A) \leftarrow \neg\mathsf{prove}(A).$	
	$\mathsf{prove}(A \wedge B) \leftarrow \mathsf{prove}(A) \wedge \mathsf{prove}(B).$	
	$\mathsf{prove}(A \vee B) \leftarrow \mathsf{prove}(A) \vee \mathsf{prove}(B).$	
(Modus Ponens)	$\mathsf{prove}(A) \leftarrow \mathsf{non\_annotated}(A) \wedge \mathsf{clause}(A, B) \wedge \mathsf{prove}(B).$	
(Bottom)	$\mathsf{prove}(A\ \bot).$	
(A-Resolution)	$\mathsf{prove}(A\ \gamma) \leftarrow \gamma \sqsubseteq (\alpha \sqcup \beta) \wedge \mathsf{clause}(A\ \alpha, B) \wedge \mathsf{prove}(B) \wedge \neg(\alpha \sqsubseteq \beta) \wedge \mathsf{prove}(A\ \beta).$	
	Compiler	4.3
Predicates	$\mathsf{comp}(p(t_1, \ldots, t_n)\ \alpha) \longrightarrow p(t_1, \ldots, t_n, \alpha)$	
	$\mathsf{comp}(p(t_1, \ldots, t_n)) \longrightarrow p(t_1, \ldots, t_n)$	
Standard	$\mathsf{comp}(A) \longrightarrow A$ if A is a constraint	
	$\mathsf{comp}(\neg A) \longrightarrow \neg\mathsf{comp}(A)$	
	$\mathsf{comp}(A \wedge B) \longrightarrow \mathsf{comp}(A) \wedge \mathsf{comp}(B)$	
	$\mathsf{comp}(A \vee B) \longrightarrow \mathsf{comp}(A) \vee \mathsf{comp}(B)$	
(*Modus Ponens*)	$\mathsf{comp}(A) \leftarrow \mathsf{comp}(B)$ for each clause $(A \leftarrow B)$	
(Bottom)	$\mathsf{comp}(p(X_1, ...X_n)\ \bot)$ for each annotated predicate p with arity n	
(*A-Resolution*)	$\mathsf{comp}(A\ \gamma) \leftarrow \gamma \sqsubseteq (\alpha \sqcup \beta) \wedge \mathsf{comp}(B) \wedge \neg(\alpha \sqsubseteq \beta) \wedge \mathsf{comp}(A\ \beta)$ f.e.cl. $(A\ \alpha \leftarrow B)$	
	Temporal Annotated Logic	5
	TACLP Interpreter	5.3
(Bottom)	$\mathsf{prove}(A\ th\ [ub, lb])).$	
(*A-Resolution*)	$\mathsf{prove}(A\ th\ [t_1, t_2]) \leftarrow (s_1 \leq t_1 \wedge t_2 \leq r_2) \wedge (s_1 \leq r_1 \wedge r_1 \leq s_2 \wedge s_2 \leq r_2) \wedge$	
(*th* $\sqsubseteq$)	$\mathsf{clause}(A\ th\ [s_1, s_2], B) \wedge \mathsf{prove}(B) \wedge (r_1 = s_1 \wedge r_2 = s_2 \vee \mathsf{prove}(A\ th\ [r_1, r_2]))).$	
(*in* $\sqsubseteq$)	$\mathsf{prove}(A\ in\ [t_1, t_2]) \leftarrow (t_1 \leq s_1 \wedge s_2 \leq t_2) \wedge \mathsf{clause}(A\ in\ [s_1, s_2], B) \wedge \mathsf{prove}(B).$	
(*ti* $\sqsubseteq$)	$\mathsf{prove}(A\ in\ [t_1, t_2]) \leftarrow (t_1 \leq s_2 \wedge s_1 \leq t_2) \wedge \mathsf{clause}(A\ th\ [s_1, s_2], B) \wedge \mathsf{prove}(B).$	
Negation	$\mathsf{prove}((\neg A)\ th\ I) \leftarrow \neg\mathsf{prove}(A\ in\ I).$	
	$\mathsf{prove}((\neg A)\ in\ I) \leftarrow \neg\mathsf{prove}(A\ th\ I).$	
Conjunction	$\mathsf{prove}((A \wedge B)\ th\ I) \leftarrow \mathsf{prove}(A\ th\ I) \wedge \mathsf{prove}(B\ th\ I)$	
	$\mathsf{prove}((A \wedge B)\ in\ [t_1, t_2]) \leftarrow t_1 \leq s_2 \wedge s_1 \leq t_2 \wedge \mathsf{p}(A\ th\ [s_1, s_2]) \wedge \mathsf{p}(B\ th\ [s_1, s_2])$	
Disjunction	$\mathsf{prove}((A \vee B)\ in\ I) \leftarrow \mathsf{prove}(A\ in\ I) \vee \mathsf{prove}(B\ in\ I)$	
	$\mathsf{prove}((A \vee B)\ th\ I) \leftarrow \mathsf{prove}(A\ th\ I) \vee \mathsf{prove}(B\ th\ I)$	
	TACLP Compiler	5.4
(Bottom)	$\mathsf{comp}(p(X_1, ...X_n)\ th\ [ub, lb])$	
(*A-Resolution*)	$\mathsf{comp}(A\ th\ [t_1, t_2]) \leftarrow (s_1 \leq t_1 \wedge t_2 \leq r_2) \wedge (s_1 \leq r_1 \wedge r_1 \leq s_2 \wedge s_2 \leq r_2) \wedge$	
(*th* $\sqsubseteq$)	$\mathsf{comp}(B) \wedge (r_1 = s_1 \wedge r_2 = s_2 \vee \mathsf{comp}(A\ th\ [r_1, r_2])))$ f.e.cl. $(A\ th\ [s_1, s_2] \leftarrow B)$	
(*in* $\sqsubseteq$)	$\mathsf{comp}(A\ in\ [t_1, t_2]) \leftarrow (t_1 \leq s_1 \wedge s_2 \leq t_2) \wedge \mathsf{comp}(B)$ f.e.cl. $(A\ in\ [s_1, s_2] \leftarrow B)$	
(*ti* $\sqsubseteq$)	$\mathsf{comp}(A\ in\ [t_1, t_2]) \leftarrow (t_1 \leq s_2 \wedge s_1 \leq t_2) \wedge \mathsf{comp}(B)$ f.e.cl. $(A\ th\ [s_1, s_2] \leftarrow B)$	
Negation	$\mathsf{comp}(\neg A)\ th\ I) \longrightarrow \neg\mathsf{comp}((A\ in\ I))$	
	$\mathsf{comp}(\neg A)\ in\ I) \longrightarrow \neg\mathsf{comp}((A\ th\ I))$	
Conjunction	$\mathsf{comp}((A \wedge B)\ th\ I) \longrightarrow \mathsf{comp}(A\ th\ I) \wedge \mathsf{comp}(B\ th\ I)$	
	$\mathsf{comp}((A \wedge B)\ in\ [t_1, t_2]) \longrightarrow t_1 \leq s_1 \wedge s_2 \leq t_2 \wedge \mathsf{c}(A\ th\ [s_2, s_1]) \wedge \mathsf{c}(B\ th\ [s_2, s_1])$	
Disjunction	$\mathsf{comp}((A \vee B)\ in\ I) \longrightarrow \mathsf{comp}(A\ in\ I) \vee \mathsf{comp}(B\ in\ I)$	
	$\mathsf{comp}((A \vee B)\ th\ I) \longrightarrow \mathsf{comp}(A\ th\ I) \vee \mathsf{comp}(B\ th\ I)$	

```
% The Workshop Program
%...                                                    % (1) time periods
coffee_break th [3:00,3:25].
talk(1,'Hunon','Algebraic Semantics...') th [3:25,3:50].
talk(2,'...','...') th [3:50,4:15].
talk(3,'Maringer','...') th [4:15,4:40].
talk(4,'Lepov','P = NP') th [4:40,5:05].
%...

% The Murder of Prof. Lepov

found_dead('Lepov') at 5:35.                            % (2) time point

murdered(X) in [T1-1:30,T2-1:00] :-                     % (3) indefiniteness
        found_dead(X) in [T1,T2].

% Dr. Kosta's Alibi

board_plane('Kosta') at 5:10.                           % (4) time point

   shuttle at 0:00.                                     % (5) recurrence
   shuttle at T+30 :-
        (in [0:00,11:00]) =< (at T), shuttle at T.

on_shuttle(X) th [T1,T2] :-                             % (6) indefiniteness
        T2 = T1+50, shuttle at T1, board_plane(X) in [T2,T2+50].

% Dr. Maringer's Alibi

copying('Maringer') th I :-                             % (7) durations
        I = [T1,T1+5+5+15+5], talk(2,_,_) th I.

% Whodunnit ?

murder(X,Y) :-                                          % (8) time periods
        murdered(Y) in I, involved(X), not (alibi(X) th I).

        involved('Kosta'). involved('Lepov'). involved('Maringer').

alibi(X) th I :-
        (on_shuttle(X) ; copying(X) ; talk(_,X,_)) th I.
```

Figure 2. The Workshop Murder Mystery as TACLP Program.

constraint logic program. We have shown this for the generic case and for the specific case of temporal annotated logic. In both cases we have given an interpreter and a compiler that can be implemented in constraint logic programming languages.

The clausal fragment allows for arbitrary formulas in the body of a clause. We can allow indefiniteness even in the head of a clause in the form of an *in* annotated atom. However, such indefinite predicates should not appear in negated body formulas, since negation as failure cannot cope well with indefiniteness. With TACLP clauses, we cannot express other types of indefinitess, namely that either A or B holds at a time point or throughout a time period ("Either John or Richard were born in 1964") and that A and B hold at the same, unknown time point(s) ("John is born in the same year as Suzy").

We have not yet developed a constraint theory for nested annotations, i.e. expressions like $rain\ th\ [3pm, 5pm]\ in\ [12.Dec, 19.Dec]$ ("it rained from 3 to 5 pm sometime in the week from December 12 to December 19"). We think it will be straightforward to define instances of temporal annotated logic that explicitly supports a limited kind of nesting or notions such as time granularity and recurrency. We would like to experiment with realistic applications. All this is left for future work.

Acknowledgements. Thanks to the ESPRIT LAC working group members for encouragement and helpful discussions on draft versions of this paper, especially to Camilla Schwind, Christoph Brzoska and Barry Richards, as well as to Christel Seguin and Jan Chomicki. The final version benefited greatly from detailed comments of the reviewers. Last but not least thanks to my wife Andrea and my daughter Anna for their patience.

References

M. Abada, Z. Manna (1989) Temporal Logic Programming, *J. Symbolic Computation* **8**, 277–295.

J. F. Allen (1984) Towards a General Theory of Action and Time, *Artificial Intelligence* **23**, 123–154.

J. K. van Benthem (1983) *The Logic of Time*, Synthese Library **156**, D. Reidel Publications.

C. Brzoska (1993) Temporal Logic Programming with Bounded Universal Goals, 10th ICLP, Budapest, Hungary, MIT Press, 1993.

M. Fisher, R. Owens (1992) From the Past to the Future: Executing Temporal Logic Programs, *Proceedings LPAR 92*, Springer LNCS 624, 369–380.

M. Fisher, R. Owens (Eds.) (1995) *Executable Modal and Temporal Logics*, Springer LNAI 897, March 1995.

A. M. Frisch, R. B. Scherl (1991) A General framework for Modal Deduction, *Proceedings 2nd KR '91*, Morgan Kaufmann, 196–207.

T. Frühwirth et al. (1992) Constraint Logic Programming - An Informal Introduction, *Logic Programming in Action*, Springer LNCS 636.

T. Frühwirth (1994) Annotated Constraint Logic Programming Applied to Temporal Reasoning, *Programming Language Implementation and Logic Programming (PLILP)*, Springer LNCS 844, 230–243.

T. Frühwirth (1995) Temporal Logic and Annotated Constraint Logic Programming, *Executable Modal and Temporal Logics*, M. Fisher and R. Owens (Eds.), Springer LNAI 897, 58–68.

A. Galton (Ed.) (1987) *Temporal Logics and Their Applications*, Academic Press.

A. Galton (1990) A Critical Examination of Allen's Theory of Action and Time, *Artificial Intelligence* **42**, 159–188.

J. Jaffar, M. J. Maher (1994) Constraint Logic Programming: A Survey, *Journal of Logic Programming* **19,20**, 503–581.

M. Kifer, V.S. Subrahmanian (1992) Theory of Generalized Annotated Logic Programming and its Applications, *Journal of Logic Programming* **12**, 335–367.

S. Kono (1995) A Combination of Clausal and Non Clausal Temporal Logic Programs, *Executable Modal and Temporal Logics*, M. Fisher and R. Owens (Eds.), Springer LNAI 897, 40–57.

S. M. Leach, J. J. Lu (1994) Computing Annotated Logic Programs: Theory and Implementation, *Proceedings 11th ICLP*, MIT Press.

J. J. Lu, T. Frühwirth (1994) Private communication at the 11th ICLP, Santa Margherita Ligure, Italy, 1994.

D. McDermot (1982) A Temporal Logic for Reasoning about Processes and Plans, *Cognitive Science* **6**,101–155.

M. Meier, J. Schimpf et al. (1994) ECLiPSe 3.4 User Manual and Extensions User Manual, ECRC Munich, Germany.

I. Meiri (1991) Combining Qualitative and Quantitative Constraints in Temporal Reasoning, *Proceedings AAAI 91*, 260–267.

B. Moszkowski (1986) *Executing Temporal Logic Programs*, Cambridge University Press.

M. A. Orgun, W. Ma (1994) An Overview of Temporal and Modal Logic Programming, Department of Computing, Macquarie University, Sydney, Australia.

L. Sterling, E. Shapiro (1994) "Interpreters", Chapter 17 in *The Art of Prolog*, Second Edition, MIT Press.

P. van Hentenryck (1991) Constraint Logic Programming, *The Knowledge Engineering Review* **6**, 151–194.

A. Proofs of Theorems about Temporal Annotations

The proofs for the theorems in sections 3.2 and 5.3.2.

Proof. $[\vee in]$

$(A \vee B)\ in\ I \ \Leftrightarrow$
$\exists t\ (t \in I \wedge (A \vee B)\ at\ t) \ \Leftrightarrow$
$\exists t\ (t \in I \wedge (A\ at\ t \vee B\ at\ t)) \ \Leftrightarrow$
$\exists t\ ((t \in I \wedge A\ at\ t) \vee (t \in I \wedge B\ at\ t)) \ \Leftrightarrow$
$((A\ in\ I) \vee (B\ in\ I))$

The proof is analogous for the th annotation. □

Proof. $[\vee th]$

$(A \vee B)\ th\ I \ \Leftrightarrow$
$\forall t\ (t \in I \rightarrow (A \vee B)\ at\ t) \ \Leftrightarrow$
$\forall t\ (t \in I \rightarrow (A\ at\ t \vee B\ at\ t)) \ \Leftrightarrow$
$\forall t\ ((t \in I \rightarrow A\ at\ t) \vee (t \in I \rightarrow B\ at\ t)) \ \Leftarrow$
$\forall t\ (t \in I \rightarrow A\ at\ t) \vee \forall t\ (t \in I \rightarrow B\ at\ t) \ \Leftrightarrow$
$(A\ th\ I \vee B\ th\ I)$

The proof is analogous for the in annotation. □

Proof. $[\cup th]$

$A\ th\ I \cup J \ \Leftrightarrow$
$\forall t\ (t \in I \cup J \rightarrow A\ at\ t) \ \Leftrightarrow$
$\forall t\ ((t \in I \vee t \in J) \rightarrow A\ at\ t) \ \Leftrightarrow$
$\forall t\ ((t \in I \rightarrow A\ at\ t) \wedge (t \in J \rightarrow A\ at\ t)) \ \Leftrightarrow$
$\forall t\ (t \in I \rightarrow A\ at\ t) \wedge \forall t\ (t \in J \rightarrow A\ at\ t) \ \Leftrightarrow$
$(A\ th\ I \wedge A\ th\ J)$

The proof is analogous for the in annotation. □

Proof. $[\wedge inth]$

$(A \wedge B)\ in\ I \ \Leftrightarrow$
$\exists t\ (t \in I \wedge (A \wedge B)\ at\ t) \ \Leftrightarrow$
$\exists t\ (t \in I \wedge A\ at\ t \wedge B\ at\ t) \ \Leftrightarrow$
$\exists t\ (in\ I \sqsubseteq at\ t\ \wedge A\ at\ t \wedge B\ at\ t) \ \Leftrightarrow$
$\exists J\ (in\ I \sqsubseteq th\ J\ \wedge A\ th\ J \wedge B\ th\ J)$ □