Annotated Constraint Logic Programming Applied to Temporal Reasoning

Thom Frühwirth

ECRC-94-22

technical report ECRC-94-22

Annotated Constraint Logic Programming Applied to Temporal Reasoning

Thom Frühwirth

European Computer-Industry Research Centre GmbH (Forschungszentrum)
Arabellastrasse 17
D-81925 Munich
Germany
Tel. +49 89 9 26 99-0
Fax. +49 89 9 26 99-170
Tlx. 52 69 10

For more information please contact : Thom Frühwirth (thom@ecrc.de)

Abstract

Annotated constraint logic programming (ACLP) combines constraint logic programming (CLP) and generalized annotated programming (GAP). With ACL we propose a first order logic with constraints where formulas can be annotated. ACL comes with inference rules for annotated formulas and a constraint theory for handling annotations. We describe an implementation based on the standard interpreter for logic programs. The inference rules of ACL are turned into clauses of the interpreter, and the constraints on annotations are solved by a suitable constraint solver. Then we optimize the interpreter.

We also introduce an instance of ACLP for reasoning about time. Temporal ACLP is conceptually simple while covering substantial parts of temporal logic. Temporal annotations avoid the proliferation of variables and quantifiers of standard first-order approaches. In TACLP, the model of time can be freely chosen since it is represented in the constraint theory. Both qualitative and quantitative (metric) temporal reasoning with time points (instants) and periods (temporal intervals) are supported. TACLP is implemented as an instance of the generic interpreter. An example, the "Workshop Murder Mystery", forms a guideline through the paper.

This paper was presented at the Sixth International Conference on Programming Language Implementation and Logic Programming (PLILP'94), Madrid (Spain), September 14-16, 1994, Proceedings to appear in Springer LNCS.

Part of this work was supported by ESPRIT Project 5291 CHIC.

1 Introduction

As a starting point for our investigation, we may take the following citation from [KiSu92]: "We see that there is a close relationship between annotated logic programming and constraint logic programming.[...] there is also a close connection between annotated programs and certain fragments of temporal logics. Thus, there is hope that [...] a single unifying framework for multivalued, temporal and constraint logic programming will emerge."

1.1 Annotated Constraint Logic

One contribution of our paper is that we integrate generalized annotated programming (GAP) and constraint logic programming (CLP) into annotated constraint logic programs (ACLP) and implement it with a generic interpreter. In GAP [KiSu92], atomic formulas can be annotated and special inference rules for the annotated formulas exist. In CLP [JaMa94, VH91], certain predicates are considered to be constraints and are solved efficiently by a special purpose constraint solver. As a basis for ACLP, we propose ACL, a first order logic with constraints where formulas can be annotated. We consider annotations as distinguished terms with a special structure. ACL extends first-order constraint logic with inference rules for annotated formulas and a constraint theory for handling annotations.

Our approach is to provide an implementation of ACLP on top of existing (CLP) languages. Another approach is to define a an SLD-style proof theory and implement it from scratch or by modifying a CLP compiler. Our approach is flexible and straightforward: We can implement a generic interpreter for ACLP in CLP, since the inference rules of ACL can be written as CLP clauses. This does not imply that annotations are just syntactic sugar for a CLP language, as our example will illustrate.

As for the other approach, in [KiSu92] an SLD-style proof theory for GAP is developed. However an implementation would be difficult since the proof theory involves a possibly infinite number of clauses called "reductants". Recently, "ca-resolution" to compute annotated logic programs was proposed in [LeLu94] and implemented in C. The idea is to compute dynamically and incrementally the reduction (that resulted in the reductants in [KiSu92]) by collecting partial answers. It turns out that operationally this is similar to the workings of our ACLP interpreter (which relies on recursion to collect the partial answers). However, in [LeLu94] the class of programs considered is smaller and the intermediate stages of a reduction are not sound with respect

to the standard CLP semantics. Thus the "independence of computation rule" of SLD-resolution does not hold.

1.2 Temporal Programming Languages

Another contribution of this paper is that in the ACLP framework we can define a rather general class of temporal logic programming languages (TACLPs) and thus formalize existing approaches. The idea is straightforward and has been explored to some extent in [KiSu92]: Atomic formulas can be labeled with temporal information. *Temporal annotations* say at what time(s) formulas are valid. For example

```
promoted(hugo) at 10/11/86

worked_at(hugo,sales) th [9/12/1988,3/2/1990]

fired(hugo) in 1992
```

are the atomic formulas

```
promoted(hugo), worked_at(hugo,sales), fired(hugo)
```

annotated with temporal information

```
``at 10/11/86'', ``th [9/12/1988,3/2/1990]''[1] , ''in 1992''.
```

Conceptually, this approach is simple: Like other approaches it accounts for the special status of time and if there are no temporal annotations, we are left with ordinary CLP. We also have shown in companion papers [Fru93, Fru94b] that TACLP languages have their formal justification as expressive fragments of temporal logics.

One of the first temporal logic programming languages was TEMPLOG, a "temporal Prolog" [AbMa89]. TEMPLOG implements a fragment of first-order temporal logic (tense logic). For example, in tense logic, the temporal operator ◯ denotes the next instant of time, and thus the TEMPLOG clause

```
◯fired(X) <= stole(X,Y)
```

reads "If `X` stole something, then he was fired immediately after".
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 almost from scratch.

With the advent of constraints in logic programming, the implementation of temporal logic by using temporal constraint languages became possible. The

[1] `th` abbreviates "throughout"

idea is to translate the temporal logic into a first order logic by introducing temporal parameters and special relations and functions describing the structure of time. For example, the above TEMPLOG clause is translated into

```
fire(X,T+1) :- steal(X,Y,T)
```

where $+1$ denotes the successor function on time points. As argued in [FrSc91], 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 implemented as a constraint solver can be used, while for the first order logic part, standard deduction suffices.

In [Brz93], a powerful temporal logic (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 was unemployed since he was fired results in executing a statement[2]:

```
:- T=<S,current_date(S),fired(X,T),

for_all R ((T=<R,R=<S) implies unemployed(X,R))
```

The complexity is due to a general problem of the first order approach: The translation of a temporal logic into first order logic results in a proliferation of quantifiers, temporal variables and complex constraints. In ACLP languages, the introduction of annotations avoids these problems. For example, the above statement in TACLP is simply

```
:- current_date(S), fired(X) at T, unemployed(X) th [T,S]
```

The TACLP framework can cover the complete Horn clause fragment of extended tense logic [Fru93, Fru94b]. Furthermore, TACLP can be implemented as an instance of ACLP. The resulting constraints between annotations are simpler than those of the standard first order approach. However, we have not analyzed yet how well nested annotations can be implemented. In Brozskas programming language, temporal operators can be nested, but “eventuality” in the heads of clauses is dissallowed.

In TACLP and the above-mentioned languages predicates are time-dependent. For completeness we mention another line of work in temporal programming languages with a rather different flavor. In languages like TEMPURA [Mos86] and METATEM [FiOw92] variables are time-dependent.

[2] The actual syntax in [Brz93] is somewhat different

1.3 The Workshop Murder Mystery

We illustrate the power of our approach by means of an example. It involves reasoning about qualitative and quantitative (metric), complete and partial temporal information involving time periods, their duration, and time points.

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 partial knowledge.
There are two suspects, Dr. Kosta and Dr. Maringer. They have alibis.
(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 partial knowledge.
(7) During the 2nd talk Dr. Maringer realised 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 ?*

Structure of the Paper. In the next chapter, we quickly introduce annotated constraint logic (ACL) and derive a generic optimized interpreter for fragments of the logic. In the chapter after, we define a temporal logic programming language as an instance of the ACLP scheme and give its implementation so that the workshop murder mystery can be solved. We end with conclusions.

2 Annotated Constraint Languages

We define a first order logic with constraints and annotations. Then we introduce annotated constraint logic programs and an optimized generic interpreter for such programs.

2.1 Logics with Constraints and Annotations

We start from standard first order logic (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.

Definition. A *first order constraint logic (FOCL)* is a first order logic with a distinguished class of predicates called *relational constraints* and a distinguished class of interpreted functions called *functional constraints*. A *constraint term* is a term involving only functional constraints. A *constraint theory* is the set of all sentences involving only relational and functional constraints (and no program-defined predicates). Equality ($=$) as well as $true$ and $false$ are relational constraints.

Next we add annotations to the constraint logic. Our definitions remove most of the restrictions on constraints and annotations in generalized annotated programs of [KiSu92] and in annotated logic programs of [LeLu94].

Definition. An *(first order) annotated constraint logic (ACL)* is a first order constraint logic where formulas can be annotated with a distinguished class of constraint terms called *annotation terms*. We write the annotation (term) immediately after the formula it annotates. The class of all annotations forms an upper semilattice (every nonempty finite subset has a least upper bound). The partial ordering $\sqsubseteq$ (a transitive, reflexive and antisymmetric relation) is a relational constraint. The least upper bound operator $\sqcup$ is a functional constraint (which is idempotent, associative and commutative).

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

Constraint Theory. The lattice operations $\sqsubseteq$ and $\sqcup$ can be axiomatized by the following constraint theory CT($\sqsubseteq, \sqcup$) (all variables are assumed to be all-quantified at the outermost scope):

($\sqsubseteq$Reflexivity) $\alpha \sqsubseteq \alpha$
($\sqsubseteq$Anti-Symmetry) $\alpha \sqsubseteq \beta \wedge \beta \sqsubseteq \alpha \rightarrow \alpha = \beta$
($\sqsubseteq$Transitivity) $\alpha \sqsubseteq \beta \wedge \beta \sqsubseteq \gamma \rightarrow \alpha \sqsubseteq \gamma$
where α, β, γ are annotations.

Next we define the least upper bound $\sqcup$ in the obvious way:
($\sqcup$LUB) $\alpha \sqsubseteq (\alpha \sqcup \beta) \wedge \beta \sqsubseteq (\alpha \sqcup \beta) \ \wedge \ \forall\gamma(\alpha \sqsubseteq \gamma \wedge \beta \sqsubseteq \gamma \rightarrow (\alpha \sqcup \beta) \sqsubseteq \gamma)$

The definition of $\sqcup$ is not really constructive. It helps to keep the following theorems in mind:
($\sqcup$Idempotency) $\alpha \sqcup \alpha \ = \ \alpha$
($\sqcup$Commutativity) $\alpha \sqcup \beta \ = \ \beta \sqcup \alpha$
($\sqcup$Associativity) $\alpha \sqcup (\beta \sqcup \gamma) \ = \ (\alpha \sqcup \beta) \sqcup \gamma$

Inference Rules. In addition to modus ponens

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

we add two *finitary* inference rules AX($\sqsubseteq, \sqcup$) to our constraint logic which utilize the lattice structure of the annotations:

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

where A, B are formulas and α, β are annotations.

The ($\sqsubseteq$Annotation) 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 ($\sqcup$Annotation) 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 represents all the annotations for which a formula holds. Problems may arise if we take the closure of an infinite number of annotations - this issue is discussed at length in [KiSu92].

The axioms that define the interplay of the logical connectives and the annotations come with the specific instance of our framework. For example, we may have an axiom for distributivity of annotations over conjunction: $(A \wedge B)\ \alpha \Leftrightarrow A\ \alpha \wedge B\ \alpha$. Depending on these axioms it may not be possible to transform certain annotated formulas into the normal form of ACLP clauses.

Definition. An *ACLP clause* is of the form:

$$A\ \alpha \leftarrow C_1 \wedge \ldots \wedge C_n \wedge B_1\ \alpha_1 \wedge \ldots \wedge B_m\ \alpha_m\ (n, m \geq 0)$$

where A is the head formula, the B_i's are the body formulas, the C_j's are the relational constraints, and α, α_k's are *optional* annotations. An *ACLP program* is a finite set of ACLP clauses. In instances of ACLP formulas are often restricted to atomic formulas.

2.2 Implementing Annotated Constraint Logic Programs

In this chapter we define an executable clause fragment of first order annotated constraint logic. Our generic interpreter implements the annotation inference rules for the fragment in any CLP language that provides a suitable constraint solver for annotations.

Logic programming languages are well suited for writing interpreters as they can treat programs as data [BoKo82, StSh94]. The object program is reified, i.e. the predicates are represented by functions in the interpreter. $solve(A)$ is a unary predicate that is true if and only if A is true at the object level. ACLP clauses of the object program, say $A \leftarrow B$, are represented at the meta level as $clause(A, B)$ where $clause$ is a binary predicate.

The clauses of the standard interpreter (which is part of the Prolog folklore) handle relational constraints, negation as failure, conjunction and disjunction:

$solve(A) \leftarrow constraint(A) \wedge A.$

$solve(\neg A) \leftarrow \neg solve(A).$

$solve(A \wedge B) \leftarrow solve(A) \wedge solve(B).$

$solve(A \vee B) \leftarrow solve(A) \vee solve(B).$

The most important clause of the standard interpreter for logic programs performs an SLD resolution step and thus implements modus ponens:

$solve(A) \leftarrow clause(A, B) \wedge solve(B).$

The annotation inference rules AX($\sqsubseteq, \sqcup$) can be put into clause form easily:

$solve(A\ \alpha) \leftarrow \alpha \sqsubseteq \beta \wedge solve(A\ \beta).$

$solve(A\ \gamma) \leftarrow \gamma = (\alpha \sqcup \beta) \wedge solve(A\ \alpha) \wedge solve(A\ \beta).$

We now improve the termination behavior of the generic interpreter for ACLP and optimize it. The details and proofs for the following equivalence transformations can be found in the full version of this paper.

Unfolding $solve(A\ \beta)$ in the first clause for annotations using the second clause results in:

$solve(A\ \gamma) \leftarrow \gamma \sqsubseteq (\alpha \sqcup \beta) \wedge solve(A\ \alpha) \wedge solve(A\ \beta).$

It is easy to see that the above clause subsumes the two original clauses implementing AX($\sqsubseteq, \sqcup$) by taking either $\alpha = \beta$ or $\gamma = (\alpha \sqcup \beta)$. Note that if $\alpha = \beta$ we compute $solve(A\ \alpha)$ twice. To improve efficiency, we introduce an exclusive disjunction:

$solve(A\ \gamma) \leftarrow \gamma \sqsubseteq (\alpha \sqcup \beta) \wedge solve(A\ \alpha) \wedge (\alpha = \beta\ \vee\ \neg(\alpha = \beta) \wedge solve(A\ \beta))$.

A negated constraint like $\neg(\alpha = \beta)$ does not necessarily mean that we have to provide negation in the constraint solver. One may simply delay the negated constraint until it can be decided if it holds or not. As we will see, in specific instances of ACLP, the negated constraints can also be replaced by positive ones.

To enforce as much as possible that both α and β contribute to an upper bound which covers γ we add constraints in the second disjunct:

$solve(A\ \gamma) \leftarrow \gamma \sqsubseteq (\alpha \sqcup \beta) \wedge solve(A\ \alpha) \wedge (\alpha = \beta\ \vee$

$\neg(\gamma \sqsubseteq \beta) \wedge \neg(\gamma \sqsubseteq \alpha) \wedge \neg(\alpha = \beta) \wedge solve(A\ \beta))$.

Finally we unfold $solve(A\ \alpha)$ with the clause implementing modus ponens which results in:

$solve(A\ \gamma) \leftarrow \gamma \sqsubseteq (\alpha \sqcup \beta) \wedge clause(A\ \alpha, B) \wedge solve(B) \wedge (\alpha = \beta\ \vee$

$\neg(\gamma \sqsubseteq \beta) \wedge \neg(\gamma \sqsubseteq \alpha) \wedge \neg(\alpha = \beta) \wedge solve(A\ \beta))$.

As a consequence, we can restrict modus ponens to non-annotated formulas:

$solve(A) \leftarrow non\_annotated(A) \wedge clause(A, B) \wedge solve(B)$.

The above clauses can be specialized for the specific instance of ACL, and the additional axioms of the instance have to be implemented in a similar fashion. We will illustrate this with temporal ACLP.

3 Temporal Logic Programming

Temporal annotated constraint logic (TACLP) supports both time points (instants) and time periods (temporal intervals). Here we will start from time points and then relate periods to convex sets of points. We could also define points in terms of periods to show the duality between points and periods. For all practical purposes it does not seem to matter what notion is the basic one. For background in temporal logic and temporal programming we refer the reader to [Ben83, Gal87].

3.1 Temporal Annotations and Constraints

We may think of a time point as denoting an indivisible, duration-less instant or moment of time. We will introduce three kinds of annotations based on sets of time-points. Different from the first order approach, we will use set annotations 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 is not new, e.g. [McD82] uses a similar construction.

The annotated formula $A\ at\ t$ means that the formula A is true at time point t. We relate a formula to many time points by introducing two new temporal annotations, $th\ I$ and $in\ I$, where I is a non-empty set of time points.

If a formula A holds *throughout* a set, i.e. at *every* time point in I, we write $A\ th\ I$. The definition is:

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

If a formula A holds at *some* time point(s) (but we don't know exactly when) in I we write $A\ in\ I$. This accounts for partial or imprecise information. The first order definition of the annotated formula is:

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

Constraint Theory. Using the first order definitions directly would introduce quantified temporal variables and implication between constraints as in the programming language in [Brz93]. Instead we derive the specific instance of the constraint theory CT($\sqsubseteq, \sqcup$) for temporal annotations from the above definitions:

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

Note that at annotations are incomparable. The $(th \sqsubseteq)$ axiom means that if a formula holds for every element in a set, it also holds for every element in all subsets. The $(in \sqsubseteq)$ axiom means that if a formula holds for some elements in a set, it also holds for some elements in all supersets. The $(th\sqcup)$ axiom means that if a formula holds for every element of two sets, it holds for every element of the union of the two sets.

Note that the corresponding axiom $(in\sqcup)$ is missing. The definition $in\ I \sqcup in\ J\ =\ in\ (I \cap J)$ that seems dual to $(th\sqcup)$ does not respect axiom ($\sqcup$Annotation) and the definition of in at the same time. For example, $A\ in\ \{1,2\} \wedge A\ in\ \{2,3\} \rightarrow A\ at\ 2$ does not hold even though $in\ \{1,2\}\ \sqcup\ in\ \{2,3\}\ =\ in\ \{1,2\}\ \cap\ \{2,3\}\ =\ in\ \{2\}\ =\ at\ 2$. One may think of A meaning "Prof. Lepov gave a talk" and of 1 $(2,3)$ meaning Monday (Tuesday,Wednesday). Since Prof. Lepov may have given two talks, one on Monday and one on Wednesday, it does not follow that he gave a talk on Tuesday.

This means that the annotations are not closed under $\sqcup$, because for some annotations of the form $in\ I \sqcup in\ J$ and $th\ I \sqcup in\ J$ the result cannot be represented by an annotation. To overcome this problem, we take the closure and allow for annotations that include expressions with $\sqcup$. We add the following two obvious normalization rules to the constraint theory:

$in\ I \sqcup\ in\ J\ =\ in\ J$ if $in\ I \sqsubseteq\ in\ J$

$in\ I \sqcup\ th\ J\ =\ th\ J$ if $in\ I \sqsubseteq\ th\ J$

Time Periods. We can model time periods as convex sets of time points. For that purpose, the time points have to be a partially ordered by a relation $\leq$. The relation has to be included into the constraint theory. We write the interval $[r,s]$ for the convex set $\{t \mid r \leq t \leq s\}$. We can also choose intervals which are open or closed on either side. The definition chosen here allows for the interval $[t,t]$ which is a duration-less time period that contains a single time point.

The advantage of intervals is that they provide a finite representation of infinite convex sets of time-points and that set relations and operations over intervals 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. This problem is avoided by taking sets of intervals instead of intervals. By slight abuse of notation, we let $\{[r_1,s_1],\ldots,[r_n,s_n]\} = [r_1,s_1] \cup \ldots \cup [r_n,s_n]$.

Related Work. Temporal ACLP can describe the theory about action and time as proposed in [Gal90], which is a critical examination of [All84]. In these works, time is dense (continuous) and linear. Time periods corresponding to single time-points are excluded. 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 I' is the convex set of time points denoted by the interval I. Some additional axioms and temporal constraints have to be added, so that all 13 Allen interval relations are available and not just those that can be derived from $\sqsubseteq$.

Workshop Murder Mystery. We can now express the murder mystery as TACLP program. 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". Inequality constraints over finite domain variables are identified by adding the "#" character in front, i.e. #<. The program should be self-explanatory. The idea is that we try to find a person that is involved in the case and does not have an alibi during the time Prof. Lepov was murdered.

```
% The Workshop Program
%...
coffee_break th [3:00,3:25]. % (1)

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)

murdered(X) in [T-(1:30),T-(1:00)] :- % (3)
        found_dead(X) at T.

% Dr. Kosta's Alibi

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

on_shuttle(X) th [T1,T2] :- % (6)
        T2 #= T1+50, T2 #< T3, T3 #< T2+50,
        shuttle at T1, board_plane(X) at T3.

   shuttle at 0:00. % (5)
   shuttle at T+30 :-
        0#<=T,T#<11:00, shuttle at T.
```

```
% Dr. Maringer's Alibi

copying('Maringer') th I :- % (7)
        I=[T1,T2], T2 #= T1+5+5+15+5,
        talk(2,_,_,_) th I.

% Whodunnit ?

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

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

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

3.2 Implementing Temporal ACLP

We are now ready to implement temporal ACLP and solve our murder mystery. For our implementation, we will use only a subset of temporal ACLP (TACLP) as introduced in the previous section. For simplicity of the presentation, we will only annotate atomic formulas and simplify the annotations. We present the resulting constraint theory which can be mapped onto existing constraint solvers. We then implement TACLP as an instance of the generic interpreter.

Annotations. The simplification is that we only allow convex sets in annotations. For th annotations, this is not a loss of generality, but means to introduce least upper bounds, e.g. $th`\{[1,2],[4,7]\}$ is represented by $th\ [1,2] \sqcup th\ [4,7]$. However, for in annotations, convexity amounts to a restriction. In general, we can only approximate an in annotation by a convex set. For example, $in\ \{[1,2],[4,7]\}$ is approximated by $in\ [1,7]$.

We can represent an annotated formula $A\ th\ I_1 \sqcup \ldots \sqcup in\ I_n$ by an equivalent conjunction $A\ th\ I_1 \wedge \ldots \wedge A\ in\ I_n$. When this transformation is applied to clauses, the head formula may contain a conjunction, i.e. $(A\ th\ I_1 \wedge \ldots \wedge A\ in\ I_n) \leftarrow B$. Such formulas can be rewritten into a set of clauses $A\ th\ I_1 \leftarrow B, \ldots, A\ in\ I_n \leftarrow B$. The disadvantage of this approach is that some unnecessary choices (between the rewritten clauses) and repeated work (through the conjunctions) may be introduced. However, in many applications, the number of conjuncts is rather small. The advantage is that the

constraint theory is simpler, in particular, the inference rule ($\sqcup$Annotation) (performing the closure) needs only be applied to th annotations.

Constraint Theory. We could now map the temporal constraint theory onto set constraints. However, we would like to use a more common and richer constraint theory, that also allows us to reflect the structure of time. It is a common choice to map time points onto numbers so that the resulting constraints are arithmetic expressions. The advantages of this mapping are that we can express rather complex relationships between time points and that we can reason about the bounds and duration of time periods. In strict linear time, integers are a popular choice as annotations for discrete time, and real or rational numbers are usually used for continuous (dense) time. For the former, we may use a finite domain constraint solver, and for the latter a constraint solver implementing the Simplex algorithm for solving linear inequalities.

We now specialize the constraint theory to the simplified representation (where t, t_1, t_2 are time points and $I = [s_1, s_2]$, $J = [r_1, r_2]$):

$$(thth\sqcup) \quad th\ I \sqcup th\ J \quad = \quad th\ [min(s_1, r_1), max(s_2, r_2)] \quad \Leftrightarrow \quad s_1 \leq r_2 \wedge r_1 \leq s_2$$

$$\begin{array}{llcll}
(thth \sqsubseteq) & th\ I & \sqsubseteq & th\ J & \Leftrightarrow \quad r_1 \leq s_1 \wedge s_2 \leq r_2 \\
(that \sqsubseteq) & th\ I & \sqsubseteq & at\ t & \Leftrightarrow \quad t = s_1 \wedge s_2 = t \\
(thin \sqsubseteq) & th\ I & \sqsubseteq & in\ J & \Leftrightarrow \quad r_1 = r_2 \wedge r_1 = s_1 \wedge s_2 = r_2 \\
(atth \sqsubseteq) & at\ t & \sqsubseteq & th\ J & \Leftrightarrow \quad r_1 \leq t \wedge t \leq r_2 \\
(atat \sqsubseteq) & at\ t_1 & \sqsubseteq & at\ t_2 & \Leftrightarrow \quad t_1 = t_2 \\
(atin \sqsubseteq) & at\ t & \sqsubseteq & in\ J & \Leftrightarrow \quad r_1 = r_2 \wedge r_1 = t \\
(inth \sqsubseteq) & in\ I & \sqsubseteq & th\ J & \Leftrightarrow \quad r_1 \leq t \wedge t \leq r_2 \wedge s_1 \leq t \wedge t \leq s_2
\end{array}$$

$$\begin{array}{llcll}
(inat \sqsubseteq) & in\ I & \sqsubseteq & at\ t & \Leftrightarrow \quad s_1 \leq t \wedge t \leq s_2 \\
(inin \sqsubseteq) & in\ I & \sqsubseteq & in\ J & \Leftrightarrow \quad r_1 \leq s_1 \wedge s_2 \leq r_2
\end{array}$$

The Interpreter. The generic interpreter for ACLP can be specialized to temporal ACLP in a straightforward way: The standard clauses are the same as in the general case. The clause dealing with the inference rules of annotated formulas can be specialized using the constraint theory developed above. Because the upper bound is only defined for th annotations, in all other cases we can take the clause

$solve(A\ \alpha) \leftarrow \alpha \sqsubseteq \beta \wedge clause(A\ \beta, B) \wedge solve(B)$
and generate a specific version for each $\sqsubseteq$ definition in the constraint theory. For example, the instance of the above clause for $(inin \sqsubseteq)$ is

$$solve(A\ in\ [r_1, r_2]) \leftarrow r_1 \leq s_1 \wedge s_2 \leq r_2 \wedge clause(A\ in[s_1, s_2], B) \wedge solve(B)$$

Only for the $thth$ case we need to take the clause that performs the closure and replace the lattice operations accordingly to arrive at:

$$solve(A\ th\ [t_1, t_2]) \leftarrow min(s_1, r_1) \leq t_1 \wedge t_2 \leq max(r_1, r_2) \wedge s_1 \leq r_2 \wedge r_1 \leq s_2 \wedge$$

$clause(A\ th\ [s_1, s_2], B) \wedge solve(B) \wedge ((s_1 = r_1 \wedge s_2 = r_2) \vee$

$(\neg(r_1 \leq t_1 \wedge t_2 \leq r_2) \wedge \neg(s_1 \leq t_1 \wedge t_2 \leq s_2) \wedge \neg(s_1 = r_1 \wedge s_2 = r_2) \wedge$

$solve(A\ th\ [r_1, r_2])))$

Without loss of generality, we can require that $s_1 \leq r_1 \leq s_2 \leq r_2$ and get the negation-free clause:

$solve(A\ th\ [t_1, t_2]) \leftarrow s_1 \leq t_1 \wedge t_2 \leq r_2 \wedge s_1 \leq r_1 \wedge r_1 \leq s_2 \wedge s_2 \leq r_2 \wedge$

$clause(A\ th\ [s_1, s_2], B) \wedge solve(B) \wedge ((s_1 = r_1 \wedge s_2 = r_2) \vee$

$(r_1 > t_1 \wedge t_2 > s_2 \wedge solve(A\ th\ [r_1, r_2])))$

The actual interpreter for TACLP as implemented in the constraint logic programming platform ECLiPSe [MaSch94] is given below. The clauses solving for the same kind of annotation have been merged using disjunctions and the order of the constraints has been optimized:

```
solve(A):- constraint(A),call(A).
solve(not A):- not solve(A).
solve((A,B)):- solve(A),solve(B).
solve((A;B)):- solve(A);solve(B).
solve(A):- non_annotated(A),clause(A,B),solve(B).

solve(A in [T1,T2]):-
        T1#<=T2,
        (clause(A in [T3,T4],B),T3#<=T4 % (inin)
        ;clause(A at T3,B),T3=T4 % (inat)
        ;clause(A th [T5,T6],B),T5#<=T3,T3=T4,T4#<=T6),   % (inth)
        T1#<=T3,T4#<=T2,
        solve(B).
solve(A at T):-
        (clause(A at T1,B),T1#=T % (atat)
        ;clause(A in [T1,T2],B),T1#=T,T#=T2 % (atin)
        ;clause(A th [T1,T2],B),T1#<=T,T#<=T2), % (atth)
        solve(B).
solve(A th [T1,T2]):-
        T1#<=T2,
        (clause(A th [T3,T4],B),T3#<=T1,T1#<=T4 % (thth)
        ;clause(A at T4,B),T4#=T1 % (that)
        ;clause(A in [T3,T4],B),T4#=T1,T3#=T4), % (thin)
        solve(B),
        (T2#<=T4 ; solve(A th [T4+1,T2])). % closure
```

The Murder Mystery Solved. Asking `:- solve(murder(X,Y))` returns two solutions `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.

4 Conclusions

We integrated GAP and constraint logic programming (CLP) into annotated constraint logic programs (ACLP). With ACL we proposed a first order logic with constraints where formulas can be annotated. We implemented a generic optimized interpreter for ACLP in CLP. We illustrated the unifying power of the ACL paradigm with an instance of ACLP for reasoning about time. Temporal ACLP is conceptually simple while covering substantial parts of temporal logic. Annotations avoid the proliferation of variables and quantifiers of the standard first order logic approach.

Our work at this stage can only be the starting point. For example, we want to provide nested annotations. We are aiming at a generic compiler for ACL. The simplicity of the interpreter indicates that compilation is simple as well, and indeed we already have achieved some promising preliminary results.

The mapping of temporal logics into annotated constraint logic and its limitations have to be further investigated. The relationship to other theories about time such as event calculus and situation calculus has to be examined. We are also considering extending the available temporal constraints along the lines of [DMP91, Mei91, Fru94a], but have to consider carefully the interaction with annotations.

Acknowledgements. Thanks to Joachim Schimpf for suggesting the use of a murder mystery example to illustrate the power of TACLP and for his comments. Thanks to James Lu for discussing the paper and its topics in detail. Thanks to anonymous referees for their helpful comments.

Bibliography

[AbMa89] M. Abadi and Z. Manna, Temporal Logic Programming, Journal of Symbolic Computation (1989) 8, pp 277-295.

[All84] J. F. Allen, Towards a General Theory of Action and Time, Artificial Intelligence, Vol. 23, 1984, pp 123-154.

[Ben83] J.F.A.K. van Benthem, The Logic of Time, Synthese Library, Vol. 156, D. Reidel Pub., Holland, 1983.

[BoKo82] K. A. Bowen and R. A. Kowalski, Amalgamating Language and Metalanguage in Logic Programming, In Logic Programming, K.A. Clark and S.A. Tarnlund (eds), pp. 153-173, Academic Press, 1982.

[Brz93] Ch. Brzoska, Temporal Logic Programming with Bounded Universal Goals, 10th ICLP, Budapest, Hungary, MIT Press, 1993.

[DMP91] R. Dechter, I. Meiri and J. Pearl, Temporal Constraint Networks, Artificial Intelligence, Vol. 49, pp. 61-95, 1991.

[FiOw92] M. Fisher and R. Owens, From the Past to the Future: Executing Temporal Logic Programs, LPAR 92, St. Petersburg, Russia, Springer LNCS 624, July 1992, pp 369-380.

[FrSc91] A. M. Frisch and R. B. Scherl, A General framework for Modal Deduction, 2nd KR '91, pp. 196-207, Cambridge, Mass., Morgan Kaufmann, 1991.

[Fru93] T. Frühwirth, Temporal Annotated Constraint Logic Programming, Workshop on Executable Modal and Temporal Logics at IJCAI 93, Chambery, France, August 1993.

[Fru94a] T. Frühwirth, Temporal Reasoning with Constraint Handling Rules, Technical Report ECRC-94-05, ECRC Munich, Germany, January 1994.

[Fru94b] T. Frühwirth, Annotating Formulas with Temporal Information, Workshop on Logic and Change at ECAI 94, Amsterdam, The Netherlands, August 1994.

[Gal87] A. Galton (ed), Temporal Logics and Their Applications, Academic Press, 1987.

[Gal90] A. Galton, A Critical Examination of Allen's Theory of Action and Time, Artificial Intelligence, Vol. 42, 1990, pp. 159-188.

[JaMa94] J. Jaffar and M. J. Maher, Constraint Logic Programming: A Survey, Journal of Logic Programming, to appear.

[KiSu92] M. Kifer and V.S. Subrahmanian, Theory of Generalized Annotated Logic Programming and its Applications, Journal of Logic Programming, April 1992.

[LeLu94] S. M. Leach and J. J. Lu, Computing Annotated Logic Programs: Theory and Implementation, 11th ICLP, Santa Margherita Ligure, Italy, MIT Press, 1994.

[MaSch94] M. Meier, J. Schimpf et al., ECLiPSe 3.4 User Manual and Extensions User Manual, ECRC Munich, Germany, January 1994.

[McD82] D. McDermot, A Temporal Logic for Reasoning about Processes and Plans, Cognitive Science 6:101-155, 1982.

[Mei91] I. Meiri, Combining Qualitative and Quantitative Constraints in Temporal Reasoning, AAAI 91, July 1991, pp 260-267.

[Mos86] B. Moszkowski, Executing Temporal Logic Programs, Cambridge University Press, 1986.

[StSh94] L. Sterling and E. Shapiro, “Interpreters”, Chapter 17, The Art of Prolog, Second Edition, MIT Press, 1994.

[VH91] P. van Hentenryck, Constraint Logic Programming, The Knowledge Engineering Review, Vol 6:3, 1991, pp 151-194.