Two Semantics for Temporal Annotated Constraint Logic Programming
Extended Abstract

Alessandra Raffaetà
Dipartimento di Informatica, Università di Pisa
Corso Italia, 40, I-56125 Pisa, Italy
raffaeta@di.unipi.it

Thom Frühwirth
Institut für Informatik, Ludwig-Maximilians-Universität (LMU)
Oettingenstrasse 67, D-80538 Munich, Germany
fruehwir@informatik.uni-muenchen.de
http://www.pst.informatik.uni-muenchen.de/~fruehwir/

Abstract
We investigate semantics of a considerable subset of Temporal Annotated Constraint Logic Programming (TACL), a class of languages that allows us to reason about qualitative and quantitative, definite and indefinite temporal information using time points and time periods as labels for atoms. After illustrating the power of TACL with some examples, TACL is given two different kinds of semantics, an operational one based on meta-logic (top-down semantics) and a fixpoint one based on an immediate consequence operator (bottom-up semantics).

1 Introduction
Temporal reasoning is at the heart of human activity and not surprisingly it has raised a lot of interest in computer science, be it in the form of temporal logics [35, 19, 15], temporal programming languages [28, 30, 16, 1, 6, 9, 13] or temporal databases [32, 34, 3, 5, 7, 18, 29]. No matter if one programs with temporal information or stores data with temporal information, in most cases the formal underpinnings will be logic, and often be variants or extensions of first order logic.

In a logical formulation and formalization of temporal information and reasoning it is quite natural to think of formulae that are labelled with temporal information and about proof procedures that take into account these labels. In other words, we separate the temporal from the non-temporal aspects. What D. Gabbay enthusiastically says in the introduction of his book on labelled deductive systems [17], that allow to label a general class of logics, “This sounds very simple but it ... makes a serious difference, like the difference between using one hand only or allowing for the coordinated use of two hands”, can apply to the special case of temporal labels as well.

In our case, the logic and the labels are familiar structures: First-order logic (FOL) and lattices. The labels are called annotations, and the overall class of logics is called annotated
logics [23]. Based on this framework and on constraint logic programming concepts [21, 22, 26, 27, 14], the family of temporal annotated constraint logic programming (TACLPL) languages has been developed in [12, 11, 13, 31, 25].

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. The annotations of TACLPL make time explicit but avoid the proliferation of temporal variables and quantifiers of the first order approach. In this way, TACLPL supports qualitative and quantitative (metric) temporal reasoning involving both time points and time periods (time intervals) and their duration. Moreover, it allows us to represent definite, indefinite and periodic temporal information.

In [13] TACLPL is presented as an instance of annotated constraint logic (ACL) for reasoning about time. ACL is a generalization of generalized annotated programs [23, 24], and extends first-order languages with a distinguished class of predicates, called constraints, and a distinguished class of terms, called annotations, used to label formulae. Moreover ACL provides inference rules for annotated formulae and a constraint theory for handling annotations. One advantage of a language in the ACL framework is that its clausal fragment can be efficiently implemented: Given a logic in this framework, there is a systematic way to make a clausal fragment executable as a constraint logic program. Both an interpreter and a compiler can be generated and implemented in standard constraint logic programming languages.

Constraint logic programming (CLP) [21, 22, 26, 27, 14] is an extension of logic programming, where in addition to predicates, which are defined by clauses and reasoned about by resolution (a form of Modus Ponens), there is a distinguished class of predicates called constraints. Their meaning is defined by a constraint theory whose reasoning capability is implemented by some efficient algorithm in the so-called constraint solver. In this way, efficient special-purpose algorithms can be integrated in a sound way into logic programming.

In Figure 1, an example from [12] illustrates the expressiveness and conceptual simplicity of TACLPL. As shown in [12] by asking the query murder(x, y) we obtain two solutions (i.e., two suspects) of the workshop murder mystery: 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. The second answer means that Dr. Maringer could be the murderer, because his alibi does not hold: The copying would have taken 30 minutes, so it cannot have happened during a talk of 25 minutes.

Overview of the paper. In this paper, the TACLPL language is given two different kinds of semantics, an operational one based on meta-logic (top-down semantics) using a meta-interpreter and a fixpoint one obtained by extending the definition of the immediate consequence operator of CLP to deal with annotated atoms (bottom-up semantics). The full paper of this extended abstract contains soundness and completeness proofs relating the two semantics presented here, it has been submitted elsewhere.

The paper is organized as follows. Section 2 introduces the TACLPL framework. Section 3 defines the two semantics for TACLPL. Section 4 presents related work and Section 5 concludes the paper.
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.

- coffeebreak 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].

(2) Prof. Lepov was found dead in his hotel room at 5:35pm - time point.

- founddead(Lepov) th [5:35].

(3) The doctor said he was dead for one to one and a half hours - duration and indefinite information.

- murdered(p) in [t₁ - 1:30, t₂ - 1:00] ← founddead(p) in [t₁, t₂]

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.

- boardplane(Kosta) th [5:10].

(5) The shuttle from the hotel leaves every half hour between noon and 11pm - recurrent (periodic) data.

- shuttle at 0:00.
- shuttle at t + 30 ← 0:00 ≤ t, t ≤ 11:00, shuttle at t

(6) It takes at least 50 minutes to get to the airport - duration and indefinite information.

- onshuttle(p) th [t₁, t₂] ← t₂ = t₁ + 50, shuttle at t₁,
- boardplane(p) in [t₂, t₂ + 50]

(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.

- copying(Maringer) th [t₁, t₂] ← t₂ = t₁ + 5 + 5 + 30 * 0.5 + 5,
- talk(2, p, w) th [t₁, t₂]

(8) Who murdered Prof. Lepov? The murderer is a person who is involved in the case and does not have an alibi during the time of murder.

- murder(p₁, p₂) ← murdered(p₂) in [t₁, t₂], involved(p₁), ¬(alibi(p₁) th [t₁, t₂])
- involved(Kosta), involved(Lepov), involved(Maringer).
- alibi(p) th [t₁, t₂] ← (onshuttle(p) ∨ copying(p) ∨ talk(n, p, w)) th [t₁, t₂]

Figure 1: The Workshop Murder Mystery
2 Temporal Annotated Constraint Logic Programming

This subsection briefly reviews TACL. In this paper, we consider the subset of TACL, where time points are totally ordered, sets of time points are convex and non-empty, and only atomic formulae can be annotated. Moreover clauses are free of negation. These restrictions will become clear during this section. For a more detailed treatment of TACL and for the general theory of ACL we refer the reader to [13].

An annotated formula is of the form $A \alpha$ where $A$ is a first order formula and $\alpha$ an annotation. In TACL, there are three kinds of annotations based on (sets of) time points. Let $t$ be a time point and let $I$ be a set of time points.

(at) The annotated formula $A \mathit{at} t$ means that $A$ holds at time point $t$.

(th) The annotated formula $A \mathit{th} I$ means that $A$ holds throughout, i.e., at every time point in the set $I$. The definition of a th-annotated formula in terms of at is:

$$A \mathit{th} I \iff \forall t \in I \rightarrow A \mathit{at} t.$$ 

(in) The annotated formula $A \mathit{in} I$ means that $A$ holds at some time point(s) - but we do not know exactly when - in the set $I$. The definition of an in-annotated formula in terms of at is:

$$A \mathit{in} I \iff \exists t \in I \land A \mathit{at} t.$$ 

The in temporal annotation accounts for indefinite temporal information.

The set of annotations is endowed with a partial order relation $\subseteq$ which turns it into a lattice. Given two annotations $\alpha$ and $\beta$, the intuition is that $\alpha \subseteq \beta$ if $\alpha$ is "less informative" than $\beta$ in the sense that for all formulae $A$, $A \beta \rightarrow A \alpha$.

More precisely, being an instance of ACL, in addition to Modus Ponens, TACL has two further inference rules: The rule $(\subseteq)$ and the rule $(\cup)$. The rule $(\subseteq)$ states that if a formula holds with some annotation, then it also holds with all annotations that are smaller according to the lattice ordering. The rule $(\cup)$ says that if a formula holds with some annotation and the same formula holds with another annotation then it holds with the least upper bound of the annotations. These three inference rules can be merged into a single rule called $A$-resolution:

$$A \alpha \quad B \quad (A \beta \leftarrow B) \quad \gamma \subseteq (\alpha \cup \beta) \quad (A\text{-Resolution})$$

Time can be discrete or dense. Time points are totally ordered by the relation $\leq$. We call the set of time points $D$. We assume that the time-line is left-bounded by the number 0 and open to the future, with the symbol $\infty$ used to denote a time point that is later than any other. A time period is an interval $[r, s]$ with $0 \leq r \leq s \leq \infty$, $r \in D$, $s \in D$ that represents the convex, non-empty set of time points $\{t \mid r \leq t \leq s\}^1$. Thus the interval $[0, \infty)$ denotes the whole time line.

The constraint theory for temporal annotations over time points and time periods contains an axiomatization of the total order relation $\leq$ on $D$ and the following axioms defining the partial order on temporal annotations:

---

1The results in this paper naturally extend to time lines that are bounded or unbounded in other ways and to time periods that are open on one or both sides.
\[\begin{align*}
\text{(at th)} & \quad \text{at } t = \text{th}[t, t] \\
\text{(at in)} & \quad \text{at } t = \text{in}[t, t] \\
\text{(th } \subseteq \text{)} & \quad \text{th}[s_1, s_2] \subseteq \text{th}[r_1, r_2] \iff r_1 \leq s_1, s_1 \leq s_2, s_2 \leq r_2 \\
\text{(in } \subseteq \text{)} & \quad \text{in}[r_1, r_2] \subseteq \text{in}[s_1, s_2] \iff r_1 \leq s_1, s_1 \leq s_2, s_2 \leq r_2
\end{align*}\]

The first two axioms state that \(\text{th} I\) and \(\text{in} I\) are equivalent to \(\text{at } t\) when the time period \(I\) consists of a single time point \(t\). Next, if a formula holds at every element of a time period, then it holds at every element in all sub-periods of that period ((\(\text{th } \subseteq\)) axiom). On the other hand, if a formula holds at some points of a time period then it holds at some points in all periods that include this period ((\(\text{in } \subseteq\)) axiom).

To summarize the partial order relation on annotations, the axioms can be arranged in the following chain, assuming \(r_1 \leq s_1, r_1 \leq s_2, s_2 \leq r_2\):

\[\text{in}[r_1, r_2] \subseteq \text{in}[s_1, s_2] \subseteq \text{in}[s_1, s_1] = \text{at } s_1 = \text{th}[s_1, s_1] \subseteq \text{th}[s_1, s_2] \subseteq \text{th}[r_1, r_2]\]

Now we axiomatize the least upper bound \(\cup\) of temporal annotations over time points and time periods. As explained in [13], the annotations are not closed under \(\cup\). From a theoretical point of view, this problem can be overcome via a closure operation which includes in the lattice expressions with \(\cup\). In practice, it suffices to consider the least upper bound for time periods that produce a different time period. Therefore we can restrict ourselves to \(\text{th}\) annotations with overlapping time periods that do not include one another:

\[\text{(th } \cup\text{)} \quad \text{th}[s_1, s_2] \cup \text{th}[r_1, r_2] = \text{th}[s_1, r_2] \iff s_1 < r_1, r_1 \leq s_2, s_2 < r_2\]

We can now define the clausal fragment of TACL that can be used as an efficient temporal programming language. A \textit{TACL} program is a finite set of ACL clauses. A \textit{TACL clause} is a TACL formula of the form:

\[A \alpha \leftarrow C_1, \ldots, C_n, B_1 \alpha_1, \ldots, B_m \alpha_m \quad (n, m \geq 0)\]

where \(A\) is an atom (not a constraint), \(\alpha\) and \(\alpha_i\) are (optional) temporal annotations, the \(C_j\)'s are the constraints and the \(B_i\)'s are atomic formulae. Constraints \(C_j\) cannot be annotated. As in logic programming syntax, commas "\," denote conjunctions. The conclusion of the implication is called the head of the clause and the premise the body of the clause. Variables in a clause are implicitly assumed to be universally quantified at the outermost scope.

We conclude the introduction of TACL with some examples.

**Example 1** In a company, there are managers and a secretary who has to manage their meetings. A manager is busy if he is in a meeting or if he is out.

\[\begin{align*}
\text{busy}(p) \text{ th } [t_1, t_2] & \leftarrow \text{in}\text{-meeting}(p) \text{ th } [t_1, t_2] \\\n\text{busy}(p) \text{ th } [t_1, t_2] & \leftarrow \text{out}\text{-of}\text{-office}(p) \text{ th } [t_1, t_2]
\end{align*}\]

\(^2\)Especially in dense time, one may disallow singleton periods and drop the two axioms. This restriction has no effects on the results of the paper.

\(^3\)Actually the least upper bound exists but sometimes may be "too large". For instance, according to the lattice, \(\text{th}[1, 2] \cup \text{th}[4, 5] = \text{th}[1, 5]\), but according to the definition of \(\text{th}\)-annotated formulae \(A\) in terms of \(\text{at}\), in the right-hand side \(A \text{ at } 3\) holds, but it does not hold in the left-hand side of the equation. Also, \(\text{in}[1, 2] \cup \text{in}[2, 3] = \text{in}[2, 2]\), but in the right-hand side \(A \text{ at } 2\) holds, but not in the left-hand side, since e.g. \(A \text{ at } 1\) and \(A \text{ at } 3\) may hold.
Suppose the schedule for today to be the following: Smith and Jones have a meeting at 9am and at 9:30am respectively, each lasting one hour. In the afternoon Smith goes out for lunch at 2pm and comes back at 3pm:

\[ \text{in-meeting(Smith) th [9am, 10am]} \]
\[ \text{out-of-office(Smith) th [2pm, 3pm]} \]
\[ \text{in-meeting(Jones) th [9:30am, 10:30am]} \]

If the secretary wants to know whether Smith is busy between 9:30am and 10:30am she can ask for \( \text{busy(Smith) in [9:30am, 10:30am]} \). Since Smith is in a meeting from 9am till 10am, one can indeed derive that Smith is busy. This query exploits indefinite information, since if Smith is busy in one instant of the period [9:30am, 10:30am], then the secretary cannot schedule an appointment for him for that period.

On the other hand, \( \text{busy(Smith) th [9:30am, 10:30am]} \) does not hold, because Smith is not busy between 10am and 10:30am. Also \( \text{busy(Smith) in [10:30am, 1:30pm]} \) does not hold, because Smith is not busy in that time period at all.

The query \( \text{busy(Smith) th [t_1, t_2], busy(Jones) th [t_1, t_2]} \) reveals that both managers are busy throughout the time period [9:30am, 10am], because this is the largest interval that is included in the time periods where both managers are busy.

Now assume that we define

\[ \text{busy th [t_1, t_2]} \leftarrow \text{busy(p th [t_1, t_2]} \]

Then busy holds when either manager is busy, namely for the intervals [9am, 10:30am] (which is the least upper bound of the time periods for the two overlapping meetings of Smith and Jones) and [2pm, 3pm].

In [31] TACLCP is successfully applied to a system for calculating the liquid flow in a network of water tanks from some events specifying when the taps were switched on and off. The following example involving continuous change is also presented.

**Example 2** We model information about the growth of trees.

1. Tree 1 sprouts at time 3.5 (the middle of year 3).
   \[ \text{sprouts(Tree1) at 3.5} \]

2. Tree 1 is an oak tree.
   \[ \text{tree_type(Tree1, Oak)} \]

3. The growth rate of oak trees is 3 meters per year.
   \[ \text{growth_rate(Oak, 3)} \]

4. If a tree is of a type that has a given growth rate \( r \), and the tree sprouts at time \( s \) then at time \( t \) it has a height, where \( h = (t - s) \times r \).
   \[
   \begin{align*}
   \text{height(tree, h) at t} & \leftarrow \\
   h &= (t - s) \times r, \\
   \text{tree_type(tree, type), growth_rate(type, r),} \\
   \text{sprouts(tree) at s}
   \end{align*}
   \]
5. If a tree has height \( h \) at time \( t \), where \( h \geq 6.75 \), then it is mature.

\[
mature(\text{tree}) \th[t, \infty] \leftrightarrow h \geq 6.75, \text{height(\text{tree}, h)} \text{ at } t
\]

In the last clause, the maturity of the tree at an instant is implied by a constraint on the height of the tree at that instant. Height is the continuously changing quantity. The query

\[
mature(\text{Tree1}) \th[6, 7]
\]

can be proved. This means that Tree1 is mature throughout the time period which begins at year 6 and ends at year 7.

The query

\[
mature(\text{Tree1}) \th[t_1, t_2]
\]

yields \( t_1 \geq 5.75, t_2 = \infty \).

3 Semantics of TACL\textit{P}

In this section we define the operational (top-down) semantics of the language TACL\textit{P} by presenting a meta-interpreter for it. Then we provide TACL\textit{P} with a fixpoint (bottom-up) semantics, based on the definition of an immediate consequence operator.

In the definition of the semantics, without loss of generality, we assume all atoms to be annotated with \( \th \) or \( \in \) labels. \( \at \) annotations can be replaced with \( \th[t, t] \) by exploiting the \( \at(\th) \) axiom. Each atom which is not annotated in the object level program is intended to be true throughout the whole temporal domain, and thus can be labelled with \( \th[0, \infty] \). Constraints stay unchanged.

3.1 Operational Semantics via Meta-Interpreter

The \textit{vanilla} meta-interpreter [33] is the simplest application of meta-programming in logic. A general formulation of the \textit{vanilla} meta-interpreter can be given by means of the \textit{demo} predicate used to represent provability. \textit{demo}(\textit{g}) means that the formula \textit{g} is provable in the object program.

\[
\text{demo}() \text{Empty).}
\]
\[
\text{demo}((b_1, b_2)) \leftarrow \text{demo}(b_1), \text{demo}(b_2)
\]
\[
\text{demo}(a) \leftarrow \text{clause}(a, b), \text{demo}(b)
\]

The unit clause states that the empty goal, represented by the constant symbol \textit{Empty}, is always solved. The second clause deals with conjunctive goals. It states that a conjunction \((B_1, B_2)\) is solved if \(B_1\) is solved and \(B_2\) is solved. Finally, the third clause deals with the case of atomic goal reduction. To solve an atomic goal \(A\), a clause from the program is chosen whose head unifies with \(A\) and the body of the clause is recursively solved. An object level program \(P\) is represented at the meta-level by a set of axioms of the kind \textit{clause}(\(A, B\), one for each object level clause \(A \leftarrow B\) in \(P\).

The extended meta-interpreter for our subset of TACL\textit{P} is defined by the following clauses:

\[
\text{demo}() \text{Empty).}
\]
\[ \text{demo}((b_1, b_2)) \leftarrow \text{demo}(b_1), \text{demo}(b_2) \]  
\[ \text{demo}(a \text{ th } [t_1, t_2]) \leftarrow s_1 \leq t_1, t_2 \leq s_2, t_1 \leq t_2, \] 
\[ \text{clause}(a \text{ th } [s_1, s_2], b), \text{demo}(b) \]  
\[ \text{demo}(a \text{ th } [t_1, t_2]) \leftarrow s_1 \leq t_1, t_1 < s_2, s_2 < t_2, \] 
\[ \text{clause}(a \text{ th } [s_1, s_2], b), \text{demo}(b), \text{demo}(a \text{ th } [s_2, t_2]) \]  
\[ \text{demo}(a \text{ in } [t_1, t_2]) \leftarrow t_1 \leq s_2, s_1 \leq t_2, t_1 \leq t_2, \] 
\[ \text{clause}(a \text{ in } [s_1, s_2], b), \text{demo}(b) \]  
\[ \text{demo}(c) \leftarrow \text{constraint}(c), c \]  
\[ \text{clause}(A \alpha, B) \leftarrow t_1 \leq t_2. \]  

A clause \( A \alpha \leftarrow B \) of a TACL program \( P \) is represented at the meta-level by

\[ \text{clause}(A \alpha, B) \leftarrow t_1 \leq t_2. \]

where \( \alpha = \text{th } [t_1, t_2] \) or \( \alpha = \text{in } [t_1, t_2] \).

This meta-interpreter can be written in any CLP language that provides a suitable constraint solver for temporal annotations (see Section 2 for the constraint theory). Hence the first difference with the vanilla meta-interpreter is that our meta-interpreter handles constraints which can either occur explicitly in its clauses, e.g. \( s_1 \leq t_1, t_1 \leq t_2, t_2 \leq s_2 \) in clause (3), or can come from the resolution steps. The latter kind of constraints is managed by clause (7) which passes each constraint \( C \) to be solved directly to the constraint solver.

The second difference is that our meta-interpreter implements not only Modus Ponens but the more powerful A-resolution rule, which is the combination of Modus Ponens itself with rule (\( \square \)) and rule (U). This is the reason why the third demo clause of the vanilla meta-interpreter is now split into four clauses. Clauses (3), (5) and (6) implement the inference rule (\( \square \)): The atomic goal to be solved is required to be labelled with an annotation which is smaller than the one labelling the head of the clause used in the resolution step. For instance, clause (3) states that given a clause \( A \text{ th } [s_1, s_2] \leftarrow B \) whose body \( B \) is solvable, we can derive the atom \( A \) annotated with any \( \text{th } [t_1, t_2] \) such that \( \text{th } [t_1, t_2] \subseteq \text{th } [s_1, s_2] \), i.e., according to axiom (\( \square \)), \( [t_1, t_2] \subseteq [s_1, s_2] \), as expressed by the constraint \( s_1 \leq t_1, t_2 \leq s_2, t_1 \leq t_2 \). Clauses (5) and (6) are built in an analogous way by exploiting axioms (in \( \square \)) and (in \( \square \)), respectively.

Rule (U) is implemented by clause (4). According to the discussion in Section 2, it is applicable only to \( \text{th} \) annotations with overlapping time periods which do not include one another. More precisely, clause (4) states that if we can find a clause \( A \text{ th } [s_1, s_2] \leftarrow B \) such that the body \( B \) is solvable, and if moreover the atom \( A \) can be proved throughout the time period \( [s_2, t_2] \) (i.e., \( \text{demo}(A \text{ th } [s_2, t_2]) \) is solvable) then we can derive the atom \( A \) labelled with any annotation \( \text{th } [t_1, t_2] \subseteq \text{th } [s_1, s_2] \). The constraints on temporal variables ensure that the time period \( [t_1, t_2] \) is a new time period different from \( [s_1, s_2] \) and \( [s_2, t_2] \) and their subintervals.

Finally, in the meta-level representation of object clauses, clause (8), we have to add the constraint \( t_1 \leq t_2 \) to ensure that the head of the object clause has a well-formed, namely non-empty, annotation.
Example 3 Consider a library database containing information about loans. Mary first borrowed the book Hamlet from May 12, 1995 to June 12, 1995 and then on June 12, 1995 she extended her loan:

\[
\text{borrow}(\text{Mary, Hamlet}) \text{ th } [\text{May 12 1995, Jun 12 1995}].
\]

\[
\text{borrow}(\text{Mary, Hamlet}) \text{ th } [\text{Jun 12 1995, Aug 1 1995}].
\]

The period of time in which Mary borrowed Hamlet can be obtained by the query

\[
\text{demo}([\text{borrow}(\text{Mary, Hamlet}) \text{ th } [t_1, t_2]]).
\]

By using clause (4), it is possible to derive the interval [May 12 1995, Aug 1 1995] (more precisely, the constraints May 12 1995 \(\leq\) \(t_1\), \(t_1\) \(<\) Jun 12 1995, Jun 12 1995 \(<\) \(t_2\), \(t_2\) \(\leq\) Aug 1 1995 are derived) that otherwise would be never generated. In fact, by applying clause (3) alone, it is possible to prove only that Mary borrowed Hamlet in the intervals [May 12 1995, Jun 12 1995] and [Jun 12 1995, Aug 1 1995] separately.

Compilation of TACL programs into CLP . . .

In [13] a compiler for TACL has been defined by means of a compilation function \(\text{comp}\) which translates an annotated formula into its CLP form. The essential step is the inclusion of the temporal annotation of an atom in the corresponding predicate as an extra-argument.

\[
\text{comp}(p(t_1, \ldots, t_n) \alpha) = p(t_1, \ldots, t_n, \alpha).
\]

Now we can basically read off the other rules of the translation function \(\text{comp}\) directly from the meta-interpreter defined in the previous section.

A constraint \(C\) is compiled into itself, i.e., \(\text{comp}(C) = C\), and a conjunction of formulae is compiled into the conjunction of the compiled version of such formulae, i.e. \(\text{comp}(B_1, B_2) = \text{comp}(B_1) \land \text{comp}(B_2)\).

Finally, the compilation of a program clause is defined in the following way:

- for each clause of the form \(A \text{ th } [s_1, s_2] \leftarrow B\) the compiler generates three clauses
  - \(\text{comp}(A \text{ th } [t_1, t_2]) \leftarrow s_1 \leq t_1, t_2 \leq s_2, t_1 \leq t_2, \text{comp}(B)\)
  - \(\text{comp}(A \text{ th } [t_1, t_2]) \leftarrow s_1 \leq t_1, t_1 < s_2, s_2 < t_2, \text{comp}(B), \text{comp}(A \text{ th } [s_2, t_2])\)
  - \(\text{comp}(A \text{ in } [t_1, t_2]) \leftarrow t_1 \leq s_2, s_1 \leq t_2, t_1 \leq t_2, s_1 \leq s_2, \text{comp}(B)\)

- for each clause of the form \(A \text{ in } [s_1, s_2] \leftarrow B\) the compiler generates a clause
  - \(\text{comp}(A \text{ in } [t_1, t_2]) \leftarrow t_1 \leq s_1, s_1 \leq s_2, s_2 \leq t_2, \text{comp}(B)\)

The result of the compilation is a standard CLP program.

3.2 Fixpoint semantics

There are several ways of defining a bottom-up semantics of TACL, related to the different possible choices of the semantic domain where the immediate consequence operator is defined. The simpler solution consists in using the powerset \(\mathcal{P}(\mathcal{A}\text{-base}^4 \times \mathcal{Ann})\) with set-theoretic inclusion, disregarding the partial order structure of the set of annotations \(\mathcal{Ann}\).

\(^4\)The formal definition of \(\mathcal{A}\text{-base}\) is given later. Briefly, it is the natural generalization of the notion of Herbrand Base in constraint logic programming.
Alternative solutions (as for generalized annotated programs in [23]) may consider a more abstract domain, which is obtained by endowing \( \textit{A-base} \times \textit{Ann} \) with the product order (induced by the discrete order on \( \textit{A-base} \) and the order on \( \textit{Ann} \)) and then by taking as elements of power domain only those subsets of annotated atoms which satisfy some closure properties with respect to such an order. For instance, one can require “downward-closedness”, which amounts to including subsumption in the \( T_P \) operator. Another possible property is “limit-closedness”, namely the presence of the least upper bound of all directed sets which, from a computational point of view, amounts to consider computations which possibly require more than \( \omega \) steps. For space limitations, we treat here the first, simpler solution.

The intended interpretation of constraints is defined by fixing a structure \( \mathcal{A} \). In our case \( \mathcal{A} \) surely contains a structure \( \mathcal{D} \) (with domain \( D \)) in which we interpret the temporal constants and functions. However, TACL P programs can have constraints not only on temporal data, hence in general the structure \( \mathcal{A} \) will be multi-sorted.

Let \( \text{Dom}_A \) the domain of the structure \( \mathcal{A} \). An \( \mathcal{A} \)-valuation is a (multi-sorted) mapping from variables to \( \text{Dom}_A \), and its natural extension maps terms to \( \text{Dom}_A \) and formulates whose predicates have arguments ranging over \( \text{Dom}_A \). An \( \mathcal{A} \)-ground instance \( A' \) of an atom \( A \) (resp. of a constraint or of a clause) is obtained by applying an \( \mathcal{A} \)-valuation to the atom (resp. to the constraint or to the clause), thus producing a construct of the form \( p(a_1, \ldots, a_n) \) with \( a_1, \ldots, a_n \) elements from \( \text{Dom}_A \). We denote by \( \text{ground}_A(P) \) the set of \( \mathcal{A} \)-ground instances of clauses from a program \( P \).

We first define the standard fixpoint operator of constraint logic programming and then extend it to deal with TACL P. An \( \mathcal{A} \)-interpretation for a CLP(\( \mathcal{A} \)) program \( P \) is a subset of the \( \mathcal{A} \)-base of \( P \), written \( \mathcal{A} \)-base\(_P\), which is the set

\[
\left\{ p(a_1, \ldots, a_n) \mid p \text{ is a } n \text{-ary user-defined predicate in } P \text{ and each } a_i \text{ is an element of } \text{Dom}_A \right\}
\]

Then the standard immediate consequence operator[26] for a CLP(\( \mathcal{A} \)) program \( P \) is a function \( T^A_P : \mathcal{P}(\mathcal{A} \text{-base}_P) \rightarrow \mathcal{P}(\mathcal{A} \text{-base}_P) \) defined as follows:

\[
T^A_P(I) = \left\{ A \mid A \leftarrow C_1, \ldots, C_k, B_1, \ldots, B_n, \in \text{ground}_A(P), \{B_1, \ldots, B_n\} \subseteq I, \mathcal{A} \models C_1, \ldots, C_k \right\}
\]

The operator \( T^A_P \) is continuous[26], and therefore it has least fixpoint which can be computed as the least upper bound of the chain \( \{ (T^A_P)^f \}_{f \geq 0} \) of the iterated applications of \( T^A_P \) starting from the empty set. The fixpoint is denoted by \( (T^A_P)^\omega \).

To generalize the above operator to deal with temporal annotations we consider a kind of extended interpretations, basically consisting of sets of annotated elements of \( \mathcal{A} \)-base. Formally we define the set of (semantical) annotations

\[\text{Ann} = \{ \text{th}[t_1, t_2], \text{in}[t_1, t_2] \mid t_1 \in D, t_2 \in D, D \models t_1 \leq t_2 \}\]

Then given a TACL P program \( P \), the lattice of interpretations is defined as \( (\mathcal{P}(\mathcal{A}-\text{base}_P \times \text{Ann}), \subseteq) \) where \( \mathcal{P} \) is the powerset operator and \( \subseteq \) is the usual relation of set-theoretic inclusion.

\(^5\)Formally, for a function \( T : \mathcal{P}(S) \rightarrow \mathcal{P}(S) \) we define \( T^0 = \emptyset \) and \( T^{i+1} = T(T^i) \).
Definition 1 Let $P$ be a TACL program, the function $\varphi^A : \wp(\text{A-base}_P \times \text{Ann}) \rightarrow \wp(\text{A-base}_P \times \text{Ann})$ is defined as follows.

\[
\varphi^A(I) =
\begin{cases}
(A, \alpha) & \text{if } A \alpha \leftarrow C_1, \ldots, C_k, B_1 \alpha_1, \ldots, B_n \alpha_n \in \text{ground}_A(P),
\begin{cases}
\{(B_1, \beta_1), \ldots, (B_n, \beta_n)\} \subseteq I,
A \models C_1, \ldots, C_k, \alpha_1 \subseteq \beta_1, \ldots, \alpha_n \subseteq \beta_n, s_1 \leq s_2
\end{cases}
\end{cases}
\]

This definition properly extends the standard definition of the immediate consequence operator. In fact, in a sense, it captures not only the Modus Ponens rule, as the standard operator does, but also rule (\upuparrows) (second set in the above definition). In addition, rule (\updownarrows) is used to prove that an annotated atom holds in an interpretation: To derive the head $A \alpha$ of a clause it is not necessary to find in the interpretation exactly the atoms $B_1 \alpha_1, \ldots, B_n \alpha_n$ occurring in the body of the clause, but it suffices to find atoms $B_i \beta_i$ which implies $B_i \alpha_i$, i.e., such that each $\beta_i$ is an annotation stronger than $\alpha_i$ ($A \models \alpha_i \subseteq \beta_i$). Finally, notice that $\varphi^A(I)$ is not downward closed, namely, it is not true that if $(A, \alpha) \in \varphi^A(I)$ then for all $(A, \gamma)$ such that $\gamma \subseteq \alpha$, we have $(A, \gamma) \in \varphi^A(I)$. However such a closure is done at the end of the computation of the fixpoint of $\varphi^A$. In this way the $\Delta$-resolution rule which combines Modus Ponens with (\upuparrows) and (\updownarrows) rules is completely captured.

An important property of the $\varphi^A$ operator, which is at the core of the definition of the fixpoint semantics, is continuity over the lattice of interpretations.

Theorem 1 (Continuity) Let $P$ be a TACL program. The function $\varphi^A$ is continuous (on $\wp(\text{A-base}_P \times \text{Ann})$, $\subseteq$).

Proof. The proof is a direct consequence of the definition of $\varphi^A$ and of the partial order $\subseteq$ on the interpretations. For more details see the full version of the paper. \qed

The bottom-up semantics for a program $P$ is defined as the downward closure of the least fixpoint of $\varphi^A$ which by theorem 1 is the least upper bound of the chain $\{(\varphi^A)^i\}_{i \geq 0}$.

Definition 2 Let $P$ be a TACL program. Then the fixpoint semantics of $P$ is defined as

\[
\mathcal{F}^A(P) = \{(A, \alpha) \mid (A, \beta) \in (\varphi^A)^\omega, A \models \alpha \subseteq \beta\}
\]

where $(\varphi^A)^\omega = \bigcup_{i \geq 0} (\varphi^A)^i$. 

136
4 Related Work

The first temporal logic programming language was Templog [1]. In [23], Templog and an interval based temporal logic are translated into annotated logic programs. The annotations used there correspond to the th annotations of TACL. To implement the annotated logic language, the paper proposed to use “reductants”, additional clauses which are derived from existing clauses to express all possible least upper bounds. The problem was that a finite program may generate infinitely many such reductants. Then, “ca-resolution” for annotated logic programs was proposed [24]. The idea is to compute dynamically and incrementally the least upper bounds by collecting partial answers. Operationally this is similar to the meta-interpreter presented here which relies on recursion to collect the partial answers. However, in [24] the intermediate stages of the computation are not sound with respect to the standard CLP semantics.

Moreover, in [23] two fixpoint semantics are presented for generalized annotated programs (GAP). The first one, called $T_P$, is based on interpretations which associate to each element of the Herbrand Base of the program $P$ a set of annotations which is an ideal, i.e., a set downward closed and closed with respect to finite least upper bounds. The computed ideal is the least one containing the annotations $\alpha$ of annotated atoms $A \alpha$ which are heads of (instances of) clauses whose body holds in the interpretation. The other one $R_P$ is based on interpretations which associate to each atom of the Herbrand Base a single annotation which is the least upper bound of the set of annotations computed as in the previous case. Our fixpoint operator for TACL works similarly to the $T_P$ operator: at each step we close with respect to (representable) finite least upper bounds, and, although we perform the downward closure only at the end of the computation, this does not reduce the set of derivable consequences. The main difference resides in the language: TACL is an extension of CLP, taking from GAP the handling of annotations, which focuses on the temporal aspects, whereas GAP is a general language with negation and arbitrary annotations but without constraints.

Our temporal annotations correspond to some of the predicates proposed by Galton in [20], which is a critical examination of Allen's classical work on a theory of action and time [2]. Galton provides for both time points and time periods in dense linear time. Assuming that the intervals $I$ are not singletons, Galton's predicate holds-in($A,I$) can be mapped into TACL's $A \text{ in } I$, holds-on($A,I$) into $A \text{ th } I$, and holds-at($A,t$) into $A \text{ at } t$, where $A$ is an atomic formula.

From this mapping it becomes clear that TACL can be seen as reified FOL where annotated formulae, e.g. $\text{born}(\text{john}) \text{ at } t$, correspond to binary meta-relations between predicates and temporal information, e.g. $\text{at(} \text{born}(\text{john}), t \text{)}$. But also, TACL can be regarded as a modal logic, where the annotations are seen as parameterized modal operators, e.g. $\text{born}(\text{john}) \text{ (at } t \text{)}$.

In [6], a powerful temporal logic named MTL (tense logic extended by parameterized temporal operators) is translated into first order constraint logic. The resulting language subsumes Templog, as does TACL. The parameterized temporal operators of MTL correspond to the temporal annotations of TACL. The constraint theory of MTL is rather complex as it involves quantified variables and implication, whose treatment goes beyond standard CLP implementations. On the other hand, TACL inherits an efficient standard constraint-based implementation of annotations from the ACL framework.

The ACL framework shares ideas with the work in [8]. There, a proof system for a large
class of (propositional and) quantified modal logics is formalized. The class is modular both
with respect to properties of the accessibility relation in the Kripke frame and leads to a
simple implementation of a modal logic theorem prover in standard logical frameworks.
Obviously, this approach is motivated by the same ideas that motivated ACL, though
on a much wider and thus more abstract class of languages. What [8] calls base logic,
corresponds to FOL in ACL, what is called relational theory, corresponds to the constraint
theory of ACL. The relational theory defines the accessibility relation in the Kripke frame
by (Horn) rules, while the constraint theory can be any FOL theory, as long as there are
efficient algorithms to implement it. Already in [10] it has been argued that the accessibility
relation can be regarded as constraint and the associated axioms as constraint theory.

In [4], translations between signed logic and FOL are investigated. Annotated logic [23]
can be embedded in signed logic. Both formalisms work with labels whose structure forms
a lattice. The two rules of regular unit resolution for signed logic correspond exactly to
A-resolution of constraint annotated logics. In [4], the lattice structure is encoded by giving
FOL clauses for each pair of lattice elements (i.e., labels). This is only feasible for finite
lattices. In ACL, the use of a constraint theory also allows for infinite lattices using a small
number of axioms.

5 Conclusions

We investigated semantics of a considerable subset of the language TACL that allows us
to reason about qualitative and quantitative, definite and indefinite temporal information
using time points and time periods. Its expressive power has been illustrated with some
examples. We defined the operational (top-down) semantics of TACLW by presenting a
meta-interpreter for it. Then we provided TACL with a fixpoint (bottom-up) semantics,
based on the definition of an immediate consequence operator.

Here we considered the subset of TACL, where time points are totally ordered, sets
of time points are convex and non-empty, and only atomic formulae can be annotated.
Furthermore clauses are free of negation. In general, in TACL arbitrary formulae can
be annotated. In some cases, as shown in [13], the annotations can be pushed inside
disjunctions, conjunctions and negation. This means that the omission of negation is the
main restriction of the current work. Consequently, we want to investigate next how the
semantics can be adapted to deal with negation.

Another interesting direction for future research is the representation and handling of
spatio-temporal data using annotations.

Acknowledgments: We thank Paolo Baldan and Roberta Gori for their useful comments
and suggestions. This work has been partially supported by Esprit Working Group 28115
- DeduGIS.

References


