Temporal Annotations for a Contextual Logic Programming Language

Vitor Nogueira and Salvador Abreu

Universidade de Évora and CENTRIA, Portugal
{vbn,spa}@di.uevora.pt

Abstract In this paper we propose the combination of modularity and temporal reasoning using logic programming as common ground. Moreover, we consider that the usage of a given module is influenced by temporal constraints, i.e. modularity and temporal reasoning are strongly connected. Besides illustrative examples, we also present the operational semantics and corresponding compiler for this language.

1 Introduction

The importance of representing and reasoning about temporal information is well known not only in the database community but also in the artificial intelligence one. In the past decades the volume of temporal data has grown enormously, making modularity a requisite for any language suitable for developing applications for such domains. One expected approach in devising a language with modularity and temporal reasoning is to consider that these characteristics co-exist without any direct relationship (see for instance the language MuTACL P [BMRT02] or [NA06]). Nevertheless we can also conceive a scenario where modularity and time are more integrated, for instance where the usage of a module is influenced by temporal conditions. In this paper we follow the later approach in defining a temporal extension to a language called Contextual Logic Programming (CxLP) [MP93]. This language is a simple and powerful extension of logic programming with mechanisms for modularity. Recent work not only presented a revised specification of CxLP together with a new implementation for it but also explained how this language could be seen as a shift into the Object-Oriented Programming paradigm [AD03]. Finally, CxLP structure is very suitable for integrating with temporal reasoning since its quite straightforward to add the notion of time of the context and let that time help deciding if a certain module is eligible or not to solve a goal.

For temporal representation and reasoning we chose Temporal Annotated Constraint Logic Programming (TACL P) [Frü94Frü96] since this language supports qualitative and quantitative (metric) temporal reasoning involving both time points and time periods (time intervals) and their duration. Moreover, it allows one to represent definite, indefinite and periodical temporal information.

The remainder of this article is structured as follows. In Sects. 2 and 3 we briefly overview CxLP and TACL P, respectively. Section 4 presents the temporal
extension of CxLP and Sect. 5 relates it with other languages. Conclusions and proposals for future work follows.

2 An Overview of Contextual Logic Programming

For this overview we assume that the reader is familiar with the basic notions of Logic Programming. Contextual Logic Programming (CxLP) [MP93] is a simple yet powerful language that extends logic programming with mechanisms for modularity. In CxLP a finite set of Horn clauses with a given name is designated by unit. Using the syntax of GNU Prolog/CX (recent implementation for CxLP [AD03]) consider a unit named employee to represent some basic facts about university employees, using ta and ap as an abbreviation of teaching assistant and associate professor, respectively:

\[
\begin{align*}
\text{:- unit(employee(NAME, POSITION)).} \\
\text{item :- employee(NAME, POSITION).} \\
\text{employee(bill, ta).} \\
\text{employee(joe, ap).} \\
\text{name(NAME).} \\
\text{position(POSITION).}
\end{align*}
\]

The main difference between the example above and a plain logic program is the first line that declares the unit name (employee) along with the unit arguments (NAME, POSITION). Unit arguments help avoid the annoying proliferation of predicate arguments, which occur whenever a global structure needs to be passed around. A unit argument can be interpreted as a “unit global” variable, i.e. one which is shared by all clauses defined in the unit. Therefore, as soon as a unit argument gets instantiated, all the occurrences of that variable in the unit are replaced accordingly.

Suppose also that each employee’s position has an associated index (integer) that will be used to calculate the salary. Such relation can be easily expressed by the following unit index:

\[
\begin{align*}
\text{:- unit(index(POSITION, INDEX)).} \\
\text{item :-} \\
\text{\quad index(POSITION, INDEX).} \\
\text{\quad index(ta, 12).} \\
\text{\quad index(ap, 20).} \\
\text{\quad index(INDEX).} \\
\text{\quad position(POSITION).}
\end{align*}
\]
A set of units is designated as a *contextual logic program*. With the units above we can build the program \( P = \{ \text{employee}, \text{index} \} \).

Given that in the same program we can have two or more units with the same name but different arities, to be more precise besides the unit name we should also refer its arity i.e. the number of arguments. Nevertheless, since most of the times there is no ambiguity, we omit the arity of the units. If we consider that *employee* and *index* designate sets of clauses, then the resulting program is given by the union of these sets.

For a given CxLP program, we can impose an order on its units, leading to the notion of *context*. Contexts are implemented as lists of unit designators and each computation has a notion of its *current context*. The program denoted by a particular context is the union of the predicates that are defined in each unit. Moreover, we resort to the *override semantics* to deal with multiple occurrences of a given predicate: only the topmost definition is visible.

To construct contexts, we have the *context extension* operation denoted by :>. The goal \( U :> G \) extends the current context with unit \( U \) and resolves goal \( G \) in the new context. For instance to obtain Bill’s position we could do:

\[
?- \text{employee(bill, P)} :> \text{item}.
\]

\[
P = \text{ta}
\]

In this query we extend the initial empty context \([\text{[]}\) with unit *employee* obtaining context \([\text{employee(bill, P)}]\) and then resolve query *item*. This leads to \( P \) being instantiated with *ta*.

Suppose also that the employee’s salary is obtained by multiplying the index of its position by the base salary. To implement this rule consider the unit *salary*:

\[
:\text{-unit(salary(SALARY)).}
\]

\[
\text{item} :-
\quad \text{position(P),}
\quad [\text{index(P, I)}] :< \text{item},
\quad \text{base_salary(B),}
\quad \text{SALARY is I*B.}
\]

\[
\text{base_salary(100).}
\]

The unit above introduces a new operator (\(<\)) called *context switch*: goal \([\text{index(P, I)}] :< \text{item}\) invokes *item* in context \([\text{index(P, I)}]\). To better grasp the definition of this unit consider the goal:

\[
?- \text{employee(bill, P)} :> (\text{item, salary(S)} :> \text{item}).
\]

\[1\) In the GNU Prolog/CX implementation the empty context its not entirely *empty* since it contains all the standard Prolog predicates such as \=/2.\]
Since we already explained the beginning of this goal, let's see the remaining part. After \texttt{salary/1} being added, we are left with the context \([\texttt{salary(S), employee(bill,ta)}]\). The second \texttt{item} is evaluated and the first matching definition is found in unit \texttt{salary}. Goal \texttt{position(P)} is called and because there is no rule for this goal in the current unit (\texttt{salary}), a search in the context is performed. Since \texttt{employee} is the topmost unit that has a rule for \texttt{position(P)}, this goal is resolved in the (reduced) context \([\texttt{employee(bill, ta)}]\). In an informal way, we queried the context for the position of whom we want to calculate the salary, obtaining \texttt{ta}. Next, the index corresponding to such position is computed, i.e. \([\texttt{index(ta, I)}] \iff \texttt{item} obtaining I = 12\). Finally, to calculate the salary, we just need to multiply the index by the base salary, getting \(S = 1200\) as answer and \([\texttt{salary(1200), employee(bill, ta)}]\) as the final context.

3 Temporal Annotated Constraint Logic Programming

This section presents a brief overview of Temporal Annotated Constraint Logic Programming (TACLP) that follows closely Sect. 2 of [RF00]. For a more detailed explanation of TACLP see for instance [Fri96].

We consider the subset of TACLP 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.

Time can be discrete or dense. Time points are totally ordered by the relation \(\leq\). We call the set of time points \(D\) and suppose that a set of operations (such as the binary operations +, −) to manage such points is associated with it. We assume that the time-line is left-bounded by the number 0 and open to the future (\(\infty\)). A time period is an interval \([r, s]\) with \(0 \leq r \leq s \leq \infty\), \(r \in D\), \(s \in D\) and represents the convex, non-empty set of time points \(\{t \mid r \leq t \leq s\}\). Therefore the interval \([0, \infty]\) denotes the whole time line.

Definition 1 (Annotated Formula). An annotated formula is of the form \(A\alpha\) where \(A\) is an atomic formula and \(\alpha\) an annotation. Let \(t\) be a time point and \(I\) be a time period:

(at) The annotated formula \(A\) at \(t\) means that \(A\) holds at time point \(t\).

(th) The annotated formula \(A\) th \(I\) means that \(A\) holds throughout \(I\), i.e. at every time point in the period \(I\).

A th-annotated formula can be defined in terms of at as: \(A\) th \(I\) \iff \(\forall t\ (t \in I \rightarrow A\) at \(t)\)

(in) The annotated formula \(A\) in \(I\) means that \(A\) holds at some time point(s) in the time period \(I\), but there is no knowledge when exactly. The in annotation accounts for indefinite temporal information.

An in-annotated formula can also be defined in terms of at: \(A\) in \(I\) \iff \(\exists t\ (t \in I \land A\) at \(t)\).

The set of annotations is endowed with a partial order relation \(\sqsubseteq\) which turns into a lattice. Given two annotations \(\alpha\) and \(\beta\), the intuition is that \(\alpha \sqsubseteq \beta\) if \(\alpha\) is “less informative” than \(\beta\) in the sense that for all formulae \(A\), \(A\beta \Rightarrow A\alpha\).
In addition to Modus Ponens, TACLP has the following two inference rules:

\[
\frac{A\alpha \gamma \sqsubseteq \alpha}{A\gamma} \quad \text{rule (\sqsubseteq)} \quad \frac{A\alpha A\beta \gamma = \alpha \sqcup \beta}{A\gamma} \quad \text{rule (\sqcup)}
\]

The rule (\sqsubseteq) 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 (\sqcup) says that if a formula holds with some annotation and the same formula holds with another annotation then it holds in the least upper bound of the annotations. Assuming \( r_1 \leq s_1 \), \( s_1 \leq s_2 \) and \( s_2 \leq r_2 \), we can summarize the axioms for the lattice operation \( \sqsubseteq \) by:

\[
in[r_1, r_2] \sqsubseteq in[s_1, s_2] \sqsubseteq in[s_1, s_1] = at \ s_1 = th[s_1, s_1] \sqsubseteq th[s_1, s_2] \sqsubseteq th[r_1, r_2]
\]

The axioms of the least upper bound \( \sqcup \) can be restricted to:

\[
\begin{align*}
th[s_1, s_2] \sqcup th[r_1, r_2] &= th[s_1, r_2] \iff s_1 \leq r_1, r_1 \leq s_2, s_2 \leq r_2
\end{align*}
\]

A TACLP program is a finite set of TACLP clauses. A TACLP clause is a formula of the form \( A\alpha \leftarrow C_1, \ldots, C_n, B_{1\alpha_1}, \ldots, B_{m\alpha_m} \) \( (m, n \geq 0) \) where \( A \) is an atom, \( \alpha \) and \( \alpha_i \) are optional temporal annotations, the \( C_j \)'s are the constraints and the \( B_i \)'s are the atomic formulae. Moreover, besides an interpreter for TACLP clauses there is also a compiler that translates them into Constraint Logic Programming (CLP).

4 Temporal Annotations and Contextual Logic Programming

In CxLP with overriding semantics, to solve a goal \( G \) in a context \( C \), a search is performed until the topmost unit of \( C \) that contains clauses for the predicate of \( G \) is found. We propose to adapt this basic mechanism of CxLP (called context search) in order to incorporate the temporal reasoning. To accomplish this we add temporal annotations to contexts and to units and it will be the relation between those two types of annotations that will help to decide if a given unit is eligible to match a goal during a context search.

The addition of time to a context is rather intuitive: instead of a list of unit designators \([u_1, \ldots, u_n]\) we now have a temporally annotated list of units designators \([u_1, \ldots, u_n]\alpha\). This annotation \( \alpha \) is called the time of the context and by default, contexts are implicitly annotated with the current time.

We could follow an approach for units similar to the one proposed for contexts, i.e. to add a temporal annotation to a unit’s declaration. Hence we could have units definitions like: \( \text{unit}(\text{foo}(X)) \ \text{th} \ [1,4] \).

Nevertheless, units and more specifically, units with arguments allow for a refinement of the temporal qualification, i.e. instead of a qualifying the entire unit, we can have several qualifications, one for each possible argument instantiation. For the unit \text{foo} above we could have:

\[\uparrow\] The least upper bound only has to be computed for overlapping \text{th} annotations.
:- unit(foo(X)).
foo(a) th [1,2].
foo(b) th [3,4].

Where the first annotated fact states that unit foo with its argument instantiated to a has the annotation th [1,2]. With these annotations, unit foo will be eligible to match a goal in the context [..., foo(a), ...] in [1,4] but its not eligible in the context [..., foo(b), ...] th [3,6] since in [1,4] ⊑ th[1,2] and th[3,6] ⊈ th[3,4]. We call those annotated facts the temporal conditions of the unit.

Each unit defines one temporally annotated predicate with the same name as the unit and arity equal to the number of the unit arguments. For the case of atemporal (timeless) units, it is assumed by default that we have the most general unit designator annotated with the complete time line.

We decided that these temporal annotations can only appear as heads of rules whose body is true, i.e. facts. Such restriction is motivated by efficiency reasons since this way we can compute the least upper bound (∪) of the th annotated facts before runtime and this way checking the units temporal conditions during a context search is simplified to the verification of partial order (⊑) between annotations. Moreover, as we shall see in the examples, such restrictions are not limitative since the expressiveness of contexts allow us to simulate TACLP clauses.

Revisiting the employee example, units employee and index with temporal information can be written as:

```prolog
:- unit(employee(NAME, POSITION)).
employee(bill, ta) th [2004, inf].
employee(joe, ta) th [2002, 2006].
employee(joe, ap) th [2007, inf].
item.
p

:- unit(index(POSITION, INDEX)).
index(ta, 10) th [2000, 2005].
index(ta, 12) th [2006, inf].
index(ap, 19) th [2000, 2005].
index(ap, 20) th [2006, inf].
item.
p
```

As an example, consider the goal:

?- at 2005 :> employee(joe, P) :> item.
P = ta

In this goal, after asserting that the context temporal annotation is at 2005, unit employee is added to the context and goal item invoked. The evaluation of item is true as long as the unit is eligible in the current context, and this is true if P is instantiated with ta (teaching assistant), therefore P = ta.

Unit salary can be defined as:

3 The reader should notice that this way its still possible to annotate the entire unit, since we can annotate the unit most general designator, for instance we could have foo(_) th [1, 10].
:- unit(salary(SALARY)).
item :-
    position(P), index(P, I) :> item,
    base_salary(B), SALARY is B*I.

base_salary(100).

There is no need to annotate the goals position(P) or index(P, I) :> item since they are evaluated in a context with the same temporal annotation.
To find out Joe’s salary in 2005 we can do:

?- at 2005 :> employee(joe, P) :> salary(S) :> item.
P = ta
S = 1000

In the goal above item is evaluated in the context [salary(S), employee(joe, P)] (at 2005). Since salary is the topmost unit that defines it, the body of the rule for such predicate is evaluated. In order to use the unit employee(joe, P) to solve position(P), such unit must satisfy the temporal conditions (at 2005), that in this case stands for instantiating P with ta, therefore we obtain position(ta). A similar reasoning applies for goal index(ta, I) :> item, i.e. this item is resolved in context [index(ta, 10), salary(S), employee(joe, ta)] (at 2005). The remainder of the rule body is straightforward, leading to the answer P = ta and S = 1000.

4.1 Operational Semantics

To define the operational semantics we assume the following notation: $C, C'$ for contexts, $u$ for unit, $\theta, \sigma, \varphi, \epsilon$ for substitutions, $\alpha, \beta, \gamma$ for temporal annotations and $\emptyset, G$ for non-annotated goals.

We also assume a prior computation of the least upper bound for the units $th$ annotations. This procedure is rather straightforward and can be describe as: if $A \th I$ and $A \th J$ are in a unit $u$, such that $I$ and $J$ overlap, then remove those facts from $u$ and insert $A \th (I \sqcup J)$. This procedure stops when there are no more facts in that conditions. Moreover, the termination is guaranteed because at each step we decrease the size of a finite set, the set of $th$ annotated facts.

Null goal

\[ C\alpha \vdash \emptyset[\epsilon] \] (1)

The null goal is derivable in any temporal annotated context, with the empty substitution $\epsilon$ as result.

Conjunction of goals

\[ \frac{C\alpha \vdash G_1[\theta]}{C\alpha \vdash G_1, G_2[\theta\sigma]} \quad \frac{C\alpha \vdash G_2[\sigma]}{C\alpha \vdash G_1, G_2[\theta\sigma]} \] (2)
To derive the conjunction derive one conjunct first, and then the other in the same context with the given substitutions.

Since \( C \) may contain variables in unit designators or temporal terms that may be bound by the substitution \( \theta \) obtained from the derivation of \( G_1 \), we have that \( \theta \) must also be applied to \( C \alpha \) in order to obtain the updated context in which to derive \( G_2 \theta \).

**Context inquiry**

\[
\frac{\theta = \text{mgu}(C, C')}{C \alpha \vdash :> C' \beta[\theta]} \quad \beta \sqsubseteq \alpha
\]

In order to make the context switch operation useful, there needs to be an operation which fetches the context. This rule recovers the current context \( C \) as a term and unifies it with term \( C' \), so that it may be used elsewhere in the program. Moreover, the annotation \( \beta \) must be less (or equal) informative than the annotation \( \alpha \) \((\beta \sqsubseteq \alpha)\).

**Context switch**

\[
\frac{C' \beta \vdash G[\theta]}{C \alpha \vdash C' \beta :< G[\theta]}
\]

The purpose of this rule is to allow execution of a goal in an arbitrary temporal annotated context, independently of the current annotated context. This rule causes goal \( G \) to be executed in context \( C' \beta \).

**Reduction**

\[
\frac{(uC \alpha) \theta \sigma \vdash B \theta \sigma[\varphi]}{uC \alpha \vdash G[\theta \sigma \varphi[\text{vars}(G)]]} \quad \begin{cases} H \leftarrow B \in u \\
\theta = \text{mgu}(G, H) \\
(\theta \sigma \beta) \in u \\
\alpha \sqsubseteq \beta
\end{cases}
\]

This rule expresses the influence of temporal reasoning on context search. In an informal way we can say that when a goal \( (G) \) has a definition \((H \leftarrow B \in u \text{ and } \theta = \text{mgu}(G, H))\) in the topmost unit \( (u) \) of the annotated context \((uC \alpha)\), and such unit satisfies the temporal conditions, to derive the goal we must call the body of the matching clause, after unification. The verification of the temporal conditions stands for checking if there is a unit temporal annotation \(((\theta \sigma \beta) \in u)\) that is “more informative” than the annotation of the context \((\alpha \subseteq \beta)\), i.e. if \((\theta \sigma \beta)\) \(\alpha\) is true.

**Context traversal:**

\(^4\) The notation \( \delta[V] \) stands for the restriction of the substitution \( \delta \) to the variables in \( V \).

\(^5\) Although this rule might seem complex, that has to do essentially with the abundance of unification’s \( (\theta \sigma \varphi) \).
\[ Ca \vdash G[\theta] \]
\[ uCa \vdash G[\theta] \quad \{ \text{pred}(G) \notin \pi \} \]  

When none of the previous rules applies and the predicate of \( G \) isn’t defined in the predicates of \( u (\pi) \), remove the top element of the context, i.e. resolve goal \( G \) in the supercontext.

*Application of the rules* It is almost direct to verify that the inference rules are mutually exclusive, leading to the fact that given a derivation tuple \( Ca \vdash G[\theta] \) only one rule can be applied.

### 4.2 Compiler

The compiler for this language can be obtained by combining a program transformation with the compiler for TACL P [Frü96]. Given a unit \( u \), such transformation rewrites each predicate \( P \) in the head of a rule by \( P' \) and adds the following definition to unit \( u \):

\[ P :- \text{Temporal\_Conditions} \rightarrow P' ; ^^P . \]

This states that resolving \( P \) is equivalent to invoking \( P' \), only if the temporal conditions are satisfied. If not, \( P \) must be solved in the supercontext \( (:^P) \), i.e. \( P \) is called in the context obtained by dropping \( u \) (the topmost unit) from the current context.

The temporal condition can be formalized as the conjunction \( \langle [U\alpha], U\alpha \rangle \), where the first conjunct queries the context for its temporal annotation \( (\alpha) \) and its topmost unit \( (U) \), i.e. the current unit. The second conjunct checks if the current unit satisfies the time of the context.

As it should be expected, the compiled language is CxLP with constraints. Finally, since GNU Prolog/CX besides the CxLP primitives also has a constraint solver for finite domains (CLP(FD)), the implementation of this language is direct on such system.

### 4.3 Application to Legal Reasoning

Legal reasoning is a very productive field to illustrate the application of these languages. Not only a modular approach is very suitable for reasoning about laws but also time is pervasive in their definition.

The following example was taken from the British Nationality Act and it was presented in [BMRT02] to exemplify the usage of the language MuTACL P. The reason to use an existing example is twofold: not only we consider it to be a simple and concise sample of legal reasoning but also because this way we can give a more thorough comparison with MuTACL P. The textual description of this law can be given as a person \( X \) obtains the British Nationality at time \( T \) if:

- \( X \) is born in the UK at the time \( T \)
– T is after the commencement
– Y is a parent of X
– Y is a British citizen or resident at time T.

Assuming that the temporal unit \texttt{person} represents the name and the place where a person was born:

\begin{verbatim}
:- unit(person(Name, Country)).
person(john, uk) th ['1969-8-10', inf].
\end{verbatim}

The temporal annotation of this unit can be interpreted as the person time frame, i.e. when she was born and when she died (if its alive, we represent it by \texttt{inf}).

Before presenting the rule for the nationality act we still need to represent some facts about who is a British citizen along with who is parent of whom:

\begin{verbatim}
:- unit(british_citizen(Name)).
british_citizen(bob) th ['1940-9-6', inf].
:- unit(parent(Parent, Son)).
parent(bob, john) th ['1969-8-10', inf].
\end{verbatim}

Considering that the commencement date for this law is '1955-1-1', one formalization of this law in our language is

\begin{verbatim}
\texttt{th} [L, _] :> \texttt{person}(X, uk) :> \texttt{item}, \texttt{fd\_min}(L, T),
\texttt{'}1955-1-1\texttt{'} #=< T,
\texttt{at} T :> (parent(Y, X) :> \texttt{item},
\texttt{british\_citizen}(Y) :> \texttt{item}; british\_resident(Y) :> \texttt{item}).
\end{verbatim}

The explanation of this goal is quite simple because each line of the goal corresponds and is presented in the same order as the textual description of the law given above.

5 Related Work

Since \cite{BMRT02} relates MuTACLCP with proposals such as Temporal Datalog \cite{OM94} and the work on amalgamating knowledge bases \cite{Sub94}, we decided to confine ourselves to the comparison between MuTACLCP and our language. MuTACLCP (Multi-Theory Temporal Annotated Constraint Logic Programming) is a knowledge representation language that provides facilities for modeling and handling temporal information, together with some basic operators for combining different knowledge bases. Although both MuTACLCP and the language here proposed use TACLCP (Temporal Annotated Constraint Logic Programming) for handling temporal information, it is in the way that modularity is dealt that they differ.

\footnote{\texttt{fd\_min}(X, N) succeeds if N is the minimal value of the current domain of X.}
diverge: we follow a dynamic approach (also called programming-in-the-small) while MuTACLP engages a static one (also called programming-in-the-large).

Moreover, the use of contexts allows for a more compact writing where some of the annotations of the MuTACLP version are subsumed by the annotation of the context. For instance, one of the rules of the MuTACLP version of the example of legal reasoning is:

\[
\text{get
citizenship}(X) \text{ at } T \leftarrow T \geq \text{Jan 1 1955}, \text{born}(X, \text{uk}) \text{ at } T, \\
\text{parent}(Y, X) \text{ at } T, \\
\text{british
citizen}(Y) \text{ at } T.
\]

6 Conclusion and Future Work

In this paper we presented a temporal extension of CxLP where time influences the eligibility of a module to solve a goal. Besides illustrative examples we also provided a compiler, allowing this way for the development of applications based on these languages. Although we presented the operational semantics we consider that to obtain a more solid theoretical foundation there is still need for a fixed point or declarative definition.

Besides the domain of application exemplified we are currently applying the language proposed to other areas such as medicine, natural language and workflow management systems.

Finally, it is our goal to show that this language can act as the backbone for constructing and maintaining temporal information systems.

References


