Integrating Temporal Annotations in a Modular Logic Language

Vitor Nogueira and Salvador Abreu

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

Abstract Albeit temporal reasoning and modularity are very prolific fields of research in Logic Programming (LP) we find few examples of their integration. Moreover, in those examples, time and modularity are considered orthogonal to each other. In this paper we propose the addition of temporal annotations to a modular extension of LP such that the usage of a module is influenced by temporal conditions. Besides illustrative examples we also provide an operational semantics together with a compiler, allowing this way for the development of applications based on such 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 MuTACLP [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 extensions 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 to decide if a certain module is eligible or not to solve a goal.

For temporal representation and reasoning we chose Temporal Annotated Constraint Logic Programming (TACLP) [Frü94,Frü96] 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 TACLP, 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 `t`eaching `a`ssistant and `a`ssociate `p`rofessor, respectively:

```
:-unit(employee(NAME, POSITION)).

item :- employee(NAME, POSITION).
employee(bill, ta).
employee(joe, ap).

name(NAME).
position(POSITION).
```

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 integer rhat will be used to calculate the salary. Such relation can be easily expressed by the following unit `index`:

```
 :- unit(index(POSITION, INDEX)).

 item :-
      index(POSITION, INDEX).

 index(ta, 12).
 index(ap, 20).

index(INDEX).
position(POSITION).
```

A set of units is designated as a *contextual logic program*. With the units above we can build the program `P = {employee, 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:

```
| ?- employee(bill, P) :> item

P = ta
```

In this query we extend the initial empty context `[]` [1] with unit employee obtaining context `[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`:

```
:-unit(salary(SALARY)).

item :-
   position(P),
   [index(P, I)] :< item,
   base_salary(B),
   SALARY is I*B.

base_salary(100).
```

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

```
| ?- employee(bill, P) :> (item, salary(S) :> item).
```

[1] In the GNU Prolog/CX implementation the empty context contains all the standard Prolog predicates such as `=/2`.

Since we already explained the beginning of this goal, lets see the remaining part. After `salary/1` being added, we are left with the context `[salary(S), employee(bill,ta)]`. The second `item` is evaluated and the first matching definition is found in unit `salary`. Goal `position(P)` is called and since there is no rule for this goal in the current unit (`salary`), a search in the context is performed. Since `employee` is the topmost unit that has a rule for `position(P)`, this goal is resolved in the (reduced) context `[employee(bill, ta)]`. In an informal way, we queried the context for the position of whom we want to calculate the salary, obtaining `ta`. Next, we query the index corresponding to such position, i.e. `[index(ta, I)] :< item`. Finally, to calculate the salary, we just need to multiply the index by the base salary, obtaining `S = 1200` with the final context `[salary(1200), employee(bill, ta)]`.

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 [Frü96].

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 the future (∞). 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 α 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: $A\ th\ I \Leftrightarrow\ \forall t\ (t \in I \rightarrow A\ at\ t)$

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

The set of annotations is endowed with a partial order relation $\sqsubseteq$ which turns into a lattice. Given two annotations α and β, the intuition is that $\alpha \sqsubseteq \beta$ if α is "less informative" than β 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:

$$\frac{A\alpha \quad \gamma \sqsubseteq \alpha}{A\gamma} \ \text{rule}\,(\sqsubseteq) \qquad \frac{A\alpha \quad A\beta \quad \gamma = \alpha \sqcup \beta}{A\gamma} \ \text{rule}\,(\sqcup)$$

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 [2]:

$$th[s_1, s_2] \sqcup th[r_1, r_2] = th[s_1, r_2] \Leftrightarrow s_1 \leq r1, r_1 \leq s_2, s_2 \leq r_2$$

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, α and α_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 its CLP form.

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 include the temporal reasoning aspects. To accomplish this we add temporal annotations to contexts and to units and it will be the relation between those two 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 $[u_1, \ldots, un]$ we now have a temporally annotated list of units designators $[u_1, \ldots, un]\alpha$. This annotation α 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 `:- unit(foo(X)) 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 `foo` above we could have:

[2] The least upper bound only has to be computed for overlapping `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 $in[1,4] \sqsubseteq th[1,2]$ and $th[3,6] \not\sqsubseteq th[3,4]$. We call those annotated facts the *temporal conditions* of the unit [3].

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 ($\sqcup$) 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 ($\sqsubseteq$) 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` can be written as:

```
:- unit(employee(NAME, POSITION)).    :- unit(index(POSITION, INDEX)).
                                      index(ta, 10) th [2000, 2005].
employee(bill, ta) th [2004, inf].    index(ta, 12) th [2006, inf].
employee(joe, ta) th [2002, 2006].    index(ap, 19) th [2000, 2005].
employee(joe, ap) th [2007, inf].     index(ap, 20) th [2006, inf].
item.                                 item.
position(POSITION).                   position(POSITION).
name(NAME).                           index(INDEX).
```

As an exemplification, consider the goal:

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

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 Compiler

The compiler for this language can be obtained by combining a program transformation with the compiler for TACLP [Frü96]. Given a unit `u`, such transformation rewrites each predicate `P` in the head of a rule by `P'` and add the following clauses to `u`:

```
 P :- Temporal_Conditions, !, P'.
 P :- :^ P.
```

stating the resolving `P` is equivalent to invoke `P'`, if the temporal conditions are satisfied. Otherwise `P` must be solved in the supercontext (`:^ P`), i.e. `P` is called in the context obtained from removing `u`.

The temporal condition can be formalized as the conjunction $:< [U|\_]\ \alpha,\ U\alpha$, where the first conjunct queries the context for its temporal annotation (α) 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 a system.

4.2 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 MuTACLP. 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 MuTACLP. 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 `person` represents the name and the place where a person was born:

```
:- unit(person(Name, Country)).
person(john, uk) th [’1969-8-10’, inf].
```

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 `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:

```
    :- unit(british_citizen(Name)).  :- unit(parent(Parent, Son)).

    british_citizen(bob)             parent(bob, john)
         th [’1940-9-6’, inf].            th [’1969-8-10’, inf].
```

Considering that the commencement date for this law is ’1955-1-1’, one formalization of this law in our language is [4]:

```
th [L, _] :> person(X, uk) :> item, fd_min(L, T),
’1955-1-1’ #=< T,
at T :> (parent(Y, X) :> item,
                       (british_citizen(Y) :> item;
                        british_resident(Y) :> item)).
```

The explanation of the goal above is quite simple since each line correspond to one condition of the textual description of the law.

[4] `fd_min(X, N)` succeeds if `N` is the minimal value of the current domain of `X`.

5 Related Work

Since [BMRT02] relates MuTACLP with proposals such as Temporal Datalog [OM94] and the work on amalgamating knowledge bases [Sub94], we decided to confine ourselves to the comparison between MuTACLP and our language. MuTACLP (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 MuTACLP and the language here proposed use TACLP (Temporal Annotated Constraint Logic Programming) for handling temporal information, it is in the way that modularity is dealt that they 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:

```
get_citizenship(X) at T <- T >= Jan 1 1955, born(X, uk) at T,
                           parent(Y, X) at T,
                           british_citizen(Y) at T.
```

In [NA06] a similar argument was used when comparing with *relational frameworks* such as the one proposed by Combi and Pozzi in [CP04] for workflow management systems, where relational queries were more verbose that its contextual version.

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 provided (in the Appendix A) 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 and natural language. Finally, it is our goal to continue previous work [AN06,ADN04] and show that this language can act as the backbone for constructing and maintaining temporal information systems.

References

AD03. Salvador Abreu and Daniel Diaz. Objective: In minimum context. In Catuscia Palamidessi, editor, *ICLP*, volume 2916 of *Lecture Notes in Computer Science*, pages 128–147. Springer, 2003.

ADN04. Salvador Abreu, Daniel Diaz, and Vitor Nogueira. Organizational information systems design and implementation with contextual constraint logic programming. In *IT Innovation in a Changing World – The 10$^{\text{th}}$ International Conference of European University Information Systems*, Ljubljana, Slovenia, June 2004.

AN06. Salvador Abreu and Vitor Nogueira. Towards structured contexts and modules. In Sandro Etalle and Miroslaw Truszczynski, editors, *ICLP*, volume 4079 of *Lecture Notes in Computer Science*, pages 436–438. Springer, 2006.

BMRT02. Paolo Baldan, Paolo Mancarella, Alessandra Raffaetà, and Franco Turini. Mutaclp: A language for temporal reasoning with multiple theories. In Antonis C. Kakas and Fariba Sadri, editors, *Computational Logic: Logic Programming and Beyond*, volume 2408 of *Lecture Notes in Computer Science*, pages 1–40. Springer, 2002.

CP04. Carlo Combi and Giuseppe Pozzi. Architectures for a temporal workflow management system. In *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, pages 659–666, New York, NY, USA, 2004. ACM Press.

Frü94. T. Frühwirth. Annotated constraint logic programming applied to temporal reasoning. In M. Hermenegildo and J. Penjam, editors, *Programming Language Implementation and Logic Programming: 6th International Symposium (PLILP'94)*, pages 230–243. Springer, Berlin, Heidelberg, 1994.

Frü96. Thom W. Frühwirth. Temporal annotated constraint logic programming. *J. Symb. Comput.*, 22(5/6):555–583, 1996.

MP93. Luís Monteiro and António Porto. A Language for Contextual Logic Programming. In K.R. Apt, J.W. de Bakker, and J.J.M.M. Rutten, editors, *Logic Programming Languages: Constraints, Functions and Objects*, pages 115–147. MIT Press, 1993.

NA06. Vitor Nogueira and Salvador Abreu. Temporal contextual logic programming. In Francisco J. López Fraguas, editor, *Proceedings of the 15th Workshop on Functional and (Constraint) Logic Programming (WFLP'06)*, Madrid, Spain, November 2006. Electronic Notes in Theoretical Computer Science.

OM94. Mehmet A. Orgun and Wanli Ma. An overview of temporal and modal logic programming. In *ICTL '94: Proceedings of the First International Conference on Temporal Logic*, pages 445–479, London, UK, 1994. Springer-Verlag.

RF00. Alessandra Raffaetà and Thom Frühwirth. *Labelled deduction*, chapter Semantics for temporal annotated constraint logic programming, pages 215–243. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

Sub94. V. S. Subrahmanian. Amalgamating knowledge bases. *ACM Trans. Database Syst.*, 19(2):291–331, 1994.

A Operational Semantics

In this section we assume the following notation: C, C' for contexts, u for unit, $\theta, \sigma, \varphi, \epsilon$ for substitutions, α, β, γ 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 $B\ th\ J$ 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

$$\frac{}{C\alpha \vdash \emptyset[\epsilon]} \tag{1}$$

The null goal is derivable in any temporal annotated context, with the *empty* substitution ϵ as result.

Conjunction of goals

$$\frac{C\alpha \vdash G_1[\theta] \qquad\qquad C\alpha\theta \vdash G_2\theta[\sigma]}{C\alpha \vdash G_1, G_2[\theta\sigma\lceil vars(G_1, G_2)]} \tag{2}$$

To derive the conjunction derive one conjunct first, and then the other in the same context with the given substitutions [5].

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

Context inquiry

$$\frac{}{C\alpha \vdash\ :>C'\beta[\theta]} \begin{cases} \theta = \mathrm{mgu}(C, C') \\ \beta \sqsubseteq \alpha \end{cases} \tag{3}$$

In order to make the context switch operation (4) 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 β must be less (or equal) informative than the annotation α ($\beta \sqsubseteq \alpha$).

Context switch

$$\frac{C'\beta \vdash G[\theta]}{C\alpha \vdash C'\beta\ :< G[\theta]} \tag{4}$$

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

[5] The notation $\delta\lceil V$ stands for the restriction of the substitution δ to the variables in V.

$$\frac{(uC\alpha)\ \theta\sigma \vdash B\theta\sigma[\varphi]}{uC\ \alpha \vdash G[\theta\sigma\varphi\lceil vars(G)]}\begin{cases} H \leftarrow B \in u \\ \theta = \mathrm{mgu}(G,H) \\ (u\theta\sigma)\ \beta \in u \\ \alpha \sqsubseteq \beta \end{cases} \tag{5}$$

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 $(H \leftarrow B \in u$ and $\theta = \mathrm{mgu}(G,H))$ 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 [6]. The verification of the temporal conditions stands for checking if there is a unit temporal annotation $((u\theta\sigma)\beta \in u)$ that is "more informative" than the annotation of the context $(\alpha \sqsubseteq \beta)$, i.e. if $(u\theta\sigma)\ \alpha$ is true.

Context traversal:

$$\frac{C\alpha \vdash G[\theta]}{uC\alpha \vdash G[\theta]}\left\{ pred(G) \notin \overline{u} \right. \tag{6}$$

When none of the previous rules applies and the predicate of `G` isn't defined in the predicates of `u` $(\overline{u})$, 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 $C\alpha \vdash G[\theta]$ only one rule can be applied.

[6] Although this rule might seem complex, that has to do essentially with the abundance of unification's $(\theta\sigma\varphi)$