Branching-Time Logic Programming: The Language *Cactus* and its Applications*†

P. Rondogiannis[1], **M. Gergatsoulis**[2], **T. Panayiotopoulos**[3]

[1] Dept. of Computer Science, University of Ioannina,
P.O. BOX 1186, 45110 Ioannina, Greece,
e_mail: prondo@cs.uoi.gr

[2] Inst. of Informatics & Telecom., N.C.S.R. 'Demokritos',
153 10 A. Paraskevi Attikis, Greece
e_mail: manolis@iit.demokritos.gr

[3] Dept. of Computer Science, University of Piraeus
80 Karaoli & Dimitriou Str., 18534 Piraeus, Greece
e-mail : themisp@unipi.gr

*This work has been partially supported by the Greek General Secretariat of Research and Technology under the project "TimeLogic" of ΠΕΝΕΔ'95, contract no 1134.

†This paper appears in **Computer Languages** Vol 24, Issue 3, pages 155-178, 1998.

Abstract

Temporal programming languages provide a powerful means for the description and implementation of dynamic systems. However, most temporal languages are based on linear time, a fact that renders them unsuitable for certain types of applications (such as expressing properties of nondeterministic programs). In this paper we introduce the new temporal logic programming language **Cactus**, which is based on a branching notion of time. In Cactus, the truth value of a predicate depends on a hidden time parameter which varies over a tree-like structure. As a result, Cactus can be used to express in a natural way non-deterministic computations or generally algorithms that involve the manipulation of tree data structures. Moreover, Cactus appears to be appropriate as the target language for compilers or program transformers. Cactus programs can be executed using BSLD-resolution, a proof procedure based on the notion of *canonical temporal atoms/clauses*.

Keywords: Logic Programming, Temporal Logic Programming, Branching Time.

1 Introduction

Temporal programming languages [1, 2, 3] provide a powerful means for the description and implementation of *dynamic* systems. For example, consider the following Chronolog [4] program simulating the operation of the traffic lights:

```
first light(green).
next light(amber)  ←  light(green).
next light(red)  ←  light(amber).
next light(green)  ←  light(red).
```

However, Chronolog as well as most temporal languages [1, 5, 6, 7, 8, 9, 10] are based on linear flow of time, a fact that makes them unsuitable for certain types of applications. For example, as M. Ben-Ari, A. Pnueli and Z. Manna indicate [11], branching time logics are necessary in order to express certain properties of non-deterministic programs. Moreover, as it is argued by M. Wooldridge and M. Fisher [12], branching-time logics are suitable for describing and reasoning about multi-agent systems.

In this paper we present the new temporal logic programming language **Cactus** which is based on a tree-like notion of time; that is, every moment in time may have more than one immediate next moments. The new formalism is appropriate for describing non-deterministic computations or more generally computations that involve the manipulation of trees. The basic ideas underlying Cactus were initially introduced by the authors in [13].

Cactus supports two main operators: the temporal operator `first` refers to the beginning of time (or alternatively to the root of the tree). The temporal operator $\texttt{next}_i$ refers to the i-th child of the current moment (or alternatively, the i-th branch of the current node in the tree). Notice that we actually have a family $\{\texttt{next}_i \mid i \in \mathcal{N}\}$ of `next` operators, each one of them representing the different next moments that immediately follow the present one.

The tree in figure 1 represents the flow of time in the case where each moment has two immediate next moments. Formally, each moment in the time-tree can be represented by a list of natural numbers.

As an example, consider the following program:

```
first nat(0).
next_0 nat(Y) ←  nat(X),Y is 2*X+1.
next_1 nat(Y) ←  nat(X),Y is 2*X+2.
```

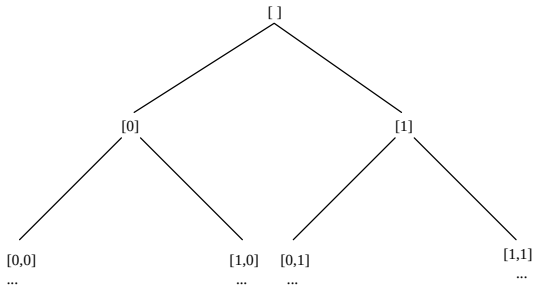

Figure 1: The tree representing the flow of branching time

The idea behind the above program is that the set of natural numbers can be mapped on a binary tree of the form shown in figure 2. More specifically, one can think of `nat` as a time-varying predicate. At the beginning of time (at the root of the tree) `nat` is true of the natural number `0`. At the left child of the root of the tree, `nat` is true of the value `1`, while at the right child it is true of the value `2`. In general, if `nat` is true of the value `X` at some node in the tree, then at the left child of that node `nat` will be true of `2*X+1` while at the right child of the node it will be true of `2*X+2`. One can easily verify that the tree created contains all the natural numbers.

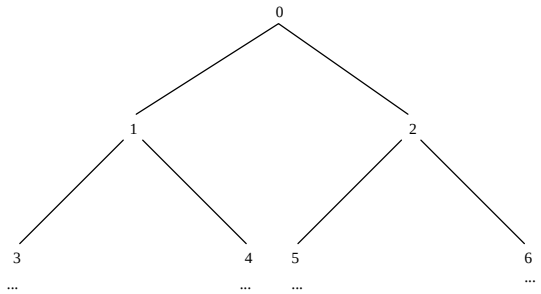

Figure 2: A mapping of the natural numbers on a binary tree

One could claim that branching time logic programming (or temporal logic programming in general) does not add much to logic programming, because *time* can always be added as an extra parameter to predicates. However, from a theoretical viewpoint this does not appear to be straightforward (see for example [14, 15] for a good discussion on this subject). Moreover, from a practical perspective, temporal languages are very expressive for many problem domains. As it will become apparent in the next sections, one can use the branching time concept in order to represent in a natural way time-dependent data as well as to reason in a lucid manner about these data.

It should be noted that the purpose of the present paper is to introduce the language Cactus, its underlying theory, and its practical applications. In this presentation we do not consider extensions of the language (such as meta or extra logical built-in predicates) that would be necessary when writing large-scale applications.

The rest of the paper is organized as follows: in section 2 we formally introduce the syntax of the language. In section 3, we present various Cactus programs which demonstrate its potential in expressing tree computations. Section 4 presents the underlying branching time logic BTL of Cactus. In section 5 we present a sound and complete proof procedure for Cactus programs. Section 6 discusses possible extensions of the language and section 7 gives the concluding remarks.

2 The syntax of Cactus programs

The syntax of Cactus programs is an extension of the syntax of Prolog programs. In the following we assume familiarity with the basic notions of logic programming [16].

The *temporal operators* of Cactus are `first` and a family of `next` operators, namely $\{\texttt{next}_i \mid i \in \mathcal{N}\}$. A *temporal atom* is an atomic formula preceded by a number (possibly 0) of temporal operators. The atomic formula of a temporal atom is also called *classical atom*. The sequence of temporal operators applied to an atom is called the *temporal reference* of that atom. A *temporal clause* is a formula of the form:

$$H \leftarrow B_1, \ldots, B_m$$

where $H, B_1, \ldots, B_m$ are temporal atoms, $m \geq 0$. If $m = 0$ then the clause is said to be a *unit temporal clause*. A *Cactus program* is a finite set of *temporal clauses*.

A *temporal goal clause* in Cactus is a formula of the form $\leftarrow A_1, \ldots, A_n$, with $n \geq 0$, where A_i, $i = 1, \ldots, n$ are temporal atoms.

Notice that the syntax of Cactus allows temporal operators to be applied on body atoms as well. For example the program defining the predicate `nat` in the introduction can be redefined as follows:

```
first nat(0).
next0 nat(Y) ←  nat(X), Y is 2*X+1.
next1 nat(Y) ←  next0 nat(X), Y is X+1.
```

The meaning of the last clause is that the value assigned to the right child of a node is the value of its left sibling plus 1.

Moreover, notice that it is not necessary for all predicates in a program to have clauses for the same number of $\texttt{next}_i$ operators.

3 Cactus Applications

In this section we present various applications showing the expressive power of branching time logic programming.

3.1 Expressing non-deterministic behaviour

Consider the non-deterministic finite automaton shown in figure 3 (similar to the one in [17], page 55) which accepts the regular language $L = (01 \cup 010)^*$. We can describe the behaviour of this automaton in Cactus with the following program:

```
first state(q0).
next0 state(q1)  ←  state(q0).
next1 state(q0)  ←  state(q1).
next1 state(q2)  ←  state(q1).
next0 state(q0)  ←  state(q2).
```

Notice that in this automaton `q0` is both the initial and the final state. Posing the goal clause:

$$\leftarrow \texttt{first next}_0\ \texttt{next}_1\ \texttt{next}_0\ \texttt{state(q0)}.$$

will return the answer `yes` which indicates that the string 010 belongs to the language L.

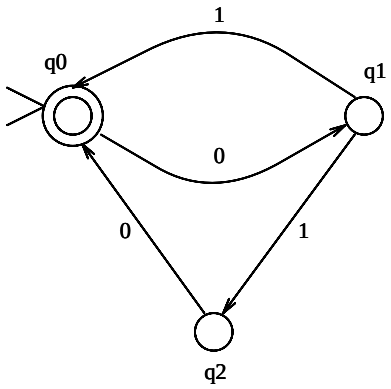

Figure 3: A non-deterministic finite automaton

The proof procedure of Cactus will be formally described in section 5. Intuitively, by using the last program clause, the above query will be transformed to the query:

$$\leftarrow \texttt{first next}_0\ \texttt{next}_1\ \texttt{state(q2)}.$$

Then, by using the fourth clause we get:

$$\leftarrow \texttt{first next}_0\ \texttt{state(q1)}.$$

which using the second program clause results in:

$$\leftarrow \texttt{first state(q0)}.$$

Finally, by resolving this with the first program clause we get an empty goal clause. Thus the query is a logical consequence of the program.

3.2 Generating sequences

One can write a simple Cactus program for producing the set of all binary sequences. The set of such sequences may be thought of as a tree, which can be described by the following program[1]:

$$\begin{array}{lll} \texttt{first binseq([\,])}. & & \\ \texttt{next}_0\ \texttt{binseq(W)} & \leftarrow & \texttt{binseq(X), append(X, [0], W)}. \\ \texttt{next}_1\ \texttt{binseq(W)} & \leftarrow & \texttt{binseq(X), append(X, [1], W)}. \end{array}$$

The goal clause:

$$\leftarrow \texttt{first next}_0\ \texttt{next}_1\ \texttt{next}_0\ \texttt{next}_1\ \texttt{binseq(S)}.$$

will produce the list `[0,1,0,1]` as a result, while the goal clause

$$\leftarrow \texttt{binseq(S)}.$$

will trigger an infinite computation which will generate all possible sequences (such goal clauses are called *open-ended* and are discussed in section 5.1).

[1]The computationally expensive calls to the well known list concatenation predicate append in the definition of binseq, can be avoided by using difference lists instead of classical ones. Notice that the definition of append is the classical one and is time independent.

One can combine the program `binseq` with the program for the nondeterministic automaton given in subsection 3.1. In this way we can produce the language recognized by the automaton. More specifically, the goal clause:

```
← state(q0), binseq(S).
```

produces the infinite set of all the binary sequences recognized by the automaton. The above goal clause (assuming a left to right computation rule) is not the classical generate-and-test procedure (not all binary sequences are generated but only those for which the automaton reaches the final state `q0`). Each successful evaluation of the goal `state(q0)` conducts the corresponding evaluation of `binseq`.

It is worthwhile noting here that in order to generate another language (on the same alphabet), one only needs to change the definition of the automaton and not the definition of `binseq`. This is due to the fact that the predicates `state` and `binseq` are completely independent (that is, `state` does not use `binseq` in its definition and vice-versa).

In ordinary logic programming (e.g. in Prolog), one could write the following program to solve the same problem:

```
first_state(q0).
next_state(q0, 0, q1).
next_state(q1, 1, q2).
next_state(q1, 1, q0).
next_state(q2, 0, q0).
state(First, [ ]) ← first_state(First).
state(New, [Move|Moves]) ← next_state(New, Move, Old),
                           state(Old, Moves).
```

The goal clause:

```
← state(q0, S).
```

corresponds to the goal clause

```
← state(q0), binseq(S).
```

in the Cactus program.

Notice here that the predicate `state` in the Prolog program depends on the predicates `first_state` and `next_state` which define the automaton. Therefore in this program the generation of the sequences is not independent of the specific automaton as it is the case in the Cactus program.

Another way of coding the same problem in Prolog, would be to add the notion of *sequence* as an extra argument to the `state` predicate. The corresponding Prolog program is shown below:

```
state(q0, [ ]).
state(q1, W) ← state(q0, X), append(X, [0], W).
state(q2, W) ← state(q1, X), append(X, [1], W).
state(q0, W) ← state(q1, X), append(X, [1], W).
state(q0, W) ← state(q2, X), append(X, [0], W).
```

Given the goal clause:

```
← state(q0, S).
```

Prolog's underlying execution engine would generate the sequences recognized by the automaton. Again, the Cactus version appears to be more concise because it separates the state transition process from the language generation one, while in the Prolog version these two processes are intermixed.

3.3 Representing trees

Branching time logic programming can be used as a tool for representing and manipulating trees. A tree can be represented in Cactus as a set of temporal unit clauses. The structure of the tree is expressed through the temporal references of the unit clauses. Moreover, the well known tree manipulation algorithms are easily expressed through Cactus programs. For example, consider the binary tree of figure 4.

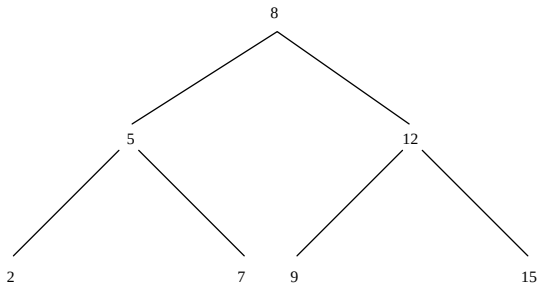

Figure 4: An (ordered) binary tree containing numeric data

A possible representation of the information included in this tree is given by the following set of Cactus unit clauses:

```
first node(8).
first next0 node(5).
first next1 node(12).
first next0 next0 leaf(2).
first next0 next1 leaf(7).
first next1 next0 leaf(9).
first next1 next1 leaf(15).
```

The above representation distinguishes between the inner nodes and the leafs of the tree by using two different predicate names, that is `node` and `leaf`.

3.4 Flattening trees

Using the representation of the last section, we can define a predicate `preorder` which collects the values of the tree nodes into a list. This definition corresponds to the left-to-right preorder traversal of the tree.

```
preorder([X]) ← leaf(X).
preorder([X|L]) ← node(X),
          next0 preorder(L1),
          next1 preorder(L2),
          append(L1,L2,L).
```

By posing the goal clause

$$\leftarrow \texttt{first preorder(L)}.$$

we will obtain the list `[8,5,2,7,12,9,15]` which corresponds to the preorder traversal of the tree shown in figure 4.

Notice that by posing the goal clause

$$\leftarrow \texttt{first next}_0\ \texttt{preorder(L)}.$$

we will obtain the list `[5,2,7]` which corresponds to the preorder traversal of the left subtree of the tree under consideration.

Generalizing the discussion above, we can say that the temporal reference appearing in goal clauses in Cactus programs, specifies a substructure of the data on which a particular computation is to be performed. Alternatively, one can access results produced by subcomputations using the temporal references.

It is worth comparing the above program with the usual way of flattening binary trees in Prolog [18]. In general a binary tree is represented in Prolog using a compound term of the form `tree(Element,Left,Right)`. In particular, the tree of figure 4 is represented by the following term:

```
tree(8, tree(5, tree(2, void,void),
                tree(7, void,void)),
        tree(12, tree(9, void,void),
                tree(15, void,void)))
```

The corresponding Prolog procedure for the predicate `preorder` is the following:

```
preorder(void,[]).
preorder(tree(X,Left,Right),[X|L]) ←
        preorder(Left,L1),
        preorder(Right,L2),
        append(L1,L2,L).
```

In the Prolog program the whole tree structure has to be passed around during execution. This is not the case in the corresponding Cactus procedure, in which the tree is represented using unit temporal clauses. The temporal operators of these clauses reflect in a natural way the structure of the tree.

3.5 Searching trees

One can easily define in Cactus a procedure for searching a binary tree. As a first step we can define a predicate `descendant(X)`, which is true in a node t of the time tree if `X` is a value which exists in the subtree rooted by t:

```
descendant(X) ← leaf(X).
descendant(X) ← node(X).
descendant(X) ← node(Y),next0 descendant(X).
descendant(X) ← node(Y),next1 descendant(X).
```

Note that the role of the body atom `node(Y)` is just to ensure termination of the program since otherwise the last two clauses of the program would produce infinite search branches. A

more efficient definition of the predicate descendant in the case where the binary tree is ordered (binary search) is shown in the following program.

$$\begin{aligned}
&\texttt{descendant(X)} \leftarrow \texttt{leaf(X)}.\\
&\texttt{descendant(X)} \leftarrow \texttt{node(X)}.\\
&\texttt{descendant(X)} \leftarrow \texttt{node(Y)}, \texttt{X} < \texttt{Y}, \texttt{next}_0\ \texttt{descendant(X)}.\\
&\texttt{descendant(X)} \leftarrow \texttt{node(Y)}, \texttt{X} > \texttt{Y}, \texttt{next}_1\ \texttt{descendant(X)}.
\end{aligned}$$

Consider the tree of section 3.3. By posing the goal clause:

$$\leftarrow \texttt{first next}_0\ \texttt{descendant(7)}.$$

we will get the answer yes, because the value 7 is in a node which represents a moment in a future time of the time point corresponding to $\texttt{first next}_0$.

Using the definition of the predicate descendant we can define the predicate search which tests if a specific numeric value is in a node of the data tree. The clause:

$$\texttt{search(X)} \leftarrow \texttt{first descendant(X)}.$$

defines the predicate search.

3.6 Multiple trees

The representation proposed in section 3.3 can be easily extended to cover the case of multiple trees. Using the predicates node and leaf one can represent and use more than one tree in the same program by adding an extra argument which corresponds to the tree identity. For example, the facts for the tree of section 3.3 become:

$$\begin{aligned}
&\texttt{first node(tree1,8)}.\\
&\texttt{first next}_0\ \texttt{node(tree1,5)}.\\
&\texttt{first next}_1\ \texttt{node(tree1,12)}.\\
&\texttt{first next}_0\ \texttt{next}_0\ \texttt{leaf(tree1,2)}.\\
&\texttt{first next}_0\ \texttt{next}_1\ \texttt{leaf(tree1,7)}.\\
&\texttt{first next}_1\ \texttt{next}_0\ \texttt{leaf(tree1,9)}.\\
&\texttt{first next}_1\ \texttt{next}_1\ \texttt{leaf(tree1,15)}.
\end{aligned}$$

where by tree1 we denote the identity of the particular tree.

As a simple example that uses this extended notation, consider the following program which defines the notion of "leftmost leaf" of a given tree:

$$\begin{aligned}
&\texttt{leftmost(Id,X)} \leftarrow \texttt{leaf(Id,X)}.\\
&\texttt{leftmost(Id,X)} \leftarrow \texttt{next}_0\ \texttt{leftmost(Id,X)}.
\end{aligned}$$

Posing the goal clause:

$$\leftarrow \texttt{first leftmost(tree1,X)}.$$

the variable X is assigned the value of the leftmost leaf of tree1.

Note that again the meaning of the predicate leftmost is more general than one would expect at a first glance. Consider for example the goal clause

$$\leftarrow \texttt{first next}_1\ \texttt{leftmost(tree1,X)}.$$

In this case we get the leftmost leaf of the right subtree of tree1. In general, the temporal reference of the goal clause specifies the subtree of tree1 whose leftmost leaf is returned.

Notice that the tree manipulation examples of the previous sections can be modified accordingly to work for the case of multiple trees.

3.7 Modeling recursion using branching time

As we have seen in the programs of the previous sections, it is often possible to write Cactus procedures with fewer arguments than the corresponding Prolog ones. One interesting question that results from this remark is whether there exists a mechanical way of transforming a Prolog program into an equivalent Cactus program whose predicates have fewer arguments than the corresponding Prolog predicates. Taking into consideration the usual WAM-based implementation of Prolog, we believe that reducing the number of arguments of the program predicates might offer opportunities for more efficient implementations.

A similar idea [19, 20, 21, 22] has been successfully applied in the context of functional programming and has offered a means for implementing functional languages on dataflow architectures.

In the following, we consider the well-known Prolog program for computing Fibonacci numbers, and transform it into a Cactus program that performs the same task. The resulting Cactus program is not an intuitive one and it certainly does not suggest a useful programming methodology in branching time logic programming. However, we believe that a generalization and formalization of the particular transformation algorithm, may lead to a new implementation technique for logic programming (in analogy to the corresponding transformation and implementation technique in functional programming).

Consider the well known (but highly inefficient) way of computing the Fibonacci numbers in Prolog:

```
fib(1,0).
fib(1,1).
fib(F,N)  ←  N1 is N-1,N2 is N-2,
             fib(F1,N1),fib(F2,N2),
             F is F1+F2.
```

Given the above program, a goal clause of the form:

```
← fib(F,10).
```

will return the tenth Fibonacci number.

During the execution of the above goal, the value of the second argument of the predicate `fib` changes in a tree-like way (because of the recursive calls in the body of `fib`). We can define in Cactus a predicate `n` which models the change of the second argument of `fib` (notice that the "initial" value of this argument is `10`):

```
first n(10).
next_0 n(Y)  ←  n(N),Y is N-1.
next_1 n(Y)  ←  n(N),Y is N-2.
```

At the beginning of time, `n` is true of the value `10`. In general, if `n` is true of the value `N` at some node in the tree, then at the left child of that node `n` will be true of `N-1` while at the right child of the node it will be true of `N-2`.

Using the predicate `n`, we can rewrite the Fibonacci program in a purely branching time way:

```
fib(1)  ←  n(0).
fib(1)  ←  n(1).
fib(F)  ←  next_0 fib(F1),next_1 fib(F2),F is F1+F2.
```

Given the above definitions for the predicates `fib` and `n`, a goal clause of the form:

$$\leftarrow \texttt{first fib(F)}.$$

will return the tenth Fibonacci number.

Notice that, the Cactus program has the number `10` "wired" into the code for `n`. This is due to the fact that the transformation under consideration compiles the source Prolog program *together* with a specific user query into a single Cactus program. This is exactly how the corresponding transformation is performed in functional programming [19, 20, 21].

The above program is definitely a less intuitive one than the original Prolog program (although this is not important as the Cactus program is meant to be the output of a transformation algorithm). Concerning the operational behaviour of the Cactus program, we can see that while the tree of `n` is constructed in a top-to-bottom way, the tree of `fib` is being built bottom-up. Actually, the first and the second arguments of `fib` in the original Prolog program would vary in exactly this way during execution, a fact that is not however expressed explicitly in the Prolog program. In other words, the Cactus program has a more operational flavour because it expresses explicitly the recursion mechanism for computing the final result.

As we realize from the above example, it is possible for certain logic programs to be transformed into branching time logic programs in which all the user defined predicates are unary. An interesting question for further investigation is whether the technique we outlined in the above example, is applicable to wide classes of logic programs. Some encouraging results in this direction are reported in [23].

4 The branching time logic of Cactus

Branching time logic programming, is based on a relatively simple *branching time logic* (BTL). In BTL, time varies over a tree-like structure. The set of moments in time can be modeled by the set $List(\mathcal{N})$ of lists of natural numbers $\mathcal{N}$. Thus, each node may have a countably infinite number of branches (`next` operators). The empty list [ ] corresponds to the beginning of time and the list $[i|t]$ (that is, the list with head $i \in \mathcal{N}$, and tail t) corresponds to the i-th child of the moment identified by the list t. BTL uses the temporal operators `first` and $\texttt{next}_i$, $i \in \mathcal{N}$. The operator `first` is used to express the first moment in time, while $\texttt{next}_i$ refers to the i-th child of the current moment in time.

The syntax of BTL extends the syntax of first-order logic with two formation rules:

- if A is a formula then so is `first` A, and
- if A is a formula then so is $\texttt{next}_i$ A.

BTL is a relatively simple branching time logic. For more on branching time logics one can refer to [11, 24, 25].

4.1 Semantics of BTL formulas

The semantics of temporal formulas of BTL are given using the notion of *branching temporal interpretation*. Branching temporal interpretations extend the temporal interpretations of the linear time logic of Chronolog [3].

Definition 4.1. A *branching temporal interpretation* or simply a *temporal interpretation* I of the temporal logic BTL comprises a non-empty set D, called the domain of the interpretation,

over which the variables range, together with an element of D for each variable; for each n-ary function symbol, an element of $[D^n \rightarrow D]$; and for each n-ary predicate symbol, an element of $[List(\mathcal{N}) \rightarrow 2^{D^n}]$.

In the following definition, the satisfaction relation $\models$ is defined in terms of temporal interpretations. $\models_{I,t} A$ denotes that the formula A is true at the moment t in the temporal interpretation I.

Definition 4.2. The semantics of the elements of the temporal logic BTL are given inductively as follows:

1. If $\mathtt{f}(e_0, \ldots, e_{n-1})$ is a term, then $I(\mathtt{f}(e_0, \ldots, e_{n-1})) = I(\mathtt{f})(I(e_0), \ldots, I(e_{n-1}))$.
2. For any n-ary predicate symbol $\mathtt{p}$ and terms $e_0, \ldots, e_{n-1}$,
   $\models_{I,t} \mathtt{p}(e_0, \ldots, e_{n-1})$ *iff* $\langle I(e_0), \ldots, I(e_{n-1}) \rangle \in I(\mathtt{p})(t)$
3. $\models_{I,t} \neg A$ *iff it is not the case that* $\models_{I,t} A$
4. $\models_{I,t} A \wedge B$ *iff* $\models_{I,t} A$ *and* $\models_{I,t} B$
5. $\models_{I,t} A \vee B$ *iff* $\models_{I,t} A$ *or* $\models_{I,t} B$
6. $\models_{I,t} (\forall \mathtt{x}) A$ *iff* $\models_{I[d/\mathtt{x}],t} A$ *for all* $d \in D$ *where the interpretation* $I[d/\mathtt{x}]$ *is the same as* I *except that the variable* $\mathtt{x}$ *is assigned the value* d.
7. $\models_{I,t} \mathtt{first}\ A$ *iff* $\models_{I,[\,]} A$
8. $\models_{I,t} \mathtt{next}_i\ A$ *iff* $\models_{I,[i|t]} A$

The semantics of formulas involving the symbols $\leftarrow$, $\rightarrow$, and $\leftrightarrow$ are defined in the usual way with respect to the semantics of $\wedge$, $\vee$ and $\neg$.

If a formula A is true in a temporal interpretation I at all moments in time, it is said to be true in I (we write $\models_I A$) and I is called a *model* of A.

Clearly, Cactus clauses form a subset of BTL formulas. It can be shown that the usual minimal model and fixpoint semantics that apply to logic programs, can be extended to apply to Cactus programs. However, such an investigation is outside the scope of this paper and is reported in a forthcoming one concerning the declarative and procedural semantics of Cactus.

4.2 Tautologies

In this section we present some useful tautologies that hold for the logic BTL, many of which are similar to those adopted for the case of linear time logics [3]. In the following, the symbol ∇ stands for either of $\mathtt{first}$ and $\mathtt{next}_i$.

Temporal operator cancellation rules: The intuition behind these rules is that the operator $\mathtt{first}$ cancels the effect of any other "outer" operator. Formally:

$$\nabla(\mathtt{first}\ A) \leftrightarrow (\mathtt{first}\ A) \tag{1}$$

Notice that this is actually a family of rules, one for each different instantiation of the operator ∇.

The above law is not difficult to prove by considering the two different cases for ∇. We show it for the case of $\texttt{next}_i$:

$$\begin{array}{ll} \models_{I,t} \texttt{next}_i\ (\texttt{first}\ A) & \textit{iff} \\ \models_{I,[i|t]}\ (\texttt{first}\ A) & \textit{iff} \\ \models_{I,[\,]}\ A & \textit{iff} \\ \models_{I,t} \texttt{first}\ A & \end{array}$$

The case for `first` is shown in a similar way.

Temporal operator distribution rules: These rules express the fact that the branching time operators of BTL distribute over the classical operators $\neg$, $\wedge$ and $\vee$. Formally:

$$\nabla(\neg A) \leftrightarrow \neg(\nabla A) \tag{2}$$

$$\nabla(A \wedge B) \leftrightarrow (\nabla A) \wedge (\nabla B) \tag{3}$$

$$\nabla(A \vee B) \leftrightarrow (\nabla A) \vee (\nabla B) \tag{4}$$

Again, each of the above rules actually represents a family of rules depending on the instantiation of ∇.

The above laws are not difficult to prove by considering all the possible alternatives. We show the conjunction rule in the case of $\texttt{next}_i$:

$$\begin{array}{ll} \models_{I,t} \texttt{next}_i\ (A \wedge B) & \textit{iff} \\ \models_{I,[i|t]}\ (A \wedge B) & \textit{iff} \\ (\models_{I,[i|t]}\ A)\ \textit{and}\ (\models_{I,[i|t]}\ B) & \textit{iff} \\ (\models_{I,t} \texttt{next}_i\ A)\ \textit{and}\ (\models_{I,t} \texttt{next}_i\ B) & \textit{iff} \\ \models_{I,t}\ (\texttt{next}_i\ A) \wedge (\texttt{next}_i\ B) & \end{array}$$

The other cases are shown in a similar way.

From the temporal operator distribution rules we see that if we apply a temporal operator to a whole program clause, the operator can be pushed inside until we reach atomic formulas. This is why we did not consider applications of temporal operators to whole program clauses.

Rigidness of variables: The following rule states that a temporal operator ∇ can "pass inside" $\forall$:

$$\nabla(\forall X)(A) \leftrightarrow (\forall X)(\nabla A) \tag{5}$$

The above rule holds because variables represent data-values which are independent of time (i.e. they are *rigid*).

Again, the above tautology is not difficult to prove by considering the two different cases for ∇. We show the tautology for the case of $\texttt{next}_i$:

$$\begin{array}{ll} \models_{I,t} \texttt{next}_i\ (\forall \texttt{x}\ A) & \textit{iff} \\ \models_{I,[i|t]}\ (\forall \texttt{x}\ A) & \textit{iff} \\ \models_{I[d/\texttt{x}],[i|t]}\ A, \textit{for all}\ d \in D & \textit{iff} \\ \models_{I[d/\texttt{x}],t}\ (\texttt{next}_i\ A), \textit{for all}\ d \in D & \textit{iff} \\ \models_{I,t}\ \forall \texttt{x}(\texttt{next}_i\ A) & \end{array}$$

The case for `first` is shown in a similar way.

We should also note that the formulas $\texttt{next}_i\ \texttt{next}_j\ A$ and $\texttt{next}_j\ \texttt{next}_i\ A$ are not equivalent in general when $i \neq j$.

5 A proof procedure for Cactus programs

Cactus programs are executed using a resolution-type proof procedure called *BSLD-resolution* (**B**ranching-time **SLD**-resolution). BSLD-resolution is a refutation procedure which extends SLD-resolution [16], and is similar to TiSLD-resolution [26], the proof procedure for Chronolog programs. The following definitions are necessary in order to introduce BSLD-resolution.

Definition 5.1. A *canonical temporal reference* is a temporal reference of the form $\texttt{first next}_{i_1} \cdots \texttt{next}_{i_n}$, where $i_1, \ldots, i_n \in \mathcal{N}$ and $n \geq 0$. A *canonical temporal atom* is a temporal atom whose temporal reference is canonical. A *canonical temporal clause* is a temporal clause whose temporal atoms are canonical.

It can be shown that every Cactus program can be transformed into a (possibly infinite[2]) set of canonical temporal clauses, which has the same set of temporal models as the initial program (see Lemma 5.1 below). Therefore, the transformation preserves the set of canonical atoms that are logical consequences of the program. The construction of this set of canonical temporal clauses is formalized by the following definitions:

Definition 5.2. The *normal form of a temporal reference* R is the temporal reference R' obtained by removing all temporal operators preceding the rightmost occurrence of the operator `first` (if any) in R.

For example, the temporal reference $\texttt{first next}_0$ is the normal form of the reference $\texttt{next}_1$ $\texttt{first next}_0$ $\texttt{first next}_0$. In the following, we assume that temporal references of all atoms in a program are in normal form.

Definition 5.3. A *canonical temporal instance* of a temporal clause C is a canonical temporal clause C' which can be obtained by applying the same canonical temporal reference to all non-canonical atoms of the normal form of C.

Intuitively, a canonical temporal instance of a clause is an instance in time of the corresponding temporal clause. Notice that the notion of canonical temporal atoms/clauses/instances have been initially introduced in the context of the linear time temporal logic programming language Chronolog [3, 6].

Notice that a canonical temporal instance of a clause can also be obtained by the following procedure: apply a canonical temporal reference to the clause itself; use the temporal operator distribution axioms to distribute the temporal reference so as to be applied to each individual temporal atom of the clause; finally eliminate any superfluous operator by applying the cancellation rules.

Example 5.1. Consider the following Cactus program:

```
first p(0).
next_0 p(s(X)) ← p(X).
next_1 p(s(s(X))) ← p(X).
```

The set of canonical temporal instances corresponding to the above program clauses is as follows:

The clause:

[2]Notice that in a real implementation, only the canonical instances of a clause that are actually needed are generated and not the possibly infinite set.

`first p(0).`
is the only canonical temporal clause corresponding to the first program clause.

The set of clauses:

$$\{\texttt{first next}_{i_1}\cdots\texttt{next}_{i_n}\ \texttt{next}_0\ \texttt{p(s(X))}\ \leftarrow\ \texttt{first next}_{i_1}\cdots\texttt{next}_{i_n}\ \texttt{p(X)}\ \mid\ i_1,\ldots,i_n\in\mathcal{N}, n\geq 0\}$$

corresponds to the second program clause. Finally the set of clauses:

$$\{\texttt{first next}_{i_1}\cdots\texttt{next}_{i_n}\ \texttt{next}_1\ \texttt{p(s(s(X)))}\ \leftarrow\ \texttt{first next}_{i_1}\cdots\texttt{next}_{i_n}\ \texttt{p(X)}\ \mid\ i_1,\ldots,i_n\in\mathcal{N}, n\geq 0\}$$

corresponds to the third program clause.

The notion of canonical instance of a clause is very important since the truth value of a given clause in a temporal interpretation, can be expressed in terms of the values of its canonical instances, as the following lemma shows:

Lemma 5.1 *Let C be a clause and I a temporal interpretation of BTL. $\models_I C$ if and only if $\models_I C_t$ for all canonical instances C_t of C.*

Proof: Let C_t be a canonical instance of C. We will prove that $\models_{I,t'} C_t$ for all t'. C_t has been obtained by applying a sequence $\texttt{first next}_{i_1}\cdots\texttt{next}_{i_n}$, with $n\geq 0$, to C. By applying definition 4.2 to $\models_{I,t'} \texttt{first next}_{i_1}\cdots\texttt{next}_{i_n}C$, we can easily see that this is equivalent to $\models_{I,[i_n,\ldots,i_1]} C$, which is true because $\models_I C$.

The converse is easily proved by contradiction since if we suppose that all canonical instances of C are valid but C is invalid then there must exist an interpretation I and a time point t in which C is false. By applying definition 4.2 (7),(8) in reverse order, we get a canonical instance of C which is false in I. □

BSLD-resolution is applied to canonical instances of program clauses and canonical goal clauses.

Definition 5.4. Two canonical temporal atoms A_1, A_2 are *unifiable* if they have the same temporal references and their classical atoms are unifiable in the classical sense. The *most general unifier (mgu)* of A_1, A_2 is the most general unifier of their classical atoms.

Definition 5.5. Let P be a program in Cactus and G be a canonical temporal goal clause. A *BSLD-derivation* of $P\cup\{G\}$ consists of a (possibly infinite) sequence of canonical temporal goals $G_0 = G, G_1, \ldots, G_n, \ldots$ a sequence $C_1,\ldots,C_n,\ldots$ of canonical instances of clauses of P (called the *input clauses*), and a sequence $\theta_1,\ldots,\theta_n\ldots$ of most general unifiers such that for all i, the goal G_{i+1} is obtained from the goal:

$G_i = \ \leftarrow A_1,\ldots,A_{m-1},A_m,A_{m+1},\ldots,A_p$

as follows:

1. A_m is a canonical temporal atom in G_i (called the *selected* atom)
2. $H\leftarrow B_1,\ldots,B_r$ is the input clause C_{i+1} (standardized apart from G_i),
3. $\theta_{i+1} = mgu(A_m, H)$
4. G_{i+1} is the goal: $G_{i+1} = \ \leftarrow (A_1,\ldots,A_{m-1},B_1,\ldots,B_r,A_{m+1},\ldots,A_p)\theta_{i+1}$

Definition 5.6. A *BSLD-refutation* of $P \cup \{G\}$ is a finite BSLD-derivation of $P \cup \{G\}$ which has the empty goal clause $\Box$ as the last clause of the derivation.

Definition 5.7. Let P be a program in Cactus and G be a canonical temporal goal. A *computed answer* for $P \cup \{G\}$ is the substitution obtained by restricting the composition $\theta_1\theta_2 \ldots \theta_n$ to the variables of G, where $\theta_1, \theta_2, \ldots, \theta_n$, is the sequence of the most general unifiers used in a BSLD-refutation of $P \cup \{G\}$.

Let us now see an example of the application of BSLD-resolution.

Example 5.2. Consider the program defining the predicate `nat` presented in the introduction:

```
(1)   first nat(0).
(2)   next_0 nat(Y)  ←  nat(X), Y is 2*X+1.
(3)   next_1 nat(Y)  ←  nat(X), Y is 2*X+2.
```

A BSLD-refutation of the canonical temporal goal:

$\leftarrow$ `first next_0 next_1 nat(N)`

is given below (in every derivation step the selected temporal atom is the underlined one):

$\leftarrow$ `first next_0 next_1 nat(N)`

using the following canonical instance of clause (3):
`first next_0 next_1 nat(Y) ← first next_0 nat(X), first next_0 (Y is 2*X+2).`

$\leftarrow$ `first next_0 nat(X), first next_0 (N is 2*X+2)`

using the following canonical instance of clause (2):
`first next_0 nat(Y) ← first nat(X), first (Y is 2*X+1).`

$\leftarrow$ `first nat(X_1), first (X is 2*X_1+1), first next_0 (N is 2*X+2)`

(`X_1 = 0`) using clause (1):

$\leftarrow$ `first (X is 2*0+1), first next_0 (N is 2*X+2)`

(`X = 1`) evaluation of the built-in predicate `is`[3]

$\leftarrow$ `first next_0 (N is 2*1+2)`

(`N = 4`) evaluation of the built-in predicate `is`

$\Box$

BSLD-resolution is a sound and complete proof procedure. In the following, we present the basic soundness and completeness theorems of BSLD-resolution, and give a sketch of their proof. The full proofs are outside the scope of this paper and are reported in a forthcoming one concerning the declarative and procedural semantics of Cactus. In order to present the soundness and completeness theorems we need the notion of *correct answer*.

[3] In order to use Cactus in practical applications it is useful to introduce certain built-in predicates which behave as in classical Prolog. The built-in procedure `is` is independent of time (rigid).

Definition 5.8. Let P be a Cactus program and $G = \leftarrow A_1, \ldots, A_n$ be a canonical temporal goal. A substitution θ is said to be a *correct answer* for $P \cup \{G\}$ iff $P \models \forall(A_1 \wedge \ldots \wedge A_n)\theta$.

Theorem 5.1 *(**Soundness of BSLD-resolution**) Let P be a program in Cactus and G a canonical temporal goal. Then every computed answer for $P \cup \{G\}$ is a correct answer for $P \cup \{G\}$.*

Proof: (Sketch) The soundness theorem can be easily proved by induction on the length of the refutation. The proof is almost identical to the proof of the soundness theorem of SLD-resolution (see [16], page 43). The only difference here is that instead of program clauses we use canonical instances of program clauses (as it is the case for the TiSLD-resolution [26], the proof procedure of Chronolog). □

Theorem 5.2 *(**Completeness of BSLD-resolution**) Let P be a program in Cactus and G a canonical temporal goal. Then for every correct answer θ for $P \cup \{G\}$, there exists a computed answer σ for $P \cup \{G\}$ and a substitution ξ such that $\theta = \sigma\xi$.*

Proof: (Sketch) The proof of the completeness theorem is similar to the proof of the corresponding completeness theorem of SLD-resolution (see [16], pages 47-49). For this proof we need a *lifting lemma* and an *mgu lemma*. Again the proofs of these lemmas as well as the proof of the completeness theorem differ from the corresponding proofs for SLD-resolution in that canonical instances of program clauses instead of program clauses, are used. □

As it is the case for SLD-resolution, BSLD-resolution can easily be shown to be independent of the selection of the specific *computation rule* (i.e. the rule which is responsible for the selection of the specific atom in the goal clause which will be used in a derivation step).

5.1 Open-ended goal clauses

When some of the temporal atoms included in a goal clause are not canonical, we say that we have an *open-ended goal clause* (e.g. the goal clauses in section 3.2). The idea behind open-ended goal clauses was first introduced in the context of Chronolog [26]. An open-ended goal clause G represents the infinite set of all canonical queries corresponding to G. Open-ended goal clauses are used to imitate non-terminating computations. An implementation strategy for executing an open-ended goal clause is by enumerating and evaluating (one by one) the set of all canonical instances of the goal clause.

The above enumeration strategy is the standard technique which has been adopted for the Chronolog family of languages. The origins of this approach date back to the functional-dataflow language *Lucid* [27], in which the basic data structures are *streams*. The Chronolog language, designed by the same research group, inherited the operators, the stream-oriented nature, and some of the implementation decisions of Lucid. A similar idea as the enumeration strategy described above, has been used in Lucid implementations. Notice that, in logic programming terms, this strategy corresponds to backtracking (with respect to time).

The enumeration idea provides an acceptable proof procedure for certain types of applications. However, there exist cases in which enumeration proves impractical and more powerful techniques are required. The authors have undertaken one such approach [28] which works directly on program clauses (and not on canonical instances). The basic idea behind this extended resolution-based proof procedure, is to extend the notion of unifier to take into account the implicit time parameter of the language (through the manipulation of temporal references).

Alternatively, constraint solving techniques which have been used in other temporal logic programming languages (see for example [8, 29]), may also be appropriate in our case.

6 Possible Extensions

In the branching time logic of Ben-Ari, Pnueli and Manna [11], richer temporal operators are defined[4]:

$\bigcirc A$: holds at t iff A is true at every immediate successor of t.

$\Diamond_G A$: holds at t iff for all paths departing from t there is some time point (node) in which A is true.

$\Diamond_F A$: holds at t iff there is some node in the subtree rooted from t at which A is true.

$\Box_G A$: holds at t iff A is true at all time points (nodes) of the subtree rooted at t (including t).

$\Box_F A$: holds at t iff there is a path departing from t such that A is true at all time points (nodes) of this path.

In the following, we show how one can imitate the use of the above operators through Cactus programs.

We suppose that we have already defined the predicate `p` in the program. In the following we restrict attention to a branching time logic with two `next` operators, namely $\texttt{next}_0$ and $\texttt{next}_1$. Then the Cactus program corresponding to $\bigcirc$ $\texttt{p}(\overline{\texttt{X}})$ is[5]:

$$\begin{aligned}\texttt{circle\_p}(\overline{\texttt{X}}) \;\leftarrow\; & \texttt{next}_0\ \texttt{p}(\overline{\texttt{X}}),\\ & \texttt{next}_1\ \texttt{p}(\overline{\texttt{X}}).\end{aligned}$$

The Cactus program corresponding to $\Diamond_G$ $\texttt{p}(\overline{\texttt{X}})$ is:

$$\begin{aligned}\texttt{diamondG\_p}(\overline{\texttt{X}}) \;&\leftarrow\; \texttt{p}(\overline{\texttt{X}}).\\ \texttt{diamondG\_p}(\overline{\texttt{X}}) \;&\leftarrow\; \texttt{next}_0\ \texttt{diamondG\_p}(\overline{\texttt{X}}),\\ &\qquad \texttt{next}_1\ \texttt{diamondG\_p}(\overline{\texttt{X}}).\end{aligned}$$

The Cactus program corresponding to $\Diamond_F$ $\texttt{p}(\overline{\texttt{X}})$ is:

$$\begin{aligned}\texttt{diamondF\_p}(\overline{\texttt{X}}) \;&\leftarrow\; \texttt{p}(\overline{\texttt{X}}).\\ \texttt{diamondF\_p}(\overline{\texttt{X}}) \;&\leftarrow\; \texttt{next}_0\ \texttt{diamondF\_p}(\overline{\texttt{X}}).\\ \texttt{diamondF\_p}(\overline{\texttt{X}}) \;&\leftarrow\; \texttt{next}_1\ \texttt{diamondF\_p}(\overline{\texttt{X}}).\end{aligned}$$

The Cactus program corresponding to $\Box_G$ $\texttt{p}(\overline{\texttt{X}})$ is:

$$\begin{aligned}\texttt{boxG\_p}(\overline{\texttt{X}}) \;\leftarrow\; & \texttt{p}(\overline{\texttt{X}}),\\ & \texttt{next}_0\ \texttt{boxG\_p}(\overline{\texttt{X}}),\\ & \texttt{next}_1\ \texttt{boxG\_p}(\overline{\texttt{X}}).\end{aligned}$$

Finally, the Cactus program corresponding to $\Box_F$ $\texttt{p}(\overline{\texttt{X}})$ is:

$$\begin{aligned}\texttt{boxF\_p}(\overline{\texttt{X}}) \;&\leftarrow\; \texttt{p}(\overline{\texttt{X}}),\\ &\qquad \texttt{next}_0\ \texttt{boxF\_p}(\overline{\texttt{X}}).\\ \texttt{boxF\_p}(\overline{\texttt{X}}) \;&\leftarrow\; \texttt{p}(\overline{\texttt{X}}),\\ &\qquad \texttt{next}_1\ \texttt{boxF\_p}(\overline{\texttt{X}}).\end{aligned}$$

[4] The operators $\Box_G$, $\Box_F$, $\bigcirc$, $\Diamond_G$ and $\Diamond_F$ are represented in [11] as $\forall G$, $\forall F$, $\forall X$, $\exists G$ and $\exists F$ respectively.

[5] By $\overline{\texttt{X}}$ we denote a tuple of variables.

It is easy to see that although the programs defining the predicates `boxG_p` and `boxF_p` are correct definitions of the corresponding temporal operators, in practice they do not work (the only cases in which each one of them will succeed, require `p` to be true in an infinite number of time points, but then the computation never terminates). However, many times in practice we are dealing with finite trees. In such cases, the above definitions can be adapted to give the desired results. More specifically, we have to modify the definitions so as that they recognize the end points of the tree for `p`. For example the following program redefines the predicate `boxG_p`.

$$\begin{aligned}
\texttt{boxG_p}(\overline{\texttt{X}}) &\leftarrow \texttt{p}(\overline{\texttt{X}}), \texttt{end_point_p}.\\
\texttt{boxG_p}(\overline{\texttt{X}}) &\leftarrow \texttt{p}(\overline{\texttt{X}}),\\
&\qquad \texttt{next}_0\ \texttt{boxG_p}(\overline{\texttt{X}}),\\
&\qquad \texttt{next}_1\ \texttt{boxG_p}(\overline{\texttt{X}}).
\end{aligned}$$

Here we suppose that the program has been supplied with a set of unit temporal clauses defining the predicate `end_point_p` of the form:

Tref `end_point_p`.

where *Tref* corresponds to the time bounds of the temporal tree for predicate `p` (i.e. the last time points in each branch of the time tree).

A more serious restriction of the above definitions of these temporal operators is that we have to supply our program with new definitions of the above form for each program predicate for which we want to use these operators. These restrictions indicate the need to extend our underlying branching time logic so as that it supports a richer set of temporal operators.

Clearly, Cactus is a superset of ordinary logic programming. A significant additional characteristic of Cactus is its ability to handle "hard-wired" data structures while in the corresponding Prolog programs one usually has to pass them around as extra arguments. However, (as one of the reviewers pointed out) it is not always possible to express all data used by an application in a static way. These dynamic data can also be handled in Cactus as in usual Prolog programs. A really useful extension of Cactus (that also the same reviewer suggested) is the quantification over the subscript of the $\texttt{next}_i$ operator. This would allow many useful programs to be coded in a compact way.

Another interesting extension to Cactus is the addition of *multiple time dimensions*. Multidimensional programming is a promising new area of research [30, 31] but had mainly been explored so far for linear time dimensions. Recent work however [22, 20] has demonstrated that multidimensional branching-time languages provide an effective means for implementing higher-order functional languages. We believe than an investigation of *multidimensional Cactus* could probably reveal further applications of the branching time logic programming paradigm.

Finally, an interesting direction for future work is the addition of negation in Cactus. Promising results in this direction (for the linear logic language Chronolog) are reported in [32].

7 Conclusions

Temporal programming languages, either functional [27] or logic [1, 3, 33], have been widely used as a means for describing dynamic systems. However, most temporal languages use a linear notion of time a fact that makes them unsuitable for certain types of applications.

In this paper we have introduced the branching time logic programming language **Cactus** which is based on a tree-like notion of time. We have demonstrated that Cactus is capable of expressing various tree-related problems in a natural way. Moreover, we have shown that Cactus retains the clarity of logic programming and has a simple procedural interpretation.

It is easy to see that Cactus is a superset of both Prolog and Chronolog. In particular, Chronolog can be obtained by restricting Cactus to have only one `next` operator.

The language has been implemented in the form of a simple meta-interpreter in which the programs of the paper have been tested. Future directions include efficient implementations (see [28] for an approach in this direction), extensions to the expressive power of the language, transformation techniques and development of applications of considerable size and complexity.

The branching time concept has been particularly successful in the functional programming domain [19, 20, 21, 34] and we believe that a similar potential exists for the area of logic programming.

Acknowledgements: We would like to thank the anonymous referees for the detailed comments and suggestions which have helped us to improve our paper. Moreover, we would also like to thank Dr. Costas Koutras for many useful discussions and remarks.

References

[1] M. A. Orgun and W. Ma. An overview of temporal and modal logic programming. In *Proc. of the First International Conference on Temporal Logics (ICTL'94)*, Lecture Notes in Computer Science (LNCS) 827, pages 445–479. Springer-Verlag, 1994.

[2] M. Fisher and R. Owens. An introduction to executable modal and temporal logics. Lecture Notes in Artificial Intelligence (LNAI) 897. Springer-Verlag, February 1995.

[3] M. A. Orgun. *Intensional logic programming.* PhD thesis, Dept. of Computer Science, University of Victoria, Canada, December 1991.

[4] W. W. Wadge. Tense logic programming: A respectable alternative. In *Proc. of the 1988 International Symposium on Lucid and Intensional Programming*, pages 26–32, 1988.

[5] T. Hrycej. A temporal extension of Prolog. *The Journal of Logic Programming*, 15:113–145, 1993.

[6] M. A. Orgun, W. W. Wadge, and W. Du. Chronolog($\mathcal{Z}$): Linear-time logic programming. In O. Abou-Rabia, C. K. Chang, and W. W. Koczkodaj, editors, *Proc. of the Fifth International Conference on Computing and Information*, pages 545–549. IEEE Computer Society Press, 1993.

[7] M. Baudinet. A simple proof of the completeness of temporal logic programming. In L. Farinas del Cerro and M. Penttonen, editors, *Intensional Logics for Programming*, pages 51–83. Oxford University Press, 1993.

[8] C. Brzoska. Temporal logic programming and its relation to constraint logic programming. In *Proc. of the Logic Programming Symposium*, pages 661–677. MIT Press, 1991.

[9] C. Brzoska. Temporal logic programming with bounded universal modality goals. In D. S. Warren, editor, *Proc. of the Tenth International Conference on Logic Programming*, pages 239–256. MIT Press, 1993.

[10] M. Gergatsoulis, P. Rondogiannis, and T. Panayiotopoulos. Disjunctive Chronolog. In M. Chacravarty, Y. Guo, and T. Ida, editors, *Proceedings of the JICSLP'96 Post-Conference Workshop "Multi-Paradigm Logic Programming"*, pages 129–136, Bonn, 5-6 Sept. 1996.

[11] M. Ben-Ari, A. Pnueli, and Z. Manna. The temporal logic of branching time. *Informatica*, pages 207–226, 1983.

[12] M. Wooldridge and M. Fisher. A first-order branching time logic of multi-agent systems. In *Proc. of the European Conference of Artificial Intelligence (ECAI), Vienna, Austria*. Willey and Sons, August 1992.

[13] P. Rondogiannis, M. Gergatsoulis, and T. Panayiotopoulos. *Cactus*: A branching-time logic programming language. In *Proc. of the First International Joint Conference on Qualitative and Quantitative Practical Reasoning, ECSQARU-FAPR'97, Bad Honnef, Germany*, Lecture Notes in Artificial Intelligence (LNAI) 1244, pages 511–524. Springer, June 1997.

[14] Dov Gabbay. Modal and temporal logic programming. In A. Galton, editor, *Temporal Logics and their applications*, pages 197–237. Academic Press, London, 1987.

[15] D. M. Gabbay, I. Hodkinson, and M. Reynolds. *Temporal Logic: Mathematical foundations and computational aspects*. Clarendon Press-Oxford, 1994.

[16] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.

[17] H. R. Lewis and C. H. Papadimitriou. *Elements of the Theory of Computation*. Prentice-Hall, Inc., 1981.

[18] L. Sterling and E. Shapiro. *The art of Prolog*. M.I.T. Press, 1986.

[19] A. Yaghi. *The intensional implementation technique for functional languages*. PhD thesis, Dept. of Computer Science, University of Warwick, Coventry, UK, 1984.

[20] P. Rondogiannis. *Higher-order functional languages and intensional logic*. PhD thesis, Dept. of Computer Science, University of Victoria, Canada, December 1994.

[21] P. Rondogiannis and W. W. Wadge. First-order functional languages and intensional logic. *Journal of Functional Programming*, 7(1):73–101, 1997.

[22] P. Rondogiannis and W. W. Wadge. Compiling higher-order functions for tagged dataflow. In *Proceedings of the IFIP International Conference on Parallel Architectures and Compilation Techniques*, pages 269–278. North Holland, 1994.

[23] P. Rondogiannis and M. Gergatsoulis. The intensional implementation technique for chain datalog programs. In *Proc. of the 11th International Symposium on Languages for Intensional Programming (ISLIP'98), May 7-9, Palo Alto, California, USA*, 1998.

[24] E. A. Emerson and J. Y. Halpern. Decision procedures and expressiveness in the temporal logic of branching time. *Journal of Computer and System Sciences*, 30(1):1–24, February 1985.

[25] E. A. Emerson and J. Y. Halpern. "Sometimes" and "Not Never" revisited: On branching versus linear time temporal logic. *Journal of the ACM*, 33(1):151–178, January 1986.

[26] M. A. Orgun and W. W. Wadge. Chronolog admits a complete proof procedure. In *Proc. of the Sixth International Symposium on Lucid and Intensional Programming (ISLIP'93)*, pages 120–135, 1993.

[27] W. W. Wadge and E. A. Ashcroft. *Lucid, the dataflow programming language*. Academic Press, 1985.

[28] M. Gergatsoulis, P. Rondogiannis, and T. Panayiotopoulos. Proof procedures for branching-time logic programs. In W. W. Wadge, editor, *Proc. of the Tenth International Symposium on Languages for Intensional Programming (ISLIP'97), May 15-17, Victoria BC, Canada*, pages 12–26, 1997.

[29] T. Fruehwirth. Annotated constraint logic programming applied to temporal reasoning. In *Proc. of the International Symposium on Programming Language Implementation and Logic Programming (PLILP'94)*, Lecture Notes in Computer Science (LNCS) 844, pages 230–243. Springer-Verlag, 1994.

[30] M. A. Orgun and W. Du. Multi-dimensional logic programming: theoretical foundations. *Theoretical Computer Science*, 158(2):319–345, 1997.

[31] E. A. Ashcroft, A. A. Faustini, R. Jagannathan, and W. W. Wadge. *Multidimensional programming*. Oxford University Press, 1995.

[32] P. Rondogiannis. Negation in Chronolog. In *Proc. of the 11th International Symposium on Languages for Intensional Programming (ISLIP'98), May 7-9, Palo Alto, California, USA*, 1998.

[33] T. Panayiotopoulos and M. Gergatsoulis. Intelligent information processing using TRLi. In *6th International Conference and Workshop on Data Base and Expert Systems Applications (DEXA' 95), (Workshop Proceedings), London, UK, 4-8 September*, pages 494–501, 1995.

[34] S. Tao. *Indexical attribute grammars*. PhD thesis, Dept. of Computer Science, University of Victoria, Canada, 1994.