Cactus: A Branching-Time Logic Programming Language*

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@zeus.cs.uoi.gr

[2] Inst. of Informatics & Telecom., N.C.S.R. 'Demokritos',
153 10 A. Paraskevi Attikis, Greece
e_mail: manolis@iit.nrcps.ariadne-t.gr

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

Abstract. Temporal programming languages are recognized as natural and expressive formalisms for describing dynamic systems. However, most such languages are based on linear flow of time, a fact that makes them unsuitable for certain types of applications. 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 has a tree-like structure. As a result, Cactus appears to be especially appropriate for expressing non-deterministic computations or generally algorithms that involve the manipulation of tree data structures.

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

1 Introduction

Temporal programming languages [OM94, Org91] are recognized as natural and expressive formalisms for describing *dynamic* systems. For example, consider the following Chronolog [Wad88] 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, Cronolog as well as most temporal languages [OM94, Hry93, OWD93, Bau93, OW92, Brz91, Brz93, GRP96] are based on linear flow of time, a fact that

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

This paper appears in the Proceedings of the First International Joint Conference on Qualitative and Quantitative Practical Reasoning; ECSQARU-FAPR'97, Bad Honnef, Germany, June 1997, Lecture Notes in Artificial Intelligence LNAI 1244, D. Gabbay and R. Kruse and A. Nonnengart and H. J. Ohlbach (eds), pp 511-524, Springer.

makes them unsuitable for certain types of applications. 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 next moments. The new formalism is appropriate for describing non-deterministic computations or more generally computations that involve the manipulation of trees.

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. Notice that we actually have a family $\{\texttt{next}_i \mid i \in N\}$ of `next` operators, each one of them representing the different next moments that immediately follow the present one.

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.
```

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 1. 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, `n` 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.

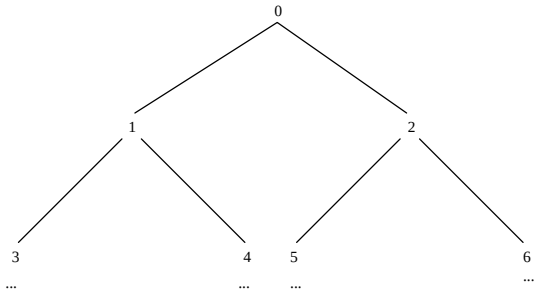

Fig. 1. A mapping of the natural numbers on a binary tree

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 [Gab87, GHR94] for a good discussion on this subject). Moreover, 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.

The rest of the paper is organized as follows: in section 2 we present various Cactus programs which demonstrate its potential in expressing tree computations. In section 3, we formally introduce the syntax of the language. Section 4 presents the underlying branching time logic BTL of Cactus. In section 5 we discuss implementation issues, and section 6 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 [Llo87].

A *temporal atom* is an atomic formula with a number (possibly 0) of applications of temporal operators. 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:

$$H \leftarrow B_1, ..., B_m$$

where $H, B_1, ..., B_m$ 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 *goal clause* in Cactus is a formula of the form $\leftarrow A_1, ..., A_n$ where A_i, $i = 1, ..., n$ 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.

As it will become clear from the semantics of Cactus, a clause is assumed to be true at every moment in time. In particular, this explains the difference between a clause of the form

```
first nat(0).
```

and the clause

```
nat(0).
```

The first clause asserts that it is always true that `nat(0)` is true at the beginning of time while the second clause indicates that it is always true that `nat` is true of 0 at every moment in time.

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 2 (taken from [LP81] 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(q2)  ←  state(q1).
next1 state(q0)  ←  state(q1).
next0 state(q0)  ←  state(q2).
```

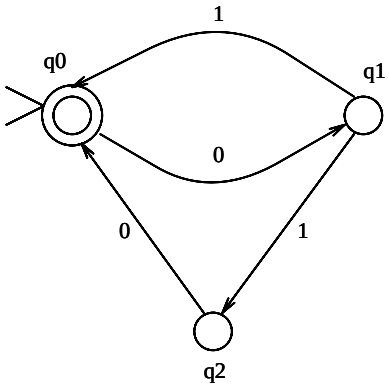

Fig. 2. A non-deterministic finite automaton

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

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

will return the answer `yes` which indicates that the string 010 is an acceptable string of the language L.

As we will see in section 5, the proof procedure of Cactus is similar in nature to the well known SLD-resolution of Horn clause logic programming [Llo87].

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:

```
first binseq([]).
next_0 binseq([0|X])  ← binseq(X).
next_1 binseq([1|X])  ← binseq(X).
```

The goal clause:

```
← binseq(S).
```

will trigger an infinite computation which will generate all possible sequences. More specifically, the underlying proof procedure of Cactus, considers the above goal clause as an infinite set of "temporally ground" goal clauses, each one corresponding to a different point of the time tree.

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`). This is due to the fact that each succesful evaluation of the goal `state(q0)` at a specific time point, triggers a corresponding evaluation of `binseq`, at the same time point.

It is worthwhile noting here that in order to generate another language one only needs to change the definition of the automaton and not the definition of `binseq`.

3.3 Representing and manipulating trees

Branching time logic programming is a powerful 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 and naturally expressed through Cactus programs. For example, consider the binary tree of figure 3.

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

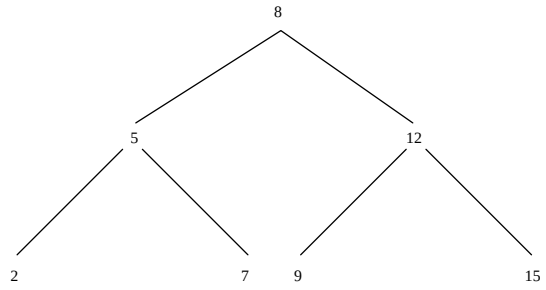

Fig. 3. An (ordered) binary tree containing numeric data

$$\begin{array}{l}
\texttt{first data(8).}\\
\texttt{first next}_0\ \texttt{data(5).}\\
\texttt{first next}_1\ \texttt{data(12).}\\
\texttt{first next}_0\ \texttt{next}_0\ \texttt{data(2).}\\
\texttt{first next}_1\ \texttt{next}_0\ \texttt{data(9).}\\
\texttt{first next}_0\ \texttt{next}_1\ \texttt{data(7).}\\
\texttt{first next}_1\ \texttt{next}_1\ \texttt{data(15).}
\end{array}$$

The following program defines the predicate `descendant(X)`. A temporal atom $< Temporal\ \ reference >$ `descendant(X)` is true if `data(X)` is true in the time represented by $< Temporal\ \ reference >$ or in a future moment of this time point.

$$\begin{array}{lcl}
\texttt{descendant(X)} & \leftarrow & \texttt{data(X).}\\
\texttt{descendant(X)} & \leftarrow & \texttt{data(Y), next}_0\ \texttt{descendant(X).}\\
\texttt{descendant(X)} & \leftarrow & \texttt{data(Y), next}_1\ \texttt{descendant(X).}
\end{array}$$

Notice that the purpose of the existence of the atom `data(Y)` in the bodies of the second and third clause is only to ensure termination of the proof procedure.

A more efficient definition of the predicate `descendant` which takes into account the fact that the binary tree is ordered (binary search) is shown in the following program.

$$\begin{array}{lcl}
\texttt{descendant(X)} & \leftarrow & \texttt{data(X).}\\
\texttt{descendant(X)} & \leftarrow & \texttt{data(Y), X} < \texttt{Y,next}_0\ \texttt{descendant(X).}\\
\texttt{descendant(X)} & \leftarrow & \texttt{data(Y), X} > \texttt{Y,next}_1\ \texttt{descendant(X).}
\end{array}$$

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 the future of `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 definition of `search` is given by the clause:

```
search(X) ← first descendant(X).
```

Let us now define a predicate `flattree` which collects the values in the tree nodes into a list. This definition corresponds to the preorder traversal of the tree.

```
flattree([]) ← data(void).
flattree([X|L]) ← data(X),
                  next_0 flattree(L1),
                  next_1 flattree(L2),
                  append(L1, L2, L).
```

Notice that the above program recognizes the tips of the tree when it encounters a `data(void)` unit clause. For this, we have to add the following unit clauses to the program[4]:

```
first next_0 next_0 next_0 data(void).
first next_0 next_0 next_1 data(void).
first next_0 next_1 next_0 data(void).
first next_0 next_1 next_1 data(void).
first next_1 next_0 next_0 data(void).
first next_1 next_0 next_1 data(void).
first next_1 next_1 next_0 data(void).
first next_1 next_1 next_1 data(void).
```

4 The branching time logic of Cactus

In this section we describe the branching time logic (BTL) on which Cactus is based. In BTL, time has an initial moment and flows towards the future in a tree-like way. The set of moments in time in BTL, can be modelled by the set $List(N)$ of lists of natural numbers. In this case, each node has a countably infinite number of branches (`next` operators). Similarly, we may choose a finite subset S of N and define the logic $BTL(S)$, which has a finite number of $\texttt{next}_i$ operators whose subscript i ranges over the set S. Intuitively, this corresponds to trees in which every node has a finite number of branches. In any case, the empty list [ ] corresponds to the beginning of time and the list $[i|t]$ (that is, the list with head i and tail t) corresponds to the i-th child of the moment identified by the list t.

[4] A more compact representation of the above tree (that avoids the use of void nodes) would be to distinguish the (inner) nodes from the leafs of the tree by using two different predicate names e.g. `node(X)` and `tip(X)` instead of the single predicate `data`. In that case we have to change slightly the definition of `flattree`.

BTL uses the temporal operators `first` and $\texttt{next}_i$, $i \in N$. 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 [BAPM83].

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 [Org91].

Definition 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 $[List(N) \rightarrow 2^{D^n}]$.

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

Definition 2. The semantics of the elements of the temporal logic BTL are given inductively as follows:

1. If $\texttt{f}(e_0, \ldots, e_{n-1})$ is a term, then $I(\texttt{f}(e_0, \ldots, e_{n-1})) = I(\texttt{f})(I(e_0), \ldots, I(e_{n-1}))$.
2. For any n-ary predicate symbol $\texttt{p}$ and terms $e_0, \ldots, e_{n-1}$,
   $\models_{I,t} \texttt{p}(e_0, \ldots, e_{n-1})$ *iff* $\langle I(e_0), \ldots, I(e_{n-1})\rangle \in I(\texttt{p})(t)$
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. $\models_{I,t} (\forall x)A$ *iff* $\models_{I[d/x],t} A$ *for all* $d \in D$ *where the interpretation* $I[d/x]$ *is the same as* I *except that the variable* x *is assigned the value* d.
7. $\models_{I,t}$ `first` A *iff* $\models_{I,[\,]} A$
8. $\models_{I,t}$ $\texttt{next}_i$ A *iff* $\models_{I,[i|t]} A$

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 paper [RGP97].

4.2 Axioms and Rules of Inference

In this section we present some useful axioms and inference rules that hold for the logic BTL, many of which are similar to those adopted for the case of linear time logics [Org91]. In the following, the symbol ∇ stands for any of `first` and $\texttt{next}_i$.

Temporal operator cancellation rules: The intuition behind these rules is that the operator `first` cancels the effect of any other "outer" operator. Formally:

$$\nabla(\texttt{first}\ A) \leftrightarrow (\texttt{first}\ A)$$

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

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:

$$\nabla(\neg A) \leftrightarrow \neg(\nabla A)$$
$$\nabla(A \wedge B) \leftrightarrow (\nabla A) \wedge (\nabla B)$$
$$\nabla(A \vee B) \leftrightarrow (\nabla A) \vee (\nabla B)$$

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

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.

Temporal operator non-commutativity rule: This rule says that the following:

$$\texttt{next}_i\ \texttt{next}_j\ A \leftrightarrow \texttt{next}_j\ \texttt{next}_i\ A$$

is not a valid axiom of the language when $i \neq j$. The essence of this rule is that in general, two operators $\texttt{next}_i$ and $\texttt{next}_j$ can not be interchanged when i and j are different.

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

$$\nabla(\forall X)(A) \leftrightarrow (\forall X)(\nabla A)$$

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

Temporal operator introduction rules: The following rule states that if A is a theorem of BTL then ∇A is also a theorem of BTL.

$$if\ \vdash A\ then\ \vdash \nabla A$$

The validity of the above axioms is easily proved using the semantics of BTL.

5 A proof procedure for branching time logic programs

Cactus programs are executed using a resolution-type proof procedure called *BSLD-resolution* (**B**ranching-time **SLD**-resolution). For practical reasons, we suppose that the underlying logic of Cactus programs is $BTL(S)$, where S is a finite subset of N (i.e. in the time tree every node has a finite number of branches). BSLD-resolution is a refutation procedure which extends SLD-resolution [Llo87], and is similar to TiSLD-resolution [OW93], the proof procedure for Chronolog programs. The following definitions are necessary in order to introduce BSLD-resolution.

Definition 3. A *canonical temporal atom* is a formula $\texttt{first next}_{i_1} \cdots \texttt{next}_{i_n}\ A$, where $i_1, \ldots, i_n \in S$ and $n \geq 0$, and A is an atom. A *canonical temporal clause* is a temporal clause whose temporal atoms are canonical temporal atoms.

As in Chronolog [Org91, OWD93], every temporal clause can be transformed into a (possibly infinite) set of canonical temporal clauses. This can be done by applying $\texttt{first next}_{i_1} \cdots \texttt{next}_{i_n}$, where $i_1, \ldots, i_n \in S$ and $n \geq 0$, to the clause and then using the axioms of BTL, presented in section 4.2, to distribute the temporal reference so as to be applied to each individual temporal atom of the clause; finally any superfluous operator is eliminated by applying the cancellation rules of BTL.

Intuitively, a canonical temporal clause is an instance in time of the corresponding temporal clause.

Example 1. Consider the following Cactus program:

$$\begin{aligned} &\texttt{first p(0).} \\ &\texttt{next}_0\ \texttt{p(s(X))} \leftarrow\ \texttt{p(X).} \\ &\texttt{next}_1\ \texttt{p(s(s(X)))} \leftarrow\ \texttt{p(X).} \end{aligned}$$

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

The clause:

$\texttt{first p(0).}$

is the only canonical temporal clause corresponding to the first program clause (because of axiom 1).

The set of clauses:

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

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

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

corresponds to the third program clause.

The notion of canonical atom/clause is very important since the value of a given formula of a branching time logic $BTL(S)$, for some finite subset S of N, in a temporal interpretation can be expressed in terms of the values of its canonical instances, as the following lemma shows:

Lemma 4. *Let A be a formula and I a temporal interpretation of $BTL(S)$. $\models_I A$ if and only if $\models_I A_t$ for all canonical instances A_t of A.*

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

Definition 5. Let P be a Cactus program and G be a canonical temporal goal. A *BSLD-derivation* from P with top goal G consists of a (possibly infinite sequence) of canonical temporal goals $G_0 = G, G_1, ..., G_n, ...$ such that for all i the goal G_{i+1} is obtained from the goal:
$G_i = \leftarrow A_1, ..., A_{m-1}, A_m, A_{m+1}, ..., A_p$
as follows:

1. A_m is a canonical temporal atom in G_i (called the *selected* atom)
2. $H \leftarrow B_1, ..., B_r$ is a canonical instance of a program clause,
3. there is a substitution θ such that $\theta = mgu(A_m, H)$
4. G_{i+1} is the goal:
   $G_{i+1} = \leftarrow (A_1, ..., A_{m-1}, B_1, ..., B_r, A_{m+1}, ..., A_p)\theta$

Definition 6. Let P be a Cactus program and G be a canonical temporal goal. A *BSLD-refutation* from P with top goal G is a finite BSLD-derivation of the null clause $\Box$ from P with top goal G.

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

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

```
← first next_0 next_1 nat(N)
```

is given below:

```
← first next_0 next_1 nat(N)
```

using clause (3)

```
← first next_0 nat(X), first next_0 (N is 2 * X + 2)
```

using clause (2)

$\leftarrow$ `first nat(`X_1`)`, `first next`$_1$ `(X is 2 * `X_1 `+ 1),`
`first next`$_0$ `(N is 2 * X + 2)`

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

$\leftarrow$ `first next`$_1$ `(X is 2 * 0 + 1)`, `first next`$_0$ `(N is 2 * X + 2)`

(`X = 1`) evaluation of the built-in predicate `is`[5]

$\leftarrow$ `first next`$_0$ `(N is 2 * 1 + 2)`

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

□

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 Cronolog [OW93]. 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 (e.g. by traversing in a breadth-first way the time-tree).

6 Conclusions

Temporal programming languages, either functional [WA85, DW90, EAAJ91] or logic [OM94, Org91, PG95], 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 introduce the branching time logic programming language **Cactus** which is based on a tree-like notion of time. We demonstrate that Cactus is capable of expressing various problems in a natural way. Moreover, we show that Cactus retains the semantic clarity of logic programming and has a simple procedural interpretation.

The branching time concept has been particularly successful in the functional programming domain [RW97, Ron94, Yag84, Tao94] and we believe that a similar potential exists for the area of logic programming.

References

[BAPM83] M. Ben-Ari, A. Pnueli, and Z. Manna. The Temporal Logic of Branching Time. *Informatica*, pages 207–226, 1983.

[5] 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 considered to be independent of time (rigid).

[Bau93] M. Baudinet. A simple proof of the completeness of temporal logic programming. In L. Farinas del Cerro and M. Penttonen, editors, *International Logics for Programming*, pages 51–83. Oxford University Press, 1993.

[Brz91] 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.

[Brz93] 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.

[DW90] W. Du and W.W.Wadge. A 3D Spreadsheet Based on Intensional Logic. *IEEE Software*, pages 78–89, July 1990.

[EAAJ91] A. A. Faustini E. A. Ashcroft and R. Jagannathan. An Intensional Language for Parallel Applications Programming. In B.K.Szymanski, editor, *Parallel Functional Languages and Compilers*, pages 11–49. ACM Press, 1991.

[Gab87] Dov Gabbay. Modal and temporal logic programming. In A. Galton, editor, *Temporal Logics and their applications*, pages 197–237. Academic Press, London, 1987.

[GHR94] D. M. Gabbay, I. Hodkinson, and M. Reynolds. *Temporal Logic: Mathematical Foundations and Computational Aspects*. Clarendon Press-Oxford, 1994.

[GRP96] 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.

[Hry93] T. Hrycej. A temporal extension of Prolog. *The Journal of Logic Programming*, 15:113–145, 1993.

[Llo87] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.

[LP81] H. R. Lewis and C. H. Papadimitriou. *Elements of the Theory of Computation*. Prentice-Hall, Inc., 1981.

[OM94] 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)*, pages 445–479. Springer Verlag, 1994. LNCS No 827.

[Org91] M. A. Orgun. *Intensional Logic Programming*. PhD thesis, Dept. of Computer Science, University of Victoria, Canada, December 1991.

[OW92] M. A. Orgun and W. W. Wadge. Towards a unified theory of intensional logic programming. *The Journal of Logic Programming*, 13(4):113–145, August 1992.

[OW93] 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.

[OWD93] 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.

[PG95] 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, 4th-8th September*, pages 494–501, 1995.

[RGP97] P. Rondogiannis, M. Gergatsoulis, and T. Panayiotopoulos. Theoretical foundations of Branching-Time Logic Programming. 1997. In preparation.

[Ron94] P. Rondogiannis. *Higher-Order Functional Languages and Intensional Logic.* PhD thesis, Dept. of Computer Science, University of Victoria, Canada, December 1994.

[RW97] P. Rondogiannis and W. W. Wadge. First-order functional languages and intensional logic. *Journal of Functional Programming*, 1997. (to appear).

[Tao94] S. Tao. *Indexical Attribute Grammars.* PhD thesis, Dept. of Computer Science, University of Victoria, Canada, 1994.

[WA85] W. W. Wadge and E. A. Ashcroft. *Lucid, the dataflow Programming Language.* Academic Press, 1985.

[Wad88] 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.

[Yag84] A. Yaghi. *The Intensional Implementation Technique for Functional Languages.* PhD thesis, Dept. of Computer Science, University of Warwick, Coventry, UK, 1984.

This article was processed using the LaTeX macro package with LLNCS style