Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

EXECUTABLE TEMPORAL SPECIFICATIONS WITH FUNCTIONAL GRAMMARS

H. Lewis Chau
D. Stott Parker

June 1988
CSD-880046

Executable Temporal Specifications with Functional Grammars

H. Lewis Chau

D. Stott Parker

Department of Computer Science
University of California
Los Angeles, CA 90024-1596

ABSTRACT

The *Stream Pattern Analyzer (SPA)* is one part of the Tangram Stream Query Processing System being developed at UCLA. It uses *functional grammars* to specify pattern analysis for streams of data.

Parallel execution events in a distributed system may be captured in an event stream for analysis. Given a set of functional grammar rules, SPA can analyze arbitrarily complex behavior patterns in this stream. At the same time a SPA grammar can act as a declarative specification of valid event histories.

We define a simple but powerful scheme that coroutines recognition of multiple patterns in an event stream. Propositional temporal logic queries can be expressed in SPA in terms of predefined temporal operators such as *eventually, implies, not_until,* etc. Thus complex history-oriented specifications can be developed easily.

Functional grammar rules by themselves act as pattern generators or specifiers, and can be used to develop parsers by compilation to Log(F). Log(F) is a combination of Prolog and a functional language called F*. We describe a simple algorithm to compile functional grammars to Log(F), and prove its correctness.

Keywords: Executable Specifications, Distributed Systems, Temporal Logic, Parsing, Prolog.

This work done under the Tangram project, supported by DARPA contract F29601-87-C-0072.

H. Lewis Chau (chau@cs.ucla.edu) (213) 825-2756
D. Stott Parker (stott@cs.ucla.edu) (213) 825-6871

Executable Temporal Specifications with Functional Grammars

H. Lewis Chau

D. Stott Parker

Department of Computer Science
University of California
Los Angeles, CA 90024-1596

1. Introduction

Tangram is a multi-paradigm environment for modeling. Model simulations generate enormous event streams, and these streams require sophisticated analysis tools. The *Tangram Stream Processor (TSP)*, the functional/stream paradigm of Tangram, has been discussed in [22].

TSP is a system founded on the abstraction of stream transducers. A transducer is a mapping from some number of input streams to one or more output streams. Transducers are the basic building blocks of TSP, and can be combined into networks. The resulting system may be used for 'database-flow' computations, a combination of 'dataflow' and database processing.

In this paper we discuss the *Stream Pattern Analyzer (SPA)*, a subsystem of TSP. It allows users to specify patterns with *functional grammars*, which are compiled to efficient transducers. Regular expressions and, more generally, path expressions [3], can be easily defined with functional grammars. As a simple example, to recognize sequences of one or more copies of *net_failure* followed by a *cpu_failure*, we can specify the pattern with the following grammar rules:

```
pattern => ([net_failure]+,[cpu_failure]).
(X+) => X.
(X+) => X, (X+).
(X,Y) => append(X,Y).
```

where '+' is the postfix pattern operator defining the Kleene plus, and ',' defines pattern concatenation.

Definite Clause Grammars (DCGs) [23] rest on Prolog, and were among the first practical logic grammars for natural language analysis. Functional grammars rest on Log(F), a combination of Prolog and a functional language called F*, developed by Sanjai Narain at UCLA [19,20]. Functional grammars provide the basis for arbitrary pattern analysis on trace data which cannot be conveniently specified by existing database query languages. We stress that Log(F) is an addition to Prolog without changing Prolog's fundamental primitives such as its unification algorithm or control strategy, and the formalism of functional grammars is more general than DCGs (see sections 2 and 3).

One interesting use of functional grammars is to describe the behavior of distributed systems. In the literature, many researchers have used temporal logic for specification [10, 13], verification

[16, 21, 27], testing/debugging [14], and synthesis [15, 29, 5] of concurrent systems. Others have expressed concurrency with a combination of formal languages, partial orders, and temporal logic [25]. The functional grammar approach presented in this paper offers increased modularity and power over previous approaches. At the specification level, arbitrary pattern analysis can be conveniently expressed by grammar rules. At the implementation level, functional grammars can be compiled straightforwardly to Log(F) and executed by standard Prolog interpreters.

Specifications can be of two basic types:

(1) They are *'object-oriented'* [2], or *'automaton-oriented'*. Specification is given in terms of behaviors (methods, transitions) of some entity (object, automaton) in response to some event (message, input). Examples of this kind of specification are finite automata, Petri nets, object-oriented simulations, Harel's Statecharts [11], etc. It arises naturally, since people often conceptualize systems in terms of a collection of entities interacting over time.

(2) They are *'history-oriented'*. Specification is given in terms of restrictions on legal histories or event traces. Examples of this kind of specification are temporal logic and path expressions. It arises often when the relationships or constraints among valid behaviors become complex.

A system related to SPA for object-oriented specifications is [7], a model for distributed systems based on graph rewriting. Many graph-rewriting systems have been proposed [8, 28]. It is interesting to consider extension of the term-rewriting foundation of SPA to graph-rewriting, but we will not investigate that here.

Section 2 describes Log(F), the programming environment of SPA, and provides an example library of transducer operations and their Prolog implementation. We henceforth assume that the reader is familiar with Prolog. A good introduction to Prolog can be found, for example, in [6]. Section 3 formally defines functional grammars, and how they apply to stream pattern analysis. Section 4 presents a stream pattern analysis approach to describe a distributed system. Section 5 then goes on to discuss implementation. A simple algorithm to compile functional grammars to Log(F) transducers is described. Finally, optimization issues will be addressed, along with avenues for future work.

2. Log(F)

Log(F) is the integration of Prolog and a functional language in which one programs using rewrite rules. This section reviews the major aspects of Log(F), and describes its advantages for stream processing [17].

2.1. Overview of F* and Log(F)

F* is a rewrite rule language. In F*, all statements are rules of the form

LHS => *RHS*

where *LHS* and *RHS* are terms (actually Prolog terms) satisfying certain modest restrictions summarized below.

Definition 2.1

A *term* is either a variable, or an expression of the form $f(t_1,..,t_n)$ where f is a n-ary function symbol, $n \geq 0$, and each t_i is a term.

Consider the following two rules, defining how lists may be appended:

```
append([],W) => W.
append([U|V],W) => [U|append(V,W)].
```

Like the Prolog rules for appending lists, this concise description provides all that is necessary.

Log(F) is the integration of F* with Prolog. In Log(F), F* rules are compiled to Prolog clauses. The compilation process is straightforward. For example, the two rules above are translated into something functionally equivalent to the following Prolog code:

```
reduce(append(A,B),C) :- reduce(A,[]), reduce(B,C).
reduce(append(A,B),C) :- reduce(A,[D|E]), reduce([D|append(E,B)],C).
```

Unlike many rewriting systems, the `reduce` rules here can operate non-deterministically, just like their Prolog counterparts. Many ad hoc function- or rewrite rule-based systems have been proposed to incorporate Prolog's backtracking, but the simple implementation of F* in Prolog shown above provides this capability as a natural and immediate feature.

An important feature of F* and Log(F) is the capability for *lazy evaluation*. With the rules above, the goal

```
?- reduce(append([1,2,3],[4,5,6]),X).
```

yields the result

```
X = [1|append([2,3],[4,5,6])].
```

That is, in one `reduce` step, only the head of the resulting appended list is computed. The tail, `append([2,3],[4,5,6])`, can then be further reduced if this is necessary. Demand-driven computation like this is referred to as lazy evaluation or delayed evaluation, and is basic to stream processing [1].

The astute reader will have noticed that, in order for the `reduce` rules above to work as we are claiming, we will have to add two definitions:

```
reduce([],[]).
reduce([X|Y],[X|Y]).
```

Definition 2.2

In F*, functors like `[]` and `[_|_]` with this property are called *constructor symbols*. Terms whose functors are constructor symbols are said to be *simplified;* they cannot be directly reduced further.

The main restriction on Log(F) rules *LHS* => *RHS* is that *LHS* must be of the form

$$f(t_1, \ldots, t_n)$$

where $n \geq 0$, and each of the t_i is either a variable or a term whose functor is a constructor symbol. This restriction guarantees efficient implementation. In order to guarantee soundness and completeness properties, restrictions on variables are also made: first, no variable may appear twice in *LHS* (the 'linearity' restriction), and second, every variable in *RHS* must also appear in *LHS* [19, 20].

There is one more important point about the integration of F* with Prolog. Where F* computations are naturally lazy because of their implementation with reduction rules, Log(F) permits some *eager computation* as well. Essentially, eager computations invoke routines outside F*. For example, in the Log(F) code

```
count([X|S],N) => count(S,N+1).
```

the subterm `N+1` is recognized by the Log(F) compiler as being eager, and the resulting code produced is equivalent to

```
reduce(count(A,N),Z) :- reduce(A,[X|S]), M is N+1, reduce(count(S,M),Z).
```

Programmers may declare their own predicates to be eager. By judicious combination of eager and lazy computation, programmers obtain programming power not available from Prolog or F* alone.

It is easy to develop programs with compact sets of rewrite rules. For example, the following is an executable Log(F) program for computing primes via the sieve of Eratosthenes:

```
primes => sieve(intfrom(2)).

intfrom(N) => [N|intfrom(N+1)].

sieve([U|V]) => [U|sieve(filter(U,V))].

:- eager multiple/2.

multiple(U,A,true) :- 0 is U mod A, !.
multiple(_,_,false).

filter(A,[U|V]) => if( multiple(U,A), filter(A,V), [U|filter(A,V)] ).
```

The `intfrom` rule generates an infinite stream of integers. The rule for `filter` uses the eager Prolog predicate `multiple`. As an example of execution, if we define the predicate

```
reducePrint(X) :- reduce(X,[H|T]), write(H - T), nl, reducePrint(T).
```

then the goal

```
?- reducePrint(primes).
```

produces the following (non-terminating) output:

```
2 - sieve(filter(2,intfrom(3)))
3 - sieve(filter(3,filter(2,intfrom(4))))
5 - sieve(filter(5,filter(3,filter(2,intfrom(6)))))
7 - sieve(filter(7,filter(5,filter(3,filter(2,intfrom(8))))))
    ...
```

For other useful examples of the combination of lazy and eager evaluation, see [18].

2.2. Advantages of Log(F)

From the examples above it is clear that the rules have a functional flavor. Stream operators are easily expressed using recursive functional programs. The syntax is convenient, and can be considered a useful query language in its own right.

Furthermore, Log(F) naturally provides *lazy evaluation*. Functional programs on lists can produce terms in an incremental way, and incremental or "call by need" evaluation is an elegant mechanism for controlling query processing.

It turns out furthermore that Log(F) has a formal foundation that captures important aspects of stream processing:

(1) Determinate (non-backtracking) code is easily detected through syntactic tests only. This avoids the overhead of "distributed backtracking" incurred by some parallel logic programming systems.

(2) Log(F) takes as a basic assumption that stream values are *ground terms*, i.e., Prolog terms without variables. Again this avoids problems encountered by other parallel Prolog systems which must attempt to provide consistency of bindings to variables used by processes on opposing ends of streams.

These features of Log(F) make it a nicely-limited sublanguage in which to write high-powered programs for stream processing and other performance-critical tasks. Special-purpose compilers can be developed for this sublanguage that produce highly-optimized code.

3. Stream Pattern Analysis

In [22], we illustrated how Log(F) is a natural system for expressing transductions of streams. Here we show how functional grammars also make a powerful language for specifying pattern analysis against streams.

3.1. Functional Grammars and the Match Transducer

Definition 3.1

Functional grammar rules have the form:

LHS => *RHS*.

Here *LHS* and *RHS* satisfy the following properties.

(1) *LHS* is any term except a variable or a simplified term (a constructor symbol with zero or more argument terms). In addition, each of the arguments of *LHS* is either a variable or a constructor symbol with variable arguments. This restriction guarantees efficient implementation.

(2) *RHS* is a term.

(3) If *RHS* is a variable, it must appear in *LHS*.

Functional grammars relax Log(F) variable restrictions to obtain the power of unification. That is, arguments of terms in rewrite rules can be used not only as inputs as in Log(F), but also as grammar outputs (see example 3.1).

Regular expressions and, more generally, path expressions, can be easily defined with grammar rules as in the following:

```
(X+) => X.
(X+) => X,(X+).
(X*) => [].
(X*) => X,(X*).
(X;Y) => X.
(X;Y) => Y.
(X,Y) => append(X,Y).
append([],X) => X.
append([X|Y],Z) => [X|append(Y,Z)].
if(true,X) => X.
if(true,X,Y) => X.
if(false,X,Y) => Y.
skipto(X) => X.
skipto(X) => [_],skipto(X).
```

Example 3.1

Suppose we wish to count the number of times one or more network failures were followed by a cpu failure. That is, we want to count the occurrences of a specific pattern in the input stream. We can use the pattern:

```
number( ([net_failure]+,[cpu_failure]), Total).
```

where we include the following grammar for `number`:

```
number(Pattern,Total) => number(Pattern,Total,0).
number(Pattern,Total,Total) => [end_of_file].
number(Pattern,Total,Count) => skipto(Pattern), number(Pattern,Total,Count+1).
```

Here the variable `Total` is used to return output values, and the variable `Pattern` is used as input any patterns. Note such a grammar cannot be developed easily using DCGs.

The rules for pattern analysis are very simple. The entire definition is based on the following *match transducer* :

```
match([],S) => S.
match([X|L],[X|S]) => match(L,S).
```

This transducer takes a pattern as its first argument, and an input stream as its second argument. If the pattern reduces to the empty list [], *match* simply succeeds. On the other hands, if the pattern reduces to some stream [X|L], it is matched against the input stream.

The match transducer can be thought of as *applying* a pattern to a stream, in an attempt to find a prefix of the stream that the grammar defining the pattern can generate. There is a certain elegance to this; the rules of the grammar by themselves act as pattern generators, but when applied with the *match* transducer they act like a parser. This acceptance/generation duality is familiar to users of DCGs, and the ability to use grammars both as acceptors and as generators has a number of uses.

Pattern analysis is signalled explicitly with the match transducer. For example,

```
match(([net_failure]+,[cpu_failure]),file_terms('experiment.output')).
```

matches the pattern 'one or more copies of `net_failure` followed by a `cpu_failure`' against the stream of terms produced by the file `'experiment.output'`.

3.2. Transformation of Mutual Exclusive Grammar Rules to IF-THEN-ELSE Construct

A limitation of Horn clause logic as a practical programming language is the lack of an explicit IF-THEN-ELSE construct [24]. The cut operator is used for this purpose in Prolog, which is more powerful in a dangerous way. In functional grammars, a structured IF-THEN-ELSE construct is available to serve a similar purpose.

```
if(true,X,Y) => X.
if(false,X,Y) => Y.
```

The conditional part of "if" is only a logical testing of a Prolog goal which returns true or false. However, sometimes we expect the conditional part be a pattern to be matched. That is, the conditional part attempts to consume a part of the input stream and return the rest of the stream if successful. We henceforth introduce a constructor symbol, "cond" which is similar to "if" except that the conditional part is a pattern. We then need one more definition of the *match* transducer.

```
match(cond(A,B,C),S) => if(match(A,S),match(B,match(A,S)),match(C,S)).
```

The declarative interpretation of the above transducer is as follows: if pattern A is matched, then SPA tries to further match pattern B, otherwise, it tries to match pattern C.

For example, the following mutual exclusive grammar rules generate $\{[a]^n[b]^n \mid n>0\}$

```
ab => a(0).
a(Count) => [a], a(Count+1).
a(Count) => [b], b(Count-1).
b(Count) => [b], b(Count-1).
b(0) => [end_of_file].
```

and could be rewritten as

```
ab => a(0).
a(Count) => cond([a],a(Count+1),([b],b(Count-1))).
b(Count) => if(Count==0,[end_of_file],([b],b(Count-1))).
```

The "cond" takes a pattern, [a], as its conditional part while the "if" takes a Prolog goal, Count==0, as its conditional part.

The following are some grammar rules together with their "cond" counterparts.

(X*) => X, (X*). (X*) => [].	(X*) => cond(X,(X*),[]).
(X;Y) => X. (X;Y) => Y.	(X;Y) => cond(X,[],Y).
skipto(X) => X. skipto(X) => [_], skipto(X).	skipto(X) => cond(X,[],([_],skipto(X))).

The use of "if" or "cond" eliminates the creation of some unnecessary choice points, i.e., it reduces the degree of nondeterminism. It encourages better programming style by its restricted use of commitment and enhances readability of the grammar rules. Furthermore, "if" or "cond" in functional grammars can be nested.

4. Stream Pattern Analysis Approach to Extract Behaviors of a Distributed System

Concurrent execution in a distributed system can be captured as a stream of interleaved events from each process. The stream pattern analysis approach we use here captures time-dependent relationships of events that occur in the execution of a concurrent program in an event stream. SPA implements the *match* transducer and permits specification of sets of functional grammar rules that can be used to discern specific patterns in an event stream, i.e., to recognize processes specified by allowable sequences of operations. SPA parses the stream, extracting specific behavior patterns of concurrent programs, such as serializability and mutual exclusion.

4.1. Specifying Propositional Temporal Logic Queries with Functional Grammars

Propositional temporal logic is an extension of classical propositional logic geared towards the description of sequences. In propositional temporal logic, there are different time points which may yield different truth values of propositions. We assume the set of time points to be finite, discrete and linearly ordered. We further assume that only one event occurs at a time. The events that occur during the set of time points can be mapped into a stream of events that abstracts a possible concurrent program execution sequences. The table below summarizes the definitions of some common temporal operators.

eventually(X)	Pattern X will be detected later
precedes(X,Y)	Pattern X precedes pattern Y
not_until(X,Y)	Y appears, and X does not appear before it
implies(X,Y)	If pattern X is detected, pattern Y will be detected later, X and Y are non-overlapping
always_implies(X,Y)	After implies(X,Y) is detected, recursively apply always_implies(X,Y) until end of history

Temporal operators can be easily defined with functional grammar rules, as in the following:

```
eventually(X) => skipto(X).
precedes(X,Y) => eventually(X), eventually(Y).
not_until(X,Y) => skipto_without(Y,X).
skipto_without(X,Y) => cond(X,[],cond(Y,fail,([_],skipto_without(X,Y)))).
implies(X,Y) => cond(eventually(X),eventually(Y),[]).
always_implies(X,Y)=>cond(eventually(X),(eventually(Y),always_implies(X,Y)),[]).
```

Note that the arguments in these grammar rules are normally instantiated to ground patterns during query processing.

Example 4.1 :

The following pattern specifies that [a] is always immediately followed by [b]

```
always_implies([a],[b]).
```

which is invoked by the Prolog query

```
?- reduce(match(always_implies([a],[b]),file_terms('input.stream')),_).
```

SPA reduces the pattern, `always_implies([a],[b])`, to a simplified term by some predefined grammar rules and matches against the stream of terms produced by the file `input.stream`.

Example 4.2 : Two-phase Locking Protocol

Let a transaction event stream be a sequence of [lock]s, [unlock]s and other events of database items. The following is one of the patterns that must be followed in the event stream of a single transaction in order for it to obey the two-phase locking protocol [12]:

```
(eventually([unlock]), not_until([lock],[end_of_file])).
```

That is, a transaction acquires locks as needed, and once it releases a lock it may issue no further lock requests.

4.2. Specifying Parallel Execution Events with Functional Grammars

In a distributed environment, concurrent execution can be modelled by arbitrary interleaving of events from each process, and these events are captured into a stream. Discerning multiple patterns in an event stream often requires sophisticated analysis tools. For efficiency purposes, we avoid multiple scans of a stream. We define a grammar that coroutines recognition of multiple patterns in an event stream, while requiring only one scan of the stream:

```
([X|L1] // [X|L2]) => [X|L1//L2].
([] // L) => L.
(L // []) => L.
```

The coroutining operator, //, takes two patterns as arguments, reduces them to [X|Xs] and [Y|Ys] respectively and rewrites them to [Z|Xs//Ys] where Z = Xθ = Yθ and θ is a most general unifier of X and Y. For example,

```
   (eventually([a]) // eventually([b]))
→ (skipto([a]) // skipto([b]))
→ ([a] // ([_], skipto([b])))
→ ([a] // append([_],skipto([b])))
→ ([a] // [_|append([],skipto([b]))])
→ [a | [] // append([],skipto([b]))]
```

The *match* transducer matches [a] against an event stream. The reduction and matching process repeat until the pattern is detected.

With this coroutining definition, we can relax the assumption that the patterns defined by `implies(X,Y)` are non-overlapping as in the following:

```
implies(X,Y) => cond((X // Y),[],cond(eventually(X),fail,[])).
```

The formalism of // is important. Suppose we have *n* specifications of a system, each of which corresponds to a sequence of events to be matched in a stream, but these sequences are independent to each other and are interleaved arbitrarily. We can specify each of these sequences separately, and combine them with // in order to get a global specification of the whole system.

Example 4.3 : Two-phase Locking Protocol

In example 4.2, if one more constraint of the two-phase locking protocol is required : the number of [lock]s must be equal to the number of [unlock]s. We can specify individual requirement as follows:

```
number([lock],N1)
number([unlock],N2)
(eventually([unlock]), not_until([lock],[end_of_file]))
```

and combine these patterns to get a global specification of the two-phase locking protocol with the constraint `N1=N2`.

```
number([lock],N) // number([unlock],N) //
(eventually([unlock]), not_until([lock],[end_of_file])).
```

Example 4.4 : Mutual Exclusion

Consider an example which is taken from [15]. Suppose we have two processes, P1 and P2, that communicate with a synchronizer S. The signals sent to S by P1 are [begin_1] (begin critical section) and [end_1] (end critical section). Similarly the signals sent to S by P2 are [begin_2] and [end_2]. The synchronizer should ensure that processes P1 and P2 are never simultaneously in their respective critical sections. The following patterns must be recognized in an event

stream in order to ensure mutual exclusion.

```
% The first signal sent by P1 is [begin_1].
not_until([end_1],[begin_1]).

% P1 alternately sends [begin_1] and [end_1].
always_implies([begin_1],not_until([begin_1],[end_1])).
always_implies([end_1],not_until([end_1],[begin_1])).

% Similar grammar rules are defined for P2.
not_until([end_2],[begin_2]).

always_implies([begin_2],not_until([begin_2],[end_2])).
always_implies([end_2],not_until([end_2],[begin_2])).

% After letting P1 proceed into its critical section by accepting
% a [begin_1], do not let P2 enter its own critical section until
% P1 has finished, and vice versa.
always_implies([begin_1],not_until([begin_2],[end_1])).
always_implies([begin_2],not_until([begin_1],[end_2])).
```

The global pattern for specifying mutual exclusion is combining all the above patterns by the coroutining operator `//`.

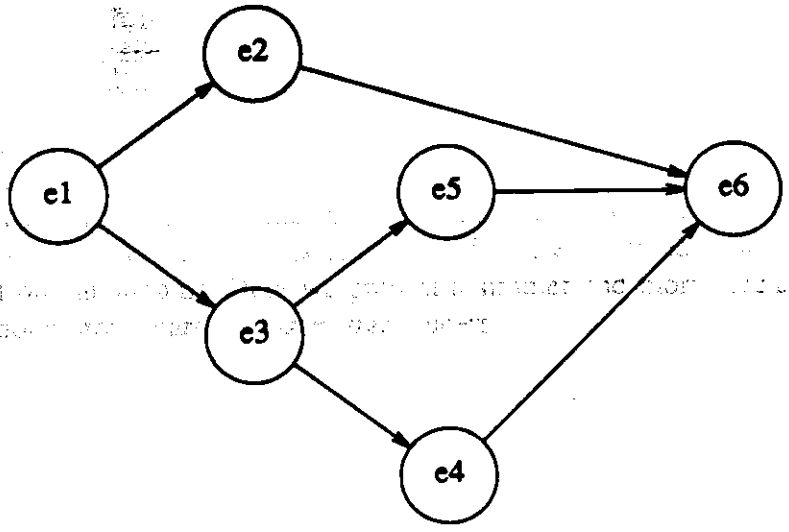
4.3. Specifying Partial Ordering Events with Functional Grammars

In a distributed environment, the computation history consisting of a partial ordering of events which abstracts from any particular interleaving of concurrent events. A directed acyclic graph (DAG) is well suited for such a model. Given a DAG with a set of nodes representing events in the system, an edge X → Y in the DAG represents an order constraint that event X occurs before event Y which can be specified by `precedes(X,Y)`. We can directly represent a global ordering of events in a system by a pattern as follows:

```
... // precedes(X,Y) // ...
```

where each `precedes(X,Y)` corresponds to an edge X → Y in the DAG.

For example, given a partial ordering of events in a system which is represented by a DAG:



The pattern that describes the partial ordering is

```
precedes(e1,e2) // precedes(e1,e3) // precedes(e3,e5) // precedes(e3,e4)
// precedes(e2,e6) // precedes(e5,e6) // precedes(e4,e6).
```

and can be optimized as

```
(eventually(e1), (eventually(e2) // (eventually(e3),
(eventually(e4) // eventually(e5)))), eventually(e6)).
```

Here // is right associative and has a higher precedence than ",".

4.4. Functional Grammars with Unbounded State Information

In previous sections, we have given examples illustrating the expressive power of functional grammars for SPA. All grammar rules predefined in the library require bounded state information. If a query requires unbounded state information, SPA summarizes all the necessary state information when an event stream is scanned and tests whether the specification is matched.

Example 4.5 : First-in First-out Service

Consider a first-order temporal logic formula which specifies first-in first-out service:

$$\forall(X,Y)\,[(\texttt{request}(X)\ \textit{Precedes}\ \texttt{request}(Y)) \supset (\texttt{serve}(X)\ \textit{Precedes}\ \texttt{serve}(Y))] \wedge$$
$$\forall(X)\,[\neg\ \texttt{serve}(X)\ \textit{Until}\ \texttt{request}(X)]$$

where `request` and `serve` are events in a stream. That is, if `request(X)` precedes `request(Y)` then `serve(X)` precedes `serve(Y)`, and a request is not served until it is made. We can capture the above formula by defining the following grammar rules:

```
fifo => fifo([]).
fifo([X|Queue]) => [serve(X)], fifo(Queue).
fifo(Queue) => cond([request(X)],fifo(append(Queue,[X])),([_],fifo(Queue))).
fifo([]) => [end_of_file].
```

When matching against an input stream, `fifo` stores any `request` events in a queue, and these events are matched against their corresponding `serve` events. Note that the state information in this problem is unbounded. Therefore, the predefined temporal grammar rules are no longer sufficient.

In the literature, many researchers have used temporal logic to specify a distributed system. Some systems are restricted to propositional temporal logic specifications, while functional grammars proposed here are not. Let us consider another example which goes beyond propositional temporal logic specifications.

Example 4.6 : Tree-protocol

If we have prior knowledge as to the order in which database items will be accessed, it is possible to construct locking protocols that are not two-phase but, nevertheless, ensure serializability. The tree-protocol [12] is one example. In the tree-protocol, we impose a partial ordering $\rightarrow$ on the set $D = \{d_1,..,d_h\}$ of all data items. If $d_i \rightarrow d_j$, then any transaction accessing both d_i and d_j must access d_i before accessing d_j. d_i is called the parent of d_j, and every data item except the

root has a unique parent.
The following rules must be obeyed for each transaction, T_i, in the tree-protocol:

(1) The first lock by T_i may be on any data item.

(2) Subsequently, a data item Q can be locked by T_i only if the parent of Q is currently locked by T_i.

(3) Data items may be unlocked at any time.

(4) A data item that has been locked and unlocked by T_i cannot subsequently be relocked by T_i.

A transaction trace is a sequence of [*lock*(X,Y)], [*unlock*(Z)] and other events of database items, where X, Y, and Z are database items and Y is the parent of X. Correct tree-protocol histories are specified by the following grammar on these events:

```
tree_protocol => not_until([unlock(_)],[lock(X,_)]), tree([(X,lock)]).

% State is a list of ordered pairs (Data_item,(lock or unlock)).
tree(State) => cond([lock(X,Y)], tree(update(State,X,Y)),
               cond([unlock(X)], tree(mark(State,X)),
               cond([end_of_file], terminate(State), ([_], tree(State))))).


update([(A,lock)|C],X,A) => [(X,lock),(A,lock)|not_member(C,X)].
update([(A,B)|C],X,Y) => if((X==A),fail,[(A,B)|update(C,X,Y)]).

mark([(X,lock)|Y],X) => [(X,unlock)|Y].
mark([X|Y],Z) => [X|mark(Y,Z)].

terminate([]) => [].
terminate([(_,unlock)|X]) => terminate(X).

not_member([],_) => [].
not_member([(X,A)|Y],Z) => if((X==Z),fail,[(X,A)|not_member(Y,Z)]).
```

Note that the state information ensures correct tree-protocol is unbounded which is stored in the argument of `tree`.

For other useful examples of stream pattern analysis, see [4].

5. Implementation

The main implementation issue in SPA is in finding a way to compile functional grammars and *match* to efficient transducers. A straightforward approach is to adapt the techniques of compilation of DCGs to Prolog [26], with certain modifications. The procedure for compiling DCGs to Prolog is simple. It translates the LHS and RHS of a grammar rule to the head and body of an appropriate Prolog clause. The basic idea is to add a difference-list to each nonterminal symbol giving the input and output streams. Here we present a simpler and more elegant algorithm for compilation of functional grammars to Log(F) transducers.

Algorithm 5.1

For each functional grammar rule $f(A_1,..,A_m)$ => RHS where f is an m-ary, $m \geq 0$, non-constructor function symbol and each of RHS and $A_1,..,A_m$ is a term, perform the following:

(1) If RHS is a variable or a simplified term, generate

$$f(A_1,\ldots,A_m,S) \Rightarrow \texttt{match}(RHS,S).$$

(2) If RHS is of the form $g(B_1,..,B_n)$ where g is an n-ary, $n \geq 0$, non-constructor function symbol and each of $B_1,..,B_n$ is a term, generate

$$f(A_1,\ldots,A_m,S) \Rightarrow g(B_1,\ldots,B_n,S).$$

where S is the input stream to be matched.

Definition 5.1

Let U, V be terms. We write

$$U \rightarrow V$$

if there is a subterm T of U, a Log(F) rule L => H, a most general unifier θ of T and L such that V is the result of replacing T in U by H and applying θ. In other words, if U = U[T] then V = U[H]θ. We say U rewrites to V in one step.

More generally we write

$$U \xrightarrow{k} V$$

if U rewrites to V in k steps, and

$$U \xrightarrow{*} V$$

if U rewrites to V in zero or more steps.

Theorem 5.1

Suppose F is a functional grammar term, and S is a stream. Let CF be the result of "compiling" F and S as follows:

$$CF = \begin{cases} f(T_1,\ldots,T_m,S) & \text{if } F = f(T_1,..,T_m) \text{ where f is not a constructor symbol} \\ \text{match}(F,S) & \text{if F is a variable or a simplified term} \end{cases}$$

If match(F,S) $\xrightarrow{n}$ R using the grammar, then CF $\xrightarrow{n}$ R using the compiled grammar where R is some suffix of S.

Proof:

By induction **on *n*, the length** of the reduction sequence of match(F,S).

Basis: $n = 1$

In this case, F must be either a variable or a simplified term [X|L] or [], since these are only two rules for reducing *match*. But then CF is just match(F,S), so CF $\xrightarrow{n}$ R as claimed.

Induction step:

Let $n = k+1$, and assume the theorem is true for all $n \leq k$.

If F is simplified, then CF=match(F,S), therefore, it is trivially true that both match(F,S) and CF reduce to the same term.

Let us consider the case where F is not simplified.
Here we assume match(F,S) $\rightarrow$ match(F',S) $\xrightarrow{k}$ R, so F $\rightarrow$ F'.
CF = f($T_1, \ldots, T_m$,S) where F = f($T_1, \ldots, T_m$).
Cases for F':

(1) F' is a variable or a simplified term.
Let CF' = match(F',S) where CF' is the result of "compiling" F' and S.
Since F $\rightarrow$ F', then L => H is in the grammar, where L=f($A_1, \ldots, A_m$), Lθ=Fθ and Hθ=F'. By (1) of algorithm 5.1, the compiled grammar rule corresponding to L => H is f($A_1, \ldots, A_m$,S) => match(H,S) where f($A_1, \ldots, A_m$,S)θ = CFθ, and match(H,S)θ=CF'.
Therefore, CF $\rightarrow$ CF', and inductively CF' $\xrightarrow{k}$ R, so CF $\xrightarrow{n}$ R.

(2) F' = g($Y_1, \ldots, Y_j$) where g is an j-ary, j $\geq$ 0, non-constructor function symbol.
Let CF' = g($Y_1, \ldots, Y_j$,S) where CF' is the result of "compiling" F' and S.
Since F $\rightarrow$ F', then L => H is in the grammar where L=f($A_1, \ldots, A_m$), H=g($B_1, \ldots, B_j$), Lθ=Fθ and Hθ=F'θ. By (2) of algorithm 5.1, the compiled grammar rule corresponding to L => H is f($A_1, \ldots, A_m$,S) => g($B_1, \ldots, B_j$,S) where f($A_1, \ldots, A_m$,S)θ = CFθ, and g($B_1, \ldots, B_j$,S)θ = CF'θ.
Therefore, CF $\rightarrow$ CF', and inductively CF' $\xrightarrow{k}$ R, so CF $\xrightarrow{n}$ R.

Therefore, both match(F,S) and CF reduce to R in k+1 reductions.

By induction, if match(F,S) $\xrightarrow{n}$ R using the grammar, then CF $\xrightarrow{n}$ R using the compiled grammar.

Q.E.D.

Example 5.1

Consider the functional grammar:

```
net_crash => ([net_failure]+,[cpu_failure]).
(X,Y) => append(X,Y).
(X+) => X.
(X+) => (X,(X+)).
```

This grammar is compiled to the following Log(F) transducers:

```
netcrash(S) => ',' ([net_failure]+,[cpu_failure],S).
 ',' (X,Y,S) => append(X,Y,S).
 '+' (X,S) => match(X,S).
 '+' (X,S) => ',' (X,(X+),S).
```

We have implemented a translator based on algorithm 5.1 to compile functional grammars to Log(F) transducers and the implementation is simpler than the procedure for compiling DCGs to Prolog. Theorem 5.1 ensures that the compiled Log(F) transducers reduce to the same term as the functional grammars. See appendix A for the implementation.

6. Conclusion

In this paper, we have given examples illustrating the expressive power of functional grammars for SPA. Regular expressions or path expressions specifying patterns can be easily expressed with functional grammars. Functional grammars are suitable for describing complex patterns in streams of event data, which cannot be conveniently specified by existing database query languages.

We have described an important application of functional grammars in specifying distributed systems. Concurrent execution is modelled by arbitrary interleaving of events from each process. There is a natural mapping between the propositional temporal logic and our event stream model. By predefining a set of grammar rules (corresponding to some common temporal operators) in the library, we can express temporal logic queries through patterns. Furthermore, a scheme exists that coroutines recognition of multiple patterns in an event stream.

This paper has concentrated on temporal logic, or 'history-oriented' specifications. However SPA can be used for many other types of specifications. In particular, it can be used in 'object-oriented' specifications. Given a set of actors (automata, concurrent objects, etc.) $A_1,...,A_n$ we can produce SPA grammars with starting patterns $S_1,...,S_n$ for these, and then use $S_1//.../S_n$ as a specification of valid histories for the entire system. SPA can therefore integrate both object-oriented and history-oriented specifications. To our knowledge, this aspect of the system is unique. Consider the following specification of a counter:

```
c(S) => cond([clear], c(0),
        cond([up], c(S+1),
        cond([down], c(S-1),
        cond([show], ([count_value(S)],c(S)), ([_],c(S)))))).
```

The point is that this specification does two things: First, it describes the behavior of a counter in terms of its reaction to specific messages. Second, it describes the sequence of valid messages a counter process history can contain.

Once a specification mechanism becomes sufficiently powerful, its 'expressive power' is no longer problematic, and other issues become important: elegance, usefulness in specifying real systems, tractability of inference problems used in verification, and so on. SPA grammars are very interesting in their properties here. Since they are grammatically based, they can be used for both acceptance and generation of histories. Acceptance is equivalent to *analysis* of histories, while generation is useful in producing hypothetical histories for *testing* [9]. Furthermore, since SPA grammars are ultimately translated to Prolog programs (in a way that does not make use of the meta-logical features of Prolog), it seems possible to perform *verification* of properties of these grammars using existing first-order logic theorem proving technology. Altogether SPA appears to hold many advantageous properties for specification.

The main implementation issue in SPA is in finding a way to compile functional grammars and *match* to efficient transducers. While grammar rules can act as pattern generators or specifiers, the transducer resulting from their compilation acts like a parser. We have described a simple algorithm to compile functional grammars to Log(F) transducers and proved that a functional grammar term together with the *match* transducer reduces to the same term that the compiled Log(F) transducer reduces to. We stress that all functional grammar specifications here are

executable. The expressiveness of functional grammars together with an implementation formalism make SPA a nice system for stream pattern analysis.

A great deal of work remains in exploring performance issues of implementing SPA. A major concern of optimization here is to reduce the degree of nondeterminism of the functional grammars. The "if" and "cond" formalism propose here is one step towards this direction. It is not difficult to observe that the order of the grammar rules is important for efficient processing of grammar reductions. Much more work needs to be done in designing heuristics to guide rule selection.

7. Acknowledgement

We thank Richard Muntz, Ted Kim, Lars Hagen, Richard Guy and Arman Bostani for their helpful comments.

[illegible] and A. Pnueli. Adequate [illegible] of Concurrent Programs. *Science of Computer Programming*, 139, Dec 1984.

[illegible] and D. S. Parker [illegible] A Computer Scientist [illegible]

Appendix A : Compilation of Functional Grammars to Log(F) Transducers

```
:- op(400,xfy,(=>)).

:- mode fg_rule_expansion(+,?).
fg_rule_expansion((H=>X),(H1=>match(X,S0))) :-
        X == [],
        !,
        macro(H,H1,S0).
fg_rule_expansion((H=>X),(H1=>match(X,S0))) :-
        var(X),
        !,
        macro(H,H1,S0).
fg_rule_expansion((H=>X),(H1=>match(X,S0))) :-
        functor(X,'.',2),
        !,
        macro(H,H1,S0).
fg_rule_expansion((H=>X),(H1=>X1)) :-
        functor(H,F,A),
        A1 is A+1,
        functor(H1,F,A1),
        arg(A1,H1,S0),
        fg_rule_copy_args(H,H1,0),
        functor(X,G,B),
        B1 is B+1,
        functor(X1,G,B1),
        arg(B1,X1,S0),
        fg_rule_copy_args(X,X1,0).

:- mode fg_rule_copy_args(+,+,+).
fg_rule_copy_args(F1,F2,I) :-
        I1 is I+1,
        arg(I1,F1,X),
        arg(I1,F2,X),
        !,
        fg_rule_copy_args(F1,F2,I1).
fg_rule_copy_args(_,_,_).

macro(H,H1,S0) :-
        functor(H,F,A),
        A1 is A+1,
        functor(H1,F,A1),
        arg(A1,H1,S0),
        fg_rule_copy_args(H,H1,0).

term_expansion(X,Y) :- fg_rule_expansion(X,Y).
```

References

1. Abelson, H. and G. Sussman, *The Structure and Interpretation of Computer Programs*, MIT Press, Boston, MA, 1985.
2. Berson, S., E. de Souza e Silva, and R.R. Muntz, "An Object-Oriented Methodology for the Specification of Markov Models," Technical Report CSD-870030, UCLA Computer Science Dept., Los Angeles, CA 90024-1596, July 1987.
3. Campbell, R.H. and A.N. Habermann, "The Specification of Process Synchronization by Path Expressions," *Lecture Notes in Computer Science*, vol. 16, pp. 89-102, Springer-Verlag, New York, 1974.
4. Chau, H.L. and D.S. Parker, "Functional Grammars: A New Formalism for Stream Pattern Analysis," Technical Report, UCLA Computer Science Dept., Los Angeles, CA 90024-1596, May 1988.
5. Clarke, E.M. and E.A. Emerson, "Synthesis of Synchronization Skeletons from Branching Time Temporal Logic," *Lecture Notes in Computer Science*, pp. 52-71, Springer-Verlag, 1981.
6. Clocksin, W.F. and C.S. Mellish, *Programming in Prolog*, Springer-Verlag, 1984.
7. Degano, P. and U. Montanari, "A Model for Distributed Systems Based on Graph Rewriting," *JACM* , vol. 34, no. 2, pp. 411-449, April 1987.
8. Ehrig, H., "Aspects of Concurrency in Graph Grammars," *Lecture Notes in Computer Science*, vol. 153, pp. 58-81, Springer-Verlag, Berlin, 1983.
9. Gorlick, M.D., C. Kesselman, D. Marotta, and D.S. Parker, "Mockingbird: A Logical Methodology for Testing," Technical Report, The Aerospace Corporation, P.O. Box 92957, Los Angeles, CA 90009-2957, May 1987. To appear, *Journal of Logic Programming*, 1988.
10. Hailpern, B. and S. Owicki, "Modular Verification of Computer Communication Protocols," *IEEE Trans. on Communications*, vol. COM-31, no. 1, Jan 1983.
11. Harel, D., "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming*, vol. 8, 1987.
12. Korth, H.F. and A. Silberschatz, *Database System Concepts*, McGraw-Hill, 1986.
13. Lamport, L., "Specifying Concurrent Program Modules," *ACM Transactions on Programming Languages, and Systems*, vol. 5, no. 2, pp. 190-222, April 1983.
14. LeDoux, C.H., "A Knowledge-Based System for Debugging Concurrent Software," Technical Report CSD-860060 (Ph.D. Dissertation), UCLA Computer Science Dept., Los Angeles, CA 90024-1596, March 1986.
15. Manna, Z. and P. Wolper, "Synthesis of Communicating Processes from Temporal Logic Specifications," *ACM Transactions on Programming Languages and Systems*, vol. 6, no. 1, pp. 68-93, January 1984.
16. Manna, Z. and A. Pnueli, "Adequate Proof Principles for Invariance and Liveness Properties of Concurrent Programs," *Science of Computer Programming*, vol. 4, no. 3, pp. 257-289, Dec 1984.
17. Muntz, R.R. and D.S. Parker, "Tangram: Project Overview," Technical Report CSD-880032, UCLA Computer Science Dept., Los Angeles, CA 90024-1596, April 1988.

18. Narain, S., "A Technique for Doing Lazy Evaluation in Logic," *J. Logic Programming*, vol. 3, no. 3, pp. 259-276, October 1986.

19. Narain, S., "LOG(F): A New Scheme for Integrating Rewrite Rules, Logic Programming and Lazy Evaluation," Technical Report CSD-870027, UCLA Computer Science Dept., Los Angeles, CA 90024-1596, 1987.

20. Narain, S., "LOG(F): An Optimal Combination of Logic Programming, Rewrite Rules and Lazy Evaluation," Ph.D. Dissertation, UCLA Computer Science Dept., Los Angeles, CA 90024-1596, 1988.

21. Owicki, S. and L. Lamport, "Proving Liveness Properties of Concurrent Programs," *ACM Transactions on Programming, Languages, and Systems*, vol. 4, no. 3, pp. 455-495, July 1982.

22. Parker, D.S., R.R. Muntz, and H.L. Chau, "The Tangram Stream Query Processing System," Technical Report CSD-880025, UCLA Computer Science Dept., Los Angeles, CA 90024-1596, March 1988.

23. Pereira, F.C.N. and D.H.D. Warren, "Definite Clause Grammars for Language Analysis," *Artificial Intelligence*, vol. 13, pp. 231-278, 1980.

24. Porto, A., "Two-level Prolog," *Proc. Intl. Conf. on Fifth Generation Computer Systems*, 1984.

25. Pratt, V., "Modelling Concurrency with Partial Orders," Technical Report STAN-CS-86-1113, Stanford Computer Science Depart., Stanford, CA 94305, June 1986.

26. Sterling, L. and E. Shapiro, *The Art of Prolog*, MIT Press, Cambridge, MA, 1986.

27. Vogt, F.H., "Event-based Temporal Logic Specifications of Services and Protocols," *IFIP*, 1982.

28. Wileden, G., "Relationships between Graphs Grammars and the Design and Analysis of Concurrent Software," *Lecture Notes in Computer Science*, vol. 73, pp. 456-463, Springer-Verlag, Berlin, 1979.

29. Wolper, P., "Specification and Synthesis of Communicating Processes using an Extended Temporal Logic," *Proc. 9th ACM Symposium on Principles of Programming Languages*, pp. 20-23, Albuquerque N.M., 1982.