LOGIC PROGRAMMING FOR REAL-TIME CONTROL OF TELECOMMUNICATION SWITCHING SYSTEMS

NABIEL A. ELSHIEWY

▷ An experiment using logic programming in the specification and implementation of a telecommunication switching system is reported, and one of the main modules in the system, a telephone-line controller, is described in detail as an illustrative example. The system is described in terms of transition relations in a labeled transition system. The programming language used is a variant of the parallel logic language PARLOG augmented with annotations to express timing constraints. The operational model of PARLOG is modified to handle time by allowing each goal-reduction process in a query to maintain its own logical clock, which can be read and set by the goal-reduction process itself. A metainterpreter is given to describe the operational behavior and an implementation scheme for the language. ◁

1. INTRODUCTION

A telecommunication switching system is a typical example of an embedded real-time computing system. It is embedded because the computing system is a part of a larger system and is in charge of controlling and monitoring physical processes and devices external to it. A real-time computing system often performs a number of parallel tasks under specified timing constraints interacting with its environment (subscribers, operators, and other exchanges). This interaction does not intentionally terminate as long as the system is in use. The system changes status according to input stimuli, producing output responses within finite and specifiable delays.

The conventional approach to describe such systems is based on finite-state transition-machine concepts. The system is first specified in terms of a state–transition-machine description. This description is then translated into explicit hand-coded programs, which undergo many optimizations for run-time processing. A state-machine description, although producing efficient programs, is hard to design and to

Address correspondence to Nabiel A. Elshiewy, Computer Science Laboratory, Ellemtel, S-125 25 Älvsjö, Sweden.

Received May 1987; accepted May 1988.

THE JOURNAL OF LOGIC PROGRAMMING

655 Avenue of the Americas, New York, NY 10010 0743-1066/90/$3.50

maintain, and modifying the description demands substantial effort. Algebraic models, e.g., CCS and CSP, provide sophisticated tools for the specification and analysis of such systems. They cannot, however, be used for producing efficient programs.

Kowalski's thesis "algorithm = logic + control" makes logic programming an attractive alternative for the specification and implementation of real-time control systems. The use of logic programming promises to allow us not only to describe, but also to code and possibly optimize, the implementations of a system in the same framework. One starts with an executable specification which can be transformed using automatic logical validation techniques into programs having the required run-time efficiency.

Logic programming is based on the concept of using predicate logic as a programming language [8] with the resolution principle [11] as the basic computational model. PROLOG was the first language to realize, in practice, the concept of logic programming. An excellent introduction to logic programming and PROLOG can be found in [13].

PROLOG has proved useful in a variety of computing applications, e.g., natural-language processing, deductive information retrieval, compiler writing, and knowledge-based systems. In recent years a great deal of research activity has been concentrated on investigating more efficient implementations of PROLOG and the design of special architectures which exploit the inherent parallelism in logic programs; see for example [18] for a survey.

Other investigations concern the extension of logic programming to express explicit parallel and communicating processes and to provide synchronization mechanisms between such processes. The most widely known family of these languages consists of those based on committed choice and stream AND parallelism. Concurrent PROLOG [14], GHC (Guarded Horn Clauses) [17], and PARLOG [2] are members of that family. An excellent overview of it can be found in [15].

At the Ericsson Telecom Computer Science Laboratory, interest in logic programming started in 1982, when a working implementation of the logic-programming language LPL [7] was released. A program written in LPL has been used to control a local telephone switching system (PABX). The interface with the hardware was defined with built-in predicates compiled in the LPL system. The experience gained from this experiment is reported in [3].

More experimental studies have been conducted since, in which we used both PROLOG and PARLOG on a larger scale than in the LPL experiment [1]. The conclusion was that the computational model of PARLOG (i.e., committed-choice stream AND parallelism) is more suitable for use in our application domain and that we needed to introduce explicit mechanisms to express timing constraints as well as the handling of erroneous and exceptional cases. It is also important to develop an environment and tools to support modular organization of programming telecommunication applications in the large.

The notion of explicit time is fundamental to distributed computations in general and to real-time computations in particular. An appropriate concept of time for distributed computing systems is that time is local for each process in the system and can be synchronized with the time on other processes.

The notions of time and event are closely related to each other. The conventional definition of an event is that it represents an action which can change the state of a

program. Considering the operational view of logic programming, we may intuitively assume that a transition from a reduction step in a goal-reduction process to the next reduction step represents an event and that a single goal-reduction process is defined as a set of events with a total ordering.

Real-time constraints are often tailored to efficiency and system performance requirements. Timing constraints may, however, be imposed on the behavior of both the system and its environment, making demands not only on the rate of system outputs (responses) but also on the rate of inputs (stimuli) coming from the environment. It is therefore necessary to be able to express maximum and/or minimum time limits on interprocess communications, that is, the allowed (maximum or minimum) time period between the occurrence of two inputs (stimuli) from the environment or between an output (response) from the system and the next input (stimulus) from the environment. It is also required to be able to delay an action for a given period of time or to wait until some specified time is reached.

This paper reports, through a real-world application, an experiment to introduce the explicit expression of time and timing constraints into the logic-programming framework. The computational model is based on the process interpretation model with nondeterministic committed-choice and stream AND parallelism in which each goal-reduction process in the computational network maintains its own logical clock which can be read and set by the goal-reduction process itself. The programming language is a variant of PARLOG augmented with operations to express timing information.

The rest of the paper is organized as follows: The programming language PARLOG-RT (Real-Time PARLOG) and its computational model are reviewed, followed by an overview of the labeled transition system model used to develop our real-time logic programs. An informal description of a simple telephone switching exchange is given. Timing primitives are introduced at their place of use during the process of describing a telephone-line controller, which is one of the main modules in a switching system. The module is described, using PARLOG-RT notation, in terms of transition relations in a labeled transition system. A metainterpreter for PARLOG-RT is then given to describe the operational behavior and an implementation scheme for the language.

2. THE LANGUAGE PARLOG-RT: AN OVERVIEW

As in PARLOG [2], a program definition is a set of guarded clauses. Using a syntax similar to that of DEC-10 PROLOG (in which a variable symbol begins with an uppercase letter and any other symbol begins with a lower case letter) a guarded clause has the general form:

Head : - Guard_Goals : Body_Goals

where **:** is the commit operator and the **Goals** are conjunctions of predicate calls. In a guarded clause, a conjunction of calls (**C1, C2**) will be reduced (evaluated) in parallel. In cases where it is important to control the order of evaluation (reduction), the sequential conjunction operator **&** may be used. In a conjunction of calls (**C1 & C2**), **C1** must successfully terminate before the evaluation (reduction) of **C1** starts. Shared variables in goals act as communication streams.

Only one guarded clause, among the alternative clauses which define a program, is chosen as a candidate to solve (reduce) a goal. Alternative guarded clauses in a program definition are separated by either a parallel search operator . or a sequential search operator ;. The last . acts as a program definition terminator. A set of .-separated clauses are searched in parallel to select a candidate clause. If two alternative clauses are separated by ;, the second clause will not be tried unless the first one has failed to be candidate. For example, suppose alternative clauses of a predicate are defined as follows:

Clause_One.

Clause_Two;

Clause_Three.

Then **Clause_One** and **Clause_Two** are first searched in parallel for a candidate clause. **Clause_Three** is tried for candidacy only if both fail.

For a program clause to be a candidate for selection, both unification of the clause head with the goal and reduction of all guard goals must succeed. Any communicated data between conjunctive goals are made visible to the receiving goal if and only if the reduction of the sending goal results in a commitment (selecting a goal clause).

Each program definition is accompanied by a mode declaration which defines for each argument in a clause head an access mode to that argument. A mode declaration has the form

mode P(M$_1$,...,M$_k$).

where **P** is the predicate name and each **M**$_i$ ($1 \leq i \leq k$) is either the symbol **?**, which means that the access to a variable appearing in the corresponding position is restricted to input, or the symbol **^**, which indicates that access to the variable is restricted to output.

Using mode declarations, each clause is compiled to a simpler kernel (standard) form. Input matching and output unification are performed by explicit primitive calls which are added to the guard and body respectively. This is done as a compilation phase in which guards are checked for safety. A safe guard is a guard that cannot bind variables which appear in input arguments.

In addition to asynchronous communication, the *back-communication* mechanism is provided to allow for synchronized communication, which is necessary for resource allocation and demand-driven input/output operations. The synchronization is provided by allowing an input message to be defined as a structure which contains an unbound variable among its components with the intention that the receiving goal process will instantiate this variable to a value. The value is then made accessible to the sending goal process.

What makes PARLOG-RT is the introduction of time into the computational model of PARLOG: Each goal-reduction process maintains its own local logical clock. A logical clock **C** is maintained by each goal-reduction process **G** and is represented by a perpetual successor function over natural numbers. The successor-function process is running in parallel with the goal-reduction process. The value of the clock **C** will change between reductions of the clock reduction process **C** itself, and changing **C**'s value does not itself constitute an explicit event in the goal process **G**

to which the clock **C** is attached. During goal reduction, each goal-reduction process can read its own clock through a display stream. To allow the handling of timing constraints, each logical clock is supplied with an alarm function which can be set and read by the owner goal-reduction process through the attached display stream during goal reduction.

A real-time computing system is modeled as a network of goal-reduction processes joined by communication links (shared variables). Each goal-reduction process executes an event-driven reduction algorithm where an event is abstracted either to the arrival of a message or to the attainment of a certain value by the clock maintained by the goal-reduction process. It is assumed that the different clocks, maintained by a conjunction of different goal-reduction processes, are running at the same rate and are synchronized to keep the same absolute time.

To read the current time of the clock during a goal reduction, the primitive `ctime` is introduced. It has the declaration

```
mode ctime(^).
```

Evaluating the goal `ctime(T)` succeeds with `T` instantiated to the value of current time read from the logical clock maintained by the goal reduction process. The goal `ctime(T)` can be called anywhere in a guard or a body part of a clause.

The time value of a logical clock, represented as a perpetual counter, is dependent on the type and speed of the processors used. Assuming that millisecond units are chosen to represent clock readings, a set of six different postfix operators are provided, for example, to express timing values in higher units: `sec`, `min`, `hr`, `day`, `week`, and `year`. For example, the goal

```
Fortnight is 2 week
```

is equivalent to the goal

```
Fortnight is (2*7*24*60*60*1000).
```

Two time guard primitives, to be used only in the guard part of a program clause, are provided to express timing constraints on input (stimuli). They are described in their place of use in the description of the main module of our telecommunication switching system. Other primitives to express timing constraints on the output (responses) of the system are described later, with examples showing their use in the timing of tasks in the switching system.

3. PROGRAMS AS LABELED TRANSITION PREDICATES

A promising approach to program development is to start with a rigorous specification of a system to be developed, in terms of executable descriptions of the system behavior. The specification is then subjected to a sequence of transformations to yield an efficient implementation of the system.

The approach used here to develop our programs is conceptually based on the structural approach to operational semantics [16]. The structural approach was originally developed to provide a simple formal framework in which programming-language semantics are specified in a way that reflects our intuitive understanding of the program execution. Only interesting execution steps are considered in such specifications, which are presented in the form of axioms and inference rules guided

by the structure of the language. The approach is based to a great extent on predicate logic and makes intensive use of pattern matching and unification, which makes logic-programming systems a natural tool for the development, execution, and validation of structural operational specifications.

In the structural operational approach, a system is described in terms of a labeled transition system. The behavior of a system (over discrete time intervals) is a sequence of configurations identified by a transition relation among configurations. The transition relation shows that when the system is in a configuration **C**, it can make a transition (a reduction step) to a configuration **C'**. The transition is normally caused by an action which gives information about what went on in the configuration during the transition (internal action) and/or about the interaction between the system and its environment (external action).

A composite system is specified in terms of the transitions of its components. The behavior of the individual components is considered to be a part of the system behavior. Parallelism is modeled in terms of nondeterministic interleaving. Transitions of several components can be combined into one system transition. Some transitions of an individual component are considered to be caused by communication actions with other components in the system. Actions related to communications between nodes in a distributed system are called labels. Nonterminating systems can simply be described in terms of recursive structures.

In the notation of PARLOG-RT a configuration is represented by a goal which comprises the state of the system and the relevant data at a particular instance of time. A transition relation is defined in terms of a predicate (program) in which the head of a clause represents the configuration before transition and the body represents the new configuration to which the system is supposed to make a transition. Alternative program clauses in a predicate definition represents the different possible transitions related to specific internal and/or external actions at any particular time. External actions are represented by pattern matching and unification of the arguments of a goal. Internal actions are guards and other local computations not observed by the environment. A composite system is defined in terms of a conjunction of goals.

4. A SIMPLE TELECOMMUNICATION SWITCHING SYSTEM

A telephone switching system is a network of telephone exchanges connected together in a variety of topological structures. The main task of a telephone exchange is to provide a subscriber to its services the ability to ring up any other subscriber connected to any of the exchanges (nodes) of the network by establishing direct connections between the subscriber terminals. Each subscriber terminal is connected to one of the (nearest) telephone exchanges by a pair of wires, referred to as the subscriber line. In addition to carrying speech signals, the subscriber line also conveys other signalling information to and from the subscriber terminal. The lines connecting exchanges to each other in the telephone network are referred to as trunk lines.

A simple model for a telephone exchange with distributed control is shown in Figure 1. A subscriber-line controller SC is provided for each subscriber terminal in the exchange. The subscriber-line controller responds to signals from the terminal

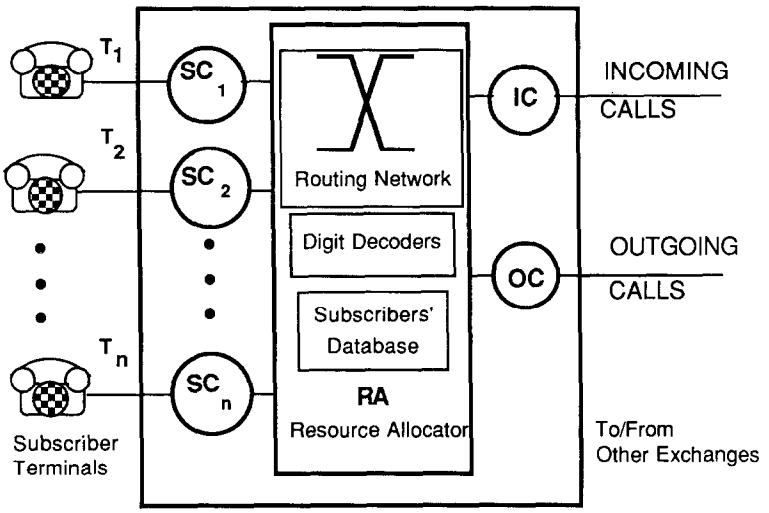

FIGURE 1. Structure of a simple telephone exchange.

and performs functions required, e.g., to set up and disconnect calls to and from the associated subscriber terminal. Two trunk-line controllers are also provided: The controller IC handles incoming call requests from other exchanges, while the controller OC handles outgoing call requests to other exchanges in the network.

To reduce the cost of establishing connections between lines, the exchange is equipped with a limited set of resources. There is, for example, a limited set of digit decoders, of which one is connected to a subscriber line to decode the signals coming from the terminal while the subscriber is dialing a number. The exchange also possesses a routing network to establish, on demand, direct speech-path connections between subscriber or trunk lines. To allocate one of these resources, subscriber-line controllers communicate their requests to the resource allocator RA. The resource allocator also keeps a record of information which is of global interest for the task of connecting subscribers: the local line numbers, the state of subscriber lines, charging information, etc.

5. THE SUBSCRIBER-LINE CONTROLLER

Each subscriber line has an identity, which corresponds to the telephone number, and the controller uses input/output streams to communicate with the subscriber terminal and the resource allocator respectively. This is specified as follows:

```
mode subscriber_controller( Line_Identity?,
                            From_Subscriber_Terminal?,
                            From_Resource_Allocator?,
                            To_Subscriber_Terminal^,
                            To_Resource_Allocator^    ).
```

Once a controller is created, it makes a silent transition to its initial configuration `idel`, which refers to the current state of the subscriber terminal:

```
subscriber_Controller(Ident, FromTerm, FromResA, ToTerm, ToResA) :-

    true :  idle(Ident, FromTerm, FromResA, ToTerm, ToResA).
```

There are two ways to initiate a call from the idle state:

The subscriber terminal sends an OFFHOOK signal to the controller (generated when the subscriber lifts the telephone handset). The request is validated first (the telephone bill may be overdue), and if accepted, the controller requests the connection of a digit decoder to the subscriber line. The line is referred to as an A party line.

A SEIZE signal is sent by the resource allocator (another subscriber requests communication). A ring current is sent to the subscriber terminal, at which the bell starts ringing to announce that communication is requested by another line. In that case, the subscriber line is referred to as a B party line:

```
mode idle(Ident?, FromTerm?, FromResA?, ToTerm^, ToResA^).

idle(Ident, [offhook|FromTerm], FromResA, ToTerm, ToResA) :-

    may_call(Type), var(FromResA):

        ToResA = [getDigD(Ident, Type)|ToResA1],

        a_party(Ident, FromTerm, FromResA, ToTerm, ToResA1).

idle(Ident, FromTerm, [seize|FromResA], ToTerm, ToResA) :-

    true : ToTerm = [ringcurrent|ToTerm1],

           b_party(Ident, FromTerm, FromResA, ToTerm1, ToResA).
```

The goal `var(FromResA)` in the first clause above expresses an explicit priority constraint imposed on the selection of a clause if both clauses are candidates at the same time. This means that completing a call connection in progress is favored over starting a new one.

6. THE CALLING A-PARTY-LINE CONFIGURATION

If the controller is in an A-party-line configuration and the connection of a digit-decoding resource is positively acknowledged by the resource allocator, the terminal receives a dial tone. If no digit decoder is available at the time of request, the subscriber is informed by a busy signal and is expected to replace the telephone handset, ending the service. The subscriber may also replace the handset spontaneously, which results in sending an ONHOOK signal from the subscriber terminal indicating that service is no longer needed. In that case, the controller releases any allocated resources, marks the subscriber line free, and makes a transition back to the `idle` configuration, waiting there for new requests to come. Note that the ONHOOK signal (replacing the handset) can be sent at any time during the call setup

procedure:

```
mode a_party(Ident?, FromTerm?, FromResA?, ToTerm^, ToResA^).
a_party(Ident, FromTerm, [done|FromResA], ToTerm, ToResA) :-
    true : ToTerm = [dialtone|ToTerm1],
           first_digit( Ident, FromTerm, FromResA, ToTerm1, ToResA,
                        Register).
a_party(Ident, FromTerm, [unable|FromResA], ToTerm, ToResA) :-
    true : ToTerm = [congestiontone|ToTerm1],
           terminate(Ident, FromTerm, FromResA, ToTerm1, ToResA).
a_party(Ident, [onhook|FromTerm], FromResA, ToTerm, ToResA) :-
    true : ToResA = [free(Ident)|ToResA1],
           idle(Ident, FromTerm, FromResA, ToTerm, ToResA1).
```

The reader may notice that there is no need to impose any priority constraints on the choice of a candidate clause here. If one of the first two clauses is a candidate clause and the third clause is also a candidate, it doesn't matter which one is chosen. The call service will be terminated in any case.

6.1. Signals Must Arrive in Time

The expected response from the subscriber terminal is to enter the first of the sequence of digits referring to the called-subscriber (the B-party-line) number. The digit is stored in a register, and its value is sent to the resource allocator for analysis. Because the controller has allocated a shared resource (the digit decoder), waiting for the subscriber to start dialing is limited to (say) 45 seconds. If the time limit elapses with no action detected from the subscriber side, the service is aborted and allocated resources are released. It can be also that the subscriber is provided a hot-line service, which means that if the subscriber hasn't started dialing a number within (say) 5 seconds, the controller will automatically request connecting the subscriber line to a predefined B-party line.

To express timeouts (maximum time limits) on input arguments of a clause, the `after` annotation is introduced, to be used only in the guard part of a clause:

```
mode after(?).
```

The appearance of a time-guard annotation `after(T)` in the guard part of a program clause will cause the resolving (candidate-clause selection) process to consider timing. `T` is instantiated to an integer value representing a time limit. The alarm function of the attached clock is set to signal an alarm when the time-limit period has elapsed.

Input matching and evaluation of other guard goals (if any) in the time-guarded clause are performed, in parallel, according to normal evaluation rules. If the clock signalled timeout while the input matching is suspended, the time-guarded clause is a candidate clause. In the presence of other guard goals in the clause, they must either succeed or suspend but not fail for the clause to be a candidate. If input matching succeeds first, the clause is a noncandidate clause.

We left our subscriber line controller in the state of waiting for the first digit to be dialed. The controller is provided with a register for storing the decoded number as dialed:

```
mode first_digit(Ident?, FromTerm?, FromResA?, ToTerm^,
                 ToResA^, Register?).
```

If the subscriber has started dialing, the dial tone is stopped, and after storing the dialed digit in the number register, the resource allocator receives the new value stored in the register for analysis. The controller then moves to the `next_digit` state configuration. The subscriber may also replace the handset, terminating the service. The dial tone is stopped and the resource allocator is requested to release any allocated resources and mark the subscriber line free. The controller moves back to the `idle` configuration, ready for new call requests:

```
first_digit(Ident, [digit(D)|FromTerm], FromResA, ToTerm, ToResA,
            Register) :-
    true : ToTerm = [stoptone|ToTerm1],
           store_digit(D, Register, Register1),
           ToResA = [analyse(Ident, Register1)|ToResA1],
           next_digit(Ident, FromTerm, FromResA, ToTerm1, ToResA1,
                      Register1).
first_digit(Ident, [onhook|FromTerm], FromResA, ToTerm, ToResA,
            Register) :-
    true : ToTerm = [stoptone|ToTerm1],
           ToResA = [free(Ident)|ToResA1],
           idle(Ident, FromTerm, FromResA, ToTerm1, ToResA1).
```

Two timeouts are watched: one for a hot-line connection if such service is provided, and one for the maximum time given to the subscriber to start dialing a number:

```
first_digit(Ident, FromTerm, FromResA, ToTerm, ToResA, Register) :-
    Hot_Limit is 5 sec, after(Hot_Limit), hot_line(B_party_No):
           ToTerm = [stoptone|ToTerm1],
           ToResA = [analyse(Ident, B_party_No)|ToResA1],
        next_digit(Ident, FromTerm, FromResA, ToTerm1, ToResA1,
                   B_party_No).
first_digit(Ident, FromTerm, FromResA, ToTerm, ToResA, Register) :-
    Dial_Limit is 45 sec, after(Dial_Limit):
           ToTerm = [timeout_tone|ToTerm1],
           ToResA = [release(Ident)|ToResA1],
           terminate(Ident, FromTerm, FromResA, ToTerm1, ToResA1).
```

If the subscriber is provided a hot-line service and no dialing takes place during 5 seconds, the dial tone is stopped and the predefined B-party-line number is sent to

the resource allocator to be analyzed and seized, if possible, for connection. If hot-line connection service is not available and 45 seconds have elapsed without dialing, the subscriber receives a timeout tone message and a request is sent to the resource allocator to release any allocated resources. The controller moves then to the configuration `terminate`, where the expected action is that the subscriber replaces the handset (an ONHOOK signal is received by the controller).

6.2. Digit Reception and Number Analysis

As shown above, the service can only proceed if a digit is received by the controller or the subscriber has requested a hot-line connection. The controller makes a transition to the `next_digit` reception configuration, in which it watches for the result of the number analysis and reacts accordingly.

The resource allocator receives the value of the number register, the number is analyzed, and the result of the analysis is sent to the subscriber controller. The possible results are: the subscriber has dialed a number which is not in use, the number is correct but the B-party line is engaged in another call, or the resource allocator is unable to find a free speech path connection in the routing network. In any of the three cases the controller responds with the appropriate message and makes a transition to the terminate configuration, waiting there for the subscriber to replace the handset:

```
mode next_digit(?, ?, ?, ^, ^, ?).
next_digit(Ident, FromTerm, [unused|FromResA], ToTerm, ToResA,
           Register) :-
    true : ToTerm = [recorded_message(unused_number)|ToTerm1],
           ToResA = [release(Ident)|ToResA1],
           terminate(Ident, FromTerm, FromResA, ToTerm1, ToResA1).
next_digit(Ident,FromTerm,[engaged|FromResA], ToTerm, ToResA,
           Register) :-
    true : ToTerm = [busytone|ToTerm1],
           ToResA = [release(Ident)|ToResA1],
           terminate(Ident, FromTerm, FromResA, ToTerm1, ToResA1).
Next_digit(Ident, FromTerm, [unable|FromResA], ToTerm, ToResA,
           Register) :-
    true : ToTerm = [congestiontone|ToTerm1],
           ToResA = [release(Ident)|ToResA1],
           terminate(Ident, FromTerm, FromResA, ToTerm1, ToResA1).
```

There are two other possible results of number analysis. The first is that the B-party-line number received is correct, the line is free, and the allocator succeeds in allocating a speech-path connection between the lines. Then the controller sends a ring tone to the associated subscriber terminal and makes a transition to the `connecting` configuration where the connection can be established. The second

possible result is requesting more digits to complete a B-party-line number:

```
next_digit(Ident,  FromTerm, [available|FromResA], ToTerm, ToResA,
           Register) :-
    true : ToTerm = [ringtone|ToTerm1],
           connecting(Ident, FromTerm, From ResA, ToTerm1, ToResA1).
next_digit(Ident, FromTerm, [more|FromResA], ToTerm, ToResA,
           Register) :-
    true : next_digit(Ident, FromTerm, FromResA, ToTerm1, ToResA1).
```

If more digits are required to complete the number or if the subscriber replaces the handset to terminate the call request, two clauses are defined similar to the first two clauses of **first_digit** above. The only difference is that there is no need to send a **stoptone** message to the subscriber terminal.

To enable releasing the allocated resources if the subscriber gives up the service request without replacing the handset, a time limit is imposed on waiting for the next digit to arrive, say, 20 seconds:

```
next_digit(Ident, FromTerm, FromResA, ToTerm, ToResA, Register) :-
    Dial_Limit is 20 sec, after(Dial_Limit):
           ToTerm = [timeout_tone|ToTerm1],
           ToResA = [release(Ident)|ToResA1],
           terminate(Ident, FromTerm, FromResA, ToTerm1, ToResA1).
```

If the time period has elapsed, the controller informs the subscriber terminal and requests the release of allocated resources, making a transition to the **terminate** configuration.

6.3. Connecting the Two Lines

The ring tone received at the A-party terminal informs the subscriber that it is ringing at the B-party line. It is expected that the called B-party will lift the handset to answer the call. In that case the A-party line controller receives an ONLINE signal from the resource allocator and then puts the line in a speech-condition state. The controller makes a transition to the **a_speech** configuration, in which the termination of service is watched for:

```
mode connecting(?, ?, ?, ^, ^).
connecting(Ident, FromTerm, [online|FromResA], ToTerm, ToResA) :-
    true : ToTerm = [speech_condition|ToTerm1],
           a_speech(Ident, FromTerm, FromResA, ToTerm1, ToResA).
```

If the called party didn't answer (e.g., nobody is home), it is not economic to leave the telephone ringing forever even if the A-party wants to. Therefore, a maximum time limit (say 5 minutes) is imposed on the ringing session so that resources can be released if call connection hasn't happened. The A-party line subscriber may also give up the request and replace the handset, in which case the call is terminated and

the controller makes a transition to the initial `idle` configuration:

```
connecting(Ident, [onhook|FromTerm], FromResA, ToTerm, ToResA) :-
    true : ToTerm = [stoptone|ToTerm1],
           ToResA A = [free(Ident)|ToResA1],
           idle(Ident, FromTerm, FromResA, ToTerm1, ToResA1).
connecting(Ident, FromTerm, FromResA, ToTerm, ToResA,
     Register) :-
    Ring_Limit is 5 min, after(Ring_Limit):
           ToTerm = [timeout_tone|ToTerm1],
           ToResA = [release(Ident)|ToResA1],
           terminate(Ident, FromTerm, FromResA, ToTerm1, ToResA1).
```

While the two lines are connected, the controller is in the `a_speech` configuration, expecting to receive an ONHOOK signal terminating the call and causing the controller to make a transition to the initial configuration `idle` and to inform the resource allocator to mark the line free:

```
mode a_speech(Ident?, FromTerm?, FromResA?, ToTerm^, ToResA^).
a_speech(Ident, [onhook|FromTerm], FromResA, ToTerm, ToResA) :-
    true : ToResA = [free(Ident)|ToResA1],
           idle(Ident, FromTerm, FromResA, ToTerm, ToResA1).
a_speech(Ident, FromTerm, [offline|FromResA], ToTerm, ToResA) :-
    true : watch_a_speech(Ident, FromTerm, FromResA, ToTerm, ToResA).
```

In many telephone systems, the B-party is allowed to replace the handset during a call without causing disconnection (termination) of the call. The B-party can lift the handset again and still be able to continue the ongoing call (e.g., the B-party may move to another room, using a different telephone set connected to the same line). In that case the A-party line controller receives an OFFLINE-signal and makes a transition to a `watch_a_speech` configuration:

```
mode watch_a_speech(?,?,?,^,^).
watch_a_speech(Ident, FromTerm, [online|FromResA], ToTerm, ToResA) :-
    true : a_speech(Ident, FromTerm, FromResA, ToTerm, ToResA).
watch_a_speech(Ident, [onhook||FromTerm], FromResA, ToTerm, ToResA) :-
    true : ToResA = [free(Ident)|ToResA1],
           idle(Ident, FromTerm, FromResA, ToTerm, ToResA1).
watch_a_speech(Ident, FromTerm, FromResA, ToTerm, ToResA) :-
    Watch_Limit is 90 sec, after(Watch_Limit):
           ToTerm = [timeouttone|ToTerm1],
           ToResA = [release(Ident)|ToResA1],
           terminate(Ident, FromTerm, FromResA, ToTerm1, ToResA1).
```

The A-line controller may receive an ONLINE signal announcing that the B-party is back with the headset in hand, leading to a transition back to the `a_speech` configuration. The B-party may also intentionally terminate the call by replacing the hand-set first. The A-party is then expected to replace the handset also. If for some reason the A-party didn't replace the hand-set and a time period of say 90 seconds has elapsed, then the allocated resources are released and the A-party is requested to terminate the call, that is, replace the handset.

7. THE CALLED B-PARTY-LINE CONFIGURATION

The subscriber line controller makes a transition from its `idle` configuration into a *B-party-line* configuration, caused by receiving a SEIZE signal from the resource allocator. The signal is sent if the resource allocator has found that the line is free and that a speech path could be reserved in the routing network. The B-party-line controller reacts by sending a ring current to the associated subscriber terminal. The B-party is expected to answer by lifting the handset (OFFHOOK signal), whereupon the ring current is switched off and the resource allocator receives an acknowledgement that the subscriber has answered. The new configuration means that the B-party is in a *speech* condition. It is also possible that the A-party replaces the handset, giving up the connection request and resulting in a disconnect command from the resource allocator. Then the ring current is switched off and the B-party-line controller makes a transition back to an `idle` configuration:

```
mode b_party(Ident?, FromTerm?, FromResA?, ToTerm^, ToResA^).
b_party(Ident, [offhook|FromTerm], FromResA, ToTerm, ToResA) :-
    true : ToTerm = [stopring|ToTerm1],
           ToResA = [answered(Ident)|ToResA1],
           b_speech(Ident, FromTerm, FromResA, ToTerm1, ToResA1).
b_party(Ident, FromTerm, [disconnect|FromResA], ToTerm, ToResA) :-
    true : ToTerm = [stopring|ToTerm1],
           idle(Ident, FromTerm, FromResA, ToTerm1, ToResA).
```

Being in a speech condition, the B-party may interrupt (or terminate) the call by replacing the telephone handset (ONHOOK signalled to the controller). A possible reconnection of the call (the B-party lifts the handset again) is awaited for a limited period of time (say 90 seconds):

```
mode b_speech(?,?,?,^,^).
b_speech(Ident, [onhook|FromTerm], FromResA, ToTerm, ToResA) :-
    true : ToResA = [interrupt(Ident)|ToResA1],
           watch_b_speech(Ident, FromTerm, FromResA, ToTerm, ToResA1).
b_speech(Ident, FromTerm, [disconnect|FromResA], ToTerm, ToResA) :-
    true : terminate(Ident, FromTerm, FromResA, ToTerm, ToResA).
```

To express minimum time limits on the interprocess communication, the time-guard annotation **before** is introduced:

```
mode before(?).
```

It is used only in the guard part of a program clause. A program clause which has a **before(T)** time-guard annotation in its guard part is searched for candidacy as follows: T is instantiated to an integer value representing a time limit. The alarm function is set to the given time-limit value. If input matching succeeds, the time-guarded clause is a candidate clause. Note that if there are other guard goals in the candidate clause, they must also succeed. If an alarm is signaled while input matching is still suspended, the time-guarded clause fails to be a candidate.

If the B-party lifts the handset (OFFHOOK) before the watching time limit has expired, the call can be reconnected again:

```
mode watch_b_speech(?,?,?,^,^).
watch_b_speech(Ident, [offhook|FromTerm], FromResA, ToTerm, ToResA) :-
    MaxTime is 90 sec, before(MaxTime), var(FromResA):
          ToResA = [answered(Ident)|ToResA1],
          b_speech(Ident, FromTerm, FromResA, ToTerm, ToResA1).
  atch_b_speech(Ident, FromTerm, [disconnect|FromResA], ToTerm, ToResA) :-
    true : ToResA = [free(Ident)|ToResA1],
          idle(Ident, FromTerm, FromResA, ToTerm, ToResA1).
```

The B-party may proceed to terminate the call by replacing the handset first. In that case the subscriber lines are disconnected either when the A-party replaces the handset or when the resource allocator discovers the intentions of the B-party (by watching the time from the reception of the INTERRUPT signal from the B-party until **MaxTime** has elapsed).

During this procedure, the B-party may lift the handset (OFFHOOK) with the intention to request a new call connection (identified in this case as an A-party). The OFFHOOK signal has, therefore, two different interpretations, depending on when it arrives. If the OFFHOOK signal arrives before timeout, it is interpreted as a request to reconnect a call in progress, If it arrives after timeout or call disconnection, it means a new call connection request. Note that if the OFFHOOK signal arrives after the time limit for watching has elapsed, it may be detected when the controller is back in the idle configuration. This motivates the introduction of the **before** time guard and the preference for its use instead of an **after** time guard.

We end up the description of our subscriber-line controller by showing the transition relation between a **terminate** configuration and an **idle** configuration. The expected actions are that the subscriber replaces the telephone handset and that the resource allocator will mark the subscriber line free:

```
terminate(Ident, [onhook|FromTerm], FromResA, ToTerm, ToResA) :-
    true : ToResA = [free(Ident)|ToResA1],
          idle(Ident, FromTerm, FromResA, ToTerm, ToResA).
```

There are many other features and services which may be offered by a subscriber-line controller and which are not mentioned here, e.g., call forwarding to another number, automatic callback to busy numbers, multiparty (conference) calls in which more than two subscribers are engaged in the call, charging and billing functions, etc.

8. MORE TIMING CONSTRAINTS

It can be necessary to delay an action for a given period of time or to wait until some specified time is reached. This usually affects the evaluation of goals in the body part of a program clause, that is, timing constraints on outputs (system responses).

If it is required to delay the evaluation of a goal for a given period of time relative to current time, the `delay` metapredicate is introduced:

```
mode delay(Period?, Goal?).
```

It takes as its first argument the delay-time value, and as its second argument a goal to be evaluated when the delay time is expired.

The `delay` metapredicate can be defined in terms of a time-guarded clause with an `after` time guard as follows:

```
delay(Period, Goal) :-
    after(period) : call(Goal).
```

The use of the `delay` metapredicate in our exchange is shown in the following example: Assume that subscribers are charged for telephone services four times a year. A recursively defined process, which handles collecting charging data and making the invoice in due time, may have the following conjunction of goals in the body of one of its program clauses:

```
Quarter is 13 week, delay(Quarter, mk_invoice(Line_Ident))
```

The first time the `delay` metagoal is reduced, it will wait for 13 weeks before the goal `mk_invoice` is evaluated (reduced). A further recursive call will cause another 13-weeks delay, causing the required effect.

There are cases where evaluating a goal has to wait until a prespecified time is reached. This can be expressed using the `delay` metapredicate after computing the delay time period relative to current time of starting goal reduction. It can, however, be convenient to provide another metapredicate to express such cases:

```
mode at(Time?, Goal?).
```

The goal `at(FutureTime, WaitingGoal)` waits until the value of the current time is equal to the value bound to `FutureTime` to reduce the `WaitingGoal`. The `at` goal fails if the value bound to `FutureTime` is less than current time; otherwise it always succeeds.

Assuming that time reference and conversion are provided, we can define a functor (data constructor)

```
date(870425,073000)
```

which represents the date of a day (a sequence of three digit pairs: year, month, and

day) and the time on that day (a sequence of three digit pairs: hour, minute, and second) at which a subscriber requested a *wake-up* service. The metagoal

```
at(date(870425,073000), ring(LineIdent,
recorded_message(wake_up)))
```

waits until the given time in the goal matches the current time and reduces itself to the goal given in its second input argument, which is reduced, with the effect of ringing the given subscriber line number to deliver a recorded *wake-up* message.

The `at` meta-predicate can be defined in terms of the `delay` metapredicate as follows:

```
at(AbsTime, Goal) :-
    true : ctime(TimeNow),
           relate(TimeNow, AbsTime, RelTime),
           delay(RelTime, Goal).
```

—assuming that the goal `relate(TimeNow, AbsTime, RelTime)` succeeds with `RelTime` instantiated to the value of converting `AbsTime` relative to `TimeNow`.

9. A METAINTERPRETER FOR PARLOG-RT

PARLOG-RT programs are expressed in an intermediate form between source and kernel PARLOG. A program in PARLOG-RT is defined by declaring its access modes, which are used, here, for syntactic check purposes. An input variable (a variable which occurs inside a head argument with input mode) may appear anywhere in the clause. The appearance of an output variable (the variable which occurs inside a head argument with output mode) in the guard part of a clause is illegal (output variables which are arguments of a guard call can be renamed and unified after commitment in the body part). This means that output unification may only appear in the body part of a clause. Instantiating input variables is achieved by input matching (one-way unification), while the instantiation of output variables employs full unification (using the full unification primitive = [5]).

9.1. Distributed Logical Clocks

Each goal-reduction process maintains a local logical clock:

```
mode clock(?).
clock(S) :-
    true : clock(0, S).
```

The logical-clock process takes a display stream `S` as an input argument and creates a clock which starts ticking from 0, and which interfaces its owner goal process

through the display stream S:

```
modeclock(?, ?).
clock(_, []).                          % discard the clock.
clock(C, [display(T)|S]) :-            % read current time.
    true : T=C, succ(C, Cs),
           clock(Cs, S).
clock(C, [set(T,Alarm)|S]) :-          % set and read Alarm.
    true : succ(C, Cs),
           alarm(T, Alarm),
           clock(Cs, S).
clock(C, S) :-                         % or just update the clock.
    var(S) : succ(C, Cs),
           clock(Cs, S).
```

The alarm function, provided with each logical clock, is specified as follows:

```
mode alarm(?, ^).
alarm(0, S) :-                         % signal timeout.
  true : S=signal.
alarm(C, Alarm) :-                     % count down the time.
    C > 0:
           pred(D, Cp),
           alarm(Cp, Alarm).
```

To synchronize a distributed system of logical clocks, assuming that the different clocks in the system are running at the same rate, the following relation can be considered as a specification of a simple synchronization algorithm:

```
mode synchronise(?, ?).
synchronise(_, []).                    % done.
synchronise(T, [S1|Rest]) :-           % create a logical clock.
    true : clock(T, S1),
           synchronise(T, Rest).
```

The predicate `synchronise` takes a value `T` and a list of display stream variables as input arguments, and is reduced to a conjunction of logical clock processes, each of which is initially set to the value `T` and is connected to the goal process which owns the clock through a display stream `S`.

In a real implementation, synchronizing clocks may require that, e.g., each clock process reliably broadcast its clock value to every other clock process in the network of conjunctive goals. There are many proposed clock synchronization algorithms which work even in the presence of faults; see e.g. [10].

The predicates succ and pred are assumed to be provided by the kernel system:

```
mode succ(C?,Cs^).
mode pred(C?,Cp^).
```

They output respectively the successor and the predecessor of a given input value. These relations resemble a counter connected to the ticks produced by a physical clock.

9.2. Goal Reduction

To reduce a goal clause P, a goal-reduction process and a logical clock are created to run in parallel and to be interfaced through a display stream S:

```
mode reduce(?).
reduce(P) :-
    true : clock(S), reduce(P,S).
mode reduce(?,^).
reduce(true, S) :-                    % An empty goal
    true : S=[].
reduce((P1, P2), S) :-                % Parallel conjunction
    true : S=[display(T)],
           synchronise(T, [S1, S2]),
           reduce(P1, S1),
           reduce(P2, S2).
reduce((P1 & P2), S) :-               % Sequential conjunction
    true : reduce(P1, S1)
           & reduce(P2, S2),
           concat(S1, S2, S).
reduce((ctime(T)), S) :-              % Read the clock
    true : S=[display(T)].
reduce(P, S) :-                       % System-defined goal
    built_in(P) : S=[], call(P).
reduce(P, S) :-                       % User-defined goal
    clauses(P, Clauses) :
           S=[display(T)|S1],
           resolve(P, Clauses, T, Body),
           reduce(Body, S1).
```

The following predicates are assumed to be provided by the kernel system:

```
mode clauses(P?,S^).
```

reads and collects a user-defined predicate P and produces at S the AND-OR tree

structure of that predicate. If an `after` or `before` goal appears in a guard part of a clause, the clause head takes the form `after(T,H)` or `before(T,H)` where T is the timing parameter and H is the clause head.

```
mode built_in(P?).
```

succeeds if P is a predefined predicate in the system.

```
mode call(P?).
```

is a metapredicate which is reduced to the given goal P. It succeeds if the reduction of P succeeds.

The reduction of an empty goal always succeeds. Each subgoal in a parallel conjunctive goal is reduced, independently in parallel, and a system of synchronized clocks is created in which each subgoal has a clock attached to it. In a sequential conjunctive goal, subgoals are reduced in sequence one after the other; each has a local display stream interfacing the clock. All local display streams are concatenated together into the clock display stream in the same order as the sequential conjunctive goals. The concatenation operator is coded as follows:

```
mode concat(?,?,^).
concat([], Ys, Zs) :-
    true : Zs=Ys.
concat([X|Xs], Ys, Z) :-
    true : Z=[X|Zs], concat(Xs, Ys, Zs).
```

Reading the time is achieved by sending a display-time command to the clock as a structure with a variable argument. Using the back-communication mechanism, the goal is reduced with the variable instantiated to the clock's current time value. A built-in system-defined goal is reduced by a metacall to it. In the case of a user-defined goal, only one clause, of all clauses defining the unifying predicate, must be chosen for reduction. This is resolved by nondeterministic guarded choice in which goal-head matching and the guard part of a clause must succeed for the clause to be a candidate of choice. It is assumed that the body of the clause which reports candidacy first is chosen for reduction. All other candidates are ignored.

9.3. Resolving Guarded Choice

It is required that all clauses which define a program be present at the time of interpretation and that they be joined to each other through the search operators (`.` or `;`) in an AND-OR tree structure. The search for a candidate clause is performed in parallel except if sequential search is forced by the sequential search operator `;`. For a single clause, the goal `new_vars` produces a new copy of the clause with new variables in it. The goal `commit` (described below) tries to unify the goal with the head of the copied clause and to reduce the guard part of the copied clause. If `commit` succeeds the goal, `resolve` returns the body of the copied clause:

```
mode resolve(?,?,?,^).
resolve(P, clause(C), T, Body) :-      % Single clause
    new_vars(C, Cn), Cn=(H:- G:B),
    clock(T, Sm), commit(P,H,G,Sm):
          Body=B.
```

```
resolve(P, (C1. C2), T, Body) :-      % Parallel search:
    resolve(P, C1, T, B):             % Resolve both the first
           Body=B.                    % and
resolve(P, (C1. C2), T, Body) :-      % the second in parallel
    resolve(P, C2, T, B):
           Body=B.
resolve(P, (C1; C2), T, Body) :-      % Sequential search:
    resolve(P, C1, T, B) :            % Resolve the first
           Body=B;
resolve(P, (C1; C2), T, Body) :-      % otherwise
    resolve(P, C2, T, B):             % the second
           Body=B.
```

9.4. Input Matching and Guard Evaluation

The choice of a candidate clause is committed to the clause of which both input matching and guard evaluation succeed first. Both input matching and guard evaluation proceed in parallel. To match a goal and a clause head we assume the following primitive is provided:

```
mode <=(Tl^, Tr?).
```

It is a matching (one-way unification) primitive which conveys only data directed from the right-hand-side term `Tr` into the left-hand-side term `Tl`. A call `Tl <= Tr` unifies `Tl` and `Tr` by binding variables in `Tl` so that `Tl` and `Tr` are syntactically identical. If the call could proceed only by binding variables in `Tr`, it suspends. The call fails if both terms are nonunifiable.

To handle timing constraints on the inputs the `commit` predicate is specified as follows:

```
mode commit(?,?,?,^).
commit(P, after(T, H), G, S) :-          % Late input matching
    true : S=[set(T, Alarm)|Sg],
           check_commit(Alarm, P, H,
                        G, Sg, timeout).
commit(P, before(T, H), S) :-            % Early input matching
    true : S=[set(T, Alarm)|Sg],
           check_commit(Alarm, P, H,
                        G, Sg, committed);
commit(P, H, G, S) :-                    % Otherwise, normal
    true : H <= P, reduce(G,S).
```

If matching a clause head with a calling goal is time-guarded, the `commit` predicate sets the clock alarm and checks the commitment. An `after` guarded commitment succeeds only if the alarm is notified first (time-out). A `before` guarded commitment succeeds only if both the matching and guard evaluation succeeded before the clock alarm. To match and evaluate the guards while watching the clock alarm the following predicate is defined:

```
mode check_commit(?,?,?,?,^,^).
check_commit(signal, _, _, _, _, Result) :-  % Alarm signals
    true : Result = timeout.
check_commit(_, P, H, G, S, Result) :-       % Matched in
                                               time
    new_vars((H,G), (Hn,Gn)),
    Hn <= P, reduce(Gn,S):
          H = Hn, Result = committed.
```

If the clock signals alarm, the first clause is a candidate to bind `Result` to `timeout`. If both matching and guard evaluation succeeded before any alarm was signaled, the output variable `Result` is bound to `committed` in the second clause. The reason for making a fresh copy `Hn` of the input argument `H`, with new variables in it, is to ensure the guard-safety property of PARLOG programs.

The evaluation of the guard call (`H <= P`) may result in binding variables in `H`, which is not allowed in PARLOG. As a consequence, variables in `G` must also be renamed in the same operation to ensure that any common variables in `G` and `H` will also be common after renaming. Making use of PARLOG's back-communication mechanism, the unification (`H = Hn`) in body part allows any possible variable substitutions to appear in `H` as an expected result of the input matching operation.

10. CONCLUSIONS AND FURTHER WORK

We have presented a language called PARLOG-RT which extends the parallel logic language PARLOG to enable logic programming of real-time computations. The operational model of the language extends that of PARLOG by allowing each goal reduction process, in a network of goals, to maintain its own logical clock. Temporal annotations, which make use of the time information available, were introduced to allow the expression of time and timing constraints on both the reception and sending of data between communicating processes (goals).

We have shown that the notions of time and event can be expressed in the framework of the process interpretation of Horn-clause logic with no need to introduce modal or temporal logic operators. A metainterpreter for the language PARLOG-RT has been defined to illustrate the computational model and to describe an implementation scheme for the language.

We believe that the system of distributed clocks can easily be mapped into any architecture which supports synchronized interprocess communication. A proof that the system of distributed logical clocks satisfies Lamport's clock conditions [9] for correctness and synchronization is given in [4].

The notation of the language PARLOG-RT has been used to describe a subscriber-line controller of a telephone switching system in terms of a labeled transition

system which truly captured the interesting aspects of the system behavior. We have used a pragmatic approach to the development of programs in our experiment. The PARLOG-RT interpreter has been modified in much the same way as the browsing simulator described in [6] to allow for user intervention to control and explore interesting transitions in the specified system behavior. In a program clause, actions which cause transitions between configurations can be annotated as user options. The annotated actions may be input data (stimuli) or time guards. Other types of actions can also be annotated as user options, if desired, as long as they are placed in the guard part of a program clause.

In running the specification, the interpreter displays, for each transition, a list of all possible actions which may cause a transition from the current configuration (the current goal to be reduced) to another configuration. After the user has chosen an action, the system makes a transition to the appropriate configuration, and a new list of possible actions is displayed. A user choice can be undone, so that the user is enabled to explore other possible transitions caused by different actions. This approach was also very helpful in debugging the system specification.

A drawback with this approach is that when the system and its interacting components grow large, the list of actions can be too long for users to be able to manage the exploring and validating of all possible transitions. A possible solution is the application of expert-system techniques along the lines of [12]. The domain-specific knowledge of the behavior of the system is stored in terms of suitable structure in a knowledge base. A specialized PARLOG-RT metainterpreter acts as an inference engine which through interaction with the knowledge base can automatically explore the possible transitions, possibly with guidance from a human application expert. When a satisfiable specification is obtained, the program is compiled and optimized into efficient code using special compilation algorithms and optimization techniques [5].

In the course of describing the telephone switching system, it was observed that because all subscriber and trunk lines are competing for the allocation of resources by sending request messages to the resource allocator, the allocator must use a fair strategy for responding to such requests. The natural way to forward the different requests is to merge all streams conveying those requests into one input stream to the resource allocator. If one stamps each request with the time of its arrival into the associated stream, a fair merge operator can be defined [4] to merge the requests for resource allocations in the order of their arrival times.

I wish to thank Bjarne Däcker, head of Ellemtel's Computer Science Laboratory, for encouragement and support. I am grateful to Seif Haridi for his help and valuable advice. I wish also to thank Steve Gregory for his careful reading of many drafts of the paper and his insightful comments, which led to many corrections and substantial improvements. Thanks are also due to Lars-Erik Thorelli, Ian Foster, Joe Armstrong, Carl W. Welin, and Peter "Per" Brand for reading and commenting earlier drafts of this paper. Special thanks to Robert Virding for a fruitful collaboration on the implementation of PARLOG and related issues.

REFERENCES

1. Armstrong, J. L., Elshiewy, N. A., and Virding, R., The Phoning Philosophers Problem or Logic Programming for Telecommunications Applications, in: *Proceedings of 3rd IEEE Symposium on Logic Programming*, Salt Lake City, 1986, pp. 28–33.

2. Clark, K. L. and Gregory, S., PARLOG: Parallel Programming in Logic, *ACM Trans. Programming Languages and Systems* 8(1):1–49 (1986).
3. Däcker, B., Elshiewy, N. A., Hedeland, P., Welin, C.-W., and Williams, M., Experiments with Programming Languages and Techniques for Telecommunication Applications, in: *Proceedings of 6th International Conference on Software Engineering for Telecommunication Switching Systems*, Eindhoven, 1986.
4. Elshiewy, N. A., Time, Clocks and Committed Choice Parallelism for Logic Programming of Real Time Computations, Research Report R-86013, Swedish Institute of Computer Science (SICS), 1986.
5. Gregory, S., *Parallel Logic Programming in PARLOG: The Language and Its Implementation*, Addison-Wesley, 1987.
6. Gregory, S., Neely, R., and Ringwood, G. A., PARLOG for Specification, Verification and Simulation, in: *Proceedings of the 7th International Symposium on Computer Hardware Description Languages and Their Applications*, North-Holland, 1985, pp. 139–148.
7. Haridi, S. and Sahlin, D., An Abstract Machine for LPL0, Research Report TRITA-CS-8302, Dept. of Telecommunications and Computer Systems, Royal Inst. of Technology, Stockholm, 1983.
8. Kowalski, R., Predicate Logic as Programming Language, in: *Proceedings of the IFIP 74*, North-Halland, 1974, pp. 569–574.
9. Lamport, L., Time, Clocks and the Ordering of Events in a Distributed System, *Comm. ACM* 21(7):558–565 (1978).
10. Lamport, L., Using Time Instead of Timeout for Fault-Tolerant Distributed Systems, *ACM Trans. Programming Languages and Systems* 6(2):254–280 (1984).
11. Robinson, J. A., A Machine-Oriented Logic Based on the Resolution Principle, *J. Assoc. Comput. Mach.* 12(1):23–41 (1965).
12. Sterling, L., Expert System = Knowledge + Meta-interpreter, Technical Report CS84-17, Dept. of Applied Mathematics, Weizmann Inst. of Science, 1984.
13. Sterling, L. and Shapiro, E., *The Art of Prolog: Advanced Programming Techniques*, MIT Press, 1986.
14. Shapiro, E., Concurrent Prolog: A Progress Report, *IEEE Computer* 19(8):44–58 (1986).
15. Takeuchi, A. and Furukawa, K., Parallel Logic Programming Languages, in: *Proceedings of the 3rd International Logic Programming Conference*, London, Lecture Notes in Comput. Sci. 240, Springer-Verlag, 1986, pp. 242–254.
16. Plotkin, G., A Structural Approach to Operational Semantics, Lecture Notes, DAIMI FN-19, Computer Science Dept., Aarhus Univ., 1981.
17. Ueda, K., Guarded Horn Clauses, in: *Proceedings of Logic Programming '85*, Lecture Notes in Comput. Sci. 221, Springer-Verlag, 1986, pp. 168–179.
18. Warren, D. H. D., Or-parallel execution models of Prolog, in: *Proceedings of TAPSOFT '87, International Joint Conference on Theory and Practice of Software Development*, Pisa, Italy, 1987, pp. 243–259.