Communicating Sequential Prolog

Xining Li, Brian Unger, John Cleary,
Greg Lomow and Darrin West
Department of Computer Science
Univerty of Calgary
Calgary, Alberta, Canada

ABSTRACT

Communicating Sequential Prolog (CSP') is a single-solution distributed logic programming language for discrete event simulation. Its primary goal is to speed up the execution of logic programs through the use of parallelism, while as far as possible preserving the semantics of standard Prolog. A CSP' program consists of a set of parallel processes, synchronized by simulation time and by message passing. The underlying interprocess communication mechanism is Time Warp. The pertinent features of parallel logic programing and Time Warp are described. The syntax of CSP' is introduced and the semantics of new predicates and their control structures are discussed. Examples are given to show the expressive power and simplicity of CSP'.

1. INTRODUCTION

Logic programing offers two kinds of parallelism: AND-parallelism is the parallel solution of more than one goal in a given goal sequence; and OR-parallelism is the parallel creation of many solutions for a given goal. These two kinds of parallelism are a consequence of nondeterminism in logic programming: we are free to choose any order in which to satisfy several subgoals in the body of a clause; and, when evaluating a selected subgoal, we are free to choose any clause which can match the subgoal.

There have been attempts to design systems using either one of these types of parallelism (or a combination of both)[Clark 1986, Shapiro 1983, Conery 1987]. OR-parallelism serves the same purpose in parallel that backtracking does in standard Prolog. Problems in the implementation of OR-parallelism are the combinatorial explosion in the number of processes and the representation of variable binding environments. AND-parallelism, although offering advantages such as being able to exploit parallelism in determinitic programs, has been difficult to implement due to the overhead involved in the handling of shared variable bindings and the problem in preserving the "don't know nondeterminism" semantics of logic programs. Proposed AND-parallel Prologs usually sacrifice the completeness of logic programs (not all possible solutions may be found) in order to minmize these overheads.

For example, in order to avoid variable binding conflicts, Concurrent Prolog[Shapiro 1983] and PARLOG[Clark 1986] adopt a model called Stream AND-parallelism. In this model, all subgoals in a goal sequence run in parallel and the bindings of shared variables are incrementally passed from a producer to consumers. In other words, only one of the processes, i.e., the producer, is eligible to bind a shared variable, and the other processes (the consumers) are suspended until the variable has been bound. However, the drawback of this model is that it is hard to implement nondeterminism. Therefore, the systems which exploit Stream AND-parallelism do not implement the conventional "don't know" nondeterministic semantics of logic programs and implement "don't care" nondeterminism, i.e., *committed-choice* OR-parallelism instead. *Committed-choice* OR-parallelism is implemented by using a guard in the body of each clause. For a selected subgoal, parallel processes are set up to evaluate the guards of all clause whose heads match the subgoal. If all guards fail, the subgoal fails. If more than one guard succeeds, exactly one of them is chosen to commit, by some mutual exclusion algorithm. Once a choice has been committed, there is no way to backtrack to find other solutions. Thus it is possible for a goal to terminate in failure even though there may be a successful refutation for the goal.

Parallel logic programming has been used for discrete event simulation[Cleary 1985, Broda 1984, Vaucher 1987]. Having its foundation in logic, a Prolog-like language encourages the programmer to describe problems in a manner that facilitates checking for correctness and consequently reduces the verification, i.e., the debugging, effort. Since most discrete event simulation models involve deterministic and sequential objects, and the most commonly used relations between those objects are the producer-consumer and the client-server relation, it seems that AND-parallelism offers potential speed up for discrete event simulation.

We focus our attention now on Time Warp[Jefferson 1985] as the communication mechanism underlying a parallel logic programming system. Time Warp implements Virtual Time using an optimistic mechanism which relies on generalised process lookahead and rollback. The definition of Virtual Time ensures that messages are received by a process in timestamp order. In a Time Warp system, each process charges ahead, blocking only when its input queue (which holds all the incoming messages with nondecreasing timestamp order) is exhausted. Whenever a message with a timestamp "in the past" arrives at a process's input queue, the Time Warp mechanism rolls the process back to a state with a time earlier than the late message. During the rollback the effect of any further messages sent during that period are cancelled and then starts the process forward again. Using the Time Warp mechanism to synchronize the execution of simulation components gives significant potential for achieving concurrency within distributed simulation[Berry 1986, Jefferson 1982, Li 1987].

This paper describes Communicating Sequential Prolog (CSP'), a single-solution distributed logical programming language for discrete event simulation. It uses a weak form of *committed-choice* called *committed-communication* together with explicit send and receive message passing primitives in order to obtain an efficient AND-parallel Prolog. Section 2 introduces the design goals, the syntax of CSP', the semantics of new predicates

and control structures, and the reasons behind these choices. Section 3 gives some examples to show the expressive power and the simplicity of CSP'.

2. COMMUNICATING SEQUENTIAL PROLOG

2.1. Overview

CSP' is an extension of Prolog for distributed discrete event simulation. Based on its declarative semantics, it encourages the programmer to describe problems in a manner that facilitates checking for correctness and consequently reduces the verification effort. It can be used both for model specification and program implementation.

Another major reason for choosing Prolog is that the backtracking facility of Prolog can be used as a part of the rollback mechanism in a Time Warp system. In order to achieve maximum speed up, the Time Warp mechanism uses a large amount of memory to remember the execution histories of a program. The major memory cost is due to saving old states. The Time Warp mechanism has no knowledge about what should be saved and what should not, and thus the entire data space of a process is saved for each snapshot. If a process manipulates a large amount of data, its memory will be exhausted very soon by saving several successive snapshots of its data space.

On the other hand, saving states is an inherent property of Prolog which uses a left-most-goal-first computation rule with depth-first search. When a goal fails, Prolog starts backtracking by traversing a sequence of goals in reverse, from right to left. As it backtracks over each goal, any variables which were bound during the evaluation of that goal are unbound. Then Prolog tries to re-evaluate the goal by choosing an alternative clause. Thus, the set of variables changed on a given computation path, i.e., the computation or process history, is completely captured by Prolog. Backtracking is very similar to rollback. By combining Prolog and the Time Warp mechanism, the cost of state saving can be reduced.

Even though CSP' exploits AND-parallelism in logic programs, it has the following major differencies with other AND-parallel logic programming languages, such as Concurrent Prolog and PARLOG.

1) CSP' is an object oriented language in that programs are decomposed into parallel processes which act as objects with their own local state.
2) CSP' preserves the sequential execution of a process and the nondeterminism local to a process.
3) Processes communicate only by message passing. There are no shared variables among processes.
4) CSP' uses *committed-communication* nondeterminism instead of *committed-choice* nondeterminism.

2.2. The Syntax and Semantics of CSP'

A CSP' program consists of a set of clauses (A simple syntax for CSP' in Extended Backus-Naur Form can be found in Appendix). Clauses are classified into sequential clauses and parallel clauses. A sequential clause has the same meaning and syntax as a clause in the standard Prolog. A parallel clause is a clause of the form

r(Name, Time, t1, ..., tk) ::- B1, B2, ..., Bn.

In the evaluation of a relation call r(n1,n2,...,nl), a match with the goal r is tried with the head of each clause in a program. If a match is found and the matched clause is a parallel clause, a new process is created and the goal succeeds immediately. This explanation only tells us that process creation is guaranteed, but it does not imply that the process will finally succeed. The "::-" notation is a control definition which is used to invoke a new logical refutation stream. On the other hand, if the matched clause is a sequential clause or an unit clause, the goal is evaluated as a normal Prolog procedure call.

A newly created process executes concurrently with other existing processes and it sequentially evaluates the goals in the tail of the parallel clause. Associated with each process is a Logical Clock. In the head of a parallel clause, there are two system defined arguments: the process name and the process creation time. The process name can be any meaningful literal which is used to direct communication. The creation time is an integer which represents a relative simulation time. When a process is created, its Logical Clock is set to sum of the creator's time and the creation time.

The CSP' programmer can give other arguments following these two. For example, one way to create a network with B-tree structure is as follows:

```
node(n(X), Ct, Layer) ::- create(X, Layer),
        /* some other conditions */.

create(_, 1).
create(X, L) :-
        Ln is X*2,
        Rn is X*2+1,
        L1 is L-1,
        node(n(Ln), 0, L1),
        node(n(Rn), 0, L1).
```

When we call the goal

?- node(n(1), 100, 4)

the root process creates process n(1) because the goal matches the head of the first clause which is defined as a parallel clause, and then terminates. Dynamically, process n(1) creates n(2) and n(3), process n(2) creates n(4) and n(5), and so on. Altogether fifteen processes are spawned by logical time 100. They have the same process body with different process names.

2.3. Built-in Predicates for Time and Communication

Once a simulation model has been decomposed into processes, one has to synchronize these processes so that events occur in a correct order. The first key notion is simulation time which determines the order in which events occur. In CSP', each process holds a local Logical Clock which runs its own simulation time. A process can increase its Logical Clock by the predicate

advance(T),

where T is an integer number representing a relative simulation time. If a process evaluates a goal advance(t1) at simulation time t2, then when the goal succeeds, the new simulation time t is defined as

t=t1+t2.

A process can get its current simulation time using

time(T)

which always instantiates variable T to the current value of the associated Logical Clock.

CSP' permits some limited nondeterminism both locally within a process and between processes. Within a

process backtracking is limited by a *committed-communication* mechanism using double-cut(!!). When this is encountered all choice points in the current process are committed. Any later attempt to backtrack to a double-cut causes it to succeed repeatedly.

Inter-process nondeterminism is controlled by interactions between double-cut and the try_send predicate. A try_send predicate sends a message from one process to another. If the message is received and the receiving process subsequently commits by executing a double-cut, then the system sends an acknowledgement back to the try_send, which succeeds and continues execution. While waiting for an acknowledgement, the sender does not remain idle. Rather it will backtrack and continue executing. This may involve further try_sends all waiting for acknowledgements. The first try_send which receives an acknowledgement causes the path leading to it to be re-executed and then execution continues from there.

This lookahead for try_sends allows some OR-parallelism in addition to the AND-parallelism between processes. It also requires an underlying mechanism for coordinating the commitment of processes and the acknowledgement of try_sends.

Four predicates are provided for inter-process communication. They are send(D,M), try_send(D,M), receive(S,M) and receive(S,M,T).

The predicate send is used for sending a message M to the destination D. The goal:

send(D, M)

succeeds if D is instantiated with the name of an existing process and M is instantiated with any data structure. The predicate send never blocks, it returns control to the next goal as soon as the message has been queued for subsequent transmission. Each message is stamped with the sender's name and the send time which is the Logical Clock value of the sender at the moment the message is sent. CSP' assumes that the transmission time of a message is zero, i.e., a message send at logical time t is guaranteed to arrive at the destination at the same logical time.

The predicate:

try_send(D,M)

sends a message M to a destination D and eventually expects an acknowledgement back from the system. If there is no such acknowledgement in the current input queue, the system delays the decision as to whether it succeeds or not and backtracks to evaluate other alternatives. When an acknowledgement comes back, the system will automatically direct the control to the successful path.

The predicate receive is used for receiving a message M from a source process S. All messages are queued in nondecreasing send time order. The goal:

receive(S, M)

tries to unify its arguments with the first message in the input queue, that is, unify S with the sender's name and unify M with the content of the message. If a unification is made, the message is removed from the input queue and the predicate succeeds. Otherwise the predicate fails. When a process executes a receive predicate at logical time t1 which receives a message with timestamp t2, then its new logical time t will be decided by the following formula:

$$t = \max(t1, t2)$$

Up to here, we can see that the simulation time of a process can be increased explicitly when the process calls the advance predicate, or implicitly when the process receives a message.

The predicate

receive(S,M,T)

is also used to receive a message M from a source S. However, when this predicate is called, the third argument T must be instantiated to an integer which indicates an interval of simulation time. Instantiating T to + means an infinite time interval. The predicate succeeds only when there is a unifiable message in the current input queue whose timestamp is less than or equal to the current simulation time plus T. Otherwise the predicate fails immediately.

3. EXAMPLES

3.1. Single server queueing model

Fig. 1 shows a typical single server queueing simulation model. Each arriving customer generates a successor and sends a request to a bank server, then the customer waits until receiving a result from the server and leaves the system. The bank serves customers in first-come-first-serve order. Requests, inter-arrival and service times are random.

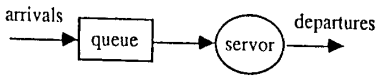

Fig. 1 A single server queueing model

```
customer(c(I), StartTime, EndTime)::-
        ranexp(0.125, Next), % negexp with mean 1/8
        generate_next(EndTime, Next, I),
        ranunif(Request),    % uniform 1-5
        send(server, Request),
        !!,
        receive(server, Result).

generate_next(EndTime, _, _):-
        time(T),
        T >= EndTime.        % stop generating
generate_next(EndTime, Next, I):-
        I1 is I+1,
        customer(c(I1), Next, EndTime).

server(server, StartTime)::-
        service.

service:-
        !!,
        receive(Customer, Request),
        process_request(Request, Result),
        send(Customer, Result),
        service.

process_request(1, open_account):- advance(10).
process_request(2, deposit):- advance(5).
process_request(3, withdraw):- advance(5).
process_request(4, cash_cheque):- advance(7).
process_request(5, credit_bill):-advance(15).
```

Suppose customers arrive from 8:30 until 15:00, the bank server starts serving from 9:00 until the last customer is served, the program can be invoked by

```
?- server(server, 540), customer(c(1), 510, 900)
```

where the time unit is minute.

3.2. Bank robbery

The following is a nondeterministic simulation model taken from T-prolog[Futo 1982]. Jim and Dick want to rob the Prolog savings bank. Jim climbs in the bank - it takes him 5 minutes, Dick waits outside. There are different safes in the bank. Jim and Dick cooperate to find the appropriate tool to open a safe. The robbery has to finish in 25 minutes. The question is which safe is to be chosen for a successful robbery.

```
jim(jim, 0)::-
        advance(5),             % climb_into_bank
        !!,
        receive(dick, Safe),
        open(Safe),
        time(T),
        T=<25.

open(milner):- advance(40).
open(wertherm):- advance(27).
open(chatwood):- advance(10).

dick(dick, 0)::-
        !!,
        has_tool(Safe),
        try_send(jim, Safe).

has_tool(milner).
has_tool(chatwood).
```

When we call

```
?- jim(jim, 0), dick(dick, 0)
```

the simulation tries to find the correct safe and tool by backtracking and attempting different safes with different opening times until one is found that can be opened in the time available.

3.3. Hierarchical Health Care System

The following is a model of health care typical of health delivery systems in developing countries.

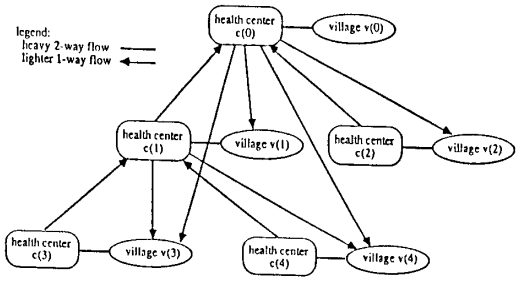

Fig. 2 A health care model

There are two kinds of objects in the above model: the villages and the health centers. The villages periodically generate patients, send them to the corresponding health centers and receive the treated people back. The health centers assess the incoming patients, if a person is treatable, then after the treatment, the person is sent back to his(her) village, otherwise, the person is sent to a higher health center.

```
create_system(5, _, _).
create_system(I, St, Et):-
        center(c(I), St),       % create Ith center process
        village(v(I), St, Et), % create Ith village process
        I1 is I+1,
        create(I1, St, Et).

/* village process */

village(v(I), St, Et)::-
        !!,
        random1(Next),          % uniform 20 to 100
        village_operation(I, Et, Q, Next).

village_operation(_, E, Q, _):-
        time(T),
        T>=E,                   % stop generating
        receive_rest(Q). % receive rest patients

village_operation(I, E, [[C, P]|T], N):-
        time(T1),
        receive(C, P, N), % receive a treated patient
        time(T2),
        TimeLeft is N-(T2-T1),
        !!,
        village_operation(I, E, T, TimeLeft).

village_operation(I, E, Q, N):-
        advance(N),
        send(c(I), [I, N]), % generate next patient
        random1(Next),
        !!,
        village_operation(I, E, Q, Next).

receive_rest([[C,P]|T]):-
        !!,
        receive(C, P),
        receive_rest(T).

/* health center process */

center(c(I), St)::-
        center_operation(I).

center_operation(I):-
        !!,
        receive(_, Patient),
        advance(2),  % assessment
        treat(I, Patient).

treat(I, [In, N]):-
        draw(N, N1),  % uniform 0 to 1
        treatable(I, N1), % if the patient is treatable
        random2(TreatTime), % uniform 5 to 10
        advance(TreatTime),
        send(v(In), TreatTime), % send the patient back
        center_operation(I).

treat(I, Patient):-
        parent_center(I1, I), % not locally treatable
        send(c(I1), Patient), % transfer to parent center
        center_operation(I).

parent_center(0, 1).
parent_center(0, 2).
parent_center(1, 3).
parent_center(1, 4).

treatable(0, _).        % all treatable in c(0)
treatable(1, T):- T=<0.75. % 0.75 treatable in c(1), c(2)
treatable(2, T):- T=<0.75.
treatable(_, T):- T=<0.5.  % 0.50 treatable in c(3), c(4)
```

This program can be invoked by using

?- create_system(0, 100, 1000)

which will create five village processes and five heath center processes. A village process stops generating patients when it reaches the end time and then goes to receive the rest treated patients; a center process repeatedly treats incoming patients. The program terminates when all process's simulation times become infinity.

4. CONCLUSION

Practical simulation involves defining a project and its goals, specifying the model, implementing it as a working computer program, verifying and validating the program, experimenting with the model, and producing documentation. The development of a large simulation model is a complex and difficult task. In many research areas, expensive computers are devoted almost exclusively to simulation. It is therefore becoming an increasing important research goal to speed up simulations by exploiting the concurrency inherent in them.

CSP' is a distributed logical programming language for discrete event simulation. It is an extension of standard Prolog combined with an optimistic communication mechanism - Time Warp mechanism. CSP' provides: the flexibility that comes from dynamic process creation; the concurrency that comes from asynchronous communication; an explicit naming facility that not only makes a program easy to understand but also makes it easier to develop large programs and libraries; and the expressive power that supports the description of concurrent operations, communication synchronizations, message segregation and nondeterministic computation. However, there are some open issues which need further investigation.

An experimental version of CSP' has been built in a distributed programming environment Jade[Joyce 1987, Unger 1986a, Unger 1986b]. Jade provides layers of tools for monitoring, debugging and graphically animating the execution of distributed programs. Our next step is to complete the implementation of CSP' and to measure the performances of CSP' programs.

APPENDIX

<clause>::=<parallel clause>.|<sequential clause>.
|<unit clause>.
<parallel clause>::=<parallel head>::-<tail>
<sequential clause>::=<sequential head>:-<tail>
<unit clause>::=<literal>
<parallel head>::=<functor>(<process name>,
<creating time>{,<term>})
<sequential head>::=<literal>
<tail>::=<literal>{,<literal>}
<process name>::=<literal>
<creating time>::=integer
<literal>::=<functor>(<term>{,<term>})
|<functor>
<functor>::= lower case identifier
<term>::=<constant>|<variable>|<list>|<literal>
<constant>::=integer|lower case identifier
<variable>::=identifier starting with an upper
case letter or a "_"
<list>::=[]|[<term>'|'<list>]

REFERENCE

[Berry 1986]
Berry O., "Performance evaluation of the Time Warp distributed simulation mechanism", Ph.D Dissertation, Univ. of Southern California, May, 1986

[Broda 1984]
Broda K. and Gregory S., "PARLOG for discrete event simulation" Research report DOC 84/5, U. of London, Mar., 1984

[Clark 1986]
Clark K.and Gregory S., "PARLOG: parallel programming in logic", ACM Tras. on programming languages and systems, Vol.8, No.1, Jan., 1986

[Cleary 1985]
Cleary J., Goh K., Unger B., "Distributed event simulation in Prolog" AI, Graphics, and Simulation, G. Birtwistle (Ed), pp 8-13, The Society of Computer Simulation

[Conery 1987]
Conery J. S., "Parallel execution of logic programs" KLUWER ACADEMIC PUBLISHERS, 1987

[Futo 1982]
Futo I. and Szeredi J., "T-Prolog: A very high level simulation system", Budapest, Oct., 1982

[Jefferson 1982]
Jefferson D. and Sowizral H., "Fast concurrent simulation using the Time Warp mechanism Part I: Local control", Tech. report, The Rand Corporation, Dec., 1982

[Jefferson 1985]
Jefferson D., "Virtual Time", ACM Tranc. on programming languages and systems 7(3), July, 1985

[Joyce 1987]
Joyce J., Lomow G., Slind K., and Unger B., "Monitoring distributed systems", ACM Tracs. on Computer Systems, Vol. 5, No. 2, May 1987

[Li 1987]
Li X. and Unger B., "Languages for Distributed Simulation" Proceedings of SCS Multiconference on AI and Simulation, Jan., 1987

[Shapiro 1983]
Shapiro E.Y., "A subset of Concurrent Prolog and its interpreter", Tech. rep. TR-003, ICOT, Tokyo, Feb., 1983

[Unger 1986a]
Unger B., Birtwistle G., Cleary J., and Dewar A., "A distributed software prototyping and simulation enviroment: Jade", Proc. of SCS Conference on Intelligent simulation Environments, San Diego, 1986

[Unger 1986b]
Unger B., Cleary J., Lomow G., Slind K., Xiao Z., and Li X., "Jade virtual time implementation manual", Res. Rep. 86/242/16, Department of Computer Science, University of Calgary, Calgary, Alberta, Canada, Oct., 1986

[Vaucher 1987]
Vaucher J. and Lapalme G., "Process-oriented simulation in Prolog" Proc. of SCS Multiconference on AI and Simulation, Jan., 1987