PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1988,
edited by ICOT.

AN EXAMINATION FOR APPLICABILITY OF FGHC: THE EXPERIENCE OF DESIGNING QUALITATIVE REASONING SYSTEM

Hayato Ohwada and Fumio Mizoguchi

Dept. of Industrial Administration
Science Univ. of Tokyo
Noda, Chiba, Japan

ABSTRACT

This paper examines the applicability of a parallel logic programming language FGHC (Flat Guarded Horn Clauses) through designing a large-scale AI application. Qualitative reasoning which we select as the application provides a powerful methodology for the next generation of AI systems. To achieve this goal, we carry out an experimental study for a step toward AI application within the framework of FGHC. The study consists of providing programming techniques for efficient search and exploring a possibility of parallel search. Furthermore, we describe a methodology for constructing a large-scale qualitative reasoning system in the parallel execution environment. The present study proposes a new methodology for employing parallel programming approach to practical AI applications in a reasonable way.

1 INTRODUCTION

Horn clause logic has been used for one of knowledge representation languages in Artificial Intelligence. It is capable of describing problems declaratively and solving the problems without specifying problem solving strategies. However, it cannot be applied to large-scale problems, since it is impossible for a programmer to specify control for searching complex search space efficiently.

In this paper, we try to apply a parallel logic programming language FGHC to designing a large-scale AI application. FGHC is a kernel language that allows parallel execution on the parallel inference machine being developed by ICOT. It is possible for users to describe important concepts for parallel programming; it is natural for expressing processes, communication and synchronization by introducing "guard" into a Horn clause program (Ueda 1985). Thus, FGHC allows a programmer to specify control for implementing efficient parallel search.

For the application, we select a qualitative reasoning system which is one of powerful tools for the next generation of AI systems. Qualitative reasoning provides a theory for predicting and explaining the behavior of a physical system (Bobrow 1984). Recent attempts to qualitative reasoning are to explore a framework for application of large-scale physical systems. The aim of using FGHC is not only increasing search speed but providing a methodology for developing large-scale qualitative reasoning systems.

In order to examine the applicability of FGHC, we carry out an experimental study for a step toward AI application. This study also aims to develop a parallel programming methodology for constructing large-scale AI systems. In the study, we show the expressive power of FGHC; FGHC provides control structure for efficient search, and parallel search in the parallel execution environment increases the efficiency of search. We also show usefulness of FGHC for applying qualitative reasoning systems to practical domains.

The next section shows the characteristic of qualitative reasoning. Section 3 presents control structure for efficient search in FGHC. In this section, two different search strategies are investigated: backtracking search and forward checking. Section 4 describes using domain-specific knowledge for improving the efficiency of the search programs. The section also includes a computation result of programs. Section 5 shows the advantage of parallel search. Section 6 describes a methodology for constructing a large qualitative reasoning system. Section 7 describes its related work. The final section has the conclusions.

2 QUALITATIVE REASONING

Qualitative reasoning provides a theory for predicting and explaining how a physical system works and for analyzing what its function is. The property of qualitativeness is useful not only for predicting the mechanism's behavior with incomplete knowledge but for producing plausible explanation about physical causality. Unlike the previous expert systems based on a shallow model, qualitative reasoning is capable of providing knowledge of deep model about the

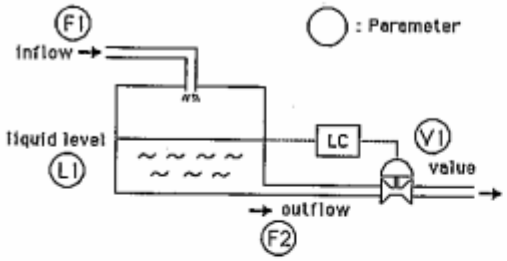

Figure 1: A model of buffer tank

```
simulate(T,F1,F2,DF,L1,V1) :-
    quasi_static(F1,F2,DF,L1,V1), !.
simulate(T,F1,F2,DF,L1,V1) :-
    new_state(T,F1,F1'), ...,
    new_state(T,V1,V1'),
    add(DF',F2',F1'), ...,
    mon_increase(L1',V1'),
    next_time(T,Next),
    simulate(Next,F1',F2',DF',L1',V1').
```

Figure 2: Prolog program of qualitative simulation

mechanism.

Predicting the behavior is called qualitative simulation. The precise definition and algorithm of qualitative simulation was described in (Kuipers 1985). We give an intuitive explanation in order to point out the characteristic of qualitative simulation.

In qualitative simulation, the structure of a physical system is described in terms of parameters and constraints among parameters. This description corresponds to qualitative version of differential equations. Each parameter has landmark values for capturing change qualitatively. Given the structural description, qualitative simulation is achieved by predicting possible next states of the parameters and checking the constraints. An important point of qualitative simulation is the non-determinism in predicting next states of a parameter. This non-determinism is due to the locality of prediction in which a next state of the parameter is determined by the current parameter value and direction of parameter change. Suppose the parameter A takes on the interval between the two landmark values a1 and a2 such that a1<a2, and the parameter is increasing now. Qualitative simulation predicts three possibilities: the parameter remains on the interval between a1 and a2, takes on the value a2 or takes a new value a* such that a1<a*<a2. These possibilities are pruned by constraints. Constraints are used for selecting plausible states of the system. Thus, qualitative simulation is regarded as a constraint satisfaction procedure.

The complexity of qualitative simulation depends on the number of parameters and constraints. Suppose there are n parameters and m constraints, we consider tree search as a typical search strategy. If no constraint prunes tuples of states, the complexity is exponential. If constraints work effective and one state is possible, the complexity is $o(m)$. In any case, tree search must search alternative paths exhaustively. The aim of our using a parallel language is to reduce large search space by searching alternative paths in parallel.

Figure 1 shows a model of simple buffer tank. This model is used for describing methods for efficient search in FGHC throughout the paper. The model has the following constraints:

`add(DF,F2,F1)`: The flow rate DF is the difference between the inflow F1 and the outflow F2.
`mul(L1,V1,F2)`: The liquid level L1 multiplied by the relative shaft position of the valve V1 is approximately the outflow F2.
`deriv(L1,DF)`: The flow rate DF is the derivative of the liquid level L1.
`mon_increase(L1,V1)`: The relative shaft position V1 is proportional to the liquid level L1.

Figure 2 shows a Prolog program implementing qualitative simulation for the buffer tank model. The predicate `simulate` searches a possible behavior by repeating prediction (performed by `new_state` predicate) and filtering (performed by the predicates associated with constraints). If directions of all changes are steady, the predicate `quasi_static` succeeds and simulation stops.

However, this program obtains only one behavior. Since qualitative simulation must predict all possible behaviors, exhaustive search is required. We design FGHC programs that track all possible behaviors exhaustively to be described in the next section. These programs simulates OR-parallelism by using AND-parallelism in FGHC.

3 QUALITATIVE SIMULATION IN FGHC

In this section, we explore methods for efficient search of qualitative simulation in FGHC. The methods can be viewed as transformation techniques that compile Horn clause programs to all-solution FGHC programs. These techniques also play a role of preserving declarative nature of logic programming in solving constraint satisfaction problems. Although the proposed techniques are used for solving constraint satisfaction problems only, developing general transformation systems may be interesting and possible.

The important point for implementing qualitative simulation in FGHC is to compile generators into all-solution predicates. Since FGHC is a committed-

```
simulate(T,F1,F2,DF,L1,V1) :-
    new_states(T,F1,F1s,[]), ...,
    new_states(T,V1,V1s,[]),
    <distribute DFs, F2s and F1s>,
    <invoke add(DF',F2',F1')>,
    <distribute L1s and V1s>,
    <invoke mul(L1',V1',F2')>,
    <invoke deriv(L1',DF')>,
    <invoke mon_increase(L1',V1')>,
    next_time(T,Next),
    simulate(Next,F1',F2',DF',L1',V1').
```

Figure 3: Scheme of backtracking search

choice language, non-deterministic procedures must be compiled to deterministic ones. For example, the original goal new_state(T,F1,F1') may be compiled to new_states(T,F1,F1s,[]), where the third and fourth arguments are d-list expressing possible next states.

The second point is to prepare some predicates that distribute the elements produced by generators for constraint checking. Before doing this, every element must be instantiated to ground terms in order not to make different bindings to the same variable. This is due to the operational semantics of FGHC. We remove the restriction by invoking AND-parallel processes that have no interaction except collecting solutions in searching alternative path of a tree. In other words, each process has no uninstantiated variables except ones used for collecting solutions. To satisfy this condition, we represent all variables in a constraint satisfaction problem as ground terms. Applying this representation to qualitative simulation, qualitative states must be represented as ground terms.

Introducing the above remedy, it is straightforward to implement qualitative simulation in FGHC. We below describe programs that simulate two types of search strategies: backtracking search and forward checking. These strategies are typical constraint satisfaction procedures.

3.1 Backtracking search

Backtracking search can be simulated by generating possible next states and checking whether each state satisfies all constraints. Since the generator part constructs lists of the states, the tester part distribute elements of the lists until all parameters included in a constraint are individuals. For example, to check the constraint add(DF,F2,F1), the element for each three parameters are distributed.

However, the distribution process need not continue until all of the parameters are individuals; otherwise the complexity will be exponential to the number of parameters. The important point for simulating backtracking search efficiently is interleaving constraints into distribution processes as soon as all arguments in a constraint become individuals. The resulting scheme is shown in Figure 3. The generator part consists of the predicate new_states which returns next states of a parameter to the third argument. The constraint part puts forward the distribution processes associated with constraints.

```
simulate(T,F1,F2,DF,L1,V1,S0,S1) :- true |
    quasi_static(F1,F2,DF,L1,V1,Ret),
    c0(Ret,T,F1,F2,DF,L1,V1,S0,S1).
c0(true,_,F1,F2,DF,L1,V1,S0,S1) :- true |
    S0=[(F1,F2,DF,L1,V1)|S1].
c0(false,T,F1,F2,DF,L1,V1,S0,S1) :- true |
    new_states(T,F1,F1s,[]), ...,
    new_states(T,V1,V1s,[]),
    c1(F1s,F2s,DFs,L1s,V1s,T,S0,S1).
c1(_,_,[],_,_,_,S0,S1) :- true | S0=S1.
c1(F1,F2,[DF|DFs],L1,V1,T,S0,S2) :- true |
    c2(F1,F2,DF,L1,V1,T,S0,S1),
    c1(F1,F2,DFs,L1,V1,T,S1,S2).
                    :
c4(F1,F2,DF,L1,V1,T,S0,S1) :- true |
    add(DF,F2,F1,Ret),
    c5(Ret,F1,F2,DF,L1,V1,T,S0,S1).
c5(true,F1,F2,DF,L1,V1,T,S0,S1) :- true |
    c6(F1,F2,DF,L1,V1,T,S0,S1).
c5(false,_,_,_,_,_,_,S0,S1) :- true | S0=S1.
                    :
c14(F1,F2,DF,L1,V1,T,S0,S1) :- true |
    next_time(T,Next),
    simulate(Next,F1,F2,DF,L1,V1,S0,S1).
```

Figure 4: FGHC program of backtracking search

Figure 4 shows a program that simulates the backtracking search. The predicate quasi_static checks whether all parameters are stable then returns its result to the six argument. If the result is true, one solution is put into d-list (S0, S1) by the predicate c0; otherwise simulation proceeds. The predicates c1 and c2 correspond to the distribution processes for the constraint add(DF,F2,F1). The predicate add has an extra argument which is used for returning the result of constraint checking. If a possible state satisfies add constraint, the predicate c6 is invoked to examine whether the state satisfies the remaining constraints or not, an empty solution is returned. Finally, if the state satisfies all of the constraints, the predicate c9 is executed and the predicate simulate is called recursively.

In this program, search of alternative paths is done by distributing processes for each path. For example,

```
simulate(T,F1,F2,DF,L1,V1) :-
    new_states(T,F1,F1s), ...,
    new_states(T,V1,V1s),
    <distribute DFs and F2s>,
    <invoke add'(DF',F2',F1s,F1s')>,
    <invoke deriv'(L1s,DF',L1s')>,
    <distribute L1s'>,
    <invoke mul'(L1',V1s,V1s')>,
    <invoke mon_increase(L1',V1s',V1s'')>,
    <distribute F1s' and V1s''>,
    next_time(T,Next),
    simulate(Next,F1',F2',DF',L1',V1').
```

Figure 5: Scheme of forward checking

the predicate c1 searches alternative paths by invoking the goals:

```
c2(F1,F2,DF,L1,V1,T,S0,S1),
c1(F1,F2,DFs,L1,V1,T,S1,S2)
```

where the goals connect each other through d-lists (S0, S1, S2). This view may be derived from the characteristic of FGHC as a process description language. A process has a candidate solution and returns the solution when no more constraints exist. This manner contributes to the simplicity of describing search problems.

In addition to process distribution, process termination can be described in a simple way. In FGHC, process termination is expressed by predicates that have no body goals except output unifications. Output unifications are explicit descriptions for returning a solution or pruning a candidate. In most procedural languages, handling a solution is implicit and is needed to use special operations.

3.2 Forward checking

Forward checking is a procedure that can prune search space by making active use of constraints, while backtracking search uses constraints in a passive manner. Unlike backtracking, this procedure checks constraints without waiting for all variables to be instantiated (Van Hentenryck 1987). In forward checking, constraints are checked when only one variable is left uninstantiated, and possible values that fail to meet the constraints are eliminated. For example, the constraint add(DF,F2,F1) can be evaluated as soon as two variables are instantiated, and possible values of the remaining variable are reduced.

Forward checking can be implemented in a similar way as the backtracking search. Distribution processes are also used. Unlike the backtracking search, constraints play a role of not determining the satisfiability of the constraints, but reducing possible values of parameters. The scheme of forward checking is shown in Figure 5. After distributing each candidate corresponding to the two parameters DF and F2, the constraint add(DF,F2,F1) tries to reduce possible states of the parameter F1. We rewrite the constraint add(DF,F2,F1) to add'(DF',F2',F1s,F1s'), where the third argument F1s is used for input and the fourth F1s' is used for output. The constraint deriv(L1,DF) is also checkable without interleaving distribution processes, because a state of the parameter DF is determined.

```
                    :
c2(F1s,F2,DF,L1,V1,T,S0,S1) :- true |
    c20(DF,F2,F1s,F1s'),
    c3(F1s',F2,DF,L1,V1,T,S0,S1).
c20(_,_,[],S) :- true | S=[].
c20(DF,F2,[F1|F1s],S0) :- true |
    add(DF,F2,F1,Ret),
    c21(Ret,F1,S0,S1),
    c20(DF,F2,F1s,S1).
c21(true,F1,S0,S1) :- true | S0=[F1|S1].
c21(false,_,S0,S1) :- true | S0=S1.
c3([],_,_,_,_,_,S0,S1) :- true | S0=S1.
c3(F1,F2,DF,L1s,V1,T,S0,S1) :- F1\=[] |
    c30(L1s,DF,L1s'),
    c4(F1,F2,DF,L1s',V1,T,S0,S1).
c30([],_,S) :- true | S=[].
c30([L1|L1s],DF,S0) :- true |
    deriv(L1,DF,Ret),
    c31(Ret,L1,S0,S1),
    c30(L1s,DF,S1).
c31(true,L1,S0,S1) :- true | S0=[L1|S1].
c31(false,_,S0,S1) :- true | S0=S1.
                    :
```

Figure 6: FGHC program of forward checking

A FGHC program simulating forward checking is shown in Figure 6. Predicates not included in the program are the same as that of the program simulating backtracking search. In the program, predicates that invoke the constraints add'(DF',F2',F1s,F1s') and derive'(L1s,DF',L1s') constraints are only included.

The predicates denoted by c_i (i is a positive integer) create a process network in which reduced values flows from a constraint to a constraint. This computation mechanism is provided by the stream-parallel computation model of FGHC. Based on parallel execution, each constraint reduces possible values in a pipelining manner. For example, the process c30(L1s,DF,L1s') which checks the constraint deriv(L1,DF) generates possible values to the argument L1s', then sends them to processes of the con-

straint mul'(L1',V1s,V1s'). Since a list of possible values are incrementally constructed from the head, constraint checking performs in parallel. This computation mechanism is useful for not only reducing computation time but implementing sophisticated search strategies. In forward checking strategy, cooperative search among constraints is possible and effective by communication via shared variables.

4 USE OF DOMAIN KNOWLEDGE

The programs presented in Section 3 are based on sequential constraint checking. Their computation complexities depend on the ordering of constraints. In qualitative reasoning, determining the ordering of constraints is called causal reasoning or causal ordering. Causal reasoning provides an intuitive explanation about the behavior of a system. It is an important point for developing computer programs that understand the mechanism.

Although causal reasoning does not intend to predict behaviors, it contributes to the efficiency of prediction. Since parameter values may be determined by causal reasoning, ordering relations among constraints improve the efficiency of the constraint satisfaction procedure. Causal reasoning is also viewed as constraint propagation which specifies dataflow from determined values to undetermined values.

In the buffer tank model, causal reasoning may be achieved as follows:

```
The liquid level is the same as the normal
and the inflow to the tank is greater than
normal.
            ↓    mon_increase(L1,V1)
The shaft position is the same as the
normal.
            ↓    mul(L1,V1,F2)
The outflow is the same as the normal.
            ↓    add(DF,F2,F1)
The flow rate is greater than the normal.
            ↓    deriv(L1,DF)
The liquid level will increase.
```

The first statement is an initial condition as input to simulation. Using the constraints, parameter values are determined qualitatively. This type of determination cannot be seen in quantitative simulation.

Causal reasoning involves non-deterministic behavior, though it was done deterministically in the above. To determine the ordering of constraints in general, some assumptions or heuristics must be introduced. This is due to the locality of the constraint propagation method. Therefore, the ordering obtained by constraint propagation is not always the best.

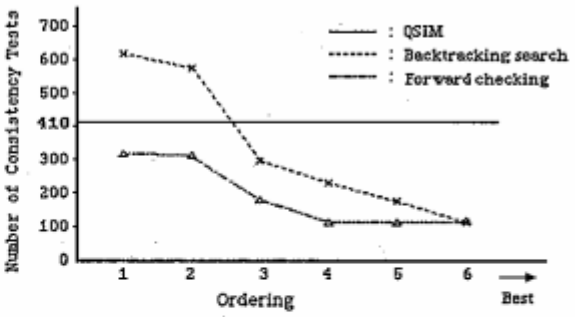

Figure 7: Performance of the two search strategies

Programs that simulate constraint satisfaction procedure are viewed as a realization of control, while causal reasoning is used as logic for improving the greater efficiency of the programs. We incorporate domain-specific knowledge as logic into the transformation systems. This way is regarded as a realization of the separation between logic and control (Kowalski 1979).

Figure 7 shows the performance data of the two search strategies in different orderings. Six ordering cases are measured for each program. The left case is the worst ordering and the right case is the best one. The number of consistency tests means total count of all constraint checking. A horizontal straight line indicates the number of consistency tests in QSIM which is a typical qualitative simulation system. QSIM does not depend on the ordering of constraints, since its constraint satisfaction is to generate the set of tuples of states and filters for each constraint.

As Figure 7 shows, better ordering achieves search more efficiently. In backtracking search, the best ordering improved the efficiency of the program by almost 5 times compared with the worst ordering. Forward checking also gained the efficiency by 3 times. This efficiency was brought by using the technique of causal reasoning. The difference in search strategies is clearly shown. Forward checking has smaller consistency tests compared with the backtracking search except for the best ordering. This observation shows the advantage of forward checking.

The advantage of our programs is that the programs prune possible solutions that fail to meet constraints before the remaining constraints are checked, while the disadvantage is that duplicate computation exits. If causal reasoning determines incorrect ordering, pathological behavior will be brought. However, the determination of the ordering is almost good as shown in Figure 7. Forward checking is efficient than QSIM even in the worst case.

Qualitative Model	Backtracking search		Forward checking	
	A*1	B*2	A	B
1	7	10	7	9
2	10	12	9	11
3	20	22	17	18
4	23	17	23	14
5	46	39	35	27
6	43	62	41	57

*1 A = Number of consistency tests per constraint
*2 B = Parallelism (= reductions/cycles)

Figure 8: Parallelism of the programs

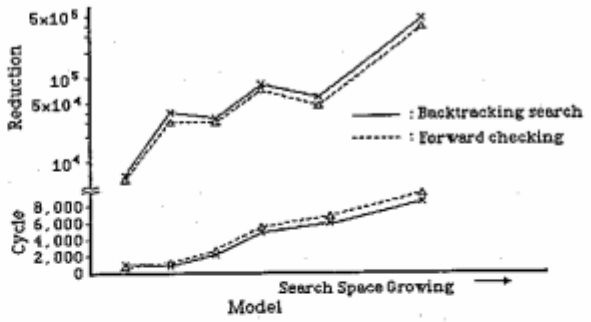

Figure 9: Reductions and cycles

5 ADVANTAGE OF PARALLEL SEARCH

This section shows the advantage of parallel search through performance data based on a parallel execution model. The model is a breadth-first execution model of programs, where all reducible goals are reduced simultaneously. In this model, reductions and cycles are useful factors for analyzing parallelism of a program. Reductions stand for the total number of goals reduced. A cycle means a simultaneous reduction of goals. Parallelism is given by the ratio of reductions to cycles.

Since the backtracking search program has no process interaction, parallelism of the program depends on the number of distributed processes (or alternative paths of a search tree). In the forward checking program, processes for checking constraints communicate, and therefore, the program involves two types of parallelism: parallelism of alternative search and that of cooperative search among constraints. However, the parallelism of the cooperative search is much smaller than that of alternative search. Thus parallelism of the two programs is approximately proportional to the number of alternative paths.

We assume that parallelism of the programs is proportional to the number of consistency tests for each constraint, since parallelism can be regarded as the average of total reduced goals. This assumption may be proved to be almost correct from the result shown in Figure 8. Figure 8 shows the measurement result on various qualitative models[1]. Parallelism of the programs corresponds to the number of consistency tests per constraint. This observation suggests that parallel search may obtain the greater efficiency for solving large constraint satisfaction problems. Figure 9 shows reductions and cycles of the programs for the same models as those in Figure 8. Figure 9 illustrates the advantage of parallel search in which parallel execution makes constraint satisfaction procedure dramatically faster than sequential execution.

[1]These models are obtained from medical domain. They are Starling equilibrium, blood pressure regulation, intraocular pressure regulation, etc. The first two are developed by Kuipers and Kassirer. The third is developed by us.

6 QUALITATIVE REASONING FOR REAL

This section describes a methodology for constructing a large-scale qualitative reasoning system in the parallel execution environment. For this purpose, we must consider the following points:

- System decomposition, and
- Stepwise refinement.

The first point means that a large-scale system consists of a number of sub-systems. In order not to make prediction of behaviors intractable, we must decompose the whole system. The second one is derived from the fact that it is difficult to build a proper qualitative model simulating the real-world. We must construct a number of consistent (not true) models and refine them into better models.

The system decomposition requires collecting the qualitative model for each component. Since each model consists of a set of parameters and constraints among them, we must collect parameters and constraints by identifying some parameters and generating constraints from component connections. However, this way cannot be applied straightforward. As a model gets large, great many constraints are generated. This is due to complete instantiation for predicting behaviors of the whole system. Intuitively, we are not concerned with the whole system for analyzing the system's behavior. We focus on components that are influenced by a given disturbance only, then aggregate to the total behavior that includes various disturbances. Thus, we developed an aggregation method that selects constraints by propagating a given disturbance. Based on this method, a number of partial

models are generated from disturbances. This means that the total behavior is predicted by simulating each model independently. The source of parallelism exits in this aggregation method.

We have built a model of intraocular pressure regulation for glaucoma diagnosis (Ohwada et. al. 1988). Glaucoma is an eye disease characterized by elevated intraocular pressure. In this model, there are 10 components. After 272 constraints were generated for the whole system, we selected 14 constraints for a disturbance. For glaucoma diagnosis, five disturbances are needed to obtain behaviors about primary glaucoma[2]. Therefore, qualitative simulation can be executed in parallel where five simulation processes are invoked. Since the degree of parallelism for each process is 21 (51080/2419)[3], 15 (79077/5202), 36 (134483/3750) and 82 (590818/7181), total parallelism becomes the sum of parallelism for each process.

The stepwise refinement requires management of multiple models. Better models can be obtained by analyzing models and observations. If the current models are inconsistent for a new observation, they are refined into several models and qualitative simulation is invoked for each model. Although multiple models are activated and simulation is invoked in parallel, multiple models must be managed for making model selection tractable. For this purpose, a model selection procedure is introduced for managing multiple models by communicating with the models.

Figure 10 shows a program of managing multiple models. The three processes are firstly invoked. The first process refines a model and invokes multiple models. The second is a manager for controlling multiple models. The third is used for getting observations from a user. In this program, the first and the second processes are regarded as search process and control process respectively. A communication channel represented by the variable HSG works as a medium between a search process and a control process. After simulation, the search process sends a message to the control process. This message consists of behaviors and uninstantiated variable Control which is instantiated by the control process. If no behavior is consistent with observations, the search process stops and this model is elimilated; otherwise, the search process refines the current model and distributes new models. The control process selects the best model among multiple models and sends a message to the model. This message is of a form:

```
fork(New,Others)
```

where New is a channel for communicating with the model and Others means channels for the others. The

[2]Primary glaucoma is a category of glaucoma.

[3]The left number means reductions and the right indicates cycles.

```
:- model(Model,HSG,[]),
   manager(HSG,OB),
   instream(OB).
model(Model,H0,H1) :- true |
    simulate(Model,Behaviors),
    H0=[(Behaviors,Control)|H1],
    model(Control,Model).
model(terminate,_) :- true | true.
model(fork(H0,H1),Model) :- true |
    refine_model(Model,NewModels),
    distribute_models(NewModels,H0,H1).
manager(Hsg,OB0) :- true |
    choose_best(Hsg,OB0,OB1,Best,Others),
    send_msg(Best,Others,OB1).
send_msg((_,Control),Others,OB) :- true |
    Control=fork(New,Others),
    manager(New,OB).
```

Figure 10: Scheme of multiple models

control process takes initiative in message passing; it sends a new channel for communicating with distributed search processes. Thus, management of multiple models can be achieved within the program.

7 RELATED WORK

We have developed a control strategy for managing search processes. In this framework, it is possible to focus on possible alternatives simultaneously and control search processes for efficient search (Ohwada 1987b). The main idea of the method is to separate a control process from search processes. The control process sends a message with communication channel to a search process, then the process is executed dependently to the message. This method decreases parallelism of original search programs, but it plays a role of preventing the search process from extra forking. This method is directly applicable to our programs. Since the qualitative reasoning system predicts behaviors of a physical system by using local interpretation in general, there are a number of similar behaviors. If a user want to aggregate them at an intermediate stage, a global checking mechanism is required. The control process is useful for this purpose.

Although we describe qualitative simulation from the viewpoint of constraint satisfaction in Section 3, we have adapted it to a distributed model. A physical system may be decomposed of several parts, and the parts are regarded as small physical objects. In this view, objects interact with each other. We have described the interaction of physical objects as a parallel model in which objects change their states through message passing (Ohwada 1987a).

The recent direction of qualitative reasoning research is to set up large-scale systems. Falkenhainer and Forbus (1988) focused on a modeling process in which multi-grain, multi-slice model of a Navy propulsion plant was used. Kuipers and Barleant (1988) presented a method for incrementally exploiting incomplete quantitative knowledge by using it to refine behaviors. Our approach to designing large-scale systems takes the same direction as these methods. We focus on the control structure of parallel qualitative reasoning to improve the efficiency of the system. We are not concerned with qualitative reasoning methods for reducing search spaces.

8 CONCLUSIONS

In this paper, we have examined the applicability of a parallel logic programming language FGHC through the experience of designing a large-scale qualitative reasoning system. This examination consists of providing programming techniques for efficient search, exploring a possibility of parallel search, and developing a methodology for constructing large-scale systems. The examination has been performed by applying to practical domains such as medical domain.

The significance of the present paper is summarized as follows:

- FGHC provides expressive power for implementing efficient search strategies such as backtracking search and forward checking. This feature is useful for solving general search problems.
- Domain-specific knowledge can be naturally incorporated into search programs, since FGHC provides two aspects of logic programming: logic and control. Design of transformation systems contributes to the separation between logic and control.
- Parallel search in the parallel execution environment dramatically increases search speed. The advantage of parallel search provides orders of magnitude computational power of parallel machine.
- System decomposition and stepwise refinement involve a large degree of parallelism. Parallel programming approach is useful as a methodology for constructing large-scale qualitative reasoning systems.

Constructing parallel systems is an important research for applying to a practical domain in which a large amount of computation is required. Our target of AI application, qualitative reasoning, is such a case. We assert that parallel programming methodologies are constructed by attacking large-scale problems, not by dealing with toy-problems. The present study proposes a new methodology for employing parallel programming approach to practical AI applications in a reasonable way. The experience given in the paper presents a starting point for the future direction.

ACKNOWLEDGEMENTS

We wish to express our thanks to Kazuhiro Fuchi and Shun-ichi Uchida for providing us with the opportunity to use FGHC systems. Special thanks are due to Koichi Furukawa and other members of ICOT Working Group FAI.

REFERENCES

[1] Bobrow, D. G., Qualitative Reasoning, *Artificial Intelligence*, Vol. 24, Special Issue, 1984.

[2] Falkenhainer, B. and Forbus, K. D., Setting up Large-Scale Qualitative Models, *Proc AAAI-88*, pp. 301-306, 1988.

[3] Kowalski, R., Algorithm = Logic + Control, *Comm. ACM*, Vol. 22, pp. 424-436, 1979.

[4] Kuipers, B., The limits of qualitative simulation, *Proc. Ninth International Conference of Artificial Intelligence*, pp. 128-136, 1985.

[5] Kuipers, B. and Berleant, D., Using Incomplete Quantitative Knowledge in Qualitative Reasoning, *Proc AAAI-88*, pp. 324-329, 1988.

[6] Ohwada, H. and Mizoguchi, F., Qualitative Simulation in Parallel Logic Programming, *Proc. Fourth Symposium on Logic Programming*, pp. 480-489, 1987.

[7] Ohwada, H. and Mizoguchi, F., Managing Search in Parallel Logic Programming, *Proc. Logic Programming'87*, K. Furukawa, H. Tanaka and T. Fujisaki (eds.), Lecture Notes in Computer Science, 315, Springer-Verlag, pp.148-177, 1988.

[8] Ohwada, H., Mizoguchi, F. and Kitazawa, Y., A Method for Developing Diagnostic Systems based on Qualitative Simulation, *Journal of Japanese Society for Artificial Intelligence*, Vol. 3, No. 5, pp. 617-626, 1988.

[9] Ueda, K., Guarded Horn Clauses, *Proc. Logic Programming'85*, E. Wada (Ed.), Lecture Notes in Computer Science, 221, Springer-Verlag, pp. 168-179, 1986.

[10] Van Hentenryck, P. and Dincbas, M., Forward Checking in Logic Programming, *Fourth International Conference on Logic Programming*, pp. 229-256, 1987.