A Simulator of the OR-Parallel Token Machine

by

Bogumil Hausman

SICS R86002
Research Report
ISSN 0283-3638

A Simulator of the OR-Parallel Token Machine

Bogumil Hausman

Swedish Institute of Computer Science*
Stockholm, Sweden

Abstract

This report is mainly meant as the documentation of the simulator of the OR-Parallel Token Machine for Horn Clause programs. The simulator has been used to investigate the dynamic characteristics of pure Horn Clause programs and to evaluate several storage structures. We start by briefly describing the virtual machine. Then we discuss the merits of a specification language Meta IV and a programming language, Simula, and also show transformations between those two. Finally, we describe the constituent parts of the simulation system, namely the underlying message passing mechanism and the three components of the machine: instruction processor, token pool and storage. The mapping of the specification into Simula shows the power of using object oriented languages for implementing abstract specifications and for simulation purposes.

* The main part of this work has been done while the author was working at The Royal Institute of Technology, Department of Computer Systems, Stockholm, Sweden

Contents

1. Introduction

Due to the advances in microcomputer technology, new machine architectures are being developed to support parallel execution of logic programs having potential parallelism. Different machine architectures has been proposed [1,2,3,4]. A suitable solution for running programs with a moderate degree of parallelism and of moderate size is proposed in [5,6,7,8,9]. Although directed towards the parallel execution of logic programs, it is believed to have a set of properties common to all parallel machines. The proposed architecture consists of a set of processing elements (PEs) connected by an order preserving packet interconnection network. Each PE consists of an instruction processor implementing an instruction set, a static memory unit holding a copy of the program to be executed, and a dynamic memory unit.

In order to evaluate the efficiency of the proposed architecture, simulations of the model were carried out. First a model of the basic machine was built, and then it was augmented by more advanced models of storage [8,11]. This report is meant as the documentation of the simulator used in [10] and [11].

2. A Short Overview of the Virtual Machine

An Or-parallel computation can be visualised as an unlimited number of independent processes, one for each alternative branch in the search tree of a program, sharing storage for binding environments and programs (Fig.1).

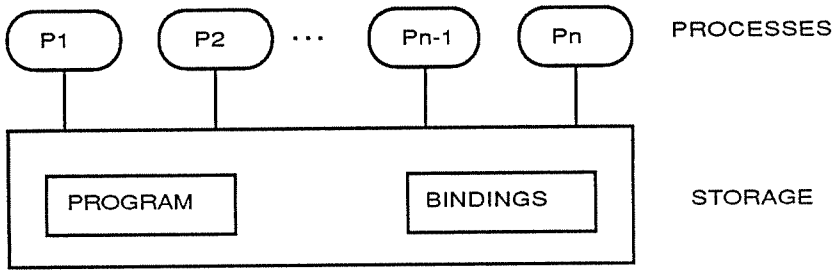


Fig.1 An Or-parallel process model.

In the virtual machine for Or-parallel execution of logic programs, the unlimited number of processes are mapped onto a finite number of processors. On this conceptual level the machine, as depicted in Fig.2, consists of a token pool, a set of processors and storage. The storage is divided into a static memory for programs and a dynamic memory for the binding environments and control information. Tokens in the pool represent processes which are ready for execution but are not allocated a processor. Processors execute processes as prescribed by the tokens and create new tokens. Processors communicate with the storage to access programs and data.

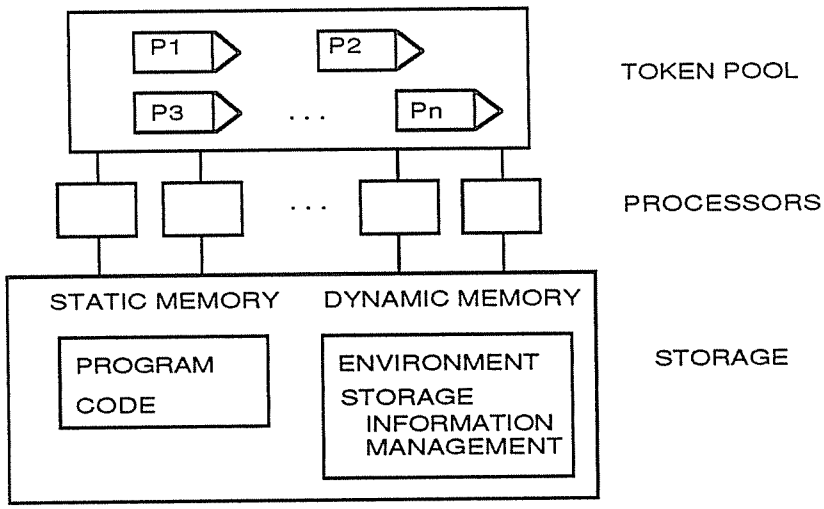


Fig.2 The virtual machine model.

The state of a process consists of a list of goals and a binding environment. Such a state is represented in our machine by a token residing in the token-pool or in one of the processors, a binding environment residing in the dynamic memory, and by a, possibly empty, list of continuation frames representing a goal list, also residing in the dynamic memory. A binding environment of a process consists of contexts for storing values of variables in clauses. A new context is created each time a clause is invoked. During the Or-parallel execution each variable may be bound to several values, still each process must have access to just one value, the one in its environment. There are several methods for maintaining a separate address space for each binding environment. The current simulator implements a version of a storage model presented in [8], which we call the naive model and some refinements presented in [11].

3. Description Tools

3.1 VDM and Meta IV

VDM [12] is a formal method for specification and development of software. Its main underlying idea is to describe a system, in our case a machine architecture, as an abstract model, catching the essential properties of the system under consideration. A model is built up of objects to be manipulated by functions. Objects represent input, output and the internal state of the system. Classes of objects are defined explicitly by domains.

The basic tool used is Meta IV, the language of the VDM, which is the notation in which the description of the model is given. It is used to describe the constituent parts of the model: syntactic objects which represent objects to be manipulated (syntactic domains), the internal state (semantic

domains), and elaboration functions which assign semantic objects (meanings) to syntactic objects.

The description of the model consists of a set of domain definitions, a set of global variable declarations and a set of functions. Each of those sets is conceptually grouped into main and auxiliary groups, to simplify the understanding of the system. In order to follow the rest of this document the reader is urged to read first a more extensive specification of Meta IV in [12] or Appendix I in [7].

The development of a software system can be carried out by proving the correctness of the specifications. As this is generally a tedious task, we preferred to embark on trying to implement the model of the OR-Parallel machine in a programming language, and test the functionality by simulating the model.

3.2 Simula

The programming language SIMULA67 [13] has been chosen as a simulation language due to the following properties:

* garbage collection implicit in the language
* Quasi-Parallel sequencing facilities
* support for process (event) oriented simulation

4. Transformation of the specifications in Meta IV into programs in Simula

Our examples are relatively large because we wanted to show the transformation of the other structures of Meta IV which were too obvious to be mentioned separately. During the transformation we added some variables which were used for collecting data during simulations. We have to mention that the program we have written is only one of many possible realisations of the specification.

4.1. Data Types

4.1.1. Basic Objects

4.1.1.1 Maps

A map can be specified in Simula as a vector. The domain of a map is transformed into the index set of the vector. Elements of the vector (references to the class objects) represent the maps range.

Example: Semantic Domains of Basic Distributed Storage Model

The maps "EnvDir" (line 2) and "VarBindings" (line 4) are translated respectively to vectors "EnvDir(1 : size)" (line 9) and "VarBindings(1 : n)" (line 16). The vectors are indexed by the sets of integers (domains), and the types of their elements (ranges) are Context and V_alue, respectively.

specification in Meta IV:

```
Env = EnvDir | fail
EnvDir = ContextName -> Loc                    /2/
Context :: NumUnbVar VarBindings
VarBindings = Var -> Value                     /4/
Value :: Cterm | unbound
Cterm :: Term ContextName
```

implementation in Simula:

```
class Env(size); integer size;
      begin
            integer used_size;
            boolean status;
            ref(Context) array EnvDir(1 : size);        /9/

            status := true;
            used_size := 0;
      end;

class Context(n); integer n;
      begin
            integer NumUnbVar;
            ref(V_alue) array VarBindings(1 : n);       /16/

            NumUnbVar := n;
      end;

class V_alue(unbound, Term, ValContextName);
      boolean unbound;
      integer Term;
      integer ValContextName;;
```

The following example shows use of the maps defined above, and also how to translate Let and Def Clauses. A Let Clause variant "**let** id E A **be s.t.** ¬(id0 E **dom** (**c** envstg)) **in** ..." can be transformed into Simula by assigning to "id" a new element of class A.

Example: function DuplicateEnvReq of Basic Distributed Storage Model

The "Let Clauses" (lines 5 and 14) are translated respectively to assignments in lines 23 and 33.

specification in Meta IV:

```
dcl envstg := [] type EnvStg;
dcl contxstg := [] type ContextStg;

type DuplicateEnvReq: EnvId => EnvId
DuplicateEnvReq (eid) =
    let eid0 E EnvId be s.t. ¬(eid0 E dom (c envstg)) in                    /5/
        (envstg := (c envstg) U [eid0 -> [] ];
        for all l E dom ((c envstg)(eid)) do
            (def cnm : (c envstg)(eid)(l);
            def mk-Context(nuv,vb) : (c contxstg)(cnm);
            if nuv = 0 then
                (envstg := (c envstg) + [eid0 -> ((c envstg)(eid0) U [l -> cnm])];
                contxstg := (c contxstg) + [cnm -> mk-Context(nuv,vb)])
            else
                let cnm0 E ContextName be s.t. ¬(cnm0 E dom (c contxstg)) in   /14/
                (envstg := (c envstg) + [eid0 -> ((c envstg)(eid0) U [l -> cnm0])];
                contxstg := (c contxstg) + [cnm0 -> mk-Context(nuv,vb)]));
        return(eid0))
```

implementation in Simula:

```
ref(Env) procedure DuplicateEnvReq(EnvId);
    ref(Env)EnvId;
    begin
        integer i, j;
        ref(Env)EnvId0;
        EnvId0 :- new Env(EnvId.size);                                      /23/
        EnvId0.used_size := EnvId.used_size;
        inspect EnvId0 do
        begin
            for i := 1 step 1 until used_size do
                if EnvId.EnvDir(i).NumUnbVar = 0
                then EnvDir(i) :- EnvId.EnvDir(i);
                else begin
                        EnvDir(i) :- new Context(EnvId.EnvDir(i).n);        /33/
                        EnvDir(i).NumUnbVar := EnvId.EnvDir(i).NumUnbVar;
                        for j := 1 step 1 until EnvDir(i).n do
                            EnvDir(i).VarBindings(j) :- EnvId.EnvDir(i).VarBindings(j);
                    end;
        end;
```

```
        DuplicateEnvReq :- EnvId0
    end;
```

4.1.1.2 Sets and Lists

Set and List types can be represented in Simula as the queue type with built-in procedures from class Simset. Elements of the queue must be prefixed by class Link.

Example: Semantic Functions of the Token Pool

The set "pool" (line 5) and list "req-list" (line 6) are translated respectively to queues: "pool" and "req_list" (lines 33 and 34). The structured clause "case" (lines 8 to 21) is translated to "inspect , when" statement (lines 38 to 69). When we want to remove an element from the set (line 14), we do it by taking the first or the last element of the queue (lines 46, 51), depending on our token distribution strategy. To add en element to the set or list (lines 11 and 21) we use the "into" procedure (lines 41 and 64) which is predefined in class Simset.

specification in Meta IV:

```
TokenStg = Token-set
Token :: InstrId ContextName EnvId ContFrameId TermListId
type TokenPool nil =>
TokenPool processor () def
    (dcl pool := {} type TokenStg;                                  /5/
    dcl req-list := <> type ∏;          /6/
    cycle input req from pid =>
    case req:                                                       /8/
        (mk-TokenReqMsg() ->                                        /9/
            (if pool = {} then
                req-list := (c req-list)^<pid>;                     /11/
            else
                (def t E pool;
                pool := (c pool) -{t};                              /14/
                output mk-TokenMsg(t) to pid;))
        mk-TokenMsg(t) ->                                           /16/
            (if pool = {} and req-list != <> then
                (output mk-TokenMsg(t ) to hd (c req-list);
                req-list := tl (c req-list); )
            else  pool := (c pool)U{t};)                            /21/
        ))
```

implementation in Simula:

```
link class Token(InstrRef, ContextName, EnvId, ContFrameId, TermList);
        integer InstrRef;
        integer ContextName;
        ref(Env)EnvId;
        ref(ContFrame)ContFrameId;
        integer TermList;;

processor class token_pool;
        begin
                ref(message) x;
                ref(head)pool, req_list;
                ref(Token)TokenId0;
                ref(processor)proc0;

                pool :- new head;                                       / 33 /
                req_list :- new head;                                   / 34 /
                while true do
                begin
                        x:- receive;
                        inspect x                                       / 38 /
                        when TokenReqMsg do                             / 39 /
                                if pool.empty
                                 then sender.into(req_list)             / 41 /
                                else
                                        begin
                                                if depth then
                                                        begin
                                                                TokenId0 :- pool.last;  / 46 /
                                                                TokenId0.out
                                                        end
                                                else
                                                        begin
                                                                TokenId0 :- pool.first; / 51 /
                                                                TokenId0.out;
                                                        end;
                                                send (sender,  new TokenMsg(current qua processor, TokenId0))
                                        end
                        when TokenMsg do                                / 56 /
                                if pool.empty  and  not  req_list.empty
                                then
                                        begin
                                                proc0 :- req_list.first;
                                                proc0.out;
                                                send (proc0, new TokenMsg(current qua processor, TokenId))
                                        end
                                else TokenId.into(pool)                 / 64 /
                        otherwise
                                begin
                                        outimage;
                                        outtext("invalid msg to token pool")
                                end;                                    / 69 /
                end;
        end token_pool;
```

4.1.2 Composite objects

A composite object can be represented in Simula as a class with an actual parameter list.

Example: Communication Domains of Basic Distributed Storage Model

The object NewContextMsg (line 4) is translated to the class NewContextMsg (line 15) with the actual parameters: EnvId, VarSet and Depth (Depth is added because of the measurement purposes).

specification in Meta IV:

```
StgReq = AssignEnvMsg | NewContxtMsg | DuplicateEnvMsg | AssignValMsg |
         GetValMsg | DisplayResultMsg | EnvStatusMsg | FailEnvMsg
AssignEnvMsg :: nil
NewContxtMsg :: EnvId Var-set                                        /4/
DuplicateEnvMsg :: EnvId
AssignValMsg :: EnvId ContextName Var Value
GetValMsg :: EnvId ContextName Var
DisplayResultMsg :: EnvId ContextName
EnvStatusMsg :: EnvId
FailEnvMsg :: EnvId
```

implementation in Simula:

```
class message(sender); ref(processor)sender;;

message class AssignEnvMsg;;

message class DuplicateEnvMsg (EnvId, Depth);
    ref(Env)EnvId; integer Depth;;

message class NewContextMsg (EnvId, VarSet, Depth);                  /15/
    ref(Env)EnvId; integer VarSet, Depth;;

message class AssignValMsg (EnvId, ContextName, Var, ValueId);
    ref(Env)EnvId; integer ContextName, Var; ref(V_alue)ValueId;;

message class GetValMsg (EnvId, ContextName, Var);
    ref(Env)EnvId; integer ContextName, Var;;

message class DisplResMsg (EnvId, ContextName);
    ref(Env)EnvId; integer ContextName;;

message class EnvStatusMsg (EnvId);
    ref(Env)EnvId;;
```

```
message class FailEnvMsg (EnvId);
    ref(Env)EnvId;;
```

4.2. Meta Processes

Meta Processes are supported directly by a class Simulation with its scheduling/rescheduling statements and a subclass Process. We have introduced a class Processor, prefixed by the class Process to build in the message passing system (procedures send and receive, queue for arriving messages, see section 5.1).

Example: Storage

The "input" and "output" statements (lines 7, 13) are translated respectively to calls of procedures "receive" and "send" (lines 46, 49).

specification in Meta IV:

```
type Storage: nil =>
Storage processor () =
    (dcl out := <> type Result;
    dcl envstg := [] type EnvStg;
    dcl contxstg := [] type ContextStg;
    cycle
        (input req from pid => HandleReq(pid,req) ))                    / 7 /

type HandleReq: ∏ StgReq =>
HandleReq (pid,req) =
    cases req:
    (mk-AssignEnvMsg() ->
        (def eid: AssignEnvReq();
        output mk-EnvMsg(eid) to pid),                                  / 13 /
    mk-NewContxtMsg(eid,vs) ->
        (def cnm: NewContxtReq(eid,vs);
        output mk-ContextNameMsg(cnm) to pid),
    mk-DuplicateEnvMsg(eid) ->
        (def eid0: DuplicateEnvReq(eid);
        output mk-EnvMsg(eid0) to pid),
    mk-AssignValMsg(eid,cnm,x,xv) ->
        AssignValReq(eid,cnm,x,xv),
```

```
mk-GetValMsg(eid,cnm,x) ->
    (def xv: GetValReq(eid,cnm,x);
    output mk-ValMsg(xv) to pid),
mk-DisplayResultMsg(eid) ->
    DisplayResultReq(eid),
mk-EnvStatusMsg(eid) ->
    (def b:EnvStatusReq(eid);
    output EnvStatusRespMsg(b) to pid),
mk-FailEnvMsg(eid) ->
    FailEnvReq(eid)
)
```

implementation in Simula (the bodies of the declared procedures are omitted):

```
processor class storage;
    begin
        ref(message) x;
        ref(Env) procedure AssignEnvReq; ... ;
        ref(Env) procedure DuplicateEnvReq(EnvId); ... ;
        ref(ObjContextName) procedure NewContextReq(EnvId, VarSet); ... ;
        ref(Env) procedure CopyEnvReq(EnvId); ... ;
        procedure AssignValReq(EnvId, ContextName, Var, ValueId); ... ;
        ref(V_alue) procedure GetValReq(EnvId, ContextName, Var); ... ;
        procedure DisplayResultReq(EnvId, ContextName); ... ;
        boolean procedure EnvStatusReq(EnvId); ... ;
        procedure FailEnvReq(EnvId); ... ;

        while true do
        begin
            x:- receive;                                                    / 46 /
            inspect x
            when AssignEnvMsg do
                send (sender, new EnvMsg(current qua processor, AssignEnvReq))    / 49 /

            when DuplicateEnvMsg do
                send (sender, new EnvMsg(current qua processor,  DuplicateEnvReq(EnvId)))
            when NewContextMsg do
                send (sender, new ContextNameMsg(current qua processor,
                        NewContextReq(EnvId, VarSet)))
            when AssignValMsg do
                AssignValReq(EnvId, ContextName, Var, ValueId)
            when GetValMsg do
                send (sender, new ValMsg(current qua processor,
                        GetValReq(EnvId, ContextName, Var)))
            when DisplResMsg do
                DisplayResultReq(EnvId, ContextName)
            when EnvStatusMsg do
                send (sender, new EnvStatusRespMsg(current qua processor,
                        EnvStatusReq(EnvId)))
            when FailEnvMsg do
                FailEnvReq(EnvId)
```

```
            otherwise
                begin
                    outimage;
                    outtext("invalid msg to storage")
                end;
        end;
    end storage;
```

5. Simulator of the Virtual Machine

The model of the OR-Parallel Token Machine is implemented using the process concept of Simula. The three components of the machine are represented by communicating processes, with the same communication features as in Meta IV, i.e. sending and receiving messages. These features are implemented using the message passing system. The simulator is run under VAX750/UNIX. The implementation of the simulator is based on a specification [7,8] written in Meta IV. The structure of the simulator is shown in Appendix I.

5.1 Message Passing System

The message passing system consists of some procedures and subclasses which are defined in the class Processor. Each simulated processor is prefixed by the class Processor and thus possesses the message passing property. The message passing system is invoked by calling procedures "send" and "receive". Each message contains the information about its sender. The quasi-parallel properties of the simulator are built around the procedures for sending and receiving messages.

5.1.2 Mechanism

The mechanism of message passing is built around the following structures and procedures:

message queue: (q.mess_queue) includes all processes which have messages to the specified process, in order of arrival. Each element in such a queue consists of a message (m) and its corresponding process (p) (fig 3).

receiver field: (q.receiver) indicates the state of a process waiting for a message to arrive.

message field: (m) is used to hold the passed message.

procedures send and receive: When a process sends a message and the "message queue" of the receiving process has other messages, it passivates itself and waits until its processing turn comes. If the receiver does not wait for a message and the "message queue" is empty, the sender passivates itself and waits in the message queue until a request for a message arrives, on the other hand if the receiver waits for a message, the message is delivered and the receiving process is activated while the sending process is placed at the end of the activation queue (fig. 4). When a process is to receive a message, while no messages has arrived, it goes into waiting state by initialising "receiver field" and passivating itself, otherwise it receives a message as described above.

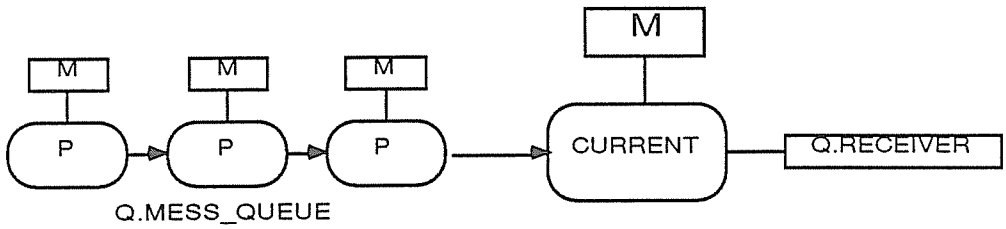


Fig. 3 Structures for message passing system.

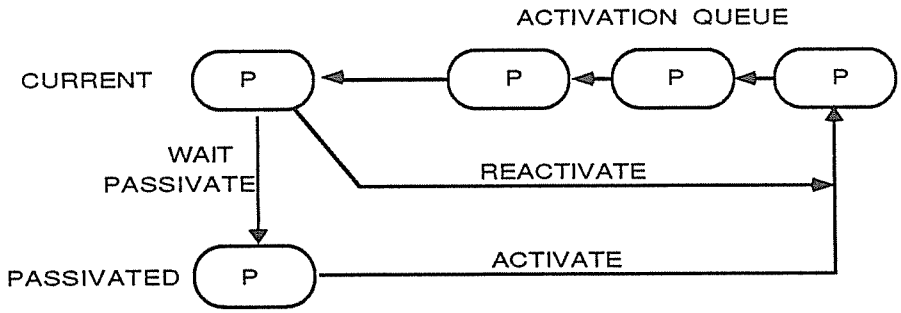


Fig. 4 Process scheduling schema.

5.1.3 Source Code

```
class mess_info;
    begin
        ref(head)mess_queue;
        ref(processor)receiver
    end mess_info;

class message(sender);
    ref(processor)sender;;

process class processor;
    begin
        ref(message)m;
        ref(mess_info)q;

        procedure send(p, mess); ref(processor)p; ref(message)mess;
            begin
                current qua processor.m :- mess;
                if not p.q.mess_queue.empty or p.q.receiver == none then
                    wait(p.q.mess_queue)
                else
                    begin
                        p.q.receiver.m :- current qua processor.m;
                        activate p.q.receiver delay 0;
                        p.q.receiver :- none;
                        reactivate current delay 0
                    end
            end send;
```

```
        ref(message)procedure receive;
            begin
                ref(processor)p;
                if current qua processor.q.mess_queue.empty then
                    begin
                        current qua processor.q.receiver :- current;
                        passivate
                    end
                else
                    begin
                        p :- current qua processor.q.mess_queue.first;
                        p.out;
                        current qua processor.m :- p.m;
                        activate p delay 0;
                        reactivate current delay 0
                    end ;
                receive :- current qua processor.m
            end receive;

        current qua processor.q :- new mess_info;
        current qua processor.q.mess_queue :- new head;

    end processor;
```

5.2 Instruction Processors

Upon the start, all instruction processors are loaded with a copy of the compiled program. One of the processors starts execution of the first instruction and creates new tokens which are sent to the token pool. Then all instruction processors repetitiously request tokens from the token pool. When a processor receives a token, it executes the instruction referred by it. The execution results in the change of the binding environment, or/and the control information, and finally in creation of 0, 1 or more new tokens. Then tokens are placed in the token pool. The tokens in the token pool represent processes which are ready for execution but not allocated a processor. The code for the instruction processors is generated by a compiler written in Prolog, capable of generating code for several virtual machines [14].

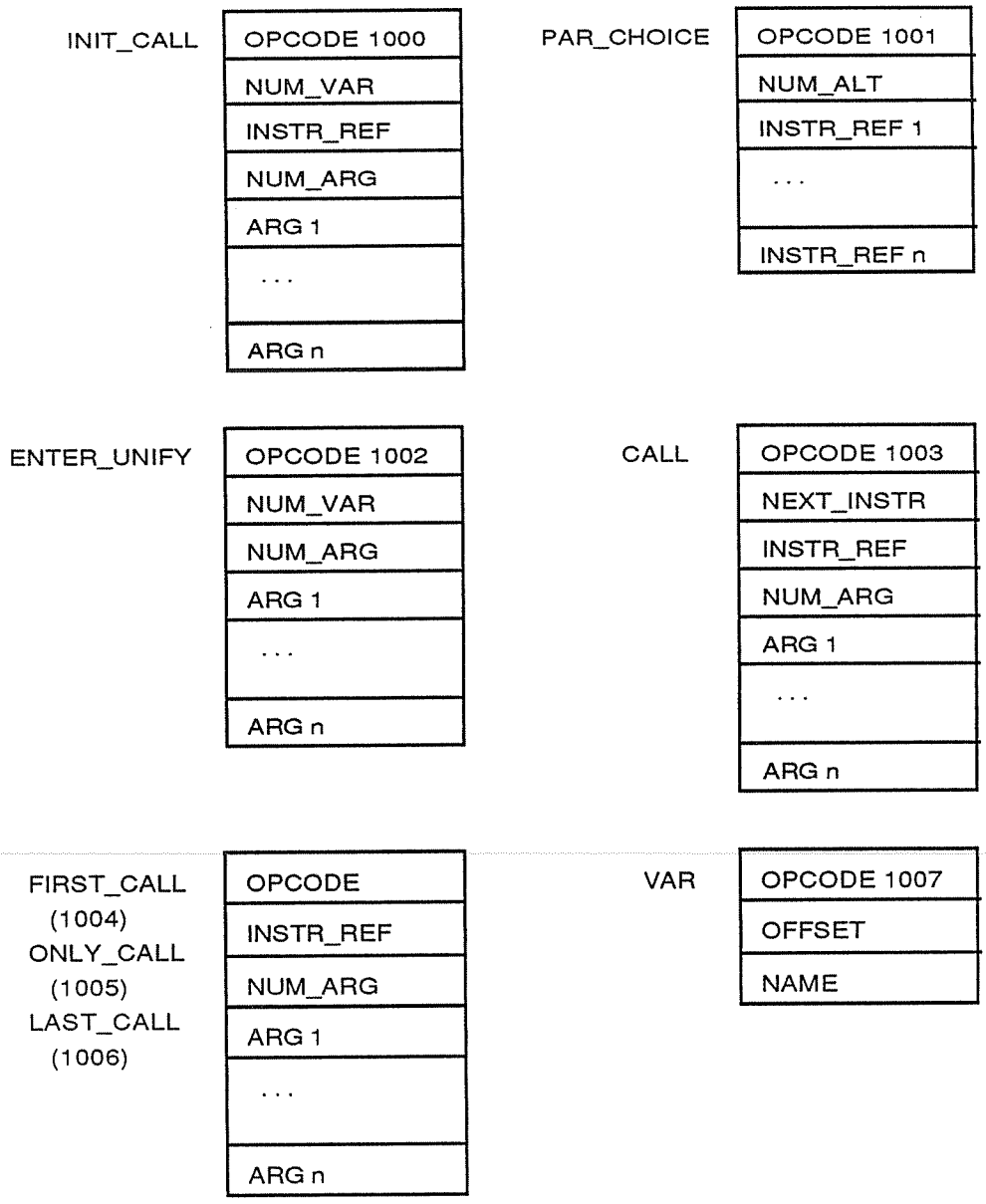
5.2.1 Implementation Considerations

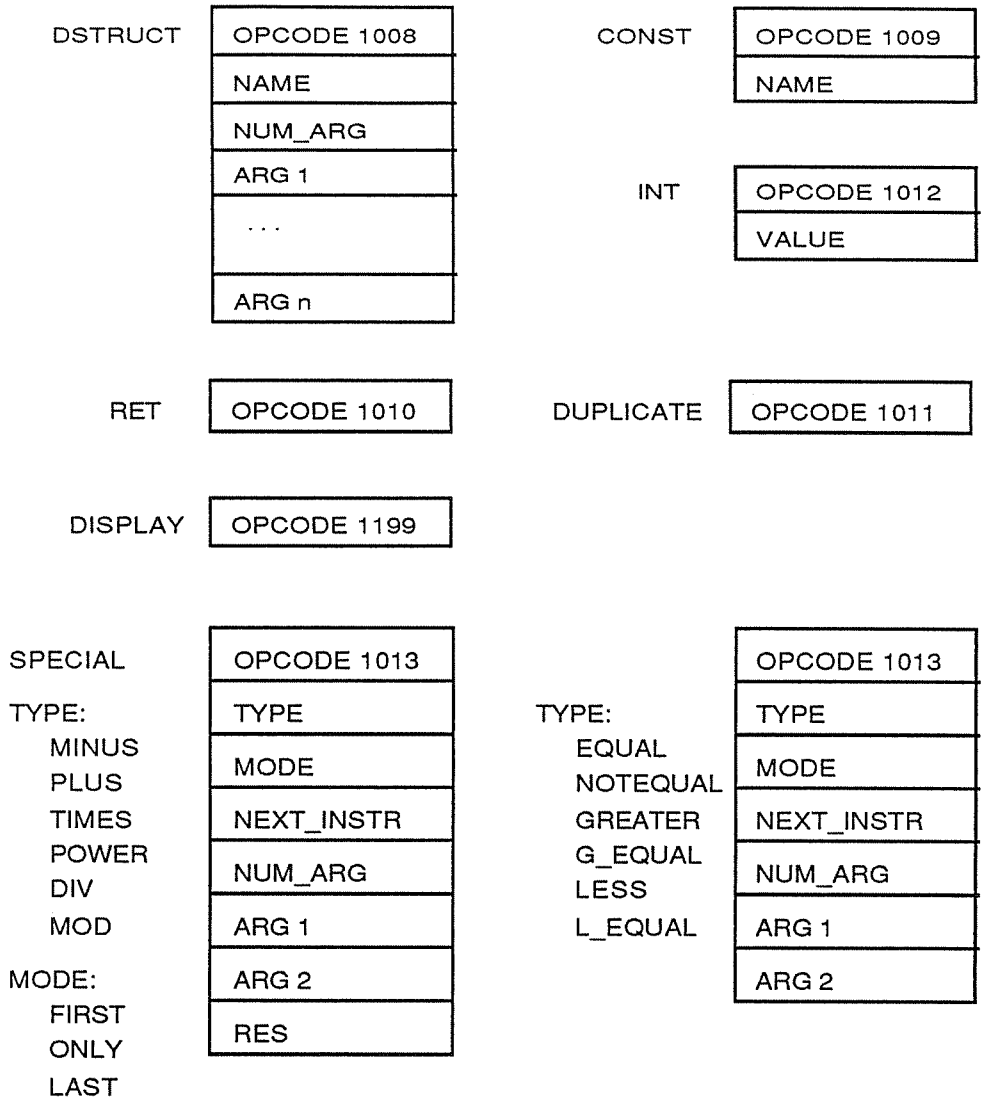
Comparing to the abstract specification we had to make some implementation decisions to be able to run reasonable programs. We implemented a number of arithmetic operations and improved the output. Our basic arithmetic operations require instantiated numeric operands. Operation "equal" is executed as a unification call and the negation is understood as negation by failure. We can also specify the number of expected solutions. The source code of the instruction processor is presented in Appendix II.

5.2.2 Instruction Set

The instruction set is based on a specification of the OR-Parallel Token Machine [7] with the additions named above.

Structure of the instructions:





5.3 Token Pool

The token pool is an abstraction of the load distribution and scheduling in the distributed systems. It was implemented in Simula as an ordered set. The source code is shown in chapter 4.1.1.2. In comparison to the specification we added features more specific for the Or-parallel execution, such as the choice between depth first and breadth first search for solutions. The depth first search is implemented by using a stack (LIFO), and the breadth first by using a queue (FIFO) as the token pool.

5.4 Storage

The dynamic storage contains environments, contexts and continuation frames. The first two categories define the environment in which the computation steps are carried out. The continuation frames are elements which define the sequence of computations. We have implemented four refinements of the basic storage model (straight forward, directory trees, hashing on contexts and

hashing on variables) described in [8,11]. The source code of the procedures of the basic storage model is shown in chapter 4.2. Here we present only an implementation of the data structures (fig 5).

The source code of the data structures is shown in 4.1.1.1.

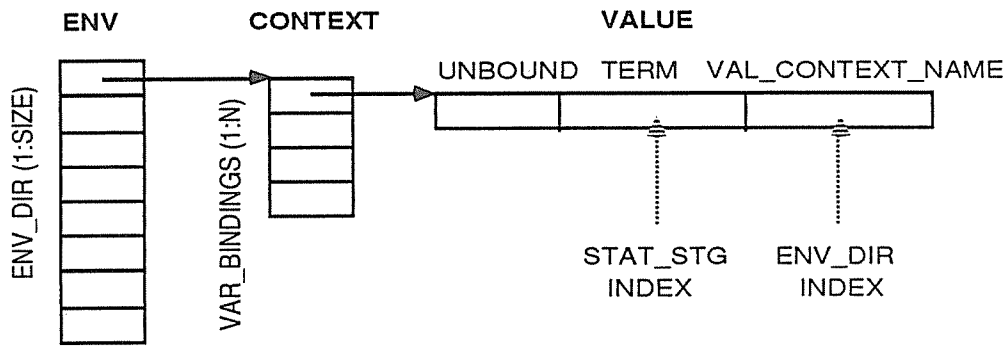


Fig. 5 Structures of the basic storage model.

6. Conclusions

Given the specification in Meta IV, choosing the high level language Simula67 with its abstract data types (classes) and object oriented properties helped to achieve our goal - to build an efficient simulator. We could almost directly map structures of Meta IV into Simula. It shows the power of using object oriented languages like Simula or Smalltalk for simulation purposes.

This implementation of the OR-Parallel Token Machine gave important information regarding the properties of this abstract model and improved the model by adding some "real world" features (output, arithmetic, negation).

Acknowledgements

I am very grateful to Andrzej Ciepielewski, who supervised the project, for the support and the discussions.

References

[1] D.S.Warren, Efficient Prolog Memory Management for Flexible Control Strategies, in proceedings of 1984 Intl. Symposium on Logic Programming, Atlantic City, also in New Generation Computing no.4 1984.

[2] G.Lindstrom, Or-Parallelism on Applicative Architectures, in proceedings of the Second Intl. Logic Programming Conference, Uppsala 1984.

[3] P.Borgwart, Parallel Prolog Using Stack Segments on Shared-Memory Multiprocessors, in proceedings of 1984 Intl. Symposium on Logic Programming, Atlantic City.

[4] G.H.Pollard, Parallel Execution of Horn Clause Programs, PhD Thesis, Imperial College of Science and Technology, University of London, 1981.

[5] S.Haridi, A.Ciepielewski, An Or-parallel Token Machine, Logic Workshop 83, Algarve/Portugal, also TRITA-CS-8303, Royal Institute of Technology, Stockholm 83.

[6] A.Ciepielewski, S.Haridi, Control of Activities in the Or-parallel Token Machine, in proceedings of 1984 Intl. Symposium on Logic Programming, Atlantic City.

[7] A.Ciepielewski, Towards a Computer Architecture for Or-parallel Execution of Logic Programs, PhD Thesis, Royal Inst. of Tech., May 1984, TRITA-CS-8401.

[8] A.Ciepielewski, S.Haridi, Storage Models for Or-parallel execution of Logic Programs - DRAFT, CSALAB Working Paper 821121, TRITA-CS-8301, Royal Institute of Technology, Stockholm 1983.

[9] A.Ciepielewski, S.Haridi, A Formal Model for Or-parallel execution of Logic Programs, in IFIP 83, North Holland P. C., Mason (ed).

[10] A.Ciepielewski, B.Hausman, S.Haridi, Initial Evaluation of a Virtual Machine for OR-Parallel Execution of Logic Programs, in proceedings of IFIP TC-10 Working Conference on Fifth Generation Computer Architecture, Manchester 1985 (also in proceedings of the Third Japanese-Swedish Workshop, Tokyo 1985).

[11] A.Ciepielewski, B.Hausman, Performance Evaluation of a Storage Model for OR-Parallel Execution of Logic Programs, submitted to Third Symposium on Logic Programming, Salt Lake City, 1986.

[12] D.Bjørner, C.B.Jones (eds), The VDM: The Meta Language, LNCS 98, Springer-Verlag 1980.

[13] Decsystem-10/20 Simula Language Handbook, Part I, II, III, Swedish National Defence Research Institute, October 1976.

[14] T.Sjöland, Transforming LPL0 Programs Using LPL0, CSALAB, TRITA-CS-8405.

Appendix I

The structure of the simulator (the mentioned sections do not contain the complete code).

```
external class simulation;
simulation class common;
    begin
        ...
        definitions of classes, data structures, global variables
        see sections 4.1.1.1, 4.1.2, 5.1.3
        ...
    end;

begin
external class common;
common
    begin
        processor class token_pool;
            begin
                ...
                see section 4.1.1.2
                ...
            end;

        processor class storage;
            begin
                ...
                see section 4.2
                ...
            end;

        processor class ip(t, s, start, prnumber);
            ref(processor) t, s;
            boolean start;
            integer prnumber;
            begin
                ...
                see Appendix II
                ...
            end;

        ... input ...

        a :- new token_pool;
        b :- new storage;
        c :- new ip(a, b, true, 1);
        d :- new ip(a, b, false, 2);
        e :- new ip(a, b, false, 3);

        activate a;
        activate b;
        activate c;
        activate d;
        activate e;

        ... output ...

    end or_parallel_token_machine;
end;
```

Appendix II

The source code for the instruction processor (the bodies of the declared procedures are omitted).

```
processor class ip(t,s,start,prnumber);
    ref(processor) t, s;
    boolean start;
    integer prnumber;
    begin
        ref(Env) procedure AssEnv; ... ;

        ref(Env) procedure DuplicateEnv (EnvId);
            ref(Env)EnvId; ... ;

        ref(ContFrame) procedure NewContFrame(InstrRef, ContextName, ContFrameId);
            integer InstrRef, ContextName;
            ref(ContFrame)ContFrameId; ... ;

        ref(ObjContextName) procedure NewContext (EnvId, VarSet);
            ref(Env) EnvId; integer VarSet; ... ;

        procedure DisplayResult(EnvId, ContextName);
            ref(Env)EnvId; integer ContextName; ... ;

        ref(V_alue) procedure GetVal (EnvId, ContextName, Var);
            ref(Env) EnvId; integer ContextName, Var; ... ;

        procedure AssignVal (EnvId, ContextName, Var, ValueId);
            ref(Env)EnvId; integer ContextName, Var; ref(V_alue)ValueId; ... ;

        boolean procedure FailEnv(EnvId);
            ref (Env) EnvId; ... ;

        procedure SignalFailEnv (EnvId);
            ref(Env) EnvId; ... ;

        procedure SetAccNum (EnvId, Num);
            ref(Env)EnvId; integer Num; ... ;

        procedure SendToken (TokenId);
            ref(Token) TokenId; ... ;

        ref(Token) procedure FetchToken; ... ;

        boolean procedure unify (pl, EnvId, flag);
            ref(head) pl;
            ref(Env) EnvId;
            boolean flag; ... ;

        ref(V_alue) procedure GetNumber (EnvId, ContextName, Arg);
            ref(Env) EnvId; integer ContextName, Arg; ... ;

        boolean procedure DoArithmetic (EnvId, ContextName, Res, Arg1, Arg2, T_ype);
            ref(Env)EnvId;
            integer ContextName, Res, Arg1, Arg2, T_ype; ... ;
```

```
boolean procedure DoRelation (EnvId, ContextName, Arg1, Arg2, T_ype);
      ref(Env)EnvId;
      integer ContextName, Arg1, Arg2, T_ype; ... ;

ref(Env)EnvId0;
integer TermList0, ContextName0, m, InstrRef0;
ref(ObjContextName) ObjContextNameId0;
ref(ContFrame) ContFrameId0;
ref(Token) TokenId0;
ref(head) unify_list;
boolean owntoken;

unify_list :- new head;
owntoken := false;

if start then
      begin
            owntoken := true;
            EnvId0 :- AssEnv;
            ObjContextNameId0 :- NewContext (EnvId0, StatStg(4));
            InstrRef0 := StatStg(5);
            ContextName0 := ObjContextNameId0.ContextName;
            ContFrameId0 :- NewContFrame (2, ContextName0, none);
            TermList0 := 6;
      end;

while action do
begin
if not owntoken then
   begin
      owntoken := true;
      TokenId0 :- FetchToken;
      InstrRef0 := TokenId0.InstrRef;
      ContextName0 := TokenId0.ContextName;
      EnvId0 :- TokenId0.EnvId;
      ContFrameId0 :- TokenId0.ContFrameId;
      TermList0 := TokenId0.TermList;
   end;

if StatStg(InstrRef0) = FIRST_CALL then
   begin
      ContFrameId0 :- NewContFrame(InstrRef0 + StatStg(InstrRef0 + NUM_ARG_f_c) + 3,
                  ContextName0, ContFrameId0);
      TermList0 := InstrRef0 + NUM_ARG_f_c;
      InstrRef0 := StatStg(InstrRef0 + INSTR_REF)
   end
else if StatStg(InstrRef0) = CALL then
   begin
      ContFrameId0 :-  NewContFrame(StatStg(InstrRef0 + NEXT_INSTR),
                  ContextName0, ContFrameId0.ContFrameId);
      TermList0 := InstrRef0 + NUM_ARG_c;
      InstrRef0 := StatStg(InstrRef0 + INSTR_REF_c)
   end
else if StatStg(InstrRef0) = LAST_CALL then
   begin
      ContFrameId0 :- ContFrameId0.ContFrameId;
      TermList0 := InstrRef0 + NUM_ARG_f_c;
      InstrRef0 := StatStg(InstrRef0 + INSTR_REF)
   end
```

```
else if StatStg(InstrRef0) = ONLY_CALL then
   begin
      TermList0 := InstrRef0 + NUM_ARG_f_c;
      InstrRef0 := StatStg(InstrRef0 + INSTR_REF)
   end
else if StatStg(InstrRef0) = PAR_CHOICE then
   begin
      owntoken := false;
      SetAccNum (EnvId0, StatStg(InstrRef0 + NUM_ALT));
      for m := 2 step 1 until StatStg(InstrRef0 + NUM_ALT) do
            SendToken (new Token (
                  StatStg(InstrRef0 + m + 1),
                  ContextName0,
                  EnvId0,
                  ContFrameId0,
                  TermList0
                  ));
      SendToken (new Token (
            StatStg(InstrRef0 + 2),
            ContextName0,
            EnvId0,
            ContFrameId0,
            TermList0
            ))
   end
else if StatStg(InstrRef0) = RET then
   begin
      InstrRef0 := ContFrameId0.InstrRef;
      ContextName0 := ContFrameId0.ContextName;
      TermList0 := 0
   end
else if StatStg(InstrRef0) = DUPLICATE then
   begin
      InstrRef0 := InstrRef0 + INSTR_REF;
      EnvId0 :- DuplicateEnv(EnvId0)
   end
else if StatStg(InstrRef0) = DISPLAY then
   begin
      owntoken := false;
      DisplayResult (EnvId0, ContextName0)
   end
else if StatStg(InstrRef0) = ENTER_UNIFY then
   begin
      ObjContextNameId0 :- NewContext (EnvId0, StatStg(InstrRef0 + NUM_VAR));
      unify_list.clear;
      for m := 1 step 1 until StatStg (InstrRef0 + NUM_ARG_e_u) do
            new unify_unit (StatStg(TermList0 + m), ContextName0, StatStg(InstrRef0 + m + 2),
                  ObjContextNameId0.ContextName).into(unify_list);
      if unify (unify_list, ObjContextNameId0.EnvId, false) then
         begin
            InstrRef0 := InstrRef0 + StatStg(InstrRef0 + NUM_ARG_e_u) + 3;
            ContextName0 := ObjContextNameId0.ContextName;
            EnvId0 :- ObjContextNameId0.EnvId;
            TermList0 := 0;
         end
      else
            owntoken := false;
   end
else if StatStg(InstrRef0) = SPECIAL then
```

```
begin
  if StatStg(InstrRef0 + TYPE_a) = MINUS or StatStg(InstrRef0 + TYPE_a) = PLUS or
    StatStg(InstrRef0 + TYPE_a) = TIMES or StatStg(InstrRef0 + TYPE_a) = POWER or
    StatStg(InstrRef0 + TYPE_a) = DIV or StatStg(InstrRef0 + TYPE_a) = M_OD
  then
    begin
      if DoArithmetic(EnvId0, ContextName0, StatStg(InstrRef0 + ARG_REF+2),
          StatStg(InstrRef0 + ARG_REF), StatStg(InstrRef0 + ARG_REF + 1),
          StatStg(InstrRef0 + TYPE _a))
        then
          begin
            m := StatStg(InstrRef0 + NEXT_INSTR_a);
            if StatStg(InstrRef0 + MODE) = FIRST
            then ContFrameId0 :- NewContFrame(NEXT_INSTR_a,
                                ContextName0, ContFrameId0)
            else if StatStg(InstrRef0 + MODE) = L_AST
            then
              begin
                ContFrameId0 :- ContFrameId0.ContFrameId;
                m := ContFrameId0.InstrRef;
                ContextName0 := ContFrameId0.ContextName
              end
            else if StatStg(InstrRef0 + MODE) = ONLY
            then
              begin
                m := ContFrameId0.InstrRef;
                ContextName0 := ContFrameId0.ContextName
              end;
            InstrRef0 := m;
            TermList0 := 0
          end
        else
            owntoken := false;
      end
  else if StatStg(InstrRef0 + TYPE_a) = EQUEAL then
      begin
        unify_list.clear;
        new unify_unit(StatStg(InstrRef0 + ARG_REF), ContextName0,
            StatStg(InstrRef0 + ARG_REF + 1),  ContextName0).into(unify_list);
        if unify (unify_list, EnvId0, false)
        then
          begin
            m := StatStg(InstrRef0 + NEXT_INSTR_a);
            if StatStg(InstrRef0 + MODE) = FIRST
            then ContFrameId0 :- NewContFrame(NEXT_INSTR_a,  ContextName0,
                                                    ContFrameId0)
            else if StatStg(InstrRef0 + MODE) = L_AST
            then
              begin
                ContFrameId0 :- ContFrameId0.ContFrameId;
                m := ContFrameId0.InstrRef;
                ContextName0 := ContFrameId0.ContextName
              end
            else if StatStg(InstrRef0 + MODE) = ONLY
            then
              begin
                m := ContFrameId0.InstrRef;
                ContextName0 := ContFrameId0.ContextName
              end;
```

```
            InstrRef0 := m;
            TermList0 := 0;
          end
      else
            owntoken := false;
   end
else if StatStg(InstrRef0 + TYPE_a) = NOTEQ
then
    begin
      unify_list.clear;
      new unify_unit(StatStg(InstrRef0 + ARG_REF), ContextName0,
            StatStg(InstrRef0 + ARG_REF + 1), ContextName0).into(unify_list);
      if not unify (unify_list, EnvId0, true)
      then
        begin
          m := StatStg(InstrRef0 + NEXT_INSTR_a);
          if StatStg(InstrRef0 + MODE) = FIRST
           then ContFrameId0 :- NewContFrame(NEXT_INSTR_a, ContextName0,
                                                          ContFrameId0)
          else if StatStg(InstrRef0 + MODE) = L_AST
          then
            begin
              ContFrameId0 :- ContFrameId0.ContFrameId;
              m := ContFrameId0.InstrRef;
              ContextName0 := ContFrameId0.ContextName
            end
          else if StatStg(InstrRef0 + MODE) = ONLY
          then
            begin
              m := ContFrameId0.InstrRef;
              ContextName0 := ContFrameId0.ContextName
            end;
          InstrRef0 := m;
          TermList0 := 0;
        end
      else
        begin
          SignalFailEnv(EnvId0);
          owntoken := false;
        end;
   end
else if StatStg(InstrRef0 + TYPE_a) = GREATER or StatStg(InstrRef0 + TYPE_a) = G_E or
      StatStg(InstrRef0 + TYPE_a) = LESS or  StatStg(InstrRef0 + TYPE_a) = L_E
then
   begin
      if DoRelation(EnvId0, ContextName0, StatStg(InstrRef0 + ARG_REF),
            StatStg(InstrRef0 + ARG_REF + 1), StatStg(InstrRef0 + TYPE_a))
      then
        begin
          m := StatStg(InstrRef0 + NEXT_INSTR_a);
          if StatStg(InstrRef0 + MODE) = FIRST
          then ContFrameId0 :- NewContFrame(NEXT_INSTR_a, ContextName0,
                                                          ContFrameId0)
          else if StatStg(InstrRef0 + MODE) = L_AST
          then
            begin
              ContFrameId0 :- ContFrameId0.ContFrameId;
              m := ContFrameId0.InstrRef;
              ContextName0 := ContFrameId0.ContextName
```

```
                    end
                else if StatStg(InstrRef0 + MODE) = ONLY
                then
                    begin
                      m := ContFrameId0.InstrRef;
                      ContextName0 := ContFrameId0.ContextName
                    end;
                InstrRef0 := m;
                TermList0 := 0
              end
            else
                owntoken := false;
          end;
      end
    else
      begin
        outimage;
        outint(prnumber, prnumber * 3);
        outtext(" wrong InstrRef = ");
        outint(InstrRef0, 3);
        outimage;
        action := false
      end;
  end while;
end ip;
```