A Formal Specification of the PVM Architecture*

Egon Börger[a] and Uwe Glässer[b]

[a]Dipartimento di Informatica, Università di Pisa, Corso Italia, 40, 56125 Pisa, Italy, boerger@di.unipi.it

[b]FB Mathematik-Informatik, Universität-GH Paderborn, Warburgerstr. 100, 33098 Paderborn, Germany, glaesser@uni-paderborn.de

We develop a mathematically precise yet transparent definition of the Parallel Virtual Machine PVM. Our model, based upon Gurevich's notion of concurrent evolving algebras, directly supports the basic intuitions of heterogeneous distributed computing.

Keyword Codes: D.3.1
Keywords: Programming Languages, Formal Definitions and Theory; Distributed Systems

Introduction

PVM (Parallel Virtual Machine) is a software system[2] that serves as a general purpose environment for heterogeneous distributed computing [?, ?]. We develop here a *mathematical definition* of PVM at a level of abstraction and precision which is tailored to the needs of a programmer who wants to be brought, fast and reliably, to a correct understanding of the system at the C–interface. We build our model in such a way that it can also be used as basis for a series of stepwise refinements, leading in a provably correct way to actual PVM code. Our specification is easily adaptable to extensions and modifications of single features, parts or interfaces of the system; such ease with extensions seems to us to be a particularly important goal for specifying a complex still changing system.

Our specification methodology is based on Gurevich's concept of *evolving algebra*. This method allows to avoid formal overhead, enabling the reader to follow a precise definition without any specific previous formal training. For details and the background of sequential and concurrent evolving algebras see [?, ?]. The present definition is based on PVM Version 3, the current system release [?]. Due to space limitations we treat here only a few—but typical—routines for message passing and task creation. For a full treatment see [?].

*In: B. Pehrson and I. Simon (Eds.) *IFIP 13th World Computer Congress 1994*, Volume I: *Technology/Foundations*, Elsevier, Amsterdam.

[2]The PVM software is public domain and can be obtained from Oak Ridge National Laboratory (ORNL) by sending electronic mail to *netlib@ornl.gov* with the message *send index from pvm3*.

The PVM System

Under PVM, a heterogeneous collection of physically interconnected and concurrently operating computers of a great variety of *architectures* (including serial, parallel, and vector computers) appears logically as a single distributed-memory computer. This abstract parallel computer is called the *virtual machine*. The constituting member computers, called *host* machines, can be dynamically added to or deleted from the virtual machine – except a designated master host, the one on which PVM is started and which keeps control of the dynamic evolution of the overall machine configuration.

Concurrently running application programs can enroll into PVM as *tasks*, the basic computing units of PVM, similar to processes in Unix. Such programs use the virtual machine through a message–passing interface that provides uniform access to hosts.[3] The underlying message-passing model does not distinguish between local and global intertask communication. The communicating tasks do not even need to know whether their communication partners reside on the same host or not. To support this view, a homogeneous global address space is used in which all tasks are uniformly addressed through unique task identifiers (*tids*).

PVM installs on each host machine a *daemon* process, called *pvmd*, which acts as a local supervisor in operations that require task management or intertask communication. Installation and management of the pvmds (maintenance of the overall machine configuration) is effected through a distinguished daemon — *demiurge*[4] — the one residing on the master host. In particular he has to create or delete pvmds on new or deleted hosts, administrating a corresponding *host table* an updated read-only copy of which he broadcasts to the other pvmds each time the configuration changes.

Parallelization of PVM tasks is possible by using message-passing constructs from the *standard interface library* of PVM that supports common concurrent processing paradigms in the form of well-defined primitives. These primitives, which offer the necessary communication, synchronization, and control features, have to be embedded in the procedural host language (Fortran77 or C).

1. Basic Data Structures

Our concurrent *PVM Algebra* has a dynamic set of $HOST$s, $master \in HOST$, each of an architecture indicated by a function $arch$ into the static domain $ARCH$[5]. The members of the dynamic set $DAEMON$ work concurrently, each on a different host assigned by a bijective function $host : DAEMON \to HOST$, supervising the activities of $TASK$s on this host. $DAEMON$ contains a distinguished element $demiurge$, $host(demiurge) = master$, which is in charge of pvmd management as explained above.

Processes being enrolled into PVM as tasks are globally addressed through functions

$$tid : TASK \to TID \qquad task : TID \to TASK \qquad pid : TASK \to PID$$

which yield identifiers in appropriate identifier sets. We assume $tid, task$ to be inverse to

[3] According to this view, a host takes the role of a processing element in a loosely coupled multiprocessor system. This analogy does not apply to the case of a host itself being a multiprocessor system.
[4] Plato (see Timaeus, 40c.) describes demiurge as creative force that shaped the material world.
[5] $ARCH$ consists of the predefined architecture names to be used with PVM 3 as specified in [?].

each other. A task identifier is supposed to encode also the unique daemon under control of which the task is operating, using a function $pvmd : TID \to DAEMON$. This function justifies to speak about the list TID_x of all $t \in TID$ of tasks which are concurrently running under daemon x, formally such that $pvmd(t) = x$.

In order to describe supervisor actions of daemons in a way which directly supports basic intuitions about distributed message–passing computing, we introduce an external dynamic function[6] $event : DAEMON \to EVENT$ which assigns to each daemon the instruction or message he is supposed to execute or read. The rules below are therefore formulated with conditions containing $event(x) = instruction/message$ expressing that when daemon x is going to execute/read the instruction/message, then.... Thus we leave it abstract how a daemon "walks through his sequence of instructions," assuming, for each x, an event updating mechanism to be given for the sequential evolving algebra formed by the module of all rules of x.[7]

Certain PVM instructions result in distributed operations involving two or more hosts the pvmds of which perform interactions through message–passing. We model inter–pvmd communication in the transition rules below by abstract updates '*forward* $\langle RequestMsg \rangle$ *to* y' or '*return* $\langle ReplyMsg \rangle$ *to* x,' where x, y refer to the interacting pvmds. They are supposed to trigger corresponding events for the receiving pvmds. For that reason we will have a number of communication related transition rules, distinguished by the suffix '*_msg*', any of which describes the reaction of a pvmd x when receiving a message from another pvmd y.

Delays might occur when a pvmd requests another pvmd for service. In order to avoid blocking due to waiting, the requesting pvmd stores its waitcontext, accessed by a unique *wait-id (wid)* which is passed along with the request and returned with the reply. The waitcontext typically includes information about the requesting task ($req\_info : WID \to TASK$), the reply data ($rep\_info : WID \to REPLY^*$), and a request count ($waitcount : WID \to INT$, indicating the number of replies a pvmd is still waiting for).

2. Message-Passing Interface

For the specification of PVM's message–passing interface—which offers point-to-point communication from one task to another as well as multicast to a set of tasks—we extend the basic model by abstract domains related to messages and buffers: $MESSAGE$, TAG, $DATA$, $BUFID$, $ENCODING$. PVM specifies no limit to the size or number of messages[8]. To send a message, a task first packs the message into a send buffer and then calls one of the send functions. To selectively receive messages, a task invokes one of various receive functions determined by a specified *receive context*.

The *basic* message-passing routines of PVM apply a simple communication model[9]

[6] Such a function, as explained in [?], is not updated by rules of the system to which it is considered to be external, but it nevertheless might change its values dynamically, due to actions which are external to the rule system.

[7] We use the notion of runs of concurrently working sequential evolving algebras, defined in [?].

[8] Our specification could easily incorporate conditions which reflect constraints coming from physical limitations of the underlying hardware and software components.

[9] For more sophisticated communication mechanisms additional routines and options, not discussed here, allow to tailor the basic model to individual application requirements, for example direct task-to-task

that is based on two fundamental assumptions: for each task there is only one send buffer and one receive buffer; any message transfer between tasks is handled by the responsible pvmds. The following description of the message-passing model assumes two basic integrity constraints that are guaranteed by the corresponding routines embedded in the virtual machine: message-passing is *reliable* and *order-preserving*.

2.1. Message Buffers

Message buffers are addressed through unique identifiers from $BUFID$, $0 \notin BUFID$. The content of a buffer may be any sequence in $DATA$ accessed by $cont : BUFID \to DATA^*$. An $encoding : BUFID \to ENCODING = \{PvmDataDefault, PvmDataRaw, PvmDataInPlace\}$, associated to send buffers, specifies the method used for packing messages: *PvmDataDefault* refers to External Data Representation (XDR); *PvmDataRaw* to the original data format; *PvmDataInPlace* means that the data items have to be copied directly out of the user's memory (for details cf. [**?**]). Two injective functions $sendbuf$, $recvbuf$ from $TASK$ into $BUFID$ yield the send and receive buffers of tasks.

The routine pvm_initsend() creates, for the task which wants to start a sending operation, an empty send buffer with the specified encoding scheme and returns the buffer identifier. pvm_pk*($Pointer, Nitem, Stride$)—there is one pack routine for each individual data type $* \in \{byte, cplx, dcplx, double, float, int, long, short\}$[10]—packs the number $Nitem$ of data into the send buffer; $Pointer$ refers to the location of the first data item and $Stride$ to the relative distance to the next one. The formal definitions of these routines are given by the rules[11]:

pvm_initsend()

if $event(x) = initsend(Encoding)$ *from Task*
then
  extend $BUFID$ **by** b **with**
    $sendbuf(Task) := b$
    $cont(b) := \langle\rangle$
    $encoding(b) := Encoding$
    return $\langle b \rangle$ *to Task*
  endextend

pvm_pk*()

if $event(x) = pk^*(Pointer, Nitem, Stride)$ *from Task*
then
  $cont(sendbuf(Task)) :=$
    $append(data^*(Pointer, Nitem, Stride),$
      $cont(sendbuf(Task)))$

2.2. Sending

A message consists of receiver and sender tid, an integer tag, and the data, accessible through functions defined on $MESSAGE$ with values in TID ($recvtid$, $sendtid$), TAG ($msgtag$), $DATA^*$ ($msgdata$). Thus each pvmd x holds a sequence $msgseq(x)$ of quadruples $\langle RecvTid, SendTid, MsgTag, BufId \rangle$ representing the messages waiting—in the order of their arrival at x—to be received by one of its local tasks. *BufId* identifies the local buffer containing the message data.

---

communication, certain group functions, or multiple send and receive buffers.

[10] For packing strings a simpler routine is used which we do not describe here.

[11] Note that the updates in evolving algebra rules are thought of as being executed simultaneously.

pvm_send(*Tid, Tag*) puts the information, stored in the send buffer of the sending task, into a newly created message that is sent to *Tid* with label *Tag*. If the daemon of the receiver is the daemon of the sender, sending the message means to enqueue it into the daemon's message queue; if the message is expected by the receiver—expressed by a function $expecting : TASK \rightarrow \langle (TID + \{-1\}) \times (TAG + \{-1\}) \rangle$ as a combination of options for the message *Tag* and sender *Tid*, where '-1' indicates matching every possible combination—it is immediately put into the receiver's receive buffer. Otherwise the message will be forwarded as intertask–message. Upon arrival an intertask–message is put into the receiver's receive buffer (if expected by the receiver) or enqueued by the receiver's daemon. This meaning of the routine *pvm_send()* is formally defined by the rules[12]:

pvm_send()

if $event(x) = send(Tid, Tag)$ *from Task*
then
$CREATE\ \ mssg\ m : (Tid, Tag, Task)$
$WITH\ \ sendmsg(m, Tid, Tag, Task)$
$through\ x$

where

$sendmsg(M, Tid, Tag, Task)\ through\ x$
$\equiv$ **if** $pvmd(Tid) = x$
then $enqueue(M, msgseq(x))$
else *forward* $intertask\_msg\langle M \rangle$ *to* $pvmd(Tid)$

$CREATE\ \ mssg\ M : (Tid, Tag, Task)$
$WITH\ \ Updates$
$\equiv$ **extend** *MESSAGE* **by** M **with**
$msgdata(M) := cont(sendbuf(Task))$
$recvtid(M) := Tid$
$msgtag(M) := Tag$
$sendtid(M) := tid(Task)$
$Updates$
endextend

$enqueue(M, msgseq(x))$
$\equiv$ **extend** *BUFID* **by** b **with**
$cont(b) := msgdata(M)$
$msgseq(x) := msgseq(x)^\frown \langle Labels, b \rangle$
if *MatchRecvContext*
then
$recvbuf(receiver) := b$
$expecting(receiver) := undef$
endextend

intertask_msg$\langle\rangle$

if $event(x) = intertask\_msg\langle M \rangle$ *from* y
thenif *MatchRecvContext*
then
extend *BUFID* **by** b **with**
$cont(b) := msgdata(M)$
$recvbuf(receiver) = b$
endextend
$expecting(receiver) := undef$
else $enqueue(M, msgseq(x))$

where $Labels \equiv recvtid(M), sendtid(M), msgtag(M)$, $receiver \equiv task(recvtid(M))$,
$MatchRecvContext \equiv matching(\langle sendtid(M), msgtag(M) \rangle, expecting(receiver))$

2.3. Receiving

PVM supports blocking and non–blocking receive routines. Blocking *recv* requested from *Task* sets the receive buffer of *Task* to the bufid of an expected message and returns bufid to *Task*, if an appropriate message is actually waiting in the message queue;

[12]The corresponding multicast routine *pvm_mcast()* can be handled in almost the same way.

otherwise, the given receive context is used to update the *expecting* function of *Task*. Non–blocking *nrecv* does the same except for returning 0 to *Task* in case there is no message of the expected type in *msgseq*. Formally this is expressed by two rules, where the function:

$$\lceil : \ \langle TID \times TID \times TAG \times BUFID \rangle^* \times \langle TID \times (TID + \{-1\}) \times (TAG + \{-1\}) \rangle \rightarrow BUFID^*$$

selects, out of *msgseq*, the messages corresponding to *expecting* :

pvm_recv()

if $event(x) = recv(Tid, Tag)$ *from Task*
thenif $msgseq(x) \lceil MsgSelect = [\,]$
  then
    $expecting(Task) := \langle Tid, Tag \rangle$
  else
    $RecvMsg(Tid, Tag, Task)$ *Through* x

pvm_nrecv()

if $event(x) = nrec(Tid, Tag)$ *from Task*
thenif $msgseq(x) \lceil MsgSelect = [\,]$
  then
    return $\langle 0 \rangle$ *to Task*
  else
    $RecvMsg(Tid, Tag, Task)$ *Through* x

where $MsgSelect \equiv \langle tid(Task), Tid, Tag \rangle$
$RecvMsg(Tid, Tag, Task)$ *Through* x
$\equiv$ **let** $msgseq(x) \lceil MsgSelect = [b|bufids]$
return $\langle b \rangle$ *to Task*
$recvbuf(Task) := b$
$msgseq(x) :=$ *delete* $\langle tid(Task), Tid, Tag, b \rangle$ *from* $msgseq(x)$

3. Process Control

To exemplify the formal treatment of process control constructs, we present two significant examples.

3.1. Creating Tasks

The routine pvm_spawn() enables dynamic subtask creation. On initiating a spawn operation the spawning *task* becomes *parent* of the (*Ntask* many) new subtasks to be created, each of which is assumed to run a copy of a given executable *File* (possibly with a list of *Arg*ument*s*). The spawning task may affect the selection of hosts to spawn on through a mode parameter: in *transparent* mode tasks are automatically executed on the most appropriate hosts w.r.t. certain load measures; in *architecture-dependent* mode the calling task specifies the architecture; in *low-level* mode it specifies a particular host. Parameters *Flag* and *Where* are used to specify a combination of options as a sum of: *0 PvmTaskDefault* - PVM chooses where to spawn the processes; *1 PvmTaskHost* - the *Where* argument specifies a particular host to spawn on; *2 PvmTaskArch* - *Where* specifies a type of architecture to spawn on using ARCH; *4 PvmTaskDebug* - starts these processes up under debugger; *8 PvmTaskTrace* - the PVM calls in these processes will generate trace data[13].

In assigning tasks to hosts upon spawning, PVM's in principle non–deterministic choice depends in particular upon the environment (e.g. the operating system) and the internal

[13]This feature is not yet implemented, cf. [?].

load balancing scheme (which is transparent to the user). We abstract from details of this complex selection procedure by using a dynamic *external* function *hostselect* which we assume, as integrity constraint, to be consistent with the options set by the user (through *Flag, Where, Ntask*).

The outcome of an attempt to spawn a task which has been assigned to a host depends on the availability of resources, and on whether a suitable copy of the relevant *File* is present on that host. We model the behaviour of the system using again a dynamic external function *try_to_spawn* which provides the necessary SPAWNREPORT. Since computing this function may be rather time consuming and thus is not an atomic action, spawning has two phases, modeled by separate rules: a spawn operation is triggered by calling *try_to_spawn* with the appropriate argument values computed using *hostselect*. On completion of this operation *try_to_spawn* returns, to the calling pvmd, the resulting list of pids, one for each successfully created process required to spawn a new task, by generating an *ext_spawn_report* event.

if $event(x) =$
  $spawn(File, Args, Flag, Where, Ntask)$
    from Task
thenif $hostselect(Flag, Where, Ntask, x) =$
    $\langle local, remote \rangle$
  & $local = \langle x, n_0 \rangle$
  & $remote = \langle pvmd_1, n_1 \rangle, \ldots, \langle pvmd_m, n_m \rangle$
then
  let $tid = tid(Task)$
  extend *WID* **by** *wid* **with**
    $req\_info(wid) := tid$
    $rep\_info(wid) := \langle\rangle$
    $waitcount(wid) := n_m + signum(n_0)$
    if $n_0 > 0$
    then *trigger*
      $try\_to\_spawn\langle params, n_0, wid \rangle$
    if $n_m > 0$
    then *forward*
      $spawn\_req\_msg\langle params, n_i, wid, tid \rangle$
        to $pvmd_i \quad (i = 1, \ldots, m)$
  endextend

where $params \equiv File, Args, Flag, Where$

if $event(x) =$
  $ext\_spawn\_report\langle Wid, \langle Pid_1, ..., Pid_n \rangle \rangle$
then
  let $tid = req\_info(Wid)$
  extend TID_x **by** $tid_1, \ldots, tid_n$ **with**
    let $Tids = tid_1, \ldots, tid_n$
    extend *TASK* **by** $t_1, \ldots, t_n$ **with**
      $\vdots$
      $pvmd(tid_j) := x$
      $enroll(t_j, tid_j, tid, Pid_j)$
      $\vdots$
      if $pvmd(tid) = x$
      then $RepInfoSpawn(Tids, Wid)$
      else
        return $spawn\_rep\_msg\langle Tids,$
          $rep\_info(Wid)\rangle$ *to* $pvmd(tid)$
    endextend
  endextend
  where $enroll(Task, Tid, Parent, Pid)$
      $\equiv$ $tid(Task) := Tid$
        $parent(Task) := Parent$
        $pid(Task) := Pid$

When receiving an *ext_spawn_report*, the pvmd enrolls the reported processes as tasks. Depending on whether the task that has initiated the spawn request is local or not, the resulting list of new tids either is appended to the local wait context (in *RepInfoSpawn*) or it is returned to the corresponding remote pvmd. We omit the straightforward formalization of $RepInfoSpawn(Wid, Tids)$: the returned wait identifier *Wid* provides access to the local wait context where the request counter $waitcount(Wid)$ and the list *rep_info(Wid)*

of tids of successfully spawned tasks are updated; upon final completion, the result of the distributed spawn operation is returned to the calling task *req_info*(*Wid*).

The rules for interaction between the spawning task pvmd and the pvmds on selected remote hosts are:

spawn_req_msg⟨⟩

if *event*(*x*) =
  spawn_req_msg⟨*Params*, *Wid*, *Tid*⟩ *from y*
then
  extend *WID* **by** *wid* **with**
    req_info(*wid*) := *Tid*
    rep_info(*wid*) := *Wid*
  endextend
  trigger try_to_spawn⟨*Params*, *wid*⟩
where *Params* ≡ *File*, *Args*, *Flag*, *Where*, *N*

spawn_rep_msg⟨⟩

if *event*(*x*) =
  spawn_rep_msg⟨*Tids*, *Wid*⟩ *from y*
then
  RepInfoSpawn(*Tids*, *Wid*)

3.2. Killing Tasks

The routine *pvm_kill*(*Tid*) causes the pvmd to kill the task identified by *Tid*. In the formal description a symbolic system command, *kill_process*, is used to express the resulting interaction between PVM and the operating system. If the task to be killed resides on a remote processor, the local pvmd forwards a kill message to the corresponding remote pvmd. Upon receiving a kill_msg a pvmd acts as if it had received a *pvm_kill* by one of its local tasks.

pvm_kill()

if *event*(*x*) = *kill*(*Tid*) *from Task*
thenif *pvmd*(*Tid*) = *x*
then
  delete Tid from TID
  kill_process(*pid*(*task*(*Tid*)))
else
  forward kill_msg⟨*Tid*⟩ *to pvmd*(*Tid*)

kill_msg⟨⟩

if *event*(*x*) = *kill_msg*⟨*Tid*⟩ *from y*
then
  delete Tid from TID
  kill_process(*pid*(*task*(*Tid*)))

REFERENCES

1. G. A. Geist and V. S. Sunderam. Network-based concurrent computing on the PVM system. *Concurrency: Practice and Experience*, 4(4):293–311, June 1992.
2. C. C. Douglas, T. G. Mattson, and M. H. Schultz. Parallel Programming Systems for Workstation Clusters. Technical Report YALEU/DCS/TR-975, Dept of Computer Science, Yale University, August 1993.
3. Y. Gurevich. Evolving algebras – a tutorial introduction. *Bulletin of the EATCS*, (43):264–284, February 1991.
4. Y. Gurevich. Evolving Algebra 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*. Oxford University Press, 1994 (to appear).

5. A. Geist et al. PVM 3 User's Guide and Reference Manual. Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831, May 1993.
6. E. Börger and U. Glässer. A formal specification of the PVM architecture. Technical Report, 1994 (to appear).