7

Efficient composition and automatic initialization of arbitrarily structured PVM programs

J.Y. Cotronis
Dept. of Informatics, University of Athens
TYPA Builds., Panepistimiopolis, 157 71 Athens, GREECE
tel.: +30 1 7291885 fax: +30 1 7219 561 e-mail: cotronis@di.uoa.gr

Abstract

There are significant programming and methodological problems when developing PVM programs, the process communication structure of which does not form trees but arbitrary graphs. We present a design methodology, called Ensemble, and the appropriate PVM techniques and tools for the efficient composition of arbitrarily structured PVM programs. In Ensemble PVM programs are described by annotated Process Communication Graphs (PCGs) and the sequential program components are designed with open communication interfaces. The annotated PCGs are interpreted by a universal PVM program Loader which spawns processes and sets values to their communication interfaces, thus establishing the program communication structure. The program components are reusable without any modification in other PVM programs. Annotated PCGs are produced from PVM program scripts. The methodology may be applied to any message passing environment by developing specific annotations of the PCG, reusable program components and the program loader.

Keywords

PVM, program composition, reusable components, annotated process communication graphs

1 INTRODUCTION

PVM allows for the most general form of MIMD parallel computation, as programs in PVM may possess arbitrary control and dependency structures (Geist et al., 1994). At any point in the execution of a PVM program, the processes in existence may have arbitrary relationships between each other and any process may communicate and/or synchronize with any other. As

I. Jelly et al. (eds.), *Software Engineering for Parallel and Distributed Systems*

with all programming environments there are program categories that are well suited to the PVM characteristics, making them easy to implement, and others that are not well suited and are much more difficult to implement. Let us overview PVM's fundamental characteristics and examine their influence on the design and implementation of programs:

1. The underlying architecture of PVM is any host system running UNIX and some special cases for Massively Parallel architectures, which are viewed as virtual machines.
2. Hosts may run the PVM console, which allows the user to interactively start, query, modify the virtual machine. PVM programs may use the complete host system; distinct programs may use the same hosts.
3. A PVM process is a UNIX process running on a host machine. A process is spawned by its parent process. To run a PVM program the user spawns a root process.
4. Processes are identified by unique integer identifiers, called task identifiers (tid), which are generated upon process creation by pvm_spawn. The tid is only known to the spawning process and the spawned process may obtain its father's and its own tid by function calls.
5. Processes may be spawned at specific hosts. If no host is specified PVM chooses where to spawn them. There are also some other spawning options.
6. Process communication and synchronization are of two categories: 1) requiring process tids and possibly some message tag identifiers (tags), such as point to point asynchronous communication (pvm_send, pvm_recv, etc.) and muticast, sending the same value to a list of processes, and 2) requiring group definitions, such as bcast, sending the same value to processes in a group and barriers, where groups of processes synchronize.

Programming applications forming, in general, tree-like process communication dependencies, where each process communicates only with its parent and its children processes, is easy to program in PVM, as for example SPMD and master/slave programs. The parent process spawns its children processes and each child process obtains its parent's tid. However, programming arbitrarily structured programs, in which the dependency structures of processes form arbitrary graphs is not, in general, an easy task in PVM. Establishing graph-like process communication dependencies in PVM requires a substantial programming effort is in two directions:

1. Creating the processes according to the parent-child model.
2. Establishing the full graph communication. Processes have to obtain the tids of the processes with which they need to communicate. They already know their children's tids and they may easily obtain their parent's tid. The programmer has to program processes to obtain the tids of the rest of the processes with which it needs to communicate.

As arbitrary process graph structures are to be established, ad-hoc programming is used which depends on the specific communication dependencies of the program in hand. The explicit programming of the order of creation of processes and of establishing the communication has the following disadvantages:

1. It is an overhead effort, since it is enforced by PVM (parent-child process creation and identification of processes upon their creation) and not by the program specification.
2. It burdens the design and implementation of programs, since the extra coding makes programs more difficult to understand, to debug and to modify.
3. It limits the reusability and scalability of program components, since components involve code which relies on specific dependency structures.

In this paper we present the Ensemble methodology and its techniques and tools for the efficient composition and initialization of arbitrarily structured static PVM programs overcoming the above disadvantages. The Ensemble methodology comprises three facets:

1. The annotated Process Communication Graphs (PCGs). We use general PCGs, as a natural structure, representing the processes as nodes and the communication dependencies between them as arcs. PCGs have been extensively used in modeling (Andrews, 1991), in dynamic analysis and simulation (Pouzet et al., 1994; Schneider and Schaefers, 1993), in mapping techniques (Norman and Thanish, 1993), etc. We annotate nodes and arcs of PCGs with information a PVM program needs for the creation and communication of its processes. We consider the annotated PCGs as interpretable structures specifying the composition of PVM programs. The annotated PCGs are produced from program scripts, but may also be produced by a graphical tool.

2. The reusable program components. Processes in PVM are spawned by loading instantiations of executable program components. We have developed programming structures and principles for program components which permit their reusability as executable library components. Such program components do not assume any specific communication structure in which the processes instantiated from them are involved. They specify a general parametric interface with the type and possibly the number of its communication dependencies and all the actual parameters of the communication procedures, such as pvm_send and pvm_recv, refer to elements of the interface. A reusable program component may have any number of instantiations in the same PVM program, as well as in other PVM programs, each instantiation having its own communication dependencies. When a process is instantiated it should be given appropriate information setting values to its interface. This information is annotating the PCG and is sent by the PVM Loader.

3. The PVM Loader. A universal PVM process which automatically initializes PVM programs by interpreting the annotated PCGs. The PVM Loader visits the nodes of the annotated PCGs and spawns the appropriate processes (instantiations of reusable components) according to the annotation on the nodes. The Loader then sends the process interface information annotating the PCG to the processes it spawned.

The structure of the paper is as follows: in section 2 we present the annotated Process Communication Graphs and their script representation; in section 3 we present the structure and design principles of reusable programs in PVM; in section 4 we present the PVM Loader; in section 5 we demonstrate the methodology by composing PVM programs all consisting of two types of reusable components. In section 6 we present our conclusions and plans for future work.

2 THE ANNOTATED PROCESS COMMUNICATION GRAPHS

Before we define the PCGs and their annotation let us describe a distributed application which we shall use as a demonstrating example.

2.1 A distributed application: Get Maximum

There are processes instantiated from a terminal component which possess a value; all terminal processes or simply terminals need to get the maximum value possessed by any of them. To

limit the number of messages the terminals do not broadcast their own value to all others; instead, there are processes instantiated from a relay component to which groups of terminals send their values, for simplicity their tids. The relay processes or simply relays cooperate to find the maximum of the tids, which they then send to their respective groups of terminals.

The terminals have one communication dependency, that with their associated relay, which we call S (Server) type. The relays have two types of communication dependencies, one with their groups of terminals, which we call C (Client) type, and one with the relays, which we call P (Propagation) type. A relay may have any non negative number of C dependencies and P dependencies. The main actions of terminals and relays are:

The actions of a terminal	**The actions of a relay**
send tid to relay (to S type)	receive tids from the client terminals (from C type)
receive maximum tid from relay (from S type)	find the local maximum LM of tids
	send LM to all other relays (to P type)
	receive LMs from all other relays (from P type)
	find the global maximum GM
	send GM to its client terminals (to C type)

We like the program to be easily configurable, that is, to be possible to add or remove any number of terminal and/or relay processes, without any modification of the program components, i.e. the terminal and relay executables..

2.2 The elements of the PCG and their annotation

Processes will be depicted on PCGs by nodes comprised of two concentric circles (Figure 1). On the inner circle the type of dependencies are indicated. The inner circle depicts the general interface type of the program components. The arcs leaving the nodes indicate communication dependencies (of a specific type) with other processes. The points where the arcs cut the outer circle depict the actual interface of processes to other processes. Each point of intersection is called a **port** and is indexed by a unique positive integer within a port type. The arcs of the PCG connect ports of nodes. Under this scheme terminals and relays will be depicted on PCGs as in Figure 1 (a) and (b) respectively.

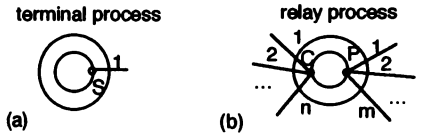

Figure 1 Graphical depiction of terminal and relay processes.

Let us assume, for example, that we have a configuration of eight terminals connected to four relays. The three C type ports of relay R[1] are connected with the S type ports of three terminals T[1], T[2] and T[3]; the two C type ports of relay R[2] are connected with the S type ports of two terminals T[4] and T[5]; the two C type ports of relay R[3] are connected with the S type ports of two terminals T[6] and T[7]; and finally the single C type port of R[4] is connected with the S type port of T[8]. All relays are connected to each other via their P ports. The PCG depicting the process dependencies is shown on Figure 2. The ports are indexed and connected according to the described configuration. As a matter of convenience the nodes are

indexed by positive integers. The elements of the PCG described so far specify a general PCG independent of any parallel implementation system.

Arcs on a PCG represent communication dependencies. For a complete communication specification in PVM, request identifiers, called tags, are needed which are used by both sending and receiving processes. The tag identifiers annotate the arcs of the PCG. In Figure 2 the arcs are annotated by unique positive integers, shown in bold.

Nodes may be further annotated by allocation information, if a process is to be spawned on a particular host. Finally, nodes are annotated by the full path name of the executable from which the process it represents will be instantiated. For reasons of simplicity allocations and executable path names are not depicted on Figure 2.

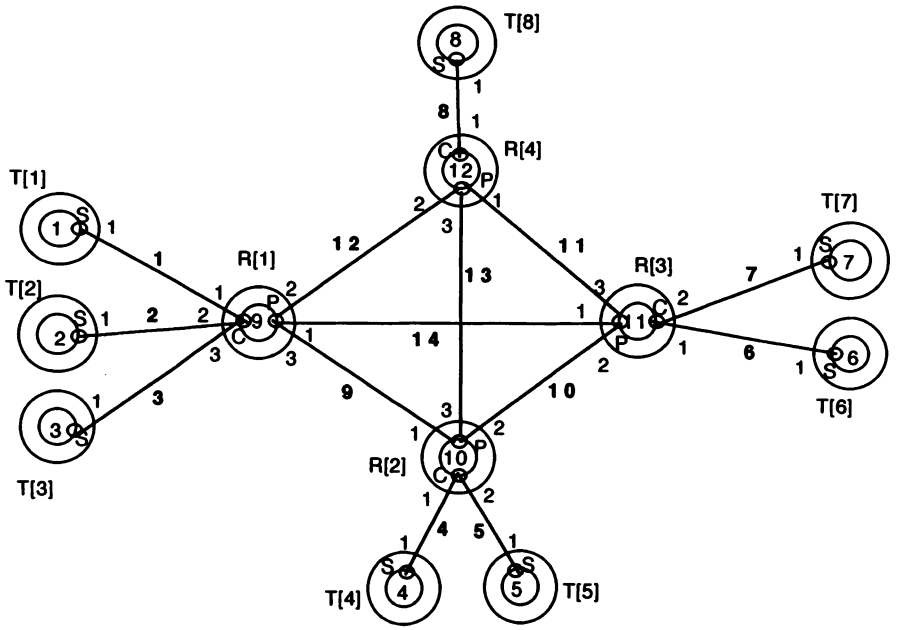

Figure 2 The annotated PCG of the application Get Maximum.

The annotated PCG may be interpreted by the PVM Loader to initiate the program. The annotated PCG may be produced by a graphical tool or by a textual description. We have developed a script language and programs which read a program script and produce the annotated PCG. A program script has three sections: the first describes the general PCG, the second the annotation of the PCG specific to a parallel environment (in this case PVM) and the third the annotation specific to the sequential components.

The script generating the annotated PCG of Figure 2, is presented in two columns in Figure 3. The first section, headed with `PCG`, defines the nodes and the number of ports for each type (e.g. all T nodes have one port of type S); it also defines the connections between the ports. The second section, headed with `Parallel System` defines the specific PCG annotation for the PVM. The compulsory annotation for RequestID, annotating the arcs, is specified; here the

default specifies the annotation of the arcs by unique positive integers, but generating algorithms or direct annotations may be defined. Also optional annotation may be specified; here all processes are allocated on specific hosts. The third section, headed with `Sequential System`, annotates the nodes of the PCG with the file locations of the executables of the sequential components from which processes are to be instantiated. From the program scripts annotated PCGs are produced which are interpreted by the PVM Loader initiating the PVM program.

```
Application Get_Maximum
PCG
Components
/* specify for each process the
number of ports of each type*/
 T[1], T[2], T[3], T[4], T[5],
 T[6], T[7], T[8] #ports = S:1;
 R[1]        #ports = C:3, P:3;
 R[2], R[3]  #ports = C:2, P:3;
 R[4]        #ports = C:1, P:3;
Connections
/* Connect process ports */
 T[1].S[1] <-> R[1].C[1];
 T[2].S[1] <-> R[1].C[2];
 T[3].S[1] <-> R[1].C[3];
 T[4].S[1] <-> R[2].C[1];
 T[5].S[1] <-> R[2].C[2];
 T[6].S[1] <-> R[3].C[1];
 T[7].S[1] <-> R[3].C[2];
 T[8].S[1] <-> R[4].C[1];
 R[1].P[1] <-> R[3].P[1];
 R[1].P[2] <-> R[4].P[2];
 R[1].P[3] <-> R[2].P[1];
 R[2].P[2] <-> R[3].P[2];
 R[2].P[3] <-> R[4].P[3];
 R[3].P[3] <-> R[4].P[1];
Parallel System
   environment PVM3;
   PVM3_annotation
   RequestID : default; /* annotate arcs
               by integer request Ids */
   PVM3_allocation
   /* specify the hosts on which
      processes are to be spawned */
   R[1], T[1], T[2], T[3]  at orion;
   R[2], T[4], T[5] at zeus;
   R[3], T[6], T[7] at ismini;
   R[4], T[8] at adonis;

Sequential System
   Location
  /* full path and name of executables */
    R:
    "/home/users/easy_spawn/bcast/relay";
    T:
 "/home/users/easy_spawn/bcast/terminal";
```

Figure 3 The script of the PVM program for Get Maximum

3 THE DESIGN OF REUSABLE PVM PROGRAM COMPONENTS

Reusability of compiled program components in a message passing environment demands that their process instantiations should be possible to establish the communication dependencies required by parallel programs. As the number of process instantiations and their communication dependencies cannot be fixed, the program components should specify the number and type of communication dependencies in a general way. They should only provide the means for establishing communication between any process instantiated from it with any other processes via an interface.

For establishing a point-to-point communication between PVM processes two values are needed in each process: the tid of the other process and the common tag identifier. Therefore, we define a data structure, called **component port**, having two elements in which (tid, tag) pairs may be stored. A program component may have any number of component ports of the same type, which are organized in an array. Finally, a program component may have many types of component ports. The types of ports form the array **Interface**, the elements of which

point to their array of ports. Each port is now identified by its type and its port index within the type.

Upon their creation processes should fix their interface. This involves two actions: the creation of the appropriate number of ports for each type and the setting of value pairs (tid, tag) to the port structures. We permit flexible process interfaces, as program components only fix the type of ports and not the actual number of the ports within types. Each process may have any number of ports of each type. Processes in our methodology are created by the PVM Loader which visits the PCG nodes and spawns processes according to the annotation of the node. The PVM Loader sends the number of ports of each type (depicted on the PCG node) to the process just created. The first action of a process is to receive the number of ports of each type and make the appropriate number of ports of each type in its Interface. This is coded in the `MakePorts` routine.

The value pairs (tid, tag) for each port cannot, in general, be sent at the time of process creation, as a process with which it needs to communicate may have not been spawned yet and its tid would not be known. The (tid, tag) pairs are send to the processes after all of the processes have been spawned, together with the type and index of the port. The processes receive the type, port number, tid and tag and set their Interface accordingly. This activity is coded in the `SetInteface` routine.

In Figure 4 we present the general structure of a reusable program component, which consists of a declaration of the `Interface` structure having N types of dependencies and as actions: a call of `MakePorts`, receiving from the Loader and making the appropriate number of ports of each dependency type; a call of `SetInterface`, receiving from the Loader and setting the values of the ports; and a call of `RealMain` which starts the main activity of the component. All PVM reusable components have the same structure; the programmer has only to replace N for the specific number of the types of ports and code the component activity in the `RealMain`, in which the parameters of the communication routines `pvm_send` and `pvm_recv` are expressions of the form `Interface[S].port[p].tid` and `Interface[S].port[p].tag`, where S is a port type and p is the number of port. By the time a process calls its `RealMain` its actual interface would be fixed.

```
void main()
{     InterfaceType Interface[N];
      MakePorts(Interface);
      SetInterface(Interface);
      RealMain(Interface);
}
```

Figure 4 The structure of reusable components in PVM

Having defined the annotated PCGs and the structure of the reusable components, we may describe the final facet of the methodology, the PVM Loader.

4 THE PVM LOADER

The PVM Loader is a universal PVM program by which PVM programs composed according to the methodology are initiated. The PVM Loader takes as input an annotated PCG and visits all its nodes; at each node the Loader spawns an instantiation of the executable file annotating the node. Then sends to the process just created the number of ports of each type and annotates the PCG node with the tid of the process. Having visited all nodes and created all processes,

the PVM Loader visits the nodes once more and sends the port interface information (port type, port number, tid, tag) to the processes.

Suppose, in our example (Figure 2), that the PVM Loader visits the node R[2] identified by 10: spawns process R[2], an instantiation of the program component relay and sends to it the number of its ports of each type, as shown in the first column of the following table:

actual values	explanation
C, 2	type C has 2 ports
P, 3	type P has 3 ports

The PVM Loader also annotates the node by the process tid, say tid(10). In its second visit to the node the Loader sends to the process identified by tid(10) information to set its interface. The values are shown in the first column of the following table:

actual values	explanation
C, 1, tid(4), 4	port 1 of type C is connected to tid(4) with tag 4
C, 2, tid(5), 5	port 2 of type C is connected to tid(5) with tag 5
P, 1, tid(9), 9	port 1 of type P is connected to tid(9) with tag 9
P, 2, tid(11), 10	port 2 of type P is connected to tid(11) with tag 10
P, 3, tid(12), 13	port 3 of type P is connected to tid(12) with tag 13

Running the PVM Loader with the annotated PCG as input we get the following output; the first column is produced by the Loader and the second by the terminal processes:

```
Spawn process 1 (terminal) tid=  c0005 [80004]  The maximum tid is 140005
Spawn process 2 (terminal) tid=  c0006 [100003] The maximum tid is 140005
Spawn process 3 (terminal) tid=  c0007 [80005]  The maximum tid is 140005
Spawn process 4 (terminal) tid= 140004 [140004] The maximum tid is 140005
Spawn process 5 (terminal) tid= 140005 [c0005]  The maximum tid is 140005
Spawn process 6 (terminal) tid=  80004 [c0006]  The maximum tid is 140005
Spawn process 7 (terminal) tid=  80005 [c0007]  The maximum tid is 140005
Spawn process 8 (terminal) tid= 100003 [140005] The maximum tid is 140005
Spawn process 9 (relay)  tid= c0008
Spawn process 10 (relay) tid= 140006
Spawn process 11 (relay) tid= 80006
Spawn process 12 (relay) tid= 100004
```

As the twelve processes, eight terminal and four relay are spawned, the PVM Loader prints their tids; the terminal processes print the global maximum of their tids. All terminal processes print the same maximum of #140005 which was the tid of process 5.

For a PVM program to behave correctly, the nodes on the PCG and the actual program components must be compatible, that is, they should specify the former virtually and the latter actually the same number of types of ports. Furthermore, the connections between ports should be of compatible type, that is connected components agree on the type of messages they exchange and their management. The present version of the PVM Loader does not check the compatibility of the connections. We are currently investigating formal methods for describing and testing component compatibility, which will be integrated in the PVM Loader.

The script language is flexible and permits the rapid composition of PVM programs. It is straight forward to edit scripts to scale a program, by adding and connecting new components,

to change the allocation of processes to hosts, change the topology of the components, etc., without modifying the program components.

5 VARIATIONS ON THE GET MAXIMUM PROGRAM

The specification for the Get Maximum program in section 2 did not specify any particular topology by which the relay processes should be connected. In our solution of section 2 we had adopted a topology in which all relay processes are connected with each other. We may achieve the same program functionality by adopting different relay topologies. We shall present two variations, one in which relay processes form a star topology and a second in which they form a tree topology. For these variations we will modify the scripts and not the components.

5.1 Get Maximum by star topology

In this solution we use an extra relay process to which the old four relay processes will be connected. The four relay processes have now only one P (propagation) port, through which they send the maximum value received from their terminals. The new relay process, let us call it central, has four ports of type C (clients). The P type ports of the four relay processes are connected to the C type ports of the central process! Let us note, that the C and the P ports of the relay processes are compatible, as only one value is sent and one value received through them. The PCG for this configuration is depicted in Figure 5:

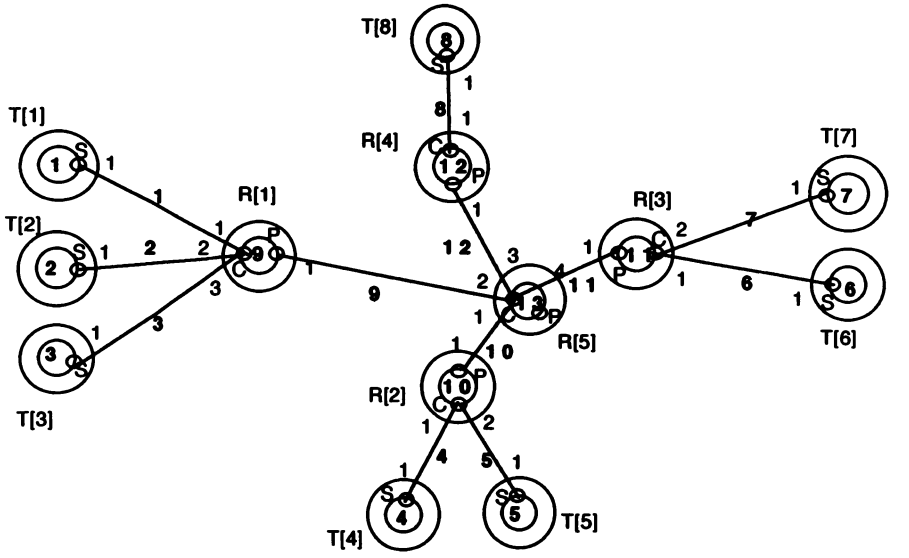

Figure 5 The PCG of Get Maximum by Star Topology.

Let us describe the behavior of the program in such configuration. The four relay processes, as before, select the maximum of the tids of their clients but now propagate it through their single port of type P to the central relay process. The central relay process receives tids from its C

ports and selects their maximum. There are no P ports to send the maximum. It then sends its maximum to its C ports. What actually sends is the global maximum, as it is the maximum of all maxima. On its C ports there are the four relay processes. Each receives the global maximum, but, according to the algorithm, they know that it is only the maximum of the central relay process. They compare it with their own maximum, select the value they have received and send it to their client ports. For this solution no changes were made to the terminal or to the relay program components, but only to the script. The executables of the terminal and relay program component were reused. The new program script was produced rapidly by modifying the program script of the version of section 2. From the script the annotated PCG was produced, which was given as input to the PVM Loader. The PCG part of the modified script and the final output of the program are in Figure 6:

Get-Maximum-Star	Program output
PCG	Spawn process 1 (terminal) tid= c000e
Components	Spawn process 2 (terminal) tid= c000f
T[1], T[2], T[3], T[4],T[5],	Spawn process 3 (terminal) tid= c0010
T[6], T[7], T[8]#ports = S:1;	Spawn process 4 (terminal) tid= 14000b
R[1] #ports = C:3, P:1;	Spawn process 5 (terminal) tid= 14000c
R[2], R[3] #ports = C:2, P:1;	Spawn process 6 (terminal) tid= 8000b
R[4] #ports = C:1, P:1;	Spawn process 7 (terminal) tid= 8000c
R[5] #ports = C:4, P:0;	Spawn process 8 (terminal) tid= 100008
Connections	Spawn process 9 (relay) tid= c0011
T[1].S[1] <-> R[1].P[1];	Spawn process 10 (relay) tid= 14000d
T[2].S[1] <-> R[1].P[2];	Spawn process 11 (relay) tid= 8000d
T[3].S[1] <-> R[1].P[3];	Spawn process 12 (relay) tid= 100009
T[4].S[1] <-> R[2].P[1];	Spawn process 13 (relay) tid= 4000a
T[5].S[1] <-> R[2].P[2];	[100008] The maximum tid is 14000c
T[6].S[1] <-> R[3].P[1];	[14000b] The maximum tid is 14000c
T[7].S[1] <-> R[3].P[2];	[14000c] The maximum tid is 14000c
T[8].S[1] <-> R[4].P[1];	[c000e] The maximum tid is 14000c
R[1].P[1] <-> R[5].S[1];	[8000b] The maximum tid is 14000c
R[2].P[1] <-> R[5].S[2];	[c000f] The maximum tid is 14000c
R[3].P[1] <-> R[5].S[3];	[c0010] The maximum tid is 14000c
R[4].P[1] <-> R[5].S[4];	[8000c] The maximum tid is 14000c

Figure 6 The PCG part of the script of the Get Maximum by Star Topology and the output.

5.2 Get Maximum by tree topology

In this variation we maintain the relationship of the eight terminals to the four relay processes having, as in the star solution, one P port. The P ports of R[1] and R[2] are connected with the C ports of R[5] and the P ports R[3] and R[4] are connected with the C ports of R[6]. Both R[5] and R[6] have two C ports and one P port; their P ports are connected to the two C ports of R[7], which does not have any P ports. The process structure is a tree of height 3: the terminal processes as leafs; R[1], R[2], R[3] and R[4] at level two; R[5] and R[6] at level one; and R[7] as the root. At each level, the relay processes receive the values from their clients, select the maximum and propagate it to the next level up. The root selects the maximum and sends it to its client processes. The relay processes below the root do the same until the maximum reaches the terminal processes. The script and the output are shown in Figure 7.

We have demonstrated the flexibility of the methodology by producing non trivial solutions for a program specification using the same reusable components. The program components were reused within the same PVM programs, as well as in other PVM programs. The only

changes required were in the program scripts. Although the script language is still under development, it has been successfully used to compose and execute programs from designs very rapidly.

Get-Maximum-Tree

```
PCG
Components
 T[1], T[2], T[3], T[4],T[5],
 T[6], T[7], T[8]    #ports = S:1;
 R[1]              #ports = C:3,P:1;
 R[2], R[3], R[5], R[6]
                   #ports = C:2,P:1;
 R[4]              #ports = C:1,P:1;
 R[7]              #ports = C:2,P:0;
Connections
   T[1].S[1] <-> R[1].C[1];
   T[2].S[1] <-> R[1].C[2];
   T[3].S[1] <-> R[1].C[3];
   T[4].S[1] <-> R[2].C[1];
   T[5].S[1] <-> R[2].C[2];
   T[6].S[1] <-> R[3].C[1];
   T[7].S[1] <-> R[3].C[2];
   T[8].S[1] <-> R[4].C[1];
   R[1].P[1] <-> R[5].C[1];
   R[2].P[1] <-> R[5].C[2];
   R[3].P[1] <-> R[6].C[1];
   R[4].P[1] <-> R[6].C[2];
   R[7].C[1] <-> R[5].P[1];
   R[7].C[2] <-> R[6].P[1];
```

Program output

```
Spawn process 1 (terminal) tid= c0016
Spawn process 2 (terminal) tid= c0017
Spawn process 3 (terminal) tid= c0018
Spawn process 4 (terminal) tid= 140011
Spawn process 5 (terminal) tid= 140012
Spawn process 6 (terminal) tid= 80011
Spawn process 7 (terminal) tid= 80012
Spawn process 8 (terminal) tid= 10000c
Spawn process 9 (relay)  tid= c0019
Spawn process 10 (relay) tid= 140013
Spawn process 11 (relay) tid= 80013
Spawn process 12 (relay) tid= 10000d
Spawn process 13 (relay) tid= 40010
Spawn process 14 (relay) tid= 40011
Spawn process 15 (relay) tid= 40012

[80011]  The maximum tid is 140012
[10000c] The maximum tid is 140012
[80012]  The maximum tid is 140012
[140011] The maximum tid is 140012
[c0016]  The maximum tid is 140012
[140012] The maximum tid is 140012
[c0018]  The maximum tid is 140012
[c0017]  The maximum tid is 140012
```

Figure 7 The PCG part of the script of the Get Maximum by Tree Topology and the output.

6 CONCLUSIONS

We have presented a design methodology, called Ensemble, by which we overcome the problems of composing arbitrarily structured static PVM programs. In the Ensemble methodology parallel PVM programs are virtually specified by annotated PCGs which are interpreted by one universal PVM Loader, spawning the PVM processes and establishing their communication dependencies. We produce PCGs from a script language. Although, the language is still under development it was shown to be flexible and permitted the rapid composition of PVM programs. It is straight forward to edit the script to scale a program, by adding and connecting new components, to change the allocation of processes to hosts, to change the topology, etc. We have proposed simple programming structures and principles for designing reusable PVM program components as library components. Program components are easy to write, as the main actions of program components are wrapped within fixed code segments. The programmer is not concerned with writing code for achieving a process topology.

We demonstrated the flexibility of the methodology, by composing various solutions to the Get Maximum problem. Having constructed the program components for the first solution we used them to compose and execute new PVM programs. This approach is related to the composition of object oriented applications by using objects and scripts (Nierstratz et al.,

1991), as it encourages a component oriented approach to application development. We shall pursue this aspect in future work.

The Ensemble methodology is not concerned with the efficiency of program execution. It supports the efficient composition and initialization of applications. The methodology affects the efficiency of the program execution only marginally; before the processes begin their main actions they have to call the `MakePorts` and `SetInterface` routines.

The Ensemble methodology may be applied to other message passing parallel environments by developing specific techniques and tools. We have applied it to the Massively Parallel architecture of PARSYTEC GC3/512 running the Parix environment (Cotronis, 1995). The Parix environment imposes altogether different constraints to programs than PVM. Parix requires different PCG annotation techniques, its own construction of reusable program components and its own Loader. We shall compare implementations of the methodology under PVM, Parix and other environments in a future report. We shall also investigate the portability of parallel programs developed with this methodology. Let us finally comment, that the script language and the structure of the reusable components are such that it seems possible to port programs by editing the annotation parts of scripts and by making new reusable components in the target environment by just changing the "wrapping code" of the `RealMain` procedure in the components.

7 REFERENCES

Andrews, G.R. (1991) Paradigms for Process Interaction in Distributed Programs, *ACM Computing Surveys*, Vol. 23, No.1, March 91.

Cotronis, J.Y. (1995) A Methodology for Initiating Arbitrary Structured Programs in Parix by Interpreting Graphs, in Proceedings of ZEUS 95 (ed. P. Fritzon and L. Finmo) IOS Press.

Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R. and Sunderam, V. (1994) PVM 3 User's guide and Reference Manual, ORNL/TM-12187, May 1994.

Nierstratz, O., Tsichritzis, D., de Mey V. and Stadelmann, M. (1991) Objects + Scripts = Applications, in Proceedings of Esprit 1991 Conference, Kluwer Academic Publishers.

Norman, M.G. and Thanisch, P. (1993) Mapping in Multicomputers. *ACM Computing Surveys*, Vol. 25, No.3.

Pouzet, P., Paris, J. and Jorrand, V. (1994) Parallel Application Design: The Simulation Approach with HASTE, in Proceedings of. HPCN, Munich, Vol II.

Scheidler, C and Schaefers, L. (1993) TRAPPER: A Graphical Programming Environment for Industrial High-Performance Applications, in Proceedings of PARLE Conf., Munich.

8 BIOGRAPHY

Dr. J.Y.Cotronis obtained his Ph.D. in Computer Science in 1982 from the Computing Laboratory, University of Newcastle-upon-Tyne, where he also worked as a Research Associate in projects in the area of parallelism. He has been involved in a number of R&D projects in industry and academia. He is an Assistant Professor and his current research interests are on methodologies and supporting tools for composing and porting parallel applications.