They take in one argument: an integer array, `ArrOfChan`, containing the values of the 'logical channels' (i.e. message identifiers) to be checked for messages. They return the buffer location, `BufID`, and the channel identity (i.e. message identifier), `ChanID`, of the message picked. Note that these routines are planted automatically by the code generator corresponding to the design.

5. Conclusion

This paper has introduced the basic concepts behind the PCSC methodology for parallel system design and briefly outlined the reasons for choosing PVM as a target implementation environment. The PCSC and PVM models were discussed and their differences compared. It was shown that the fundamental difference between the two models is the absence of the channel concept in PVM. The techniques adopted to conform PVM to PCSC model of computation were explained and illustrated. How C code for PVM is generated automatically from the PCSC design was also briefly indicated.

Currently, a working prototype of the C-PVM code translator has been implemented and is being tested. Work is in progress to implement a meta-code generator which would take code templates and automatically generate from them the appropriate code generator for the specified parallel programming language.

References

[1] Manson, G.A., and Boyle, A., A CASE Tool for Designing Parallel Systems: The Architectural Model, Proceedings of the PACTA '92, Barcelona, Spain, September 1992.

[2] Manson, G.A., and Sahib, S., An Illustration of the Parallel Communicating Sequential Methodology, Workshop on Software Engineering for Parallel Systems, World Transputer Conference '93 (WTC '93), Aachen, Germany, 20 - 22 September 93.

[3] Fisher, A.S., CASE Using Software Development Tools, John Wiley and Sons, Inc., 1988.

[4] Cachia, E.A., and Manson, G.A., CCDM - A Design Methodology for Modelling Communicating Code in Parallel Systems, Proceedings of the 16th WoTUG Technical Meeting, Sheffield, UK, March 28-31, 1993, pp. 100 - 114.

[5] Sunderam, V.S., PVM: A framework for parallel distributed computing, Concurrency: Practice and Experience, 2(4):315-339, December 1990.

[6] Galletly, J., Occam 2, Pitman Publishing, 1990.

[7] Ingevaldsson, L., JSP-A Practical Method of Program Design, Chartwel-Brat Ltd., 1986.

[8] Loh B.C. and Manson G.A., Incorporating Software Reuse into the PCSC Methodology, Poster Presentation, World Transputer Conference '94 (WTC '94), Como, Italy, Sept. 1994.

[9] Geist, A., et. al., PVM 3 User's Guide and Reference Manual, ORNL/TM-12187, May 1994.

[10] Elamvazuthi, C. and Manson, G.A., Occam, PVM and the Alternation Construct. Proceedings of the 17th WoTUG Technical Meeting, Bristol, England, 11-13 April, 1994, pp. 56-58.

[11] Elamvazuthi, C., Hussain, A.M.A., Manson, G.A. and Sahib, S., Behavioural Modelling of a Communicating Process, Proceedings of the 1994 World Transputer Conference (WTC '94), Como, Italy, Sept. 1994, pp. 568-583.

[12] Interactive Development Environments (IDE), Software through Pictures, Fundamentals of StP Release 1, February 1994.

```
#include "comm.h"

int p_inch0, p_inch1, p_inch2, p_inch3, p_outch4;
int inch0, inch1, inch2, inch3, outch4;
main(argc,argv)
int argc;
char *argv[];
{
  if ((mytid = pvm_mytid()) < 0)  exit(1);
  pvm_recv(pvm_parent(), 0);
  pvm_upkint(&num_proc, 1, 1);
  pvm_upkint(task_id, num_proc, 1);
  p_inch0  = atoi(argv[1]);/* Assignments for process */
  p_inch1  = atoi(argv[2]);/* indices */
  p_inch2  = atoi(argv[3]);
  p_inch3  = atoi(argv[4]);
  p_outch4 = atoi(argv[5]);

  inch0    = atoi(argv[6]);/* Assignments for message */
  inch1    = atoi(argv[7]);/*tags for channel uniqueness*/
  inch2    = atoi(argv[8]);
  inch3    = atoi(argv[9]);
  outch4   = atoi(argv[10]);

  ...
  ArrOfChan[0] = inch1;
  ArrOfChan[1] = inch2;
  ArrOfChan[2] = EOA;
  ChanID = alt(ArrOfChan, &BufID);
    switch(ChanID) {
      case inch1:
  ... /* receive msg1 */
  ... /* action1 */
        break;
      case inch2:
  ... /* receive msg2 */
  ... /* action2 */
        break;
      default: break;
    }
  ...
  pvm_send(task_id[p_outch4],outch4);
  ...
  pvm_exit();
}
```

Figure 8: Code fragment for P3.

The implementation of `prialt` has an additional feature. The global receive queue is scanned for the highest priority message (or channel) first. If it is not available, then the queue is scanned again for the next highest priority message until a message is picked. If there are no messages of the required types, the routine block-waits for any one message from the required set. The scan-and-pick procedure and block-waiting are handled by `pvm_nrecv` and `pvm_recv`, respectively. Here again, before invoking `pvm_recv`, the default match routine is replaced with `alt_match`.

Both alt and prialt have the following C call syntax:

```
int ChanID = alt(int ArrOfChan[], int *BufID) ;
int ChanID = prialt(int ArrOfChan[], int *BufID) ;
```

```
#include <stdio.h>  /* relevant contents of comm.h */
#include <string.h>
#include "pvm3.h"
...
#define P0  "0"   /* Unique numbers assigned to    */
#define P1  "1"   /* processes and channels during */
#define P2  "2"   /* auto-code generation.         */
#define P3  "3"
#define P4  "4"
#define P5  "5"
#define ch0 "0"
#define ch1 "1"
#define ch2 "2"
#define ch3 "3"
#define ch4 "4"
...
```

Figure 6: Relevant contents of comm.h included by all processes in the system.

```
#include "comm.h"
main()
{
 ...
 char *arg[11];

 mytid = pvm_mytid();
 ...
 arg[0] = strdup(P0);/*Runtime assignment(DFD level)*/
 arg[1] = strdup(P1);
 arg[2] = strdup(P2);
 arg[3] = strdup(P3);
 arg[4] = strdup(P4);
 arg[5] = strdup(ch0);
 arg[6] = strdup(ch1);
 arg[7] = strdup(ch2);
 arg[8] = strdup(ch3);
 arg[9] = strdup(ch4);
 arg[10] = NULL;
 pvm_spawn("P5",arg,1,"",1,&task_id[atoi(P5)]);
 ...
pvm_initsend(PvmDataDefault);
 pvm_pkint(&num_proc, 1, 1);
 pvm_pkint(task_id, num_proc, 1);
 pvm_mcast(task_id, num_proc, 0);
 ...
}
```

Figure 7: Code fragment for the master process for the DFD in Figure 3.1.

routines were written using the PVM primitives, `pvm_recv`, `pvm_nrecv` and `pvm_recvf`.

Since a channel is defined by giving the message tag a unique value, the `pvm_recv` routine can only block-wait on one channel. What is needed is for `pvm_recv` to block-wait on a number of selected channels i.e. match a prescribed set of messages. Hence a new matching function called `alt_match` having this functionality was written. Therefore the `alt` routine basically uses the `pvm_recvf` function to replace `alt_match` routine before invoking `pvm_recv`.

3.5 *Mapping Data Types*

The data model in PCSC is described using DSDs which is the data model from JSP [7]. However, the allowed basic data types in PCSC design model have been limited to a number sufficient for any design purpose. The set of data types is very similar to the one supported by occam. Proper mapping has been made from this set of basic data types onto the C-PVM basic data types. The DSD also allows abstract data types such as arrays, unions, records or the combination of them, to be described.

4. Implementation Details

The PCSC methodology is currently being implemented using a commercial CASE tool called Software through Pictures (StP) [12]. StP is a configurable CASE tool and provides sufficient leeway to test the ideas of PCSC. It already supports a number of tools (such as diagram editors) which has been customised for PCSC. All information pertaining to all aspects (such as requirements, design, etc.) of an application is stored in a central repository; this aids in enforcing the consistency requirement of PCSC.

The information captured through the design is extracted from the repository and the appropriate language code is generated and stored in files. Since each diagram editor has different properties, three separate code generators were needed. The DFD code generator generates the master process which spawns all the sub-processes. The most complex is the CSD code generator which integrates the DSD code generator. This CSD code generator translates the CSD syntax to C-PVM.

4.1 *Channel Mapping*

Implementation of the channel mapping can be split into two levels; the DFD level and the CSD level. In the DFD level, the following steps are implemented (refer to Figure 3.1, Figure 6 and Figure 7): to each process a unique number is assigned which is used to index the process identifier array containing the process identifiers of all the processes generated in the system; the master program will pass the indices (unique numbers) as arguments to the processes to be spawned; once all processes are spawned, the master program sends out to each process the process identifier array. Similarly, each channel is allocated a unique number which is passed as an argument to the process spawned. Note that the unique numbers for both processes and channels are generated automatically by the auto-code generator.

At the CSD level (refer to Figure 3.2 and Figure 8), each individual process will pick the appropriate argument values and assign them to the index variables (i.e. the formal channel names prefixed with '`p`'). These variables will be used to index into the process identifier (*task_id*) array, received from the master process, to obtain the process identifier of the relevant process in the system. Similarly, the unique channel numbers are assigned to the message tag variables (i.e. the formal channel names).

4.2 *Alternation Constructs*

To ease the mapping of the alternation constructs onto C-PVM, two routines were developed, viz., `alt` and `prialt`. The former is used for mapping the non-deterministic ALT construct and the latter, the deterministic PRIALT construct. The `alt` and `prialt`

needs to take on different values. This is illustrated in Figure 4.2. The skeletal CSD for process S (sender) contains a FOR construct which uses C-like syntax and semantics for the control expression [11]. The OUT construct is used to send out a tagged message sequentially to all the processes P0 to P[i] (in this example `i = 9`). Note that the channel `outch` (corresponding to `ch`) uses the index `i` from the FOR construct. Hence, this channel index needs to be a variable which can be assigned with values dynamically.

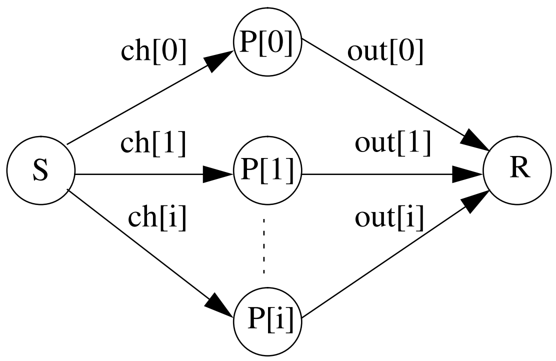

Figure 4.1: Replicated processes.

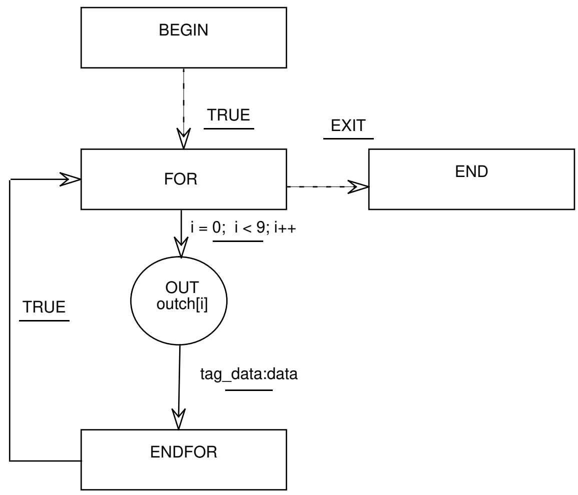

Figure 4.2: Skeletal CSD for process S in Figure 4.

3.4 Mapping Standard I/O

Figure 5 illustrates a DFD where an input process reads values from the keyboard and passes them to a sum process. The sum process, after operating on the values, passes the result to the output process which then outputs to the screen. When the DFD (i.e., the design) is mapped onto PVM, a master process is necessary to not only spawn all the three processes but also to take care of the standard I/O i.e. keyboard and screen. It must be stressed that in PVM only the master process can perform standard I/O. Note that the dotted part of the system is created by the auto-code generator and is not part of the original logical design.

This mapping was done before the `pvm_catchout` routine was made available in the later release of PVM (i.e. version 3.3) [9]. The availability of the routine does make the task of handling standard output easier since explicit messages need not be sent. However, since standard input still has to be dealt with separately, the above I/O mapping is still necessary.

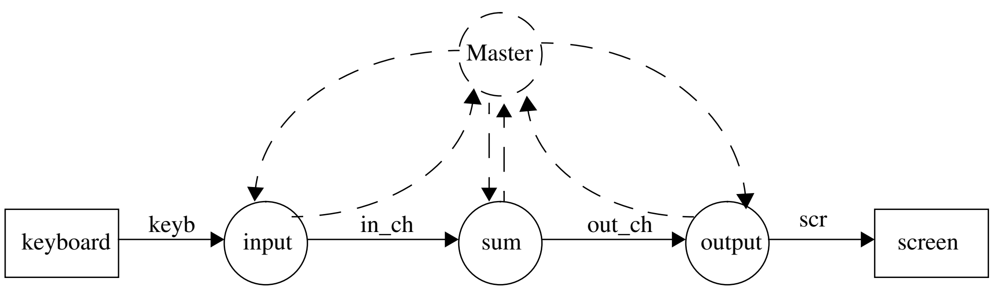

Figure 5: Standard I/O mapping for PVM.

The formal channel name in the CSD is used as the name of the message tag variable. In the above example, `outchan1` and `inch1` are the formal names for `ch1` in the CSDs of P0 and P1, respectively. In this case, both `outchan1` and `inch1` will be assigned the value of `ch1`, i.e., `1`. Again, note that the above naming convention has been adopted to be consistent with the channel names in the design although it could be made arbitrary.

Since the PVM model supports point-to-point communication (i.e. task-to-task) and guarantees message order [9], the above technique effectively maps the channel concept of the PCSC methodology onto PVM.

3.2 Mapping CSD Alternation Constructs

The PCSC model allows the selection of a channel input based on the state of the channel and any associated state variable. The CSD alternation (ALT and PRIALT) constructs are used for this purpose [10]. Consider the DFD in Figure 3.1; P0 to P5 are processes whilst `ch0` to `ch4` are channels. P5 receives messages on channels `ch0` to `ch3` and sends messages out on `ch4`. Assume that P5 wants to perform an alternation on two input channels, viz., `ch1` and `ch2`. This means that the alternation should only pick a message on `ch1` or `ch2` and leave all those messages coming in on `ch3` and `ch4` alone. The bare-bone CSD fragment for process P5 illustrating use of the ALT construct is given in Figure 3.2. A message (`msg1` or `msg2`) is picked from `inch1` (i.e. `ch1`) or `inch2` (i.e. `ch2`), operated on (`action1` or `action2`) and the `result` output to `outch4`.

The style of channel mapping discussed in the previous section allowed the development of alternation routines for use by the code generator. Note that channel uniqueness is mandatory for supporting the alternation concept.

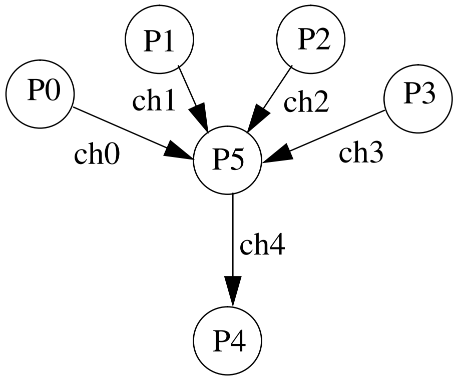

Figure 3.1: DFD to illustrate ALT construct.

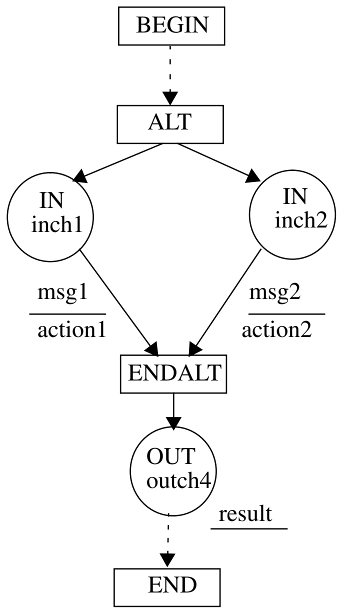

Figure 3.2: Bare-bone CSD for P5.

3.3 Mapping Process Replication

The PCSC design model supports process replication. Figure 4.1 is a DFD of a system of processes illustrating process replication. In this case the process P[i] is replicated, the code for that process is reused. Therefore at each instantiation of a P[i] process, the channel index

whilst `ch0` is a channel. P0 sends messages via `ch0` to P1; P1 receives them. In PVM, this can be written thus:

P0 sends: `pvm_send(task_id[p_outchan0], mesg_tag);`
P1 receives: `pvm_recv(task_id[p_inch0], mesg_tag);`

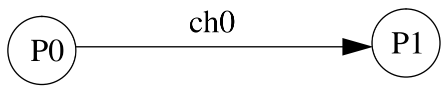

Figure 1: A simple DFD showing two processes communicating via a channel.

`task_id` is the array of all process identifiers which is received from the master process that spawns all the processes in the system. The indices (variables) into the `task_id` array are the channel names (defined within CSD) prefixed with a '`p`' to indicate *process*. This naming convention has been adopted to be consistent with the channel names in the design but it could be made arbitrary. These channel names, `outchan0` and `inch0`, are formal names (i.e. local to the CSD) as compared to `ch0` which is the actual name. The formal-actual pairing is carried out by the user during the design phase. This information is later used during automatic code generation.

Consider now the case of P0 communicating with P1 via another additional channel, `ch1`, as illustrated in Figure 2. Using the notation discussed above, this communication over `ch1` can be written thus:

P0 sends: `pvm_send(task_id[p_outchan1], mesg_tag);`
P1 receives: `pvm_recv(task_id[p_inch1], mesg_tag);`

However, since the receiving process is the same in both cases, i.e. P1, the variables `p_outchan0` and `p_outchan1` would have the same value, i.e., they would map to the same process identifier. Looking from the side of P1, the same applies to `p_inch0` and `p_inch1` since the sending process is the same in both cases, i.e. P0. Hence, this mapping alone is insufficient to preserve the channel uniqueness of the PCSC model. However, by tagging all messages on a particular channel with an unique number, the channels can be differentiated.

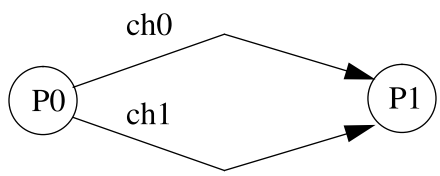

Figure 2: P0 and P1 communicating over two channels.

If the message identifier is associated with the *channel name*, all messages bearing the same identifier can be considered to be sent or received on that *logical channel*. Conversely, each channel name can be uniquely defined with an integer value, for example, `ch0 = 0`, `ch1 = 1`, and these values assigned to the messages to form the logical channels. This is illustrated below for the case of `ch1`:

P0 sends: `pvm_send(task_id[p_outchan1], outchan1);`
P1 receives: `pvm_recv(task_id[p_inch1], inch1);`

verified against the Requirements Specifications, and the Design Specifications are verified against the Analysis Specifications.

The design model is based upon the occam model of parallel computation [6]. In the occam model, an application is composed of concurrent processes that communicate between one another via channels. The designer starts off by determining the processes that make up the application and links them with appropriate unidirectional channels. This is done with Data Flow Diagrams (DFDs). Next, the channel messages are defined using Data Structure Diagrams (DSDs). The individual processes are then described with Communicating State Diagrams (CSDs). The CSD is an in-house developed technique based on the State Transition Diagram (STD) in which logic related to communication is described using CSD constructs and data structures and logic **not** related to communication are described using DSDs and Jackson Structured Diagrams (JSDs) [7]. Information that cannot be represented graphically is captured using annotations.

A design can include reusable components [8] as well as reverse engineered components. The deliverable of the design phase is the design specifications. These specifications together with those from the configuration phase are then used to generate the code automatically for the system. Once code is generated, it is executed on the target platform. If there are errors, the design is debugged and the code regenerated and executed. The methodology does not encourage the manipulation of the code. On successful execution, the system can be analysed for performance improvement. This can be achieved by optimising the design and/or the configuration.

3. Mapping of PCSC Design Model onto PVM

As mentioned earlier, the PCSC methodology has its roots in the occam model of computation. In the occam model, the only means of communication between processes is via *channels*. A channel connects only two processes. If one end of a channel is being written to, the other end must be read from, i.e. a channel is *unidirectional*. Channel communication is *synchronised*; a message is only sent when both the sending and receiving processes are ready. If one process becomes ready first, it will wait for the other to become ready. Hence, messages are not *buffer*ed. A channel is, therefore, an unbuffered, one-way, point-to-point link between two processes.

In PVM, however, the concept of channel is absent. Processes communicate by sending and receiving messages. Messages can be sent to a single task (*unicast*) or to a group of tasks (*multicast*). Message sending is *asynchronous*; the send primitives return as soon as the message is safely on its way. Messages are received by a source or label identification either in *blocking* or *non-blocking* mode. Messages are *buffer*ed. Buffers are allocated dynamically. Hence, the maximum size of a message sent or received depends on the memory available in a given host.

Comparing the programming paradigms of occam and PVM models, it can be seen that although both are message-based, it is the concept of channel (and its ramifications) that sets the occam model apart from PVM. In the occam model, messages can only be read from or written to *named* channels. In contrast, in PVM, messages are tagged uniquely and the destination (or source) process is specified when the messages are sent (or received).

3.1 Channel Mapping

Since PVM does not support the concept of channels, some method must be devised to model the channel concept for PVM. Consider the DFD in Figure 1; P0 and P1 are processes

Mapping the PCSC Design Model to PVM

G.A. Manson, C. Elamvazuthi, S. Sahib
Parallel Processing Research Group, Department of Computer Science
University of Sheffield, Regent Court, Portobello Street, Sheffield, S1 4DP, U.K.
Email: G.Manson@dcs.shef.ac.uk

Abstract: A methodology for parallel system design called Parallel Communicating Sequential Code (PCSC) is currently being developed. A CASE tool based on this methodology is also being built which automatically generates C code for the PVM environment. This paper highlights the mapping of PCSC model onto PVM to make the automatic code generation possible.

1. Introduction

An integrated CASE tool incorporating a design methodology for parallel program design is being developed at the department of Computer Science, University of Sheffield [1].

The methodology known as Parallel Communicating Sequential Code (PCSC) [2] has two basic aims:

• to produce designs that will be independent of language and platform (system software and machine architecture). The designs are to be portable to a range of parallel machine architectures as well as to heterogeneous multicomputer environments.

• to ensure consistency between design, configuration (mapping of application to processors), code and documentation of an application. This aspect is very important for the maintenance of the application.

The design information is captured in a visual-cum-textual abstract language consisting of (modified) Data Flow Diagrams (DFDs) [3], Data Structure Diagrams (DSDs) [3], Communicating State Diagrams (CSDs) [4] and annotation (text). From this abstract design language, code in various suitable programming languages is to be produced automatically.

One target environment is C-PVM [5]. The main reason for choosing PVM is its availability on a wide range of computers. Thus, a parallel design that has been translated into C-PVM should be able to run on different machine architectures without any change in the original design; hence, meeting the portability criterion. Our familiarity with C and UNIX was also a factor in the choice of PVM: the availability of source code for PVM is a plus point for it promotes a better understanding of the system and is quite useful for debugging purposes. It must also be mentioned that the free availability of PVM was also a factor in its choice.

2. PCSC Methodology

The PCSC methodology has a number of phases. Requirements Capture, Analysis and Design are tied together by the process of verification. The Analysis Specifications are

They take in one argument: an integer array, `ArrOfChan`, containing the values of the 'logical channels' (i.e. message identifiers) to be checked for messages. They return the buffer location, `BufID`, and the channel identity (i.e. message identifier), `ChanID`, of the message picked. Note that these routines are planted automatically by the code generator corresponding to the design.

5. Conclusion

This paper has introduced the basic concepts behind the PCSC methodology for parallel system design and briefly outlined the reasons for choosing PVM as a target implementation environment. The PCSC and PVM models were discussed and their differences compared. It was shown that the fundamental difference between the two models is the absence of the channel concept in PVM. The techniques adopted to conform PVM to PCSC model of computation were explained and illustrated. How C code for PVM is generated automatically from the PCSC design was also briefly indicated.

Currently, a working prototype of the C-PVM code translator has been implemented and is being tested. Work is in progress to implement a meta-code generator which would take code templates and automatically generate from them the appropriate code generator for the specified parallel programming language.

References

[1] Manson, G.A., and Boyle, A., A CASE Tool for Designing Parallel Systems: The Architectural Model, Proceedings of the PACTA '92, Barcelona, Spain, September 1992.

[2] Manson, G.A., and Sahib, S., An Illustration of the Parallel Communicating Sequential Methodology, Workshop on Software Engineering for Parallel Systems, World Transputer Conference '93 (WTC '93), Aachen, Germany, 20 - 22 September 93.

[3] Fisher, A.S., CASE Using Software Development Tools, John Wiley and Sons, Inc., 1988.

[4] Cachia, E.A., and Manson, G.A., CCDM - A Design Methodology for Modelling Communicating Code in Parallel Systems, Proceedings of the 16th WoTUG Technical Meeting, Sheffield, UK, March 28-31, 1993, pp. 100 - 114.

[5] Sunderam, V.S., PVM: A framework for parallel distributed computing, Concurrency: Practice and Experience, 2(4):315-339, December 1990.

[6] Galletly, J., Occam 2, Pitman Publishing, 1990.

[7] Ingevaldsson, L., JSP-A Practical Method of Program Design, Chartwel-Brat Ltd., 1986.

[8] Loh B.C. and Manson G.A., Incorporating Software Reuse into the PCSC Methodology, Poster Presentation, World Transputer Conference '94 (WTC '94), Como, Italy, Sept. 1994.

[9] Geist, A., et. al., PVM 3 User's Guide and Reference Manual, ORNL/TM-12187, May 1994.

[10] Elamvazuthi, C. and Manson, G.A., Occam, PVM and the Alternation Construct. Proceedings of the 17th WoTUG Technical Meeting, Bristol, England, 11-13 April, 1994, pp. 56-58.

[11] Elamvazuthi, C., Hussain, A.M.A., Manson, G.A. and Sahib, S., Behavioural Modelling of a Communicating Process, Proceedings of the 1994 World Transputer Conference (WTC '94), Como, Italy, Sept. 1994, pp. 568-583.

[12] Interactive Development Environments (IDE), Software through Pictures, Fundamentals of StP Release 1, February 1994.

```
#include "comm.h"

int p_inch0, p_inch1, p_inch2, p_inch3, p_outch4;
int inch0, inch1, inch2, inch3, outch4;
main(argc,argv)
int argc;
char *argv[];
{
  if ((mytid = pvm_mytid()) < 0)  exit(1);
  pvm_recv(pvm_parent(), 0);
  pvm_upkint(&num_proc, 1, 1);
  pvm_upkint(task_id, num_proc, 1);
  p_inch0  = atoi(argv[1]);/* Assignments for process */
  p_inch1  = atoi(argv[2]);/* indices */
  p_inch2  = atoi(argv[3]);
  p_inch3  = atoi(argv[4]);
  p_outch4 = atoi(argv[5]);

  inch0    = atoi(argv[6]);/* Assignments for message */
  inch1    = atoi(argv[7]);/*tags for channel uniqueness*/
  inch2    = atoi(argv[8]);
  inch3    = atoi(argv[9]);
  outch4   = atoi(argv[10]);

  ...
  ArrOfChan[0] = inch1;
  ArrOfChan[1] = inch2;
  ArrOfChan[2] = EOA;
  ChanID = alt(ArrOfChan, &BufID);
    switch(ChanID) {
      case inch1:
  ... /* receive msg1 */
  ... /* action1 */
        break;
      case inch2:
  ... /* receive msg2 */
  ... /* action2 */
        break;
      default: break;
    }
  ...
  pvm_send(task_id[p_outch4],outch4);
  ...
  pvm_exit();
}
```

Figure 8: Code fragment for P3.

The implementation of `prialt` has an additional feature. The global receive queue is scanned for the highest priority message (or channel) first. If it is not available, then the queue is scanned again for the next highest priority message until a message is picked. If there are no messages of the required types, the routine block-waits for any one message from the required set. The scan-and-pick procedure and block-waiting are handled by `pvm_nrecv` and `pvm_recv`, respectively. Here again, before invoking `pvm_recv`, the default match routine is replaced with `alt_match`.

Both alt and prialt have the following C call syntax:

```
int ChanID = alt(int ArrOfChan[], int *BufID) ;
int ChanID = prialt(int ArrOfChan[], int *BufID) ;
```

```
#include <stdio.h>  /* relevant contents of comm.h */
#include <string.h>
#include "pvm3.h"
...
#define P0  "0"   /* Unique numbers assigned to    */
#define P1  "1"   /* processes and channels during */
#define P2  "2"   /* auto-code generation.         */
#define P3  "3"
#define P4  "4"
#define P5  "5"
#define ch0 "0"
#define ch1 "1"
#define ch2 "2"
#define ch3 "3"
#define ch4 "4"
...
```

Figure 6: Relevant contents of comm.h included by all processes in the system.

```
#include "comm.h"
main()
{
 ...
 char *arg[11];

 mytid = pvm_mytid();
 ...
 arg[0] = strdup(P0);/*Runtime assignment(DFD level)*/
 arg[1] = strdup(P1);
 arg[2] = strdup(P2);
 arg[3] = strdup(P3);
 arg[4] = strdup(P4);
 arg[5] = strdup(ch0);
 arg[6] = strdup(ch1);
 arg[7] = strdup(ch2);
 arg[8] = strdup(ch3);
 arg[9] = strdup(ch4);
 arg[10] = NULL;
 pvm_spawn("P5",arg,1,"",1,&task_id[atoi(P5)]);
 ...
 pvm_initsend(PvmDataDefault);
 pvm_pkint(&num_proc, 1, 1);
 pvm_pkint(task_id, num_proc, 1);
 pvm_mcast(task_id, num_proc, 0);
 ...
}
```

Figure 7: Code fragment for the master process for the DFD in Figure 3.1.

routines were written using the PVM primitives, pvm_recv, pvm_nrecv and pvm_recvf.

Since a channel is defined by giving the message tag a unique value, the pvm_recv routine can only block-wait on one channel. What is needed is for pvm_recv to block-wait on a number of selected channels i.e. match a prescribed set of messages. Hence a new matching function called alt_match having this functionality was written. Therefore the alt routine basically uses the pvm_recvf function to replace alt_match routine before invoking pvm_recv.

3.5 Mapping Data Types

The data model in PCSC is described using DSDs which is the data model from JSP [7]. However, the allowed basic data types in PCSC design model have been limited to a number sufficient for any design purpose. The set of data types is very similar to the one supported by occam. Proper mapping has been made from this set of basic data types onto the C-PVM basic data types. The DSD also allows abstract data types such as arrays, unions, records or the combination of them, to be described.

4. Implementation Details

The PCSC methodology is currently being implemented using a commercial CASE tool called Software through Pictures (StP) [12]. StP is a configurable CASE tool and provides sufficient leeway to test the ideas of PCSC. It already supports a number of tools (such as diagram editors) which has been customised for PCSC. All information pertaining to all aspects (such as requirements, design, etc.) of an application is stored in a central repository; this aids in enforcing the consistency requirement of PCSC.

The information captured through the design is extracted from the repository and the appropriate language code is generated and stored in files. Since each diagram editor has different properties, three separate code generators were needed. The DFD code generator generates the master process which spawns all the sub-processes. The most complex is the CSD code generator which integrates the DSD code generator. This CSD code generator translates the CSD syntax to C-PVM.

4.1 Channel Mapping

Implementation of the channel mapping can be split into two levels; the DFD level and the CSD level. In the DFD level, the following steps are implemented (refer to Figure 3.1, Figure 6 and Figure 7): to each process a unique number is assigned which is used to index the process identifier array containing the process identifiers of all the processes generated in the system; the master program will pass the indices (unique numbers) as arguments to the processes to be spawned; once all processes are spawned, the master program sends out to each process the process identifier array. Similarly, each channel is allocated a unique number which is passed as an argument to the process spawned. Note that the unique numbers for both processes and channels are generated automatically by the auto-code generator.

At the CSD level (refer to Figure 3.2 and Figure 8), each individual process will pick the appropriate argument values and assign them to the index variables (i.e. the formal channel names prefixed with ‘`p`’). These variables will be used to index into the process identifier (*task_id*) array, received from the master process, to obtain the process identifier of the relevant process in the system. Similarly, the unique channel numbers are assigned to the message tag variables (i.e. the formal channel names).

4.2 Alternation Constructs

To ease the mapping of the alternation constructs onto C-PVM, two routines were developed, viz., `alt` and `prialt`. The former is used for mapping the non-deterministic ALT construct and the latter, the deterministic PRIALT construct. The `alt` and `prialt`

needs to take on different values. This is illustrated in Figure 4.2. The skeletal CSD for process S (sender) contains a FOR construct which uses C-like syntax and semantics for the control expression [11]. The OUT construct is used to send out a tagged message sequentially to all the processes P0 to P[i] (in this example `i = 9`). Note that the channel `outch` (corresponding to `ch`) uses the index `i` from the FOR construct. Hence, this channel index needs to be a variable which can be assigned with values dynamically.

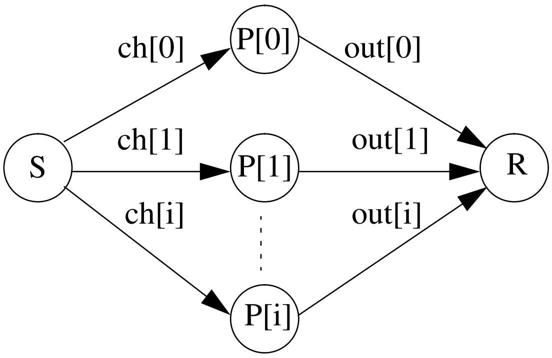

Figure 4.1: Replicated processes.

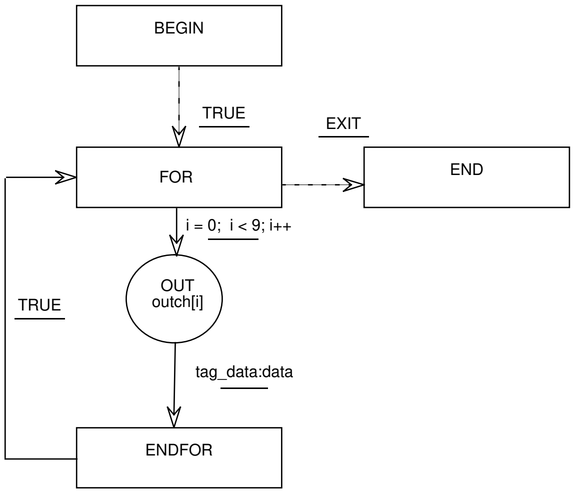

Figure 4.2: Skeletal CSD for process S in Figure 4.1

3.4 Mapping Standard I/O

Figure 5 illustrates a DFD where an input process reads values from the keyboard and passes them to a sum process. The sum process, after operating on the values, passes the result to the output process which then outputs to the screen. When the DFD (i.e., the design) is mapped onto PVM, a master process is necessary to not only spawn all the three processes but also to take care of the standard I/O i.e. keyboard and screen. It must be stressed that in PVM only the master process can perform standard I/O. Note that the dotted part of the system is created by the auto-code generator and is not part of the original logical design.

This mapping was done before the `pvm_catchout` routine was made available in the later release of PVM (i.e. version 3.3) [9]. The availability of the routine does make the task of handling standard output easier since explicit messages need not be sent. However, since standard input still has to be dealt with separately, the above I/O mapping is still necessary.

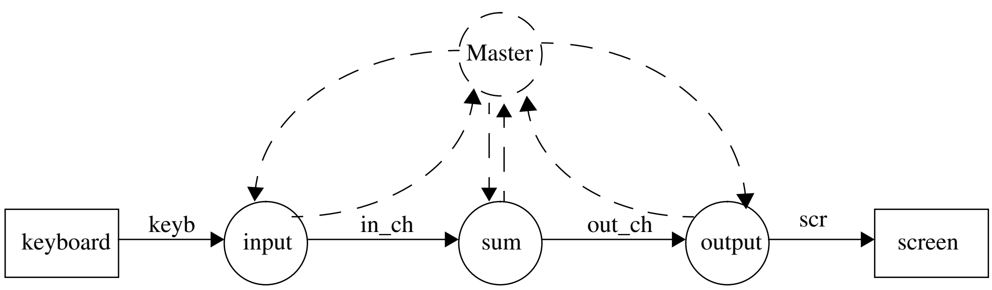

Figure 5: Standard I/O mapping for PVM.

The formal channel name in the CSD is used as the name of the message tag variable. In the above example, `outchan1` and `inch1` are the formal names for `ch1` in the CSDs of P0 and P1, respectively. In this case, both `outchan1` and `inch1` will be assigned the value of `ch1`, i.e., `1`. Again, note that the above naming convention has been adopted to be consistent with the channel names in the design although it could be made arbitrary.

Since the PVM model supports point-to-point communication (i.e. task-to-task) and guarantees message order [9], the above technique effectively maps the channel concept of the PCSC methodology onto PVM.

3.2 Mapping CSD Alternation Constructs

The PCSC model allows the selection of a channel input based on the state of the channel and any associated state variable. The CSD alternation (ALT and PRIALT) constructs are used for this purpose [10]. Consider the DFD in Figure 3.1; P0 to P5 are processes whilst `ch0` to `ch4` are channels. P5 receives messages on channels `ch0` to `ch3` and sends messages out on `ch4`. Assume that P5 wants to perform an alternation on two input channels, viz., `ch1` and `ch2`. This means that the alternation should only pick a message on `ch1` or `ch2` and leave all those messages coming in on `ch3` and `ch4` alone. The bare-bone CSD fragment for process P5 illustrating use of the ALT construct is given in Figure 3.2. A message (`msg1` or `msg2`) is picked from `inch1` (i.e. `ch1`) or `inch2` (i.e. `ch2`), operated on (`action1` or `action2`) and the `result` output to `outch4`.

The style of channel mapping discussed in the previous section allowed the development of alternation routines for use by the code generator. Note that channel uniqueness is mandatory for supporting the alternation concept.

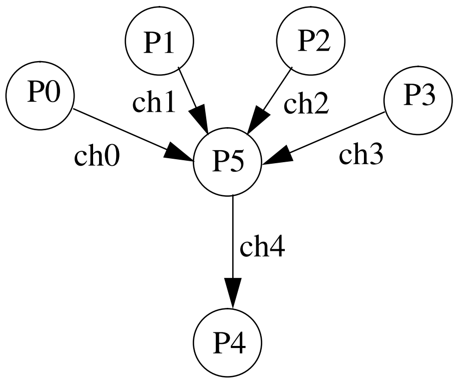

Figure 3.1: DFD to illustrate ALT construct.

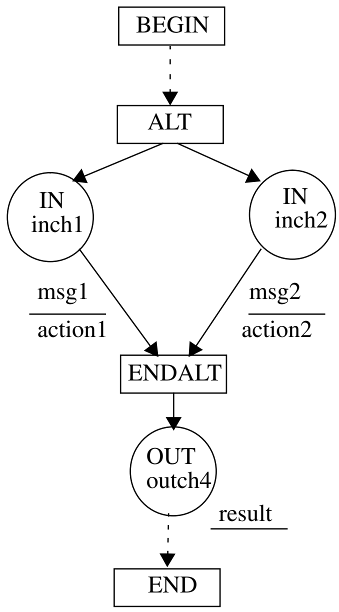

Figure 3.2: Bare-bone CSD for P5.

3.3 Mapping Process Replication

The PCSC design model supports process replication. Figure 4.1 is a DFD of a system of processes illustrating process replication. In this case the process P[i] is replicated, the code for that process is reused. Therefore at each instantiation of a P[i] process, the channel index

whilst `ch0` is a channel. P0 sends messages via `ch0` to P1; P1 receives them. In PVM, this can be written thus:

P0 sends: `pvm_send(task_id[p_outchan0], mesg_tag);`
P1 receives: `pvm_recv(task_id[p_inch0], mesg_tag);`

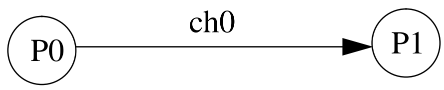

Figure 1: A simple DFD showing two processes communicating via a channel.

`task_id` is the array of all process identifiers which is received from the master process that spawns all the processes in the system. The indices (variables) into the `task_id` array are the channel names (defined within CSD) prefixed with a '`p`' to indicate *process*. This naming convention has been adopted to be consistent with the channel names in the design but it could be made arbitrary. These channel names, `outchan0` and `inch0`, are formal names (i.e. local to the CSD) as compared to `ch0` which is the actual name. The formal-actual pairing is carried out by the user during the design phase. This information is later used during automatic code generation.

Consider now the case of P0 communicating with P1 via another additional channel, `ch1`, as illustrated in Figure 2. Using the notation discussed above, this communication over `ch1` can be written thus:

P0 sends: `pvm_send(task_id[p_outchan1], mesg_tag);`
P1 receives: `pvm_recv(task_id[p_inch1], mesg_tag);`

However, since the receiving process is the same in both cases, i.e. P1, the variables `p_outchan0` and `p_outchan1` would have the same value, i.e., they would map to the same process identifier. Looking from the side of P1, the same applies to `p_inch0` and `p_inch1` since the sending process is the same in both cases, i.e. P0. Hence, this mapping alone is insufficient to preserve the channel uniqueness of the PCSC model. However, by tagging all messages on a particular channel with an unique number, the channels can be differentiated.

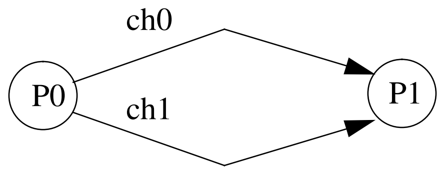

Figure 2: P0 and P1 communicating over two channels.

If the message identifier is associated with the *channel name*, all messages bearing the same identifier can be considered to be sent or received on that *logical channel*. Conversely, each channel name can be uniquely defined with an integer value, for example, `ch0 = 0`, `ch1 = 1`, and these values assigned to the messages to form the logical channels. This is illustrated below for the case of `ch1`:

P0 sends: `pvm_send(task_id[p_outchan1], outchan1);`
P1 receives: `pvm_recv(task_id[p_inch1], inch1);`

verified against the Requirements Specifications, and the Design Specifications are verified against the Analysis Specifications.

The design model is based upon the occam model of parallel computation [6]. In the occam model, an application is composed of concurrent processes that communicate between one another via channels. The designer starts off by determining the processes that make up the application and links them with appropriate unidirectional channels. This is done with Data Flow Diagrams (DFDs). Next, the channel messages are defined using Data Structure Diagrams (DSDs). The individual processes are then described with Communicating State Diagrams (CSDs). The CSD is an in-house developed technique based on the State Transition Diagram (STD) in which logic related to communication is described using CSD constructs and data structures and logic **not** related to communication are described using DSDs and Jackson Structured Diagrams (JSDs) [7]. Information that cannot be represented graphically is captured using annotations.

A design can include reusable components [8] as well as reverse engineered components. The deliverable of the design phase is the design specifications. These specifications together with those from the configuration phase are then used to generate the code automatically for the system. Once code is generated, it is executed on the target platform. If there are errors, the design is debugged and the code regenerated and executed. The methodology does not encourage the manipulation of the code. On successful execution, the system can be analysed for performance improvement. This can be achieved by optimising the design and/or the configuration.

3. Mapping of PCSC Design Model onto PVM

As mentioned earlier, the PCSC methodology has its roots in the occam model of computation. In the occam model, the only means of communication between processes is via *channels*. A channel connects only two processes. If one end of a channel is being written to, the other end must be read from, i.e. a channel is *unidirectional*. Channel communication is *synchronised*; a message is only sent when both the sending and receiving processes are ready. If one process becomes ready first, it will wait for the other to become ready. Hence, messages are not *buffer*ed. A channel is, therefore, an unbuffered, one-way, point-to-point link between two processes.

In PVM, however, the concept of channel is absent. Processes communicate by sending and receiving messages. Messages can be sent to a single task (*unicast*) or to a group of tasks (*multicast*). Message sending is *asynchronous*; the send primitives return as soon as the message is safely on its way. Messages are received by a source or label identification either in *blocking* or *non-blocking* mode. Messages are *buffer*ed. Buffers are allocated dynamically. Hence, the maximum size of a message sent or received depends on the memory available in a given host.

Comparing the programming paradigms of occam and PVM models, it can be seen that although both are message-based, it is the concept of channel (and its ramifications) that sets the occam model apart from PVM. In the occam model, messages can only be read from or written to *named* channels. In contrast, in PVM, messages are tagged uniquely and the destination (or source) process is specified when the messages are sent (or received).

3.1 Channel Mapping

Since PVM does not support the concept of channels, some method must be devised to model the channel concept for PVM. Consider the DFD in Figure 1; P0 and P1 are processes

Mapping the PCSC Design Model to PVM

G.A. Manson, C. Elamvazuthi, S. Sahib

Parallel Processing Research Group, Department of Computer Science

University of Sheffield, Regent Court, Portobello Street, Sheffield, S1 4DP, U.K.

Email: G.Manson@dcs.shef.ac.uk

Abstract: A methodology for parallel system design called Parallel Communicating Sequential Code (PCSC) is currently being developed. A CASE tool based on this methodology is also being built which automatically generates C code for the PVM environment. This paper highlights the mapping of PCSC model onto PVM to make the automatic code generation possible.

1. Introduction

An integrated CASE tool incorporating a design methodology for parallel program design is being developed at the department of Computer Science, University of Sheffield [1].

The methodology known as Parallel Communicating Sequential Code (PCSC) [2] has two basic aims:

• to produce designs that will be independent of language and platform (system software and machine architecture). The designs are to be portable to a range of parallel machine architectures as well as to heterogeneous multicomputer environments.

• to ensure consistency between design, configuration (mapping of application to processors), code and documentation of an application. This aspect is very important for the maintenance of the application.

The design information is captured in a visual-cum-textual abstract language consisting of (modified) Data Flow Diagrams (DFDs) [3], Data Structure Diagrams (DSDs) [3], Communicating State Diagrams (CSDs) [4] and annotation (text). From this abstract design language, code in various suitable programming languages is to be produced automatically.

One target environment is C-PVM [5]. The main reason for choosing PVM is its availability on a wide range of computers. Thus, a parallel design that has been translated into C-PVM should be able to run on different machine architectures without any change in the original design; hence, meeting the portability criterion. Our familiarity with C and UNIX was also a factor in the choice of PVM: the availability of source code for PVM is a plus point for it promotes a better understanding of the system and is quite useful for debugging purposes. It must also be mentioned that the free availability of PVM was also a factor in its choice.

2. PCSC Methodology

The PCSC methodology has a number of phases. Requirements Capture, Analysis and Design are tied together by the process of verification. The Analysis Specifications are

They take in one argument: an integer array, `ArrOfChan`, containing the values of the 'logical channels' (i.e. message identifiers) to be checked for messages. They return the buffer location, `BufID`, and the channel identity (i.e. message identifier), `ChanID`, of the message picked. Note that these routines are planted automatically by the code generator corresponding to the design.

5. Conclusion

This paper has introduced the basic concepts behind the PCSC methodology for parallel system design and briefly outlined the reasons for choosing PVM as a target implementation environment. The PCSC and PVM models were discussed and their differences compared. It was shown that the fundamental difference between the two models is the absence of the channel concept in PVM. The techniques adopted to conform PVM to PCSC model of computation were explained and illustrated. How C code for PVM is generated automatically from the PCSC design was also briefly indicated.

Currently, a working prototype of the C-PVM code translator has been implemented and is being tested. Work is in progress to implement a meta-code generator which would take code templates and automatically generate from them the appropriate code generator for the specified parallel programming language.

References

[1] Manson, G.A., and Boyle, A., A CASE Tool for Designing Parallel Systems: The Architectural Model, Proceedings of the PACTA '92, Barcelona, Spain, September 1992.

[2] Manson, G.A., and Sahib, S., An Illustration of the Parallel Communicating Sequential Methodology, Workshop on Software Engineering for Parallel Systems, World Transputer Conference '93 (WTC '93), Aachen, Germany, 20 - 22 September 93.

[3] Fisher, A.S., CASE Using Software Development Tools, John Wiley and Sons, Inc., 1988.

[4] Cachia, E.A., and Manson, G.A., CCDM - A Design Methodology for Modelling Communicating Code in Parallel Systems, Proceedings of the 16th WoTUG Technical Meeting, Sheffield, UK, March 28-31, 1993, pp. 100 - 114.

[5] Sunderam, V.S., PVM: A framework for parallel distributed computing, Concurrency: Practice and Experience, 2(4):315-339, December 1990.

[6] Galletly, J., Occam 2, Pitman Publishing, 1990.

[7] Ingevaldsson, L., JSP-A Practical Method of Program Design, Chartwel-Brat Ltd., 1986.

[8] Loh B.C. and Manson G.A., Incorporating Software Reuse into the PCSC Methodology, Poster Presentation, World Transputer Conference '94 (WTC '94), Como, Italy, Sept. 1994.

[9] Geist, A., et. al., PVM 3 User's Guide and Reference Manual, ORNL/TM-12187, May 1994.

[10] Elamvazuthi, C. and Manson, G.A., Occam, PVM and the Alternation Construct. Proceedings of the 17th WoTUG Technical Meeting, Bristol, England, 11-13 April, 1994, pp. 56-58.

[11] Elamvazuthi, C., Hussain, A.M.A., Manson, G.A. and Sahib, S., Behavioural Modelling of a Communicating Process, Proceedings of the 1994 World Transputer Conference (WTC '94), Como, Italy, Sept. 1994, pp. 568-583.

[12] Interactive Development Environments (IDE), Software through Pictures, Fundamentals of StP Release 1, February 1994.

```
#include "comm.h"

int p_inch0, p_inch1, p_inch2, p_inch3, p_outch4;
int inch0, inch1, inch2, inch3, outch4;
main(argc,argv)
int argc;
char *argv[];
{
  if ((mytid = pvm_mytid()) < 0)  exit(1);
  pvm_recv(pvm_parent(), 0);
  pvm_upkint(&num_proc, 1, 1);
  pvm_upkint(task_id, num_proc, 1);
  p_inch0  = atoi(argv[1]);/* Assignments for process */
  p_inch1  = atoi(argv[2]);/* indices */
  p_inch2  = atoi(argv[3]);
  p_inch3  = atoi(argv[4]);
  p_outch4 = atoi(argv[5]);

  inch0    = atoi(argv[6]);/* Assignments for message */
  inch1    = atoi(argv[7]);/*tags for channel uniqueness*/
  inch2    = atoi(argv[8]);
  inch3    = atoi(argv[9]);
  outch4   = atoi(argv[10]);

  ...
  ArrOfChan[0] = inch1;
  ArrOfChan[1] = inch2;
  ArrOfChan[2] = EOA;
  ChanID = alt(ArrOfChan, &BufID);
    switch(ChanID) {
      case inch1:
  ... /* receive msg1 */
  ... /* action1 */
        break;
      case inch2:
  ... /* receive msg2 */
  ... /* action2 */
        break;
      default: break;
    }
  ...
  pvm_send(task_id[p_outch4],outch4);
  ...
  pvm_exit();
}
```

Figure 8: Code fragment for P3.

The implementation of `prialt` has an additional feature. The global receive queue is scanned for the highest priority message (or channel) first. If it is not available, then the queue is scanned again for the next highest priority message until a message is picked. If there are no messages of the required types, the routine block-waits for any one message from the required set. The scan-and-pick procedure and block-waiting are handled by `pvm_nrecv` and `pvm_recv`, respectively. Here again, before invoking `pvm_recv`, the default match routine is replaced with `alt_match`.

Both alt and prialt have the following C call syntax:

```
int ChanID = alt(int ArrOfChan[], int *BufID) ;
int ChanID = prialt(int ArrOfChan[], int *BufID) ;
```

```
#include <stdio.h>  /* relevant contents of comm.h */
#include <string.h>
#include "pvm3.h"
...
#define P0  "0"   /* Unique numbers assigned to    */
#define P1  "1"   /* processes and channels during */
#define P2  "2"   /* auto-code generation.         */
#define P3  "3"
#define P4  "4"
#define P5  "5"
#define ch0 "0"
#define ch1 "1"
#define ch2 "2"
#define ch3 "3"
#define ch4 "4"
...
```

Figure 6: Relevant contents of comm.h included by all processes in the system.

```
#include "comm.h"
main()
{
 ...
 char *arg[11];

 mytid = pvm_mytid();
 ...
 arg[0] = strdup(P0);/*Runtime assignment(DFD level)*/
 arg[1] = strdup(P1);
 arg[2] = strdup(P2);
 arg[3] = strdup(P3);
 arg[4] = strdup(P4);
 arg[5] = strdup(ch0);
 arg[6] = strdup(ch1);
 arg[7] = strdup(ch2);
 arg[8] = strdup(ch3);
 arg[9] = strdup(ch4);
 arg[10] = NULL;
 pvm_spawn("P5",arg,1,"",1,&task_id[atoi(P5)]);
 ...
pvm_initsend(PvmDataDefault);
 pvm_pkint(&num_proc, 1, 1);
 pvm_pkint(task_id, num_proc, 1);
 pvm_mcast(task_id, num_proc, 0);
 ...
}
```

Figure 7: Code fragment for the master process for the DFD in Figure 3.1.

routines were written using the PVM primitives, `pvm_recv`, `pvm_nrecv` and `pvm_recvf`.

Since a channel is defined by giving the message tag a unique value, the `pvm_recv` routine can only block-wait on one channel. What is needed is for `pvm_recv` to block-wait on a number of selected channels i.e. match a prescribed set of messages. Hence a new matching function called `alt_match` having this functionality was written. Therefore the `alt` routine basically uses the `pvm_recvf` function to replace `alt_match` routine before invoking `pvm_recv`.

3.5 Mapping Data Types

The data model in PCSC is described using DSDs which is the data model from JSP [7]. However, the allowed basic data types in PCSC design model have been limited to a number sufficient for any design purpose. The set of data types is very similar to the one supported by occam. Proper mapping has been made from this set of basic data types onto the C-PVM basic data types. The DSD also allows abstract data types such as arrays, unions, records or the combination of them, to be described.

4. Implementation Details

The PCSC methodology is currently being implemented using a commercial CASE tool called Software through Pictures (StP) [12]. StP is a configurable CASE tool and provides sufficient leeway to test the ideas of PCSC. It already supports a number of tools (such as diagram editors) which has been customised for PCSC. All information pertaining to all aspects (such as requirements, design, etc.) of an application is stored in a central repository; this aids in enforcing the consistency requirement of PCSC.

The information captured through the design is extracted from the repository and the appropriate language code is generated and stored in files. Since each diagram editor has different properties, three separate code generators were needed. The DFD code generator generates the master process which spawns all the sub-processes. The most complex is the CSD code generator which integrates the DSD code generator. This CSD code generator translates the CSD syntax to C-PVM.

4.1 Channel Mapping

Implementation of the channel mapping can be split into two levels; the DFD level and the CSD level. In the DFD level, the following steps are implemented (refer to Figure 3.1, Figure 6 and Figure 7): to each process a unique number is assigned which is used to index the process identifier array containing the process identifiers of all the processes generated in the system; the master program will pass the indices (unique numbers) as arguments to the processes to be spawned; once all processes are spawned, the master program sends out to each process the process identifier array. Similarly, each channel is allocated a unique number which is passed as an argument to the process spawned. Note that the unique numbers for both processes and channels are generated automatically by the auto-code generator.

At the CSD level (refer to Figure 3.2 and Figure 8), each individual process will pick the appropriate argument values and assign them to the index variables (i.e. the formal channel names prefixed with ‘`p`’). These variables will be used to index into the process identifier (*task_id*) array, received from the master process, to obtain the process identifier of the relevant process in the system. Similarly, the unique channel numbers are assigned to the message tag variables (i.e. the formal channel names).

4.2 Alternation Constructs

To ease the mapping of the alternation constructs onto C-PVM, two routines were developed, viz., `alt` and `prialt`. The former is used for mapping the non-deterministic ALT construct and the latter, the deterministic PRIALT construct. The `alt` and `prialt`

needs to take on different values. This is illustrated in Figure 4.2. The skeletal CSD for process S (sender) contains a FOR construct which uses C-like syntax and semantics for the control expression [11]. The OUT construct is used to send out a tagged message sequentially to all the processes P0 to P[i] (in this example `i = 9`). Note that the channel `outch` (corresponding to `ch`) uses the index `i` from the FOR construct. Hence, this channel index needs to be a variable which can be assigned with values dynamically.

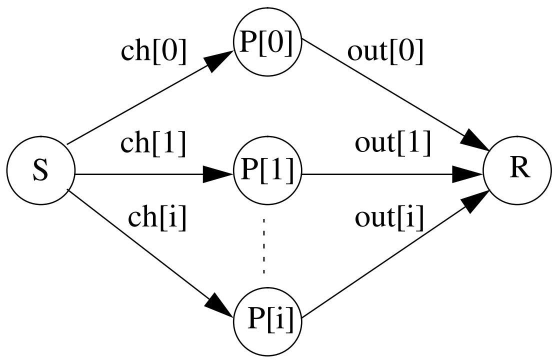

Figure 4.1: Replicated processes.

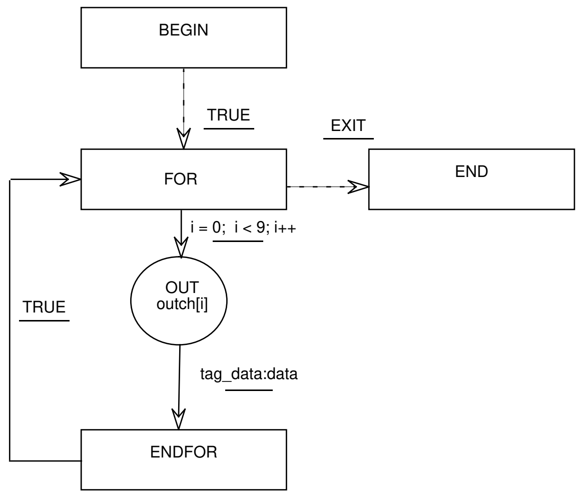

Figure 4.2: Skeletal CSD for process S in Figure 4.

3.4 Mapping Standard I/O

Figure 5 illustrates a DFD where an input process reads values from the keyboard and passes them to a sum process. The sum process, after operating on the values, passes the result to the output process which then outputs to the screen. When the DFD (i.e., the design) is mapped onto PVM, a master process is necessary to not only spawn all the three processes but also to take care of the standard I/O i.e. keyboard and screen. It must be stressed that in PVM only the master process can perform standard I/O. Note that the dotted part of the system is created by the auto-code generator and is not part of the original logical design.

This mapping was done before the `pvm_catchout` routine was made available in the later release of PVM (i.e. version 3.3) [9]. The availability of the routine does make the task of handling standard output easier since explicit messages need not be sent. However, since standard input still has to be dealt with separately, the above I/O mapping is still necessary.

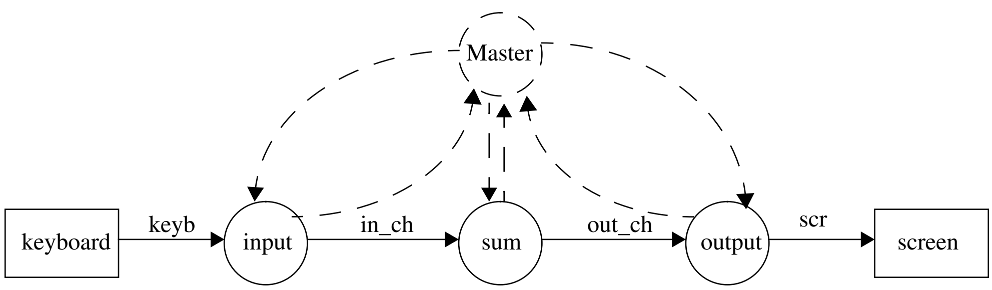

Figure 5: Standard I/O mapping for PVM.

The formal channel name in the CSD is used as the name of the message tag variable. In the above example, `outchan1` and `inch1` are the formal names for `ch1` in the CSDs of P0 and P1, respectively. In this case, both `outchan1` and `inch1` will be assigned the value of `ch1`, i.e., `1`. Again, note that the above naming convention has been adopted to be consistent with the channel names in the design although it could be made arbitrary.

Since the PVM model supports point-to-point communication (i.e. task-to-task) and guarantees message order [9], the above technique effectively maps the channel concept of the PCSC methodology onto PVM.

3.2 Mapping CSD Alternation Constructs

The PCSC model allows the selection of a channel input based on the state of the channel and any associated state variable. The CSD alternation (ALT and PRIALT) constructs are used for this purpose [10]. Consider the DFD in Figure 3.1; P0 to P5 are processes whilst `ch0` to `ch4` are channels. P5 receives messages on channels `ch0` to `ch3` and sends messages out on `ch4`. Assume that P5 wants to perform an alternation on two input channels, viz., `ch1` and `ch2`. This means that the alternation should only pick a message on `ch1` or `ch2` and leave all those messages coming in on `ch3` and `ch4` alone. The bare-bone CSD fragment for process P5 illustrating use of the ALT construct is given in Figure 3.2. A message (`msg1` or `msg2`) is picked from `inch1` (i.e. `ch1`) or `inch2` (i.e. `ch2`), operated on (`action1` or `action2`) and the `result` output to `outch4`.

The style of channel mapping discussed in the previous section allowed the development of alternation routines for use by the code generator. Note that channel uniqueness is mandatory for supporting the alternation concept.

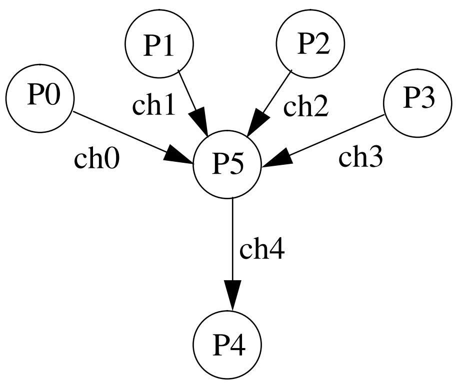

Figure 3.1: DFD to illustrate ALT construct.

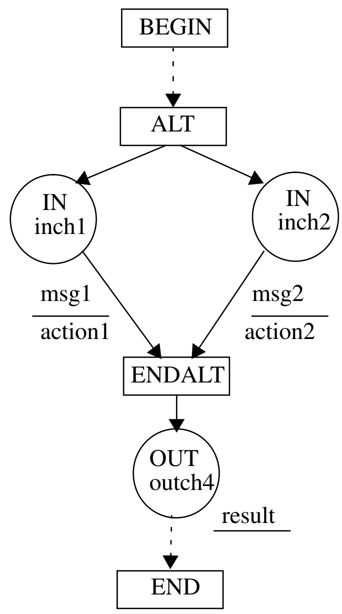

Figure 3.2: Bare-bone CSD for P5.

3.3 Mapping Process Replication

The PCSC design model supports process replication. Figure 4.1 is a DFD of a system of processes illustrating process replication. In this case the process P[i] is replicated, the code for that process is reused. Therefore at each instantiation of a P[i] process, the channel index

whilst ch0 is a channel. P0 sends messages via ch0 to P1; P1 receives them. In PVM, this can be written thus:

P0 sends: `pvm_send(task_id[p_outchan0], mesg_tag);`
P1 receives: `pvm_recv(task_id[p_inch0], mesg_tag);`

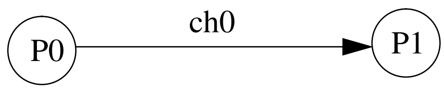

Figure 1: A simple DFD showing two processes communicating via a channel.

task_id is the array of all process identifiers which is received from the master process that spawns all the processes in the system. The indices (variables) into the task_id array are the channel names (defined within CSD) prefixed with a 'p' to indicate *process*. This naming convention has been adopted to be consistent with the channel names in the design but it could be made arbitrary. These channel names, outchan0 and inch0, are formal names (i.e. local to the CSD) as compared to ch0 which is the actual name. The formal-actual pairing is carried out by the user during the design phase. This information is later used during automatic code generation.

Consider now the case of P0 communicating with P1 via another additional channel, ch1, as illustrated in Figure 2. Using the notation discussed above, this communication over ch1 can be written thus:

P0 sends: `pvm_send(task_id[p_outchan1], mesg_tag);`
P1 receives: `pvm_recv(task_id[p_inch1], mesg_tag);`

However, since the receiving process is the same in both cases, i.e. P1, the variables p_outchan0 and p_outchan1 would have the same value, i.e., they would map to the same process identifier. Looking from the side of P1, the same applies to p_inch0 and p_inch1 since the sending process is the same in both cases, i.e. P0. Hence, this mapping alone is insufficient to preserve the channel uniqueness of the PCSC model. However, by tagging all messages on a particular channel with an unique number, the channels can be differentiated.

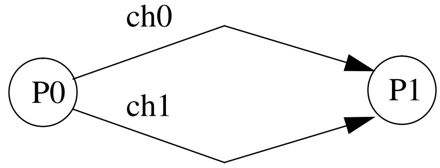

Figure 2: P0 and P1 communicating over two channels.

If the message identifier is associated with the *channel name*, all messages bearing the same identifier can be considered to be sent or received on that *logical channel*. Conversely, each channel name can be uniquely defined with an integer value, for example, ch0 = 0, ch1 = 1, and these values assigned to the messages to form the logical channels. This is illustrated below for the case of ch1:

P0 sends: `pvm_send(task_id[p_outchan1], outchan1);`
P1 receives: `pvm_recv(task_id[p_inch1], inch1);`

verified against the Requirements Specifications, and the Design Specifications are verified against the Analysis Specifications.

The design model is based upon the occam model of parallel computation [6]. In the occam model, an application is composed of concurrent processes that communicate between one another via channels. The designer starts off by determining the processes that make up the application and links them with appropriate unidirectional channels. This is done with Data Flow Diagrams (DFDs). Next, the channel messages are defined using Data Structure Diagrams (DSDs). The individual processes are then described with Communicating State Diagrams (CSDs). The CSD is an in-house developed technique based on the State Transition Diagram (STD) in which logic related to communication is described using CSD constructs and data structures and logic **not** related to communication are described using DSDs and Jackson Structured Diagrams (JSDs) [7]. Information that cannot be represented graphically is captured using annotations.

A design can include reusable components [8] as well as reverse engineered components. The deliverable of the design phase is the design specifications. These specifications together with those from the configuration phase are then used to generate the code automatically for the system. Once code is generated, it is executed on the target platform. If there are errors, the design is debugged and the code regenerated and executed. The methodology does not encourage the manipulation of the code. On successful execution, the system can be analysed for performance improvement. This can be achieved by optimising the design and/or the configuration.

3. Mapping of PCSC Design Model onto PVM

As mentioned earlier, the PCSC methodology has its roots in the occam model of computation. In the occam model, the only means of communication between processes is via *channels*. A channel connects only two processes. If one end of a channel is being written to, the other end must be read from, i.e. a channel is *unidirectional*. Channel communication is *synchronised*; a message is only sent when both the sending and receiving processes are ready. If one process becomes ready first, it will wait for the other to become ready. Hence, messages are not *buffer*ed. A channel is, therefore, an unbuffered, one-way, point-to-point link between two processes.

In PVM, however, the concept of channel is absent. Processes communicate by sending and receiving messages. Messages can be sent to a single task (*unicast*) or to a group of tasks (*multicast*). Message sending is *asynchronous*; the send primitives return as soon as the message is safely on its way. Messages are received by a source or label identification either in *blocking* or *non-blocking* mode. Messages are *buffer*ed. Buffers are allocated dynamically. Hence, the maximum size of a message sent or received depends on the memory available in a given host.

Comparing the programming paradigms of occam and PVM models, it can be seen that although both are message-based, it is the concept of channel (and its ramifications) that sets the occam model apart from PVM. In the occam model, messages can only be read from or written to *named* channels. In contrast, in PVM, messages are tagged uniquely and the destination (or source) process is specified when the messages are sent (or received).

3.1 Channel Mapping

Since PVM does not support the concept of channels, some method must be devised to model the channel concept for PVM. Consider the DFD in Figure 1; P0 and P1 are processes

Mapping the PCSC Design Model to PVM

G.A. Manson, C. Elamvazuthi, S. Sahib
Parallel Processing Research Group, Department of Computer Science
University of Sheffield, Regent Court, Portobello Street, Sheffield, S1 4DP, U.K.
Email: G.Manson@dcs.shef.ac.uk

Abstract: A methodology for parallel system design called Parallel Communicating Sequential Code (PCSC) is currently being developed. A CASE tool based on this methodology is also being built which automatically generates C code for the PVM environment. This paper highlights the mapping of PCSC model onto PVM to make the automatic code generation possible.

1. Introduction

An integrated CASE tool incorporating a design methodology for parallel program design is being developed at the department of Computer Science, University of Sheffield [1].

The methodology known as Parallel Communicating Sequential Code (PCSC) [2] has two basic aims:

• to produce designs that will be independent of language and platform (system software and machine architecture). The designs are to be portable to a range of parallel machine architectures as well as to heterogeneous multicomputer environments.

• to ensure consistency between design, configuration (mapping of application to processors), code and documentation of an application. This aspect is very important for the maintenance of the application.

The design information is captured in a visual-cum-textual abstract language consisting of (modified) Data Flow Diagrams (DFDs) [3], Data Structure Diagrams (DSDs) [3], Communicating State Diagrams (CSDs) [4] and annotation (text). From this abstract design language, code in various suitable programming languages is to be produced automatically.

One target environment is C-PVM [5]. The main reason for choosing PVM is its availability on a wide range of computers. Thus, a parallel design that has been translated into C-PVM should be able to run on different machine architectures without any change in the original design; hence, meeting the portability criterion. Our familiarity with C and UNIX was also a factor in the choice of PVM: the availability of source code for PVM is a plus point for it promotes a better understanding of the system and is quite useful for debugging purposes. It must also be mentioned that the free availability of PVM was also a factor in its choice.

2. PCSC Methodology

The PCSC methodology has a number of phases. Requirements Capture, Analysis and Design are tied together by the process of verification. The Analysis Specifications are