The Development of a Language for Specifying Structure of a Distributed and Parallel Application

Robert Dew, Peter Horan, and Andrzej Goscinski

School of Information Technology
Deakin University, Australia
{rad, peter, ang}@deakin.edu.au

Abstract. A common characteristic of distributed and parallel programming languages is that the one language is used to specify both the organisation of the application, and its functionality. Large distributed and parallel applications will benefit if connectivity and functionality of processes are specified separately. Research is being carried out to develop two new programming languages for the specification and development of distributed and parallel applications. Here, we present a language for specifying process connectivity.

1 Introduction

Existing languages and tools for developing distributed and parallel applications specify connectivity with functionality explicitly or implicitly or provide no support [8]. Furthermore, connectivity is specified in different ways.

Tightly Integrated Connectivity. A parallel Orca application starts with one process and forks to create additional processes [3]. Connectivity of an Orca application is implicit in the functionality of the application, and a specification is not captured formally during design. PVM, a library of low level tools, has a similar characteristic [7,10]. Connectivity relies on send/receive calls and programmer supplied identifiers.

Implied Connectivity. Linda [1] and SR [2] use shared memory and permit virtual memory to be in the scope of several distributed processes. A single chunk of memory is shared by all Linda processes and there is no explicit connectivity. SR uses shared memory but processes share several message queues. Connectivity in OO systems depends on objects accessing data or function members of other objects [9]. CORBA [11] delivers messages from clients to objects and returns results. Clients provide the ORB with object references, only available during run-time by object creation or as the result of a request to a directory service. Thus, connectivity is implied.

No Support for Specifying Connectivity. MPI is a popular message passing library, but not a system for specifying complete distributed applications [6]. This is because MPI lacks both process management and tools for application development.

Explicit Connectivity. HeNCE and CODE are environments specifying parallel applications as graphs which contain nodes to represent procedures and arcs, data and control flow. CODE is similar to HeNCE [4], but arcs represent data flow only.

However, large distributed and parallel applications will benefit if connectivity and functionality of processes are specified separately.

The aim of our project is to develop a method and tools for building distributed and parallel applications consisting of concurrently running processes distributed on a network, using asynchronous message passing. Messages are typed arrays of data sent by one process and automatically placed into a buffer in the receiving process [5].

The tasks to be solved to achieve this aim are: specify application connectivity and functionality independently; specify connectivity in a modular fashion to permit re-use of specifications; build applications by combining functional and connectivity specifications, and then compiling to produce object modules; distribute these modules on a cluster; and manage running distributed applications.

2 A New Approach to Programming Distributed Applications

A complete system (to show this work) is distributed over a network. Applications are managed by a run-time system implemented as a single process. Machines are loosely coupled, supported by a file system. Fig. 1 shows two applications of several processes connected by channels, and the run-time management process that builds, compiles, instantiates and manages concurrently executing applications. Connectivity is an essential characteristic of distributed applications and is independent of the internal workings of processes. So, the design of an application can be partitioned into designs of connectivity and functionality.

Fig. 1. Two applications and the run-time management process distributed on four machines

2.1 Two Languages for the Specification of Structure and Functionality

The specification of distributed applications has independent parts: structure and functionality, which can be specified by two languages: DAL (Distributed Application Language) and DAPL (DAPL is the subject of another paper), respectively. The programmer uses statements of the structure specification language to specify structure, namely, where processes execute on a network and how they are connected by communication channels. Also, this language supports modularity by allowing existing specifications to be re-used. The structure specification language does not specify functionality; it provides hooks to associate modules of functionality using a second language. These hooks allow the creation of applications that have the same structure but similar or different functionality.

In addition to these two languages, a system managing many such distributed and parallel applications at runtime is required. We use a runtime system which allows one to build, compile, instantiate and manage executing applications on a network.

2.2 Designing the DAL Language

The DAL language was designed to specify the structure of a distributed and parallel application, in particular, the connectivity of several processes and where they may execute. The DAL grammar is defined using BNF [REF5]. DAL supports:

- naming the application,
- listing machine names on which the application executes,
- naming processes and listing the order in which they commence execution,
- listing machine names on which a particular process may execute,
- declaring buffers and communication channels for asynchronous message passing,
- declaring the application's structure by associating one end of a communication channel with one process and the other with a receiving buffer of another process,
- re-using prewritten DAL specifications as components of this specification.

2.3 Implementing and Testing the DAL Language

The high level statements written in the DAL language and programs written in DAPL are translated to C code which is then used by a C compiler to generate object code. These generated object modules are used by the run-time management system to instantiate the applications processes on a network.

The DAL parser was tested to ensure that specifications written in DAL conform not only to the DAL grammar but also to the DAL language. Tests were performed by applying the DAL parser to known valid and invalid DAL specifications and comparing the parser's output with expected output. The DAL translator was tested to ensure that its generated C code is the correct C code. Tests were performed by comparing actual C code generated to expected C code that was manually developed.

3 Example

The following DAL specification specifies a simple application called `example` with processes restricted to machines `host_1`, `host_2` and `host_3`. The creation clause dictates that `proc_1` is created last.

```

application example
machines host_1, host_2, host_3
creation proc_1
processes
    process proc_1                                end
    machines host_1, host_2                        process proc_2
    buffers int buf_1                             machines host_3, host_2
    outputs int a:buf_2:proc_2                    buffers int buf_2

```

```

outputs int b:buf_1:proc_1          end
end

```

The first process clause locates process `proc_1` on either `host_1` or `host_2`, but not `host_3`. It has one integer buffer, `buf_1`, and one integer output, `a`. Its output clause specifies connection to buffer `buf_2` in `proc_2`.

The second process clause locates `proc_2`, which is very similar to `proc_1`, on either `host_3` or `host_2`. It has an integer buffer, `buf_2` and an integer output, `b`. This process is connected to `proc_1` via buffer `buf_1` of `proc_1`. This DAL specification implies that, given the proper functionality of both processes, either process can send data to the other process. The data here is simply arrays of integers.

4 Conclusion

A new programming language called DAL has been developed as part of this research. It is used to specify connectivity of distributed and parallel applications with respect to application processes and communication channels, independently of the application's functionality. The DAL language also locates processes on a set of machines, the order in which these processes begin execution, and the types of communications channels and buffers. Existing DAL specifications can be re-used when developing a new DAL specification to form a hierarchical DAL specification, enabling modular development of distributed applications.

References

1. Ahuja, S., Carriero, N. and Gelernter, D.: Linda and Friends. IEEE Computer, Vol. 19, No. 8 (1986) 26-34
2. Andrews, G. R. and Olsson, R. A.: The SR Programming Language: Concurrency in Practice. Benjamin-Cummings (1993)
3. Bal, H.: Programming Distributed Systems. Prentice-Hall (1990)
4. Browne, J. C., Dongarra, J., Hyder, S. I., Moore, K. and Newton, P.: Experiences with CODE and HeNCE in Visual Programming for Parallel Computing. IEEE Parallel and Distributed Technology, Vol. 3, No. 1 (1994) 75-83
5. Dew, R. A.: The Development of DAL and DAPL Languages for Building Distributed Applications. Deakin University (2002)
6. Dongarra, J., Otto, S. W., Suir, M. and Walker, D.: An Introduction to the MPI Standard. CS-95-274 (1995a)
7. Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R. and Sunderam, V.: PVM: Parallel Virtual Machine - A Users' Guide and Tutorial for Networked Parallel Computing. The MIT Press (1994)
8. Goscinski, A.: Finding, Expressing and Managing Parallelism in Programs Executed on Clusters of Workstations, Special Issue on Network-based Parallel and Distributed Computing, Computer Communications, Vol. 22, No. 11 (1999)
9. Horan, P.: Eiffel Assertions and the External Structure of Classes and Objects. Journal of Object Technology, Vol. 1, No. 4 (2002) 105-118
http://www.jot.fm/issues/issue_2002_09/article1
10. Sunderam, V. S.: PVM: A Framework for Parallel Distributed Computing, Concurrency: Practice and Experience, Vol. 2, No. 4 (1990) 315-339
11. Vinoski, S.: Distributed Object Computing with CORBA. C++ Report Magazine (1993)