Towards a Common Development Framework for Distributed Applications

Fethi A. Rabhi

School of Information Systems, University of New South Wales,
Sydney 2052, Australia.
Email: f.rabhi@unsw.edu.au

The last decade has witnessed a convergence in several concerns related to the development of distributed applications ranging from concurrency issues to architectural platforms. However, these concerns are often addressed within the specificities of a particular application area or a specific requirement such as performance. This paper is a contribution towards defining a common development framework specifically for distributed applications. It defines the salient characteristics of such applications and discusses emerging concepts and techniques from a number of disciplines namely concurrency theory, real-time systems, distributed systems and parallel processing. It concludes on their possible role in an integrated development framework that would be suitable for distributed applications.
keywords: Parallel processing, distributed processing, software engineering, design methods

1. INTRODUCTION

In recent years, there has been a considerable interest in distributed applications due to the availability of low cost hardware and large computer networks. By distributed application, we mean "a system of several independent software components, cooperating in a common purpose, or to achieve a common goal" (Burns and Wellings [1997]). This represents a significant proportion of computer software, including the following classes (identified by Bal, Steiner, and Tanenbaum [1989]):

—parallel and high performance applications (e.g. parallel CFD simulations)

—fault-tolerant applications and real-time systems (e.g. safety critical process control)

—applications using functional specialisation (e.g. distributed information systems)

—inherently distributed applications (e.g. World Wide Web applications)

Each of these classes has historically emerged from a distinct computing discipline such as operating systems, networks, high performance computing and real-time systems. Experiences, techniques and tools for software development are usually adapted to the particular requirements of the relevant discipline. However, there is much to learn from adapting concepts from one discipline to another since there are many common problems such as specifying the interaction between concurrent activities or mapping a process graph on a given architectural platform. Moreover, there are many applications which cannot be contained within a single discipline. For example, *metacomputations* [Smarr and Catlett 1992] are applications intended for either parallel architectures and distributed systems. Another example is in

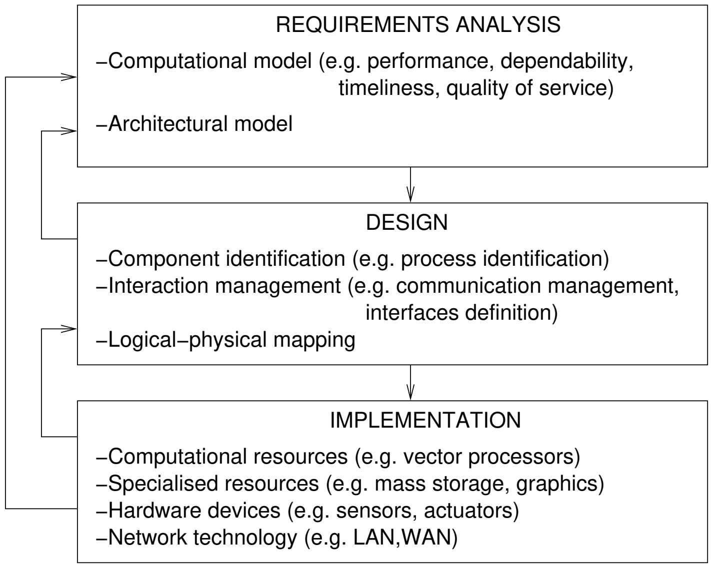

Fig. 1. Additional needs and constraints within a basic development cycle

distributed multimedia applications where real-time constraints often have to be dealt with in a distributed processing context.

Therefore, there is a need for a rigorous common development framework for distributed applications especially large and complex ones. To study this claim, the paper is divided into three parts:

—the first part (Section 2) examines the additional requirements and constraints imposed by distributed systems in relation to the sequential software development cycle

—the second part focuses on selected research areas and how they address to some extent the needs of distributed applications; these areas are concurrency theory (Section 3), real-time systems (Section 4), distributed systems (Section 5) and parallel processing (Section 6).

—the last part (Section 7) considers opportunities for generalisations and cross-discipline fertilisations that will be beneficial in the long term for the establishment of a rigorous development framework suitable for distributed applications.

2. DISTRIBUTED APPLICATION DEVELOPMENT CYCLE

For the sake of simplicity, we are limiting ourselves to the well-known three phases of requirement analysis, design and implementation in a waterfall model although issues such as maintenance and testing are still very important in this context. Figure 1 illustrates the additional needs and constraints which should be taken into account at different stages of this basic cycle when considering distributed applications. The rest of this section discusses these characteristics in more detail.

2.1 Requirements analysis

Requirement analysis is the first phase in the development process in which the requirements for the system are established and specified in detail for further development. The amount of information specified should be minimal yet complete. A common requirement for any application is its *functionality* i.e. the functions it is supposed to achieve. Distributed applications mostly differ in the *non-functional* requirements, some of which relate to the *dynamic behaviour* of the system in terms of its concurrent entities and the interaction between these entities. For these applications, the most important non-functional requirements can be grouped in the following categories:

—*performance* : this is required for parallel and high performance applications where maximum speedup and efficiency must be attained according to a specific architectural model.

—*dependability* (or robustness): includes availability, reliability, safety and security. These requirements are essential for fault-tolerant applications and those using functional specialisation.

—*timeliness*: the system must satisfy the established temporal constraints. This is an essential feature of real-time systems.

—*quality of service*: needed for applications using functional specialisation, particularly distributed multimedia. These requirements relate to the quality requirements on the collective behaviour of one or more processes. They are expressed in terms of both timeliness constraints and guarantees on measures of communication rate and latency, probabilities of communication disruptions etc. [Coulson and de Meer 1997]

—*dynamic change management*: the system must accommodate modifications or extensions dynamically. This is needed for example in *mobile systems* as the configuration of software components evolves over time [Kramer and Magee 1990].

One of the outputs of requirements analysis is a set of system models called the *requirements specification* which represents an abstraction of the system being studied and serve as a bridge between the analysis and design processes. Some formal and semi-formal models for requirements analysis will be described in Sections 3 and 4.2 respectively.

Functional and non-functional requirements relate to what the program is supposed to achieve *(computational model)*. In addition, there may be a specification of the architectural platform on which the program is to be executed *(architectural model)*. This is needed particularly for parallel and inherently distributed applications. Parallel architectural models will be discussed in Section 6.2.

2.2 Software design

Given some system requirements and models, the design stage involves developing several more detailed models of the system at lower levels of abstraction. Considering distributed applications, the main concepts that need to be embodied in every design can be grouped under these three categories:

—*Structure and component identification*: describes different components of the system such as processes, modules and data and their abstractions. In this paper, we will only concentrate on components that exhibit some concurrent behaviour.

—*Interaction management*: considers the dynamic aspects and semantics of communication. e.g. defining interfaces and communication protocols between components, which components communicate with which, when and how communication takes place, contents of communication etc.

—*Logical-physical mapping*: defines the mapping of logical entities from the computational model to physical entities from the architectural model. Such mappings can be defined statically (decided at compile-time) or dynamically (decided at run time).

Although strategies for designing sequential systems have been extensively studied, little is know about the design of distributed applications. Existing design methods that address some of the above issues will be examined in Section 4.3.

2.3 Implementation

The implementation stage consists of transforming the software design models into a set of programs or modules. Distributed applications are characterised by a wide spectrum of implementation platforms. In its simplest form, a platform consists of a set of homogeneous processing nodes connected by a network. In a more general context, a platform consists of several *resources* connected through one or several high-speed networks. There are several types of resources including:

—*Computational resources* such as PCs, workstations and vector computers.

—*Specialised resources* such as mass storage archives and graphics processors.

—*Hardware devices* such as sensors and actuators.

There are also several types of networks including custom networks (such as those in dedicated parallel architectures), Local Area Networks (LANs) and Wide Area Networks (WANs).

2.4 Conclusion

This section has outlined the specific needs of distributed applications within the development cycle. The paper now briefly reviews a selected number of research areas, focusing on their specific contribution towards the development cycle in general. The areas reviewed are concurrency theory (next Section), real-time systems (Section 4), distributed systems (Section 5) and parallel processing (Section 6).

3. CONCURRENCY THEORY

3.1 An overview of proposed theories and models

Concurrency theory whose roots are in the early 60s [Dijkstra 1968; Petri 1962] has emerged in response to problems encountered in the design of concurrent systems from different areas in computer science. One type of problems is caused by the presence of concurrent threads of control (i.e. *processes*) which lead to subtle errors such as *interference*, *deadlock* and *livelock*. The other problem is since most concurrent systems are *reactive* in nature (e.g. real-time and distributed systems), traditional notions of correctness do not apply. Therefore, formal models have an important role to play during requirements analysis particularly in specifying the dynamic behaviour of an application in terms of its concurrent activities and interaction between these activities.

One popular approach is to use *process algebra*, which are mathematical formalisms (based on set theory) that can be used to describe the interaction and synchronisation between concurrent processes. They only use few constructs such as sequential and parallel composition and non-deterministic choice but allow the expression of the full complexity of concurrent computations. Algebraic laws allow the transformation of one system into another. Amongst the formalisms available are Hoare's CSP (Communicating Sequential Processes) [Hoare 1985], Milner's CCS (Calculus of Communicating Systems)[Milner 1989] and LOTOS [Bolognesi and Brinksma 1988]. These theories employ *interleaving* i.e. they treat the concurrent execution of a program as the interleaving of the operations that constitute its processes.

An alternative to interleaving is *true concurrency* where concurrency is treated as a primitive notion i.e. the behaviour of the system is represented in terms of the causal relations between the events performed at different locations. A number of truly concurrent models with associated graphical notations exist. Amongst them Petri Nets [Peterson 1981] which were introduced in the early 60's by Petri [1962]. Petri Nets are useful in the description and analysis of synchronisation, communication and resource sharing between concurrent processes. They can be described pictorially by means of a bipartite directed graphs and have a small set of simple rules.

For dynamic systems, some theories which can model processes with evolving communication structure have been proposed. An example is Milner's π-calculus [Milner 1991; Milner et al. 1992] where a system is viewed as a collection of independent processes which may share communication links or *bindings* with other processes. These binding are referred to by unique 'names' and constitute the most primitive entities in the calculus. By naming links (instead of processes), it is possible for example to model client/server interactions (see Section 4.3) i.e. clients request a service without knowing which process will ultimately handle it.

3.2 Conclusion

Despite their advantages, formal models are still far from being widely used because formal specification and verification of a concurrent system is a non-trivial task which requires mathematical expertise and skills, particularly when tackling large and complex systems. While there has been some success in applications involving safety-critical requirements, it is not clear how current techniques could address other requirements as well.

Another problem is the lack of integration within the rest of the development cycleromising development is the increased availability of a number of tools to support the formal specification and verification of concurrent systems. These tools typically allow model checking and the verification of some properties such as absence of deadlock, livelock etc. A survey of these tools is presented by Cleaveland *et al.* [Cleaveland 1996]. Examples include the Facile system [Thomsen et al. 1996] (CCS) theπ-calculus Mobility Workbench [Victor 1994] and LTSA tool [Magee and Kramer 1999] (see example in Appendix). However, these tools are still perceived by practitioners as having limited practical value.

4. REAL-TIME SYSTEMS

We now consider another discipline, which is that of real-time systems design and implementation. We first start by briefly presenting "traditional" approaches of developing systems, which only address the implementation phase of the development cycle. Then we discuss progress in software engineering techniques specifically aimed at addressing the needs of real-time systems at the requirements and design stages.

4.1 Implementation issues

For various reasons such as easy interaction with hardware, memory size limitations, performance and time management, real-time software development has mostly been carried out at the assembly-language level. To address the problems of complexity and lack of portability, a number of "lightweight" operating systems have been proposed. These operating systems specifically address the needs of real-time systems by offering features such as support for time, exceptions and scheduling. Some operating systems such as Chorus [Rozier et al. 1988] also provide support for real-time systems on distributed platforms (more details on distributed operating systems will be given in Section 5.2).

There have also been efforts at designing programming languages specifically for real-time systems. Ada [Barnes 1995] is probably the most famous example of a high-level language commissioned by the US Defense Department to address the needs of embedded real-time systems. It has support for concurrent tasks, timing and a client/server form of interaction called the *rendezvous.* occam [Limited 1988] is another example of a programming language which supports concurrent processes and a synchronous (CSP-style) message-passing mechanism.

4.2 Semi-formal models for requirement analysis

Considering existing models for requirements analysis, most of the work has concentrated on the functional aspects of the application. Amongst the most useful models are *data flow models* (DFDs) [DeMarco 1978; Yourdon 1989] which represent the functions performed by the system and data movements between these functions. Recently, there has been a considerable interest in *object models* in which a system is decomposed into objects that represent real-world entities [Awad et al. 1996; Meyer 1988]. Various notations exist such as the one used by Booch [1994] and UML (to be described later).

Considering the dynamic behaviour of a real-time application, *state models* are useful in specifying such behaviour [Allworth and Zobel 1987; Minsky 1972]. Assuming that at any time, the system is in one of a number of possible states, an event (or stimulus) forces a transition to another state. The most basic notation for state models is a State Transition Diagram (STD) [Hopcroft and Ullman 1979]. Harel's *Statecharts* [Harel 1988] is another visual notation which can be regarded as an improvement over an STD as it supports some abstraction mechanisms such as concurrent states, hierarchical decomposition of states and aggregation of state transitions.

Other notations for specifying dynamic behaviour exist. For example, UML's *Sequence Diagram* [Eriksson and Penker 1998] shows sequences of actions performed by concurrent activities as vertical lines and interactions between actions as horizontal lines. This can be useful in showing temporal dependencies that are difficult to model using other notations. All notations discussed so far are referred to as *semi-formal*. It is also possible to use any of the formal notations discussed earlier in Section 3.1.

4.3 Design methods

Given some system requirements and models, the design stage involves developing several more detailed models of the system at lower levels of abstraction. Considering component identification, early structured methods such as JSD [Cameron 1986] and CODARTS [Gomaa 1993] provide one type of component which exhibits concurrent behaviour called a *task*. More recent structured methods such as Mascot [Simpson 1986] differentiate between lightweight concurrent entities called *threads* or *activities* and heavyweight concurrent activities usually called processes or *objects*. Another example is HRT-HOOD [Burns and Wellings 1994; Burns and Wellings 1995] which provides additional process types such as *cyclic* (or periodic) processes and *sporadic* processes. Object-oriented methods can be divided into *explicit* and *implicit* methods [Awad et al. 1996]. Explicit methods provide a separate notation for processes and objects early in the design stage. Implicit methods (such as UML [RATIONAL 1999; Eriksson and Penker 1998]) initially consider all objects as potentially concurrent entities. Later in the design stage, some objects are referred to as *active objects* i.e. they have an independent thread of control associated with them.

Processes have an external *interface* through which they interact with other processes. An advantage of (implicit) object-oriented methods is that that is no need to distinguish between interfaces for active objects and other (passive) objects. Processes also need a specification of their internal behaviour i.e. how do they react to external events. Some methods use standard techniques such as DFDs and STDs. Usually, functional and behavioural information are held separately. UML offers a rich set of notations for specifying the behaviour of objects, amongst them STDs (based on Statecharts) and *Sequence Diagrams* (mentioned in Section 4.2). Time specifications and constraints can be added these diagrams.

The most basic form of interaction in most design methods is *message-passing*. In JSD and Mascot, tasks communicate through streams or shared data specified using a *network diagram*. The network description includes the functions associated with particular events. Timing constraints can be added to the design to specify the time between input and output events. In CODARTS, *task architecture diagrams* show the decomposition of the system into concurrent tasks and the interfaces between them in the form of messages, events and modules. More elaborate forms of interaction include the *client/server* model. In such a model, a server process provides a service or a function through its interface. Several client processes send requests to the server which, once the connection is established, provides the required service (this involves data transfer in both directions).

In UML, as active objects are conceptually no different from other (passive objects), the most basic means of communication is the operation call mechanism which is equivalent to synchronous message passing. An object can also send an *explicit signal object* to another object. Such signals are interpreted as events and can be handled through the object's behaviour representation (e.g. an STD). Since signal are objects, they can carry both information and behaviour about the event and this allows for a variety of interaction forms to be modelled. Interaction with hardware devices is possible through *hardware wrapper classes* which present interfaces to the communication protocol of the devices.

In a distributed context, design methods should also provide a means for *logical-physical mapping* which refers to mapping logical entities (in the computational model) to physical entities (in the architectural model). It may also involve specifying the order of execution for processes running within a single distributable entity particularly in the presence of real-time constraints (i.e. a task scheduling strategy).

Physically distributable entities are called *virtual nodes* by Burns and Wellings [1997], *subsystems* by Gomaa [1993] or *distributable components* by Ng, Kramer, Magee, and Dulay [1996]. The mapping activity in UML is essentially a matching exercise between two diagrams: the *Component Diagram* which describes the run-time components of the system and the *Deployment Diagram* which specifies the physical resources. A Deployment Diagram shows the computers and devices (called nodes) that comprise the system. The type of network (e.g. TCP/IP, DECNET) is also shown on the diagram. Vickers and McDermid [1992] use a similar type of model for specifying a distributed real-time architecture. Their notation allows the representation of processors, memory units, communication links, information transducers and the environment. In most cases, the mapping activity is the prime responsibility of the user. HRT-HOOD is the only method which specifically provides a set of guidelines for logical-physical mapping taking into account both the non-functional requirements of the application and the constraints of the execution environment.

4.4 Conclusion

Software engineering methodologies and their real-time extensions provide a rich set of user-friendly notations for process identification and interaction management. A number of CASE tools that support these methodologies are available. Examples include STATEMATE [Harel et al. 1990] and Rhapsody [Harel and Gery 1997]. Most of these tools also translate models into code skeletons that are used as a basis for the implementation stage. The code is usually in C++ or Java.

A problem which is far from resolved is how to make best use of a methodology in a particular context or in the presence of particular requirements. Structured method usually advocate a top-down strategy which may not be suited to real-time systems since non-functional requirements play an important role. Object-oriented methods also do not come with any clear design strategies particularly when it comes to identifying active objects. In such cases, an iterative approach such as the one proposed by Burns and Lister [1991] would be more appropriate. Another problem with most of the approaches decribed earlier is the lack of formal semantics underlying their notations, which can lead to ambiguities and make it difficult to achieve automatic code generation. For instance, efforts to define a formal semantics for UML are still under way [Breu et al. 1997]. The expressive power of most established methodologies is also fairly limited. For example, they are only suitable for static systems and do not provide suitable replication structures.

Finally, logical-physical mapping is still a much neglected activity in the design process, mainly due to the lack of unified architectural models which can represent the physical resources in terms of processors, memory, communication etc.

5. DISTRIBUTED SYSTEMS

The next research area to be considered is distributed systems which have witnessed an explosive growth over the recent years thanks to the popularity of the Internet and the World Wide Web. These systems, initially provided to support sharing of resources such as printers and disks, now play a much wider role in providing a variety of services not only over local-area clusters of workstations but over a wide range of other communication networks as well.

In this section, we give a brief overview of existing approaches for implementing distributed systems followed by a selection of higher-level approaches aimed at improving the development process.

5.1 Implementation issues

Most operating systems offer a variety of concurrent and distributed processing facilities. For concurrent processing (within one workstation), routines are provided for process handling and communication management such as process creation, shared-memory access and message-passing. Recent efforts have led to a standard for specifying the creation and manipulation of local concurrent processes (i.e. threads) known as POSIX [Burns and Wellings 1997]. For distributed processing, a *remote procedure call* (RPC) mechanism is provided whereby a client process can invoke a procedure based on a remotely located server. Despite their flexibility, operating systems are perceived as too low-level and as a result, concurrent and distributed programming language design is an area which has been extensively studied and a number of surveys notably by Andrews and Schneider [1983] and Bal, Steiner, and Tanenbaum [1989] have been made. Two approaches are worth highlighting in this paper. The first one is a tendency to provide an "add-on" library to a sequential language such as C. Examples include PVM (Parallel Virtual Machine) [Geist 1994] and MPI (Message Passing Interface) [Pacheco 1996]. These standards support basic forms of message passing (e.g. send/receive) as well as group communication routines. Their main advantage is that they have ports on most distributed platforms so they come with a guarantee of code portability across a wide range of systems. The second approach that is growing in popularity is the use of object-oriented languages which support multithreaded and distributed objects [Meyer 1993; Papathomas 1995]. Examples include Java [Lea 1997] and Modula 3 [Nelson 1991]. As message-passing is the natural communication mechanism between objects, interactions evolve around the client/server model whereby a client object sends a request for a method to be invoked on a server object. For distributed programming, there have been several proposals for the provision of distributed objects either as a completely new language or supported by a distributed operating system (see next section). This is still an active research area [Cahill et al. 1997].

5.2 Distributed operating systems and middleware

A distributed operating system (DOS) provides a variety of mechanisms for the management of distributed services such as file transfer, remote execution and

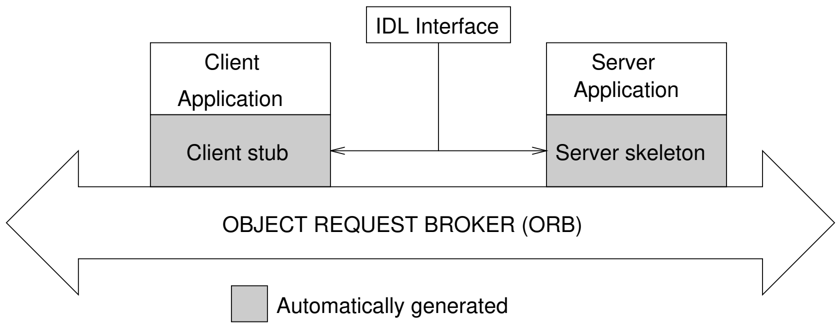

Fig. 2. The CORBA Client/Server Model of Communication

control of resource access. Distributed operating systems also offer support for *threads* (lightweight processes) and *clusters* (a grouping of several threads). Usually, threads interact through shared memory and clusters by message-passing via communication *ports*. Examples of such operating systems include Chorus [Rozier et al. 1988] and The Open Software Foundation's Distributed Computing Environment (OSF/DCE) [OSF 1992].

Most DOS support client/server forms of interaction which are more elaborate versions of the traditional RPC mechanism. Some DOS offer much richer forms of interaction. For example, clients can group several services in an *atomic transaction*. Since operations in a transaction all succeed or all fail with no effect, transactions are useful for handling updates in database systems.

Another recent development has been to enable distributed applications to cooperate with each other irrespective of the hardware or operating system being used. The most important standard that has been developed is the Common Object Request Broker Architecture (CORBA). CORBA, which is a product of the Object Management Group (OMG) [OMG 1998], is described in several recent books [Baker 1997; Orfali and Harkey 1998; Otte et al. 1995].

In CORBA, clients and servers that are potentially distributed across different platforms communicate through an *Object Request Broker* (ORB) (see figure 2). The ORB, which is at the heart of the system, is responsible for delivering requests from client applications to server applications and sending responses back to the client applications. CORBA includes an Interface Definition Language (IDL) for specifying the services provided by an object. An IDL interface consists of a set of named operations and their parameters. Once the interface is defined, it can be automatically translated into interface files for a variety of languages such as Java, C++ and Cobol. These interface files, shown in figure 2 define a client stub and a server stub (known as a server skeleton). Their role is in hiding all low-level communication aspects between the application objects and the ORB.

The CORBA standard also define a range of *services* that are available to application objects. One of the services provided is the Naming Service, which is simply a repository for object references in the system. CORBA also includes a set of *facilities* which are standardised IDL interfaces providing high-level functionality at the application level. These facilities are divided into horizontal and vertical groups. Horizontal facilities are to be used across a wide range of application and include user interface and task management facilities. Vertical facilities which are domain-based provide functionality for specific application areas such as healthcare, telecommunications and financial services [OMG 1998].

5.3 Design patterns

This section describes the concept of *design patterns*, which has consequences across several phases in the development cycle. When designing a new system (particularly a complex one), it is unusual for designers to tackle it by developing a solution from scratch. Instead, they often recall a similar problem that they have already solved and adapt its solution. The idea of design patterns, originally proposed by Gamma et al. [Gamma et al. 1995], is to facilitate the reuse of well proven solutions based on experiences from developing real systems. Given a library of common "patterns" for designing software, developers choose the pattern that is most adapted to their needs. Patterns are often associated with object-oriented systems because of their support for reusability through classes and objects.

Patterns vary greatly in aims and scope. They offer solutions ranging from high-level strategies for organising software to low-level implementation mechanisms. The documentation of design patterns is informal and varies in the literature. In most descriptions, the information associated with the pattern (such as context, problem and solution) is presented in a textual form. In [Buschmann et al. 1996], structural information is presented using object diagrams and dynamic properties are expressed using Object Message Sequence Charts, a notation similar to Sequence Diagrams mentioned in Section 4.2.

Historically, most design patterns were identified by developers of object-oriented user interfaces whose main quality criteria were usability, extensibility and portability. However, there has been a growing number of patterns which also express known concurrent behaviour of interacting entities over a possibly distributed platform [Buschmann et al. 1996; Douglass 1998; Islam and Devarakonda 1995; Schmidt 1995]. Examples include *Pipes and Filters*, *Master-Slave* and *Client-Dispatcher-Server*, illustrated in figure 3.

The design of a complex application typically involves more than one pattern (see Appendix for an example). Beside design patterns, *implementation patterns* represent higher-level forms of programming abstractions. These patterns (called *idioms* in [Buschmann et al. 1996]) refer to commonly used language-dependent techniques which can be used to model the behaviour of interacting objects. Their description is informal and includes reusable code in the form of interfaces, classes and objects. Implementation patterns are being applied in a variety of contexts from concurrent programming in Java [Lea 1997] to distributed programming in CORBA [Mowbray and Malveau 1997]. A related concept which is closely linked to object-oriented systems is that of a *framework* [Johnson 1997],

5.4 Conclusion

From the concepts presented earlier, it appears that most efforts in distributed systems design have concentrated in supporting a uniform view of services over heterogeneous resources. Most systems tend to be designed in a *bottom-up* fashion where each process is designed separately using known methods and process interaction is outside the method's scope. This reflects the fact that a system is

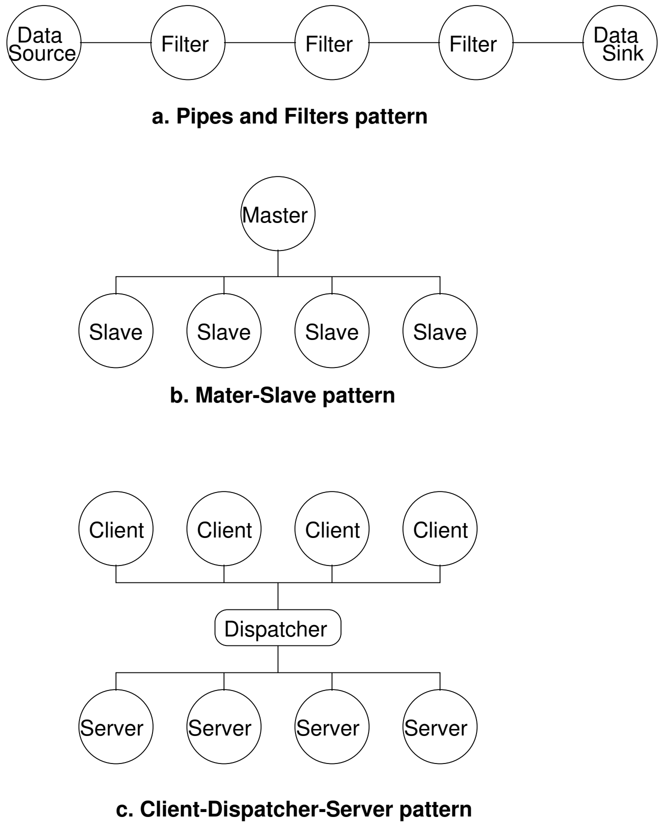

Fig. 3. Design patterns for distributed processing

often constructed using software modules that have already been developed. This explains the popularity of object models during the design phase since the ability of modelling interfaces in an abstract way makes reuse in such heterogeneous settings somewhat more manageable.

However, unlike with sequential object-oriented systems there are very few CASE tools which specifically address the needs of distributed processing, which are availability, compatibility between the communication interfaces and correct behaviour. One example is The Software Architect's Assistant [Ng et al. 1996] where the structure of the system is defined in terms of its components visually or textually using the Darwin configuration language [Magee et al. 1995]. The behaviour of each component is specified using Labelled Transition Systems (LTS) (see Section B). Reuse is encouraged by the provision of a components library. As in Parse and ADL, the distributed program is constructed by binding components together using a visual tool. Automatic code generation in C++ is also possible.

In any development environment, one of the major problems is the difficulty in providing an integrated view of a distributed application's dynamic behaviour. One "formal" solution is the use of a *configuration language* such as Polylith [Purtilo 1994], Durra [Babacci et al. 1993] and Darwin [Magee et al. 1995] for specifying the interaction between processes. A more "ad-hoc" alternative involves the use of design patterns that facilitate the reuse and sharing of successful experiences and techniques for designing distributed systems.

6. PARALLEL PROCESSING

The field of parallel processing has experienced a considerable growth since the early 80s, mainly encouraged by the high demands in computational power emerging from various science and engineering disciplines. Despite the fall in hardware costs and the availability of powerful high-performance platforms, the field has not delivered most of its promises due to the difficulty to deliver stable, efficient and portable software [Skillicorn and Talia 1998]. For this reason, the design of architectural and programming models for parallel processing as well as tools to support them is still an active research area. This section discusses some of the progress in this area and their relevance to the general software development process.

6.1 Implementation issues

Languages for parallel processing can be divided into low-level languages which are fairly close to the architectural model they are intended for and high-level languages in which additional levels of abstraction are provided.

Low-level languages for parallel processing are conceptually no different from languages for real-time systems (see Section 4.1) and distributed processing (see Section 5.1). In addition, most parallel computers are now provided with standard libraries such as PVM and MPI (see Section 5.1) which ensure code portability on both parallel and distributed platforms.

Some projects have attempted to address the low-level nature of these languages by providing visual programming tools. Programs are typically represented as graphs where processes correspond to nodes (possibly annotated with code in a given language) and where arcs represent the flow of data between processes. As in pure dataflow, the activity of processes is triggered by the arrival of messages (firing rules). Examples include CODE [Browne et al. 1989] (based on Ada, FORTRAN and C) and POKER [Snyder 1984]. TRAPPER [Schafers et al. 1995] is a more comprehensive graphical programming environment for parallel systems. It allows the user to specify both programs and hardware configurations. It also supports hierarchical decomposition, mapping, and automatic code generation in C++ with standard libraries (such as PVM) handling the communication aspects.

However, these visual programming environments are to a large extent already covered by software engineering methodologies for real-time systems (see Section 4.3). Low-level programming models and their associated tools still do not address some of the typical needs of parallel applications such as:

—predictability: although the problem of portability has been addressed to a large extent, "portability of code" does not necessarily imply "portability of efficiency". For example, porting PVM code to a different platform does not carry the guarantee that the efficiency gains will be maintained.

—"scalable" abstractions: some parallel systems consist of hundreds or thousands tasks, explicitly specifying the decomposition into tasks the actions of each task, as well as the interactions between tasks is an enormous burden to the programmer. Thus there is a need for scalable decomposition and replication abstractions to handle these large systems.

6.2 Architectural models

Since performance requirements are the main driving force for designing a parallel system, an essential feature of the development process should be the availability of a detailed architectural model, preferably with execution-time cost measures. Therefore, one of the prime contributions of parallel processing research is a wealth of architectural models, all of them providing cost information about the following three aspects:

(1) computational structure: relates to the way the execution proceeds. For example, operations can be carried out synchronously (SIMD mode) or asynchronously (MIMD mode).

(2) memory organisation: relates to the way data is stored and accessed. For example, in a *shared-memory model*, all data is kept in a central storage area and in a *distributed-memory model*, each processor keeps a portion of data locally and remote access to other portions is done through message-passing.

(3) communication provision: relates to the communication infrastructure between processors and the associated costs, which can vary depending on the underlying network used for interconnecting processors and memory modules. Networks can be of fixed topology (e.g. rings, meshes and hypercubes) or may use a dynamic routing scheme (e.g. bus or switching network).

These models are powerful enough to represent a large spectrum of architectural designs. They can be low-level where precise details of operations and communications need to be specified or they can provide some degree of abstraction over one or several of the three aspects identified earlier. A popular low-level model is the Parallel Random Access Memory (PRAM) model [Akl 1989] and its variations. It assumes an ensemble of processors acting synchronously on a uniformly accessed memory. This model requires details of the operations performed by processors and requires that memory conflicts be avoided.

A higher-level model is the Bulk Synchronous Parallel (BSP) [McColl 1995; Skillicorn et al. 1996] which is based on a distributed-memory MIMD architecture. The model requires the identification of parallel tasks and a series of *supersteps*. Communication between tasks only take effect at the the end of a superstep where all tasks engage in a barrier synchronisation. The advantages of the model is that it can provide cost measures (in terms of execution time) without any concern about the underlying interconnection network while allowing for both local and global communication to take place.

6.3 Data parallel programming models

We now examine some higher-level programming approaches which aim at providing adequate structures to handle large scale parallel systems. An important class of parallel and high performance applications is concerned with the manipulation of large data structures (e.g. vectors) in parallel. Such applications, which are referred to as *data parallel* [Hillis and Jr. 1986], include solving large systems of partial differential equations, image processing and molecular dynamics. Consequently, a number of abstractions for data parallel applications have been proposed. They include include element-wise operations (e.g. adding two vectors), broadcast and reduction (e.g. computing the sum of a vector). Some languages provide programmers with a *macroscopic view* (APL style) in which data structures can be manipulated as a whole. Alternatively, some languages allow the programmer to specify how individual data items are modified is relation to an indexing system (this is called the *microscopic view*).

The advantage of data parallel abstraction mechanisms is simplicity of programming while efficient compilers, suitable for a wide range of parallel machines, can be developed. Examples of data parallel languages include C* [Frankel 1991] and High Performance Fortran (HPF) [Loveman 1993]. Another advantage is that empirical models for performance analysis can be developed.

6.4 Skeletons

The concept of a *skeleton* [Cole 1989] is very similar to that of a pattern (see Section 5.3) except that it emerged from a different research area (parallel programming instead of object-oriented programming) and is perceived to be more formal and systematic in its approach. It is based on the observation that many parallel programs share a common set of known interaction patterns such as pipelines, master/slave, data parallel etc. for which implementations have been studied extensively on a variety of languages/machines.

Most of the work on skeletons is associated with functional languages, as these skeletons can be modelled as higher-order functional structures. Amongst the variety of skeleton-related projects are those concerned with defining *elementary skeletons* from which parallel programs can be constructed. For example, the two well-known list processing operators *map* and *reduce* form a set of elementary skeletons with inherent parallelism. Despite the fact that equivalent operators are provided in most data parallel languages (*map* corresponds to an element-wise operation and *reduce* to a reduction), the main advantage of using elementary skeletons is the availability of a formal framework for program composition. This allows for a rich set of transformations (e.g. transforming one program into a more efficient one) to be applied. In addition, cost measures can be associated with these elementary skeletons and their compositions. As an example of elementary skeletons, a group at Imperial College [Darlington et al. 1995a; Darlington et al. 1995b] proposes a range of simple skeletons for *data distribution*, *alignment*, *communication*, *etc.*, from which data parallel applications can be constructed.

The main problem with elementary skeletons is that there is little guidance as to how to compose them in order to get the best efficiency. In addition, there have been very little practical implementations due to the difficulty in adapting arbitrary composition structures to a variety of hardware platforms with very different characteristics. To address these problems, more elaborate skeletons which model complex interaction patterns can be defined. For example, the *divide-and-conquer* skeleton captures a well-known algorithmic design strategy for which several efficient implementations can be developed [Rabhi and Manson 1991]. Several of such skeletons have been developed around particular data structures. These skeletons, which are derived from category theory [Skillicorn 1992], are known as *homomorphic* *skeletons* [Skillicorn 1994]. They provide a similar level of abstraction than data parallel operators considered earlier but in addition, they also offer a more formal framework for program construction and transformation. Homomorphic skeletons have been proposed for a variety of data structures such as lists, arrays, trees and graphs (see the survey by Skillicorn and Talia [1998]). They act in a similar way to an abstract data type in providing a set of known-to-be parallel operators while hiding the internal implementation details. A related proposal is that of Parallel Abstract Data Types (PADTS) [Wu et al. 1997] which is essentially an abstract data type for irregular data structures.

In the skeletons described so far, the communication structure is implied by the (often recursive) way operators are defined. There are skeletons which work around a fixed communication structure. For example, the Static Iterative Transformation (SIT) skeleton [Rabhi 1995a] which captures a series of iterative transformations being applied to a large data structure and for which several programming environments have been proposed and implemented [Parsons and Rabhi 1995; Parsons and Rabhi 1998; Rabhi 1995b; Schwarz and Rabhi 1996].

7. CONCLUSION

This paper presented a variety of concepts and techniques from different research areas related to the development of distributed applications. This section now concludes by summarising the strengths of each discipline and then outlines remaining problems and possible directions for future work.

7.1 Strengths of each discipline and overlappings

Figure 4 summarises the role and main contribution to the development cycle (introduced in Section 2) of most of the techniques discussed in this paper. Overall, the profusion of concurrency models and tools means that they are expected to have a more important role to play during the specification and requirements analysis phase of the cycle. This study also reveals that design approaches for defining concurrent behaviour, static scheduling and interaction management have been extensively studied in the context of real-time systems. Distributed systems research offers detailed studies of dynamic behaviour (e.g. patterns) as well standard interfaces and communication protocols over a wide range of platforms. Finally, parallel processing research is the most advanced in providing concise architectural models with cost measures, a rich set of data, process and communication replication structures (e.g. skeletons); and various implementation strategies for these structures.

7.2 The case for integrated approaches

Despite their apparent disparity, we believe that there are several common issues related to all distributed applications, such as process management, communication and synchronisation, distribution and logical-physical mapping. Despite the use of different notations and terminology, many similarities exist in areas such as semantics of communication, visual display of information and automatic code generation. As an example of "overlapping" of work, most programming abstractions provided in parallel processing tools (see Section 6.1) are to a large extent already available in software engineering methodologies for real-time systems (see Section 4.3).

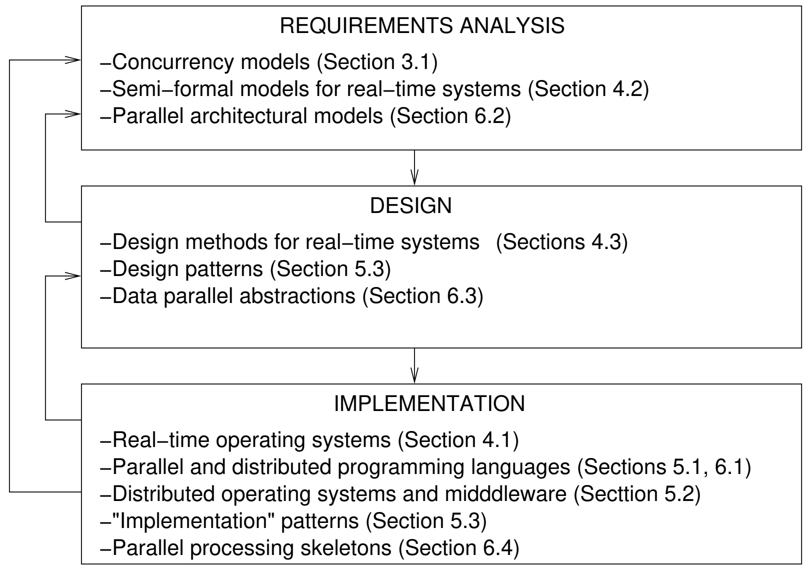

Fig. 4. Concepts and techniques within an integrated software engineering cycle

The other reason for integrated approaches is that with the greater availability and ease of use of large computer networks, there will be several applications that cross boundaries. For example, a distributed real-time system may have to consist of a large number of identical tasks for dealing with the fault-tolerance requirements and for managing identical hardware devices. This requires suitable replication structures to model and implement the concurrency, communication and distribution aspects.

The main problems that still need addressing are:

—Little is know about design strategies for distributed applicatioins. For example, we have seen that existing design strategies are not very well adapted to non-functional requirements and that logical-physical mapping is a neglected part of the development process.

—Many techniques rely on assumptions that are specific to the discipline they originate from so there are many difficulties associated with adapting concepts from one discipline to another. For example, most structured/OO design methodologies do not provide replication structures, network studies for parallel computers are not relevant to distributed processing, etc.

7.3 Future work

Future work should concentrate around the *adaptation* of concepts across disciplines and the *integration* of these concepts within all the phases of a well-defined development cycle.

Considering adaptation, most efforts at the requirements stage have concentrated on the functional requirements and the dynamic behaviour of systems. New theories and models need to be developed to express requirements such as quality of service, dynamic change management and dependability. Improved design abstractions as well as new ones are needed e.g. those with a capacity to model actors and intelligent agents capable of reactive, pro-active and co-operative behaviour etc. There is also a need for new unified architectural models which can represent physical resources in terms of processors, memory, communication etc. Finally, while middleware platforms (such as CORBA) have proven useful for applications with loosely coupled tasks and low communication requirements, their appropriateness for highly coordinated tasks with large demands on communication and synchronisation still needs investigation.

The case for integration should give a greater role for CASE tools that emphasize the role of formal notation, provide a rich set of design abstractions, allow model checking and provide automatic code generation. Integration of existing or new techniques should be achieved through formally defined, generic, reusable entities and their associated tools. Some of such entities have already been described in this paper as patterns and skeletons. This is not a new tendency but has already been happening to a large extent at the implementation level. For example, standards like CORBA and PVM can be regarded as "patterns" that support location transparency and decouple processes from the underlying communication mechanism. It is expected that similar approaches will be adopted at a much higher level in the development cycle.

REFERENCES

AKL, S. 1989. *The Design and Analysis of Parallel Algorithms.* Prentice Hall.

ALLWORTH, S. AND ZOBEL, R. 1987. *Introduction to Real-Time Software Design (2nd ed.).* Springer Verlag.

ANDREWS, G. AND SCHNEIDER, F. 1983. Concepts and notations for concurrent programming. *Computing Surveys 15*, 1 (March), 3–43.

AWAD, M., KUUSELA, J., AND ZIEGLER, J. 1996. *Object-Oriented Technology for Real-Time Systems.* Prentice Hall.

BABACCI, M., WEISTOCK, C., DOUBLEDAY, D., GARDNER, M., AND LICHOTA, R. 1993. Durra: a structure description language for developing distributed applications. *IEE Software Engineering Journal 8*, 2 (March), 83–94.

BAKER, S. 1997. *CORBA Distributed Objects Using Orbix.* Addison Wesley.

BAL, H., STEINER, J., AND TANENBAUM, A. 1989. Programming languages for distributed computing systems. *ACM Computing Surveys 21*, 3 (September), 261–322.

BARNES, J. 1995. *Programming in Ada 95.* Addison Wesley.

BOLOGNESI, T. AND BRINKSMA, E. 1988. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems 14*, 1, 25–29.

BOOCH, G. 1994. *Object Oriented Analysis and Design With Applications (2nd ed.).* Benjamin/Cummings.

BREU, R., HINKEL, U., HOFMANN, C., KLEIN, C., PAECH, B., RUMPE, B., AND THURNER, V. 1997. Towards a formalization of the unified modeling language. In M. AKSIT AND S. MATSUOKA Eds., *Proceedings of ECOOP'97 - Object-Oriented Programming, 11th European Conference, Jyvaskyla, Finland* (June 1997). Lecture Notes in Computer Science 1241, Springer Verlag.

BROWNE, J., AZAM, M., AND SOBEK, S. 1989. CODE : a unified approach to parallel programming. *IEEE Software 6*, 4 (July), 10–17.

BURNS, A. AND LISTER, A. 1991. A framework for building dependable systems. *The Computer Journal 34*, 2, 173–181.

BURNS, A. AND WELLINGS, A. 1994. HOOD: A structured design method for hard real-time systems. *Real-Time Systems 6*, 1, 73–114.

BURNS, A. AND WELLINGS, A. 1995. *A Structured Design Method for Hard Real-Time Ada Systems*. Elsevier.

BURNS, A. AND WELLINGS, A. 1997. *Real-Time Systems and Programming Languages (2nd Ed.)*. Addison-Wesley.

BUSCHMANN, F., MEUNIER, R., ROHNERT, H., SOMMERLAD, P., AND STAL, M. 1996. *A System of Patterns: Pattern-Oriented Software Architecture*. J. Wiley and Sons.

CAHILL, V., NIXON, P., TANGNEY, B., AND RABHI, F. 1997. Object models for distributed or persistent programming. *The Computer Journal 40*, 8, 513–527.

CAMERON, J. 1986. An overview of JSD. *IEEE Transactions on Software Engineering SE-12*, 2 (February), 222–262.

CLEAVELAND, R. 1996. Strategic directions in concurrency research. *Computing Surveys 28*, 4 (December), 607–625.

COLE, M. 1989. *Algorithmic skeletons: a structured approach to the management of parallel computation*. Research monographs in Parallel and Distributed Computing, Pitman.

COULSON, G. AND DE MEER, J. 1997. Special issue on quality of service. *Distributed Systems Engineering Journal 4*, 1 (March), 1–3.

DARLINGTON, J., GUO, Y., TO, H., AND YANG, J. 1995a. Functional skeletons for parallel coordination. In S. HARIDI, K. ALI, AND P. MAGNUSSIN Eds., *EuroPar'95 Parallel Processing* (August 1995), pp. 55–69. Springer-Verlag.

DARLINGTON, J., GUO, Y., TO, H., AND YANG, J. 1995b. Parallel skeletons for structured composition. In *Fifth ACM SIPLAN Simposium on Principles and Practice of Parallel Programming* (July 1995), pp. 19–28. ACM Press.

DEMARCO, T. 1978. *Structured Analysis and System Specification*. Prentice Hall.

DIJKSTRA, E. 1968. Cooperating sequential processes. In F. GENUYS Ed., *Programming Languages*, pp. 43–112. London: Academic Press.

DOUGLASS, B. 1998. *Real-Time UML: Developing Efficient Objects for Embedded Systems*. Addison-Wesley.

ERIKSSON, H. AND PENKER, M. 1998. *UML Toolkit*. J. Wiley and Sons.

FRANKEL, J. 1991. *C* Language Reference Manual*. Thinking Machines Corp., Cambridge MA, USA.

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1995. *Design Patterns: Elements of reusable Object-Oriented Software*. Addison Wesley, Professional Computing Series.

GEIST, A. 1994. *Parallel Virtual Machine (PVM): a Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press.

GOMAA, H. 1993. *Software Design Methods for Concurrent and Real-Time Systems*. Addison-Wesley.

HAREL, D. 1988. On visual formalisms. *Communications of the ACM 31*, 5, 514–530.

HAREL, D. AND GERY, E. 1997. Executable object modelling with statecharts. *IEEE Computer 30*, 7 (July), 31–42.

HAREL, D., LACHOVER, H., NAAMAD, A., PNUELI, A., POLITI, M., SHERMAN, M., SHTULL-TRAURING, A., AND TRAKHTENBROT, M. 1990. STATEMATE: a working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering 16*, 4, 403–414.

HILLIS, W. AND JR., G. S. 1986. Data parallel algorithms. *Communications of the ACM 29*, 12, 1170–1184.

HOARE, C. 1985. *Communicating Sequential Processes*. Prentice-Hall International.

HOPCROFT, J. AND ULLMAN, J. 1979. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, Reading MA.

ISLAM, N. AND DEVARAKONDA, M. 1995. An essential design pattern for fault-tolerant distributed state sharing. *Communications of the ACM 39*, 10 (October), 65–74.

JOHNSON, R. 1997. Frameworks = (components + patterns). *Communications of the ACM 40*, 10 (October), 39–42.

KRAMER, J. AND MAGEE, J. 1990. The evolving philosophers problem: dynamic change management. *IEEE Transactions on Software Engineering 16*, 11 (November).

LALONDE, W. AND PUGH, J. 1991. *Inside Smalltalk, Volume II.* Prentice-Hall.

LEA, D. 1997. *Concurrent Programming in Java : Design Principles and Patterns.* Addison Wesley Longman.

LIMITED, I. 1988. *occam 2 Reference Manual.* Prentice Hall.

LOVEMAN, D. 1993. High performance fortran. *IEEE Parallel and Distributed Technology.*

MAGEE, J., DULAY, N., EISENBACH, S., AND KRAMER, J. 1995. Specifying distributed software architectures. In *Fifth European Software Engineering Conference ESEC'95* (Barcelona, September 1995).

MAGEE, J. AND KRAMER, J. 1999. *Concurrency: State Models and Java Programs.* John Wiley and Sons.

MCCOLL, W. 1995. Bulk synchronous parallel computing. In J. DAVY AND P. DEW Eds., *Abstract Machine Models for Highly Parallel Computers*, pp. 41–63. Oxford University Press.

MEYER, B. 1988. *Object Oriented Software Construction.* Prentice Hall International.

MEYER, B. 1993. Systematic concurrent object-oriented programming. *Communications of the ACM 36*, 9 (September), 56–80.

MILNER, R. 1989. *Communication and Concurrency.* Prentice Hall.

MILNER, R. 1991. The polyadic π-calculus: a tutorial. Technical Report ECS-LFCS 91-180 (October), University of Edinburgh.

MILNER, R., PARROW, J., AND WALKER, D. 1992. A calculus of mobile processes - parts I & II. *Information and Computation 100*, 1–77.

MINSKY, M. 1972. *Computation: Finite and Infinite Machines.* Prentice Hall.

MOWBRAY, T. AND MALVEAU, R. 1997. *CORBA Design Patterns.* Wiley Computer Publishing.

NELSON, G. 1991. *Systems Programming with Modula-3.* Prentice Hall.

NG, K., KRAMER, J., MAGEE, J., AND DULAY, N. 1996. A visual approach to distributed programming. In A. ZAKY AND T. LEWIS Eds., *Tools and Environments for Parallel and Distributed Systems*, pp. 7–32. Boston: Kluwer Academic Publishers.

OMG. 1998. The common object request broker: Architecture and specification, revision 2.2. Report 98-07-01, The Object Management Group, `http://www.omg.org/`.

ORFALI, R. AND HARKEY, D. 1998. *Client/Server Programming with Java and CORBA (2nd ed.).* John Wiley and Sons.

OSF. 1992. The distributed computing environment (DCE). Technical report. Open Software Foundation, `http://osf.org/dce/index.html`.

OTTE, R., PATRICK, D., AND ROY, M. 1995. *Understanding CORBA.* Prentice Hall.

PACHECO, P. 1996. *Parallel Programming with MPI.* Morgan Kaufmann Publishers.

PAPATHOMAS, M. 1995. Concurrency in object-oriented programming languages. In O. NIERSTRASZ AND D. TSICHRITZIS Eds., *Object Oriented Software Composition*, pp. 31–68. Prentice Hall.

PARSONS, P. AND RABHI, F. 1995. Specifying problems in a paradigm-based parallel programming system. In E. D'HOLLANDER, G. JOUBERT, F. PETERS, AND D. TRYSTRAM Eds., *Parallel Computing: State-of-the-Art and Perspective* (1995), pp. 215–237. North Holland.

PARSONS, P. AND RABHI, F. 1998. Generating parallel programs from paradigm-based specifications. *Journal of Systems Architecture 45*, 4, 261–283.

PETERSON, J. 1981. *Petri Net Theory and the Modeling of Systems.* Prentice Hall.

PETRI, C. 1962. Fundamentals of a theory of asynchronous information flow. In *Information Processing 1962, Proc. IFIP Congress 1962* (Munich, 1962), pp. 386–390. North Holland Publishing Company, Amsterdam.

PURTILO, J. 1994. The polylith software bus. *ACM Transactions on Programming Languages 16*, 1 (January), 151–174.

RABHI, F. 1995a. Exploiting parallelism in functional languages : a "paradigm-oriented" approach. In J. DAVY AND P. DEW Eds., *Abstract Machine Models for Highly Parallel Computers*, pp. 118–139. Oxford University Press.

RABHI, F. 1995b. Parallel programming methodology based on paradigms. In P. NIXON Ed., *Transputer and Occam Developments*, pp. 239–252. IOS Press.

RABHI, F. AND MANSON, G. 1991. Divide-and-conquer and parallel graph reduction. *Parallel Computing 17*, 189–205.

RATIONAL. 1999. The unified modelling language (UML) resource centre. Rational Software Corporation, `http://www.rational.com/uml`.

ROZIER, M., ABROSSIMOV, V., ARMAND, F., BOULE, I., GIEN, M., GUILLEMONT, M., HERRMANN, F., KAISER, C., LANGLOIS, S., LEONARD, P., AND NEUHAUSER, W. 1988. Chorus distributed operating systems. *Computing Systems Journal, The USENIX Association 1*, 4 (December).

SCHAFERS, L., SCHEIDLER, C., AND KRAMER-FUHRMANN, O. 1995. Trapper: a graphical programming environment for parallel systems. *Future Generation Computer Systems 11*, 351–361.

SCHMIDT, D. 1995. Using design patterns to develop reusable object-oriented communication software. *Communications of the ACM 38*, 10 (October), 65–74.

SCHWARZ, J. AND RABHI, F. 1996. A skeleton-based implementation of iterative transformation algorithms using functional languages. In M. KARA, J. DAVY, D. GOODEVE, AND J. NASH Eds., *Abstract Machine Models for Parallel and Distributed Computing*, pp. 119–134. IOS Press.

SIMPSON, H. 1986. The Mascot method. *Software Engineering Journal*, 103–120.

SKILLICORN, D. 1992. The bird-meertens formalism as a parallel model. In J. KOWALIK AND L. GRANDINETTI Eds., *NATO ARW "Software for Parallel Computation"*. NATO ASI series F: computer and systems sciences; v.106, Springer Verlag.

SKILLICORN, D. 1994. *Foundations of Parallel Programming.* Cambridge International Series on Parallel Computation, vol. 6, Cambridge University Press.

SKILLICORN, D., HILL, J. M., AND MCCOLL, W. 1996. Questions and answers about BSP. Technical Report PRG-TR-15-96 (November), Oxford University Computing Laboratory.

SKILLICORN, D. AND TALIA, D. 1998. Models ans languages for parallel computation. *ACM Computing Surveys 30*, 2 (June), 123–169.

SMARR, L. AND CATLETT, C. E. 1992. Metacomputing. *Communications of the ACM 35*, 6 (June), 44–53.

SNYDER, L. 1984. Parallel programming and the POKER environment. *IEEE Computer*, 27–36.

SOMMERVILLE, I. 1996. *Software Engineering (5th ed.).* Addison-Wesley.

THOMSEN, B., LETH, L., AND KUO, T. 1996. A facile tutorial. In U. MONTANARI AND V. SASSONE Eds., *CONCUR'96: Concurrency Theory*, pp. 278–298. Lecture Notes in Computer Science, Springer Verlag.

VICKERS, A. AND MCDERMID, J. 1992. An approach to the design of software for distributed real-time systems. Technical Report YCS 211 (November), Department of Computer Science, University of York.

VICTOR, B. 1994. A verification tool for the polyadic π-calculus. Technical Report DoCS 94/50, Department of Computer Systems, Uppsala University, `http://www.docs.uu.se/~victor/mwb.html`.

WU, Q., FIELD, A., AND KELLY, P. 1997. M-tree: a parallel abstract data type for block-irregular adaptive applications. In *EuroPar'97 Parallel Processing* (1997), pp. 638–649. Lecture Notes in Computer Science 1300, Springer Verlag.

YOURDON, E. 1989. *Modern Structured Analysis.* Prentice Hall International.

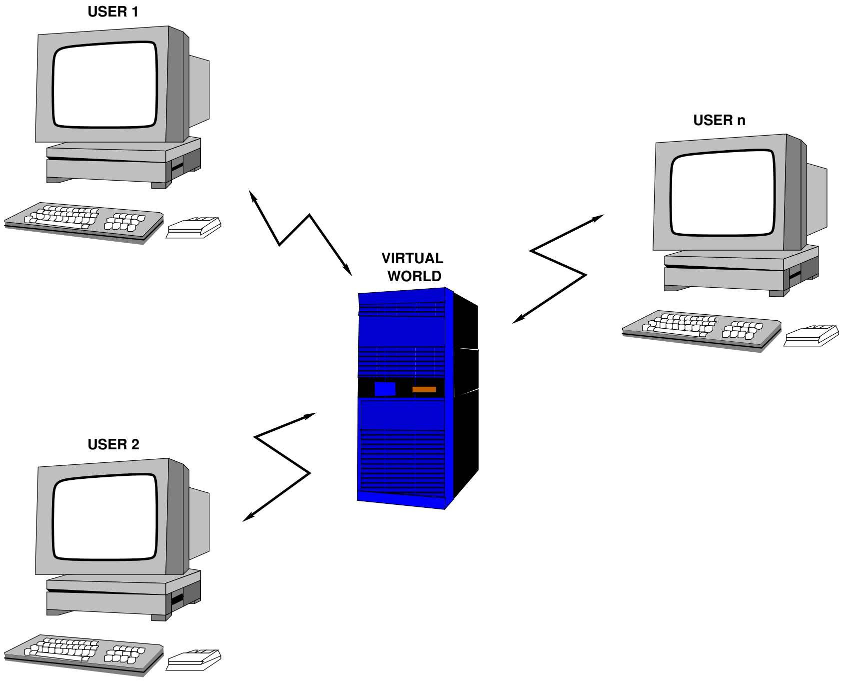

Fig. 5. A distributed virtual environment

APPENDIX

A. A DISTRIBUTED VIRTUAL ENVIRONMENT

To illustrate a distributed application that contains many of the features discussed in the paper, consider the example of a *distributed virtual environment* which is composed of a *virtual world* and several participants interacting simultaneously with it. An example application would be a surgical training system where participants include one trainer (the surgeon) and several trainees (the observers). The surgeon conducts the operation through a "virtual hand controller" and provides explanations through a microphone. All observers can visualise the model patient from various angles through a helmet mounted display as well as receive audio information from the surgeon through headphones. Another application would be a distributed video game where several participants can interact with each other and modify the virtual world concurrently.

Developing such an environment is a challenging task particularly for the inexperienced software developer. First, there are performance requirements (e.g. the virtual world has to run the simulation efficiently), real-time requirements (e.g. outputs must be generated within some time constraints), reliability requirements (e.g. the system must cope with users at different geographic locations) and dynamic changes (e.g. users joining and leaving at any time). Secondly, during the design activity, the concurrent aspects of the system must be correctly expressed, an architectural model which embodies the resources and the network(s) involved must be developed and an adequate logical-physical mapping that satisfies the re-

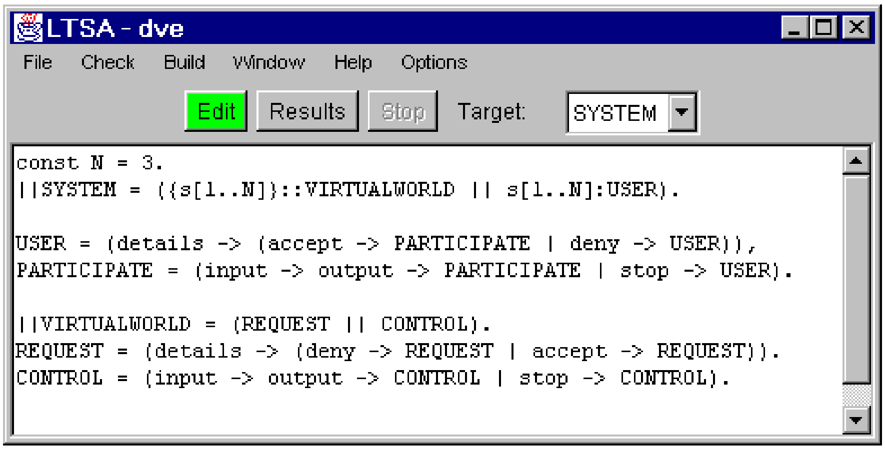

Fig. 6. FSP Specification of a Distributed Virtual Environment Protocol

quirements must be determined. Finally, the system must be implemented using a suitable combination of language(s), operating system(s) and communication protocol(s).

B. EXAMPLE OF USING A CONCURRENCY TOOL

An example of using a tool to model the dynamic concurrent behaviour of a system is illustrated in figure 6. It shows the main specification window of the LTSA tool [Magee and Kramer 1999] describing the overall protocol of users joining and leaving a virtual world (see Section A). It uses a notation called FSP (Finite State Processes)[Magee and Kramer 1999] which similar to both CCS and CSP. The specification identifies one process which represents the virtual world and N other processes which represent users. Each user can submit a request for participation (by sending some authentification details) and receives either an acceptance or a refusal. If accepted, a user generates inputs and consumes outputs until a stop event occurs. The virtual world itself consists of two concurrent processes, one responsible for registering users and another one for processing inputs/outputs. The LTSA tool also supports the conversion of FSP specifications into finite state machine descriptions called Labeled Transition Systems (LTS). Figure 7 shows the state diagram generated for one user in the system. Finally, the tool also supports formal analysis (such a safety checking) as well as the display and animation of specifications.

C. DESIGN IN UML

Considering the case study in Section A, we now model some of its design aspects in UML. Figure 8 shows an object diagram that represents some of the structural aspects of the system. This represents a refinement over the initial specification in Section B but is still in need of further improvements.

Considering the dynamic behaviour of the system, figure 9 is a UML state diagram which shows that the virtual world consists of two concurrent states, one that implements the main control loop and one that deals with registration requests. A

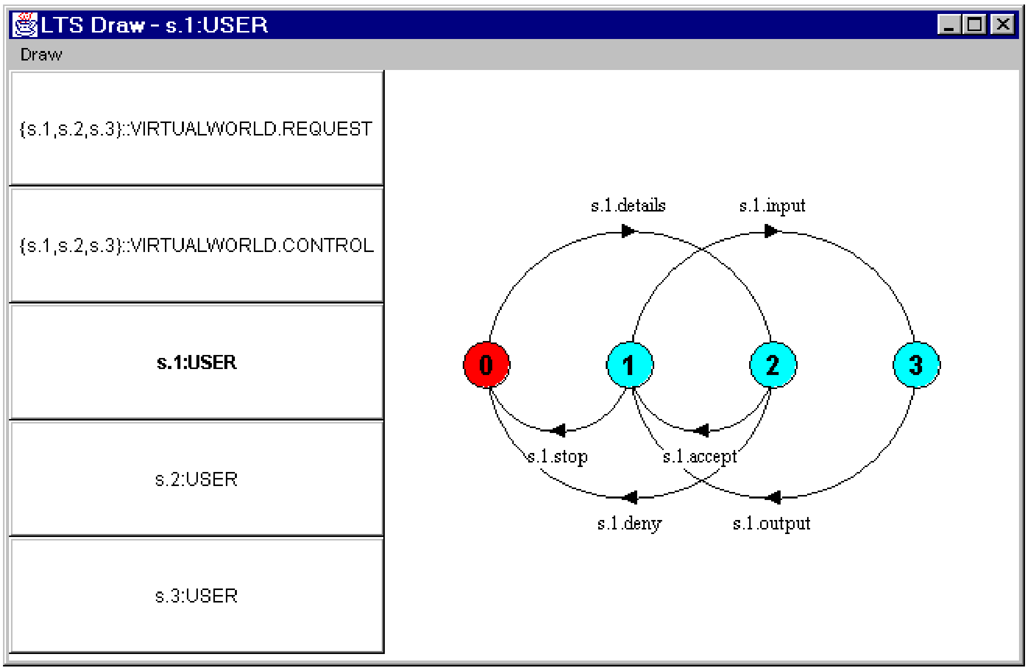

Fig. 7. A Labelled Transition Diagram generated by the LTSA tool

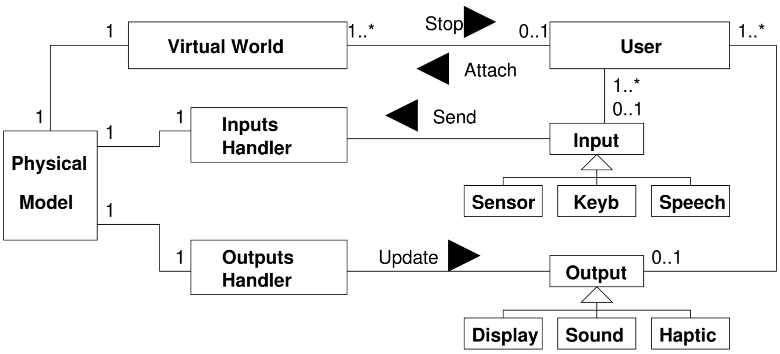

Fig. 8. Example of a UML object diagram

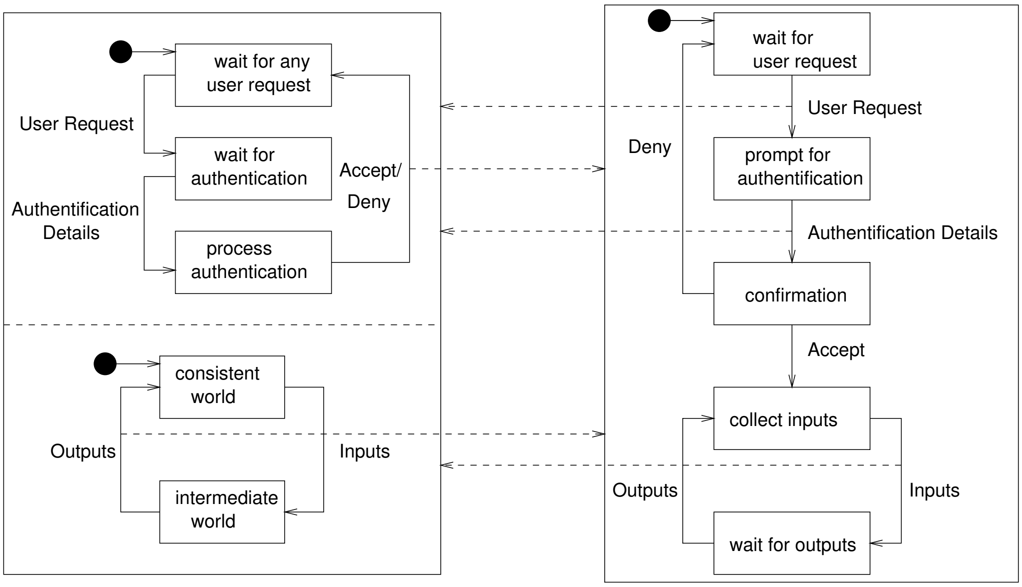

Fig. 9. Example of a UML state diagram

user can only start generating inputs and collecting results once it has passed the authentification procedure.

Once all the objects have been identified and their dynamic behaviour specified, they are grouped into components which are allocated to individual workstations. Figure 10 shows a UML deployment diagram that shows that all objects that are part of the Virtual World have been grouped into three components: one that implements the overall control algorithm, one that maintains the physical model and one that handles communication events. These components are then allocated to a single workstation which will act as a Virtual World Server. The figure also shows that all objects in the User part have been grouped into three components, one for collecting inputs, one for processing outputs and one for communicating with the Virtual World Server. The figure shows two instance users which are allocated to different workstations. All workstations communicate through an Ethernet network and one of the workstations has access to a glove subsystem.

D. DESIGN PATTERNS

The design of the distributed virtual environment case study involves a combination of several design patterns. One of them is the *Observer* pattern, whose object diagram is represented in figure 11, which specifies a server process in charge of monitoring a real-time environment and informing a set of clients processes (called the observers) when changes in the monitored values (e.g. sensor values) occur. Observers can register or detach from the system at any time. Alternatively, monitored values can be sent periodically particularly if there are risks of loosing data.

Another pattern that can be combined with this pattern is the *Model-View-Controller* pattern which divides an interactive application into three components:

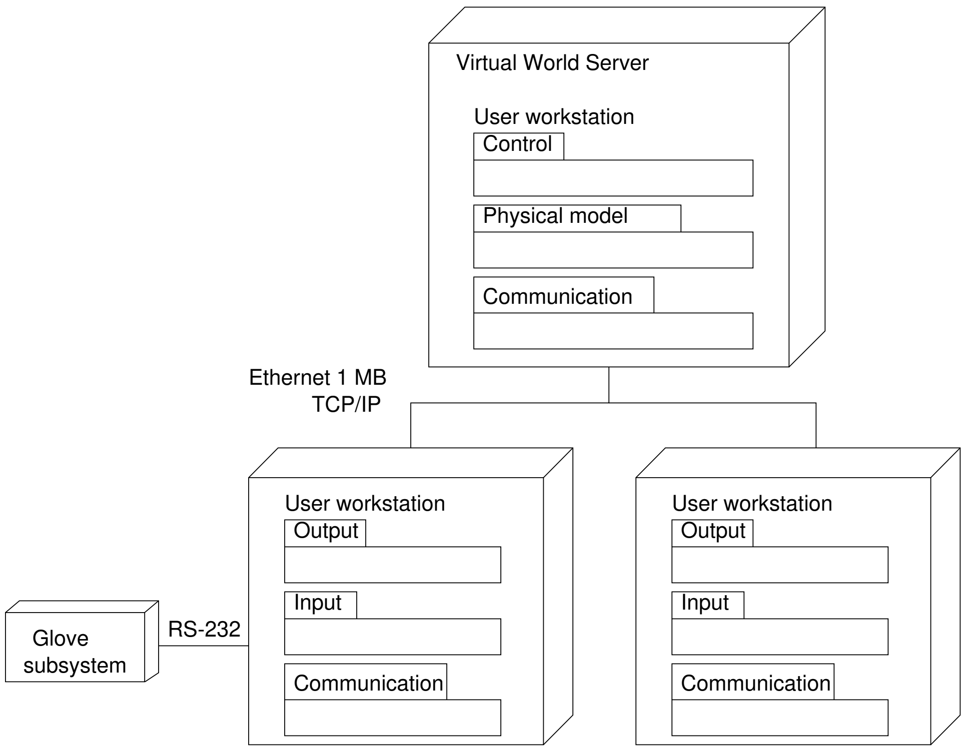

Fig. 10. Example of a UML deployment diagram

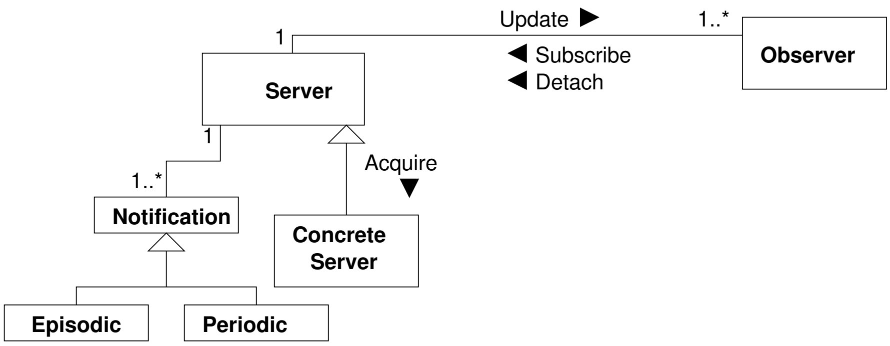

Fig. 11. Object model for the *Observer* pattern

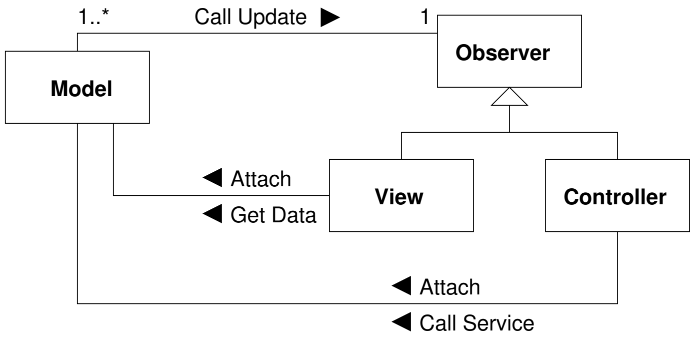

Fig. 12. Object model for the *Model-View-Controller* pattern

the model which contains the core functionality and data, the view which represents a set of outputs to the user and a controller which handles the user inputs. A UML object diagram which captures the structure of this pattern is illustrated in figure 12. Such a pattern has been used in different contexts such as in designing the Smalltalk programming environment [Lalonde and Pugh 1991].

Finally, a *Client-Dispatcher-Server* (see figure 3) can be used to illustrate the fact that users (clients) and the virtual world (server) should be able to interact regardless of their physical location and the communication medium used.