Object-Based Modeling of Parallel Programs

Ian Gorton
Division of Information Technology, CSIRO
Jonathan P. Gray
University of Wollongong
Innes Jelly
Sheffield Hallam University

Using a notation known as process graphs, this object-based approach to parallel software design represents designs simply and concisely. It then progressively refines them to capture all structural and dynamic properties of a system.

Only by using explicit parallel-programming techniques can many software applications meet their specifications. With parallelism, applications can exploit the processing power of multiprocessor systems to

- achieve high performance,
- provide fault tolerance and reliability in safety-critical and real-time systems, and
- deal with physically distributed computing resources.¹

Lack of appropriate software has significantly impeded acceptance of multiprocessor systems for general computing. Difficulties arise both in porting existing applications and in developing new software systems for parallel machines. Designers generally perceive the construction of parallel software as a daunting and time-consuming task. Although considerable research has gone into specific aspects of parallel software design and implementation, the development process still lacks coherent methodologies, techniques, and tools.²

To provide such support, the Parse project (see sidebar) has been investigating software development issues covering a range of parallel applications. Parse itself is an object-based design methodology that incorporates design management strategies based on data and function encapsulation, hierarchical decomposition, and staged refinement.

Parse represents parallel software designs with a graphical notation called process graphs. After capturing a design’s important structural fea-
**Parse project**

Begun in 1991 as a collaborative effort between Australian and UK universities, the Parse (Parallel Software Engineering) project originally aimed to devise a design methodology for supporting the construction and validation of software incorporating explicit parallel behavior. Initial work focused on using Petri nets to design parallel systems. While providing a valuable formalism with which to describe systems, Petri nets cannot help with system decomposition and structuring for large, complex systems. Essentially, Petri nets are not a sufficiently high-level abstraction for initial design activities.

We therefore designed the process graph notation to overcome these problems, providing high-level architecture- and language-independent design abstractions. We also intended it to act as a starting point from which to systematically produce more detailed designs.

To validate the Parse approach, the Parse project has applied the methodology to a number of significant parallel software developments, including:

- a parallel logic language runtime-support system,
- a parallel database engine,
- a parallel transport protocol for high-speed networks, HTTPNET.

The HTTPNET design incorporated both Petri nets and CSP to provide behavioral specification, enabling the design to be validated as deadlock-free. These projects illustrated the power of the Parse approach, and illuminated weaknesses and omissions which we have had to confront.

We have also constructed a number of prototype support tools, including a process graph editor, and code generators to automatically produce skeletal C/PVM or Occam programs. We have also expanded the core process graph notation this article describes to facilitate the description of dynamically evolving process object structures. We are testing these tools and notation extensions through practical application.

Further, we are working with a number of organizations (in the UK), in a diverse range of industries, that are interested in the Parse approach. Current feedback is very positive. The approach seems intuitive and easy to adopt, and lets designers work at a higher level of abstraction than they are accustomed to using.

We are also exploring the use of object-oriented analysis and design techniques with Parse. This is a promising area.

The project's final focus is design validation. We are working with a team at Sheffield University to investigate how to constrain Parse designs to facilitate deadlock detection without having to explore the complete state space of the program's execution. This approach seems especially promising for complex, safety-critical systems. We are also collaborating with the University of Naples to explore the integration of Parse with the Epoca environment to provide behavioral analysis using stochastic Petri nets.

**References**


Summer 1995
Figure 1. Parse's four-stage methodology.

designs into formal notations to promote verification and into programming languages for easy implementation. Because comparable approaches lack such transformations and refinement techniques, our approach significantly advances the design of explicitly parallel software.

**Parse methodology**

Figure 1 illustrates the four stages of the methodology. The Parse logical design stage enforces an object-based modeling abstraction. This requires that all behavior and data be allocated to encapsulated concurrent process objects, which can only communicate and synchronize by message passing.

Parse strictly prohibits direct access to shared data, because this compromises design security. By introducing the potential for erroneous behavior in one process object, shared access would permit errors to spread without detection into other parts of the system. The methodology therefore protects shared data with process objects that manage access to the data and ensure its integrity. The process-graphs design notation actively supports this design model, providing predefined typed objects for representing data and function encapsulation.

In stage one, the process graph notation captures a design's structural properties by identifying concurrent process objects and their modes of interaction. The methodology provides extensive guidelines for decomposing a problem into appropriate process objects and communication paths. Process graph composition rules ensure the design's logical correctness. This design phase produces a structural description of the solution expressed in terms of encapsulated process objects that communicate by sending typed messages over logical communications paths.

In the second stage, a description of the dynamic behavior of the process objects supplements the process graphs. The methodology does not prescribe precisely how to define process-object behavior. Rather, it attaches design objects known as *path constructors* to process objects. These design objects indicate how to order multiple, potentially simultaneous input messages. Then, in most cases, the methodology can apply systematic transformation rules to derive the skeletal behavior for each process object, letting the designer add additional functionality and event ordering. We have demonstrated this approach using simple finite-state machines, Petri nets, and CSP. In later examples, we use a specialized behavioral specification language—selected for its generality and ease of understanding—that we devised to ease this process.

When a formal notation for behavioral specification is used, an optional third stage permits design analysis. Safety-critical systems or applications requiring high reliability require this level of analysis. Such analysis also provides an additional level of confidence in the predicted runtime behavior of the implementation. When using a technique such as Petri nets, designers can informally simulate the design to elicit further insights into its performance. We advocate specialized formal method tool support to facilitate verification. Further, there are practical limits to the size and complexity of designs that can be submitted for verification.

The fourth stage translates a complete logical design into a physical design by adding sufficient implementation-specific information to permit code generation. Designers must consider language and
architecture issues, and may introduce shared memory to increase performance if the implementation environment permits.

As Figure 1 indicates, iteration between and within stages is inevitable as the designer experiments with alternative designs. As the design iterates and moves from one stage to the next, it becomes progressively refined through the addition of increasing detail. This detail represents many facets or dimensions of the design; we refer to these as dimensions of refinement (Figure 2).

At each stage, designers should focus their attention on selected dimensions. Thus, in stage one, their attention focuses primarily on process object identification/decomposition and patterns of interprocess communication. In subsequent stages, their attention shifts to the dimensions of dynamic behavior specification, protocol description, and timed communication. This approach avoids overburdening the designer with too many aspects or dimensions to consider at any one time.

The Parse methodology is flexible. It recognizes that different systems may require consideration of some dimensions at an unusually early development stage. For example, a real-time design may require paying attention to timed communication even at the earliest development stage. Parse leaves such decisions to the system designer's discretion.

The graphics-processing design sidebar (next page) provides an example process graph, representing a parallel design for a simple graphics processing system. This graph is typical of many designs that incorporate independent task, or farming, parallelism. The following section illustrates features of the process graphs notation with fragments taken from this example. The example graph resulted from repeated refinement of the design; some of the fragments represent earlier versions of the design's objects.

![Figure 2. Dimensions of design refinement.](image)

![Figure 3. Classification of process graph entities.](image)

**Process graph notation**

As Figure 3 shows, the process graph notation contains a number of graphical entities. These relate to the processing and communication aspects of a parallel software design.

**PROCESS OBJECTS, COMMUNICATION PATHS, AND DECOMPOSITION**

A system designer's first task in constructing a new design is to represent its principal process objects and their interconnecting communication paths. The process graph notation provides three general classes of process objects: data server, function server, and control process (Figure 4a). The notation explicitly identifies the
Graphics processing design—an example process graph

The process graph in Figure A illustrates some of the most important features of the process graph notation. It represents a design for the parallel processing of graphics data. After an input sensor collects this data, the image generated by the processing is output on a display device. The design represents the input sensor and display device as external interface objects that serve as a source and sink of messages for the five top-level process objects in the design.

The design uses independent task, or farming, parallelism. This type of parallelism “farms out” independent packets of input data for separate processing by a collection of identical, replicated “worker” processes. Although the details may differ, this is a typical parallel design for this type of application. In the interests of clarity, the diagram omits the protocols on the communication paths.

The Capture process object converts raw sensory data into input data for subsequent processing, decomposing it into a pipeline of lower-level primitive processes (Figure 8c). Process object Pool maintains a store of appropriately sized data packages that it sends to the Master process object when requested. Master is a control process; it is the most complex process object in the design. It initiates the communication operations that “drive” the design’s main processing activity. Master farms out packages of work data to the replicated collection of Worker process objects.

There are 100 instances of the Worker function server. Each instance is an identical, but independent, concurrent process object that repeatedly receives work data packages and returns result messages containing some processed image data. Master continues to farm out work, forwarding the generated display data to Filter, until Pool is exhausted of all input data and a completed image is produced. Master decomposes into three lower-level primitive process objects: Builder, Sender, and Receiver (Figure 10). Builder is another control process, whereas Sender and Receiver are function servers that send and receive work data messages and return messages to and from the Worker process objects. Communication with Worker occurs on asynchronous paths using vectored output and input ports. The Filter data server receives display data from Master, prepares the data for output, and drives the display device.

Figure A. Sample process graph.

<table>
<thead>
<tr>
<th>Function server</th>
<th>Name</th>
<th>Data server</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Passive, without state</td>
<td></td>
<td>Passive, with state</td>
<td></td>
</tr>
<tr>
<td>Control process</td>
<td>Name</td>
<td>External interface</td>
<td>Name</td>
</tr>
<tr>
<td>Active, with state</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(a)</td>
<td>Name, protocol</td>
<td>Name, protocol</td>
<td></td>
</tr>
<tr>
<td>Synchronous</td>
<td>Asynchronous</td>
<td>Name, protocol</td>
<td></td>
</tr>
<tr>
<td>(b)</td>
<td>Bi-directional synchronous</td>
<td>Broadcast</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4. Processing entities (a), and communication path types (b).

process objects that the designer creates as belonging to one of these classes. Every process object inherits some predefined behavioral properties from its general class. These classes provide abstractions that help the system designer model the problem at hand.

Two behavioral properties characterize the system-supplied general classes of process objects: an active or passive role in system behavior, and the presence or absence of persistent state. Passive objects—function servers and data servers—generally do not initiate processing activity within a system. Instead they act as a server, processing requests sent to them from active client objects (control processes):
Data servers possess a state that persists from one service request to the next. This persistent state may be updated and inspected, and can influence the data server’s behavior at subsequent service requests.

Function servers cannot possess a persistent state: they simply encapsulate some useful service or functionality needed in the system.

Control processes initiate and coordinate processing in a system. They have a persistent visible state.

All process objects might exhibit internal concurrency, and consequently might be hierarchically decomposed into lower-level process objects.

By explicitly distinguishing between the roles played by the three process object classes in a software design, the process graph notation helps the designer produce reusable modules. Typically, control processes are application-specific. By contrast, function and data servers frequently represent a general-purpose processing task and therefore are candidates for subsequent reuse. The classification of process objects into these three categories further supports the correct dynamic behavior of the system. Differentiating between active and passive processes greatly reduces the potential for introducing hidden deadlocks into the software.

Differentiating between active and passive objects reduces the chance of introducing hidden deadlocks.

Synchronous. This blocking communication for both sending and receiving process objects is unbuffered, unidirectional, and synchronous. It behaves like the CSP output (!) and input (? ) operations.

Bidirectional. This blocking communication for both sending and receiving process objects is unbuffered and synchronous. It involves the transfer of two messages, one in each direction, along the same bidirectional path. It typically serves for communication between a process object acting as an active client (which sends the first request message) and a process object acting as a passive server (which returns the second reply message).

Asynchronous. This path type has a nonblocking send operation and a blocking receive operation. It is unidirectional and assumed to have a message buffer of infinite capacity for each receiver. The receiving process object always receives the messages in the same order in which the sending process object sent them. If there is no message in the path buffer, the receive operation halts until one arrives.

Broadcast. This type is an asynchronous path with a nonblocking send operation and a blocking receive operation. The path is unidirectional and assumed to have a message buffer of infinite capacity. The receiving process objects always receive the messages in the same order in which the sending process object sent them. Because the receiving process objects are not synchronized, they all receive the same ordering of messages but at possibly different times. If there is no message in the path buffer, the receive operation halts until one arrives. The send operation can also optionally multicast to a specified subset of process objects.

All communication paths in a process graph have a name and a protocol. The protocol defines the data type and structure of the messages. The methodology defines the protocols textually using a simple protocol description language.

All objects in a process graph have both a type-name and an instance-name identifier. The type name refers to a user-defined class of process object, while the instance name refers to a particular instance of the class. To illustrate, Figure 5a (next page) shows a fragment of an early version of the sample process graph. Worker is a class name, and w1 identifies an instance. The Worker class is a specialization of the system-supplied function-server general class. (Although paths are named, the figure omits protocols, as this aspect of the design is not yet fully refined.)

The Master process object in Figure 5a contains two lower-level concurrent process objects named Builder and Sender (Figure 5b). Process objects can decompose into an arbitrary number of constituent process objects of different classes. Design refinement progresses by repeatedly applying this decomposition at many levels, resulting in a hierarchically structured graph. The terminal process objects at the lowest level of graph decomposition are primitive process objects. These
objects are sequential program structures that can receive and send messages, but contain no concurrent control constructs.

When a design is complete, the designer can, if desired, remove all of the hierarchical structure, flattening the graph to a collection of functionally equivalent primitive process objects. For a nonprimitive, decomposable, user-defined class of process object such as Master, the designer defines the class through the decomposition of the Master process object. For a user-defined class of primitive process objects, the designer defines the class by describing the process object’s dynamic behavior or sequential program structure.

In addition to process objects, external interface objects may appear. These identify a point of contact between the parallel software design and the environment with which it interacts. This interaction may arise between various aspects of the environment such as hardware I/O devices or a host operating system. The external interface object is an abstraction that hides the interaction’s details. To the design’s process objects, the external interface objects are simply sources and sinks of messages of particular protocols connected by communication paths.

**REUSABLE DESIGN COMPONENTS AND INTERFACE PORTS**

Parse pays specific attention to reusability. The ability to specify user-defined classes based on the three systemsupplied classes lets us reuse design components, both within the same design framework and in future applications. This flexibility requires an appropriate indepen-
dent process-interface mechanism. To permit each instance to connect to different communication paths, we need an anonymous way for the class to reference these connections. The process graph notation supports these anonymous connections by providing interface ports for classes.

The definition of a user-defined class includes a description of the input and output ports belonging to that class. Each port has an identifier (for example, inp1, inp2), a direction (input or output), a protocol, and a path type (synch, async, bi-di, or brd). As a communication path is drawn, it connects the output port of a sending process object instance to the input port of a receiving process object instance. Inside a decomposed class, the constituent process objects may either connect on locally named communication paths to other constituent process objects, or they may make unnamed connections to the class interface ports.

Figure 5b shows the identifiers and protocols of the interface ports. Although interface ports are present in all process object instances, Parse normally suppresses their visual presentation in a process graph. To maintain design clarity, Parse normally only shows ports when a class is defined (at decomposition for a nonprimitive process object).

**PATH CONSTRUCTORS**

For parallel software, the dynamic interaction between processes must be understood and clearly represented. Stage two of Parse refines designs to account for the behavioral aspects of the software under development. To support this refinement, the process-graph notation lets the designer specify how a process object handles the receipt of messages on multiple incoming communication paths. We call these communication objects path constructors. The ability to represent input-response handling in process graphs supports the high-level modeling of the desired system behavior. By contrast, comparable approaches such as Path Pascal, where input-sequence specification occurs at the program-code level, do not.

The notation can attach a path constructor to a process object to associate a number of its incoming communication paths. There are three path constructor types: concurrent, deterministic, and nondeterministic. Each
describes a different response to incoming messages on the associated paths (Figure 6). Designers can apply constructors, with some constraints, to both primitive and decomposed process objects. When input paths are not associated with a constructor, the handling of messages on these paths is unspecified at this level: only inspection at a lower level of definition can determine such behavior.

A concurrent constructor specifies that messages arriving on its associated set of paths can all be processed at the same time. Such a constructor therefore cannot attach to a primitive sequential process object, because some concurrent primitive process objects must exist at a lower level of decomposition to process these messages. The presence of a concurrent constructor implies that there must be at least N concurrent primitive process objects inside the decomposition, where N is the number of input paths associated with the constructor (Figure 7a).

The nondeterministic constructor specifies that when messages arrive simultaneously on the associated input paths, the sequential program structure of a primitive process object at some point makes a nondeterministic choice between them. The deterministic constructor specifies that a prioritized choice is made between messages arriving at the same time (Figure 7b). Parse usually applies nondeterministic and deterministic constructors only to a primitive process object. When it applies one of these constructors to a decomposable process object, the message conveyed on the associated paths must have the same primitive process object as the path’s destination at a lower level of decomposition.

Because a constructor is a property of a user-defined class, all instances of a class use the same constructor identifiers. As a class property, a constructor is an association of one or more input ports. References to paths occur indirectly through the ports. A process object may have many constructors attached to it, and they may be of the same or different types, but each constructor must have an identifier label. When a process object has many constructors attached to it, there is no implied combination of constructors.

**REPLICATION AND PATH RESTRICTION**

Parallel software design heuristics frequently require introducing replicated, identical processing units. Consequently, process object replication is an important feature in Parse. Replication provides a concise, convenient way of describing regular process structures such as grids and pipelines. It also provides a way of generating unique identifiers for the process object instances in a regular structure by qualification with a subscript.

With Parse, designers can expand or decompose replicated process objects. Figure 8a shows an example of process object replication in both concise and expanded forms. In the concise form, the replicator appears in square brackets next to the name of the replicated process object. Communication paths joining replicated
instances have a default interpretation of path connections for those instances. In Figure 8a, a single unreplicated Master process object connects to a replicated Worker process object; when expanded, this implies a one-to-many pattern of path connection between the instances in the regular structure.

In the expansion, the path name is the same for all connections between instances, but each instance-instance connection is qualified with the subscripted sending and receiving instance. The expansion also shows a vectored output port on the Master process object, which makes the one-to-many connections to the instances of the replicated Worker process object. Normally, a process graph fragment containing replicated process objects would remain in its concise, unreplicated form. Figure 8a illustrates how using a concise form and default interpretations can hide a considerable amount of design detail from view. Decomposition of a replicated process object is no different from that of an nonreplicated process object.

Optional path restrictions can override the default interpretation of path connections between instances of replicated process objects. A path restriction identifies particular pairs of sending and receiving instances. The restriction constrains message communication to the identified instances. The restriction can identify instances using either literal subscripts or range variables.

Path restrictions are particularly useful when defining communication patterns between instances of a replicated process object. Figure 8b shows the concise description of a process object pipeline using path restrictions. The pipeline's middle portion consists of a replicated Section process object connected by Fill and Next paths (both of the same protocol). Every instance of Section has exactly one input and one output port, but for different instances they are connected differently. This is an example of alternative path connections.

The path restrictions on the Next path specify that the output port from Section connects to either the input port of Tail or the input port of another instance of Section. Each instance of Section must connect exclusively to one or the other. Figure 8c shows how these connections define a pipeline of expanded process object instances.
**Timed Communication**

To provide the software developer with a smooth progression from outline design to detailed, nearly implementation-ready design, the process graph notation relies heavily on default interpretations of entities in the design. The application of a refinement technique often manifests itself as an override of these default interpretations. Parse supports design management, suppressing detail until it is appropriate. The methodology recognizes that the development process changes for different types of parallel software. For instance, the timed-communication dimension of a design for hard, real-time control software is likely to be refined at an early stage.

Apart from the send operations on the asynchronous and broadcast paths, send and receive operations are generally blocking operations. Sometimes it is useful to override this default behavior and indicate that the communication operations on particular paths are timed. A timed communication specifies a time interval during which an operation must occur.

If a message is not sent or received during this interval, the operation cancels, releasing the process object from its communication block and allowing it to proceed with its execution. A time interval of zero is equivalent to a nonblocking communication operation. A blocking communication operation effectively has an infinite time interval: it never times out. A positive value denotes the time delay, in implementation-determined units, that the process must wait until the communication operation either succeeds or cancels (that is, times out). All time information is local to a process object: we do not assume the availability of a global time reference. Even if a global time is available in the implementation environment, objects in the logical design might not synchronize with it.

The process graph notation denotes timed communication by a small bar placed on the path arrow next to the timed port (Figure 9a). The value of the time interval appears immediately above the bar. If no value is present, the notation assumes a default value of zero (that is, nonblocking communication). If a value is present, the notation interprets it as a constant value that applies to all process object instances. If an asterisk appears above the bar, each instance of a class might have a different value.

Figure 9 shows the permitted combinations of timed communications. Because send operations on asynchronous paths and broadcast paths assume an infinite message buffer capacity, they are nonblocking and cannot be timed. However, the process graph notation does let users specify a fixed-size buffer capacity for these paths to override this assumption, thereby permitting a timed send operation (Figure 9b). If a process graph specifies fixed-size buffer capacities, the timed send operation would normally be introduced at the physical design stage, as it is essentially an implementation consideration.

**Primitive Process Specification**

Designers specify the dynamic behavior of primitive process objects using a behavioral specification language. This language contains constructs for describing sequential program structures and includes sequences, iterations, selections, and guarded selections. Also, it contains primitive send and receive operations for each of the four communication path types.

To illustrate the language, we provide partial descriptions of the behavior of two primitive process objects taken from the example process graph: the Pool data server and the Builder control process. Figure 7 already showed some aspects of Pool's behavior regarding the deterministic constructor placed on its input paths.
Builder appears in a refined version of the decomposed Master control process (Figure 10). This refined Master process object now contains a Receiver process object to collect the result messages from the Worker process objects.

The descriptions of Pool and Builder in Figure 11 illustrate the control constructs and primitive communication operations of the behavioral specification language, but the figure omits other processing operations. We cannot fully describe the behavioral specification language here, but further information is available elsewhere. 14

Primitive operations do not refer directly to communication paths. Rather, port identifiers indirectly reference these paths. Generally, sending operations execute on output ports, and receiving operations execute on input ports. Bidirectional communication operations are an exception, because they perform both send and receive operations on the same port.

In Builder, a single bidirectional, request operation uses the outp1 output port for both sending a request token and receiving a reply message. In Pool, the two corresponding accept and reply operations use the same inp2 input port for receiving the request token and sending the reply message. The deterministic constructor applied to Pool (Figure 7b) appears in the behavioral description as the Priority Select construct.

The two input guards in this construct are the receive and accept operations. The receive textually appears before the accept, thus giving priority to messages arriving on the Supply path via port inp1 over messages arriving on path Feed via port inp2. As well as having port and message parameters, the communication operations have a second subscript parameter for
vectedored ports (zero denotes nonvectedored), and a third parameter that indicates whether communication is timed (negative value denotes not timed).

The behavioral descriptions of the Pool and Builder process objects are typical of many data servers and control processes. Generally, data servers and function servers have simple control structures, whereas control processes tend to have more complex control structures.

Designers can often reuse data servers and function servers from one system to another, whereas control processes are generally application-specific. It is possible to derive skeletal behavioral descriptions for primitive process objects from the process graph. The designer must complete these skeletal, or partial, descriptions before code is generated. The completed behavioral descriptions can translate into programming languages such as Ada, PVM/MPI, Parallel C, or Occam. In principle, this translation can be automated, and we are currently testing tools that generate Occam and C/PVM.

Parse takes a major step toward providing an industrial-strength design methodology aimed at general-purpose parallel and distributed systems. Designing systems at an abstract level, and systematically transforming designs into target languages and runtime environments, provides a high degree of design portability. In this manner, using Parse should facilitate the efficient, cost-effective construction of reliable and retargetable software systems.

REFERENCES

Ian Gorton is a project manager in the Division of Information Technology at CSIRO. His research interests include software engineering for parallel systems and exploring the application of parallelism in new applications. He received his BSc and PhD in computer science from Sheffield Hallam University in 1985 and 1989. He is a member of the IEEE. He can be reached at CSIRO Division of Information Technology, Locked Bag 17, North Ryde, New South Wales 2113, Australia; iango@dyd.dit.csiro.au.

Jonathan P. Gray is a lecturer in computer science at the University of Wollongong. His research interests are software engineering for parallel systems and CASE technology. He received his MSc and PhD in computer science from Sheffield Hallam University in 1987 and 1991. He can be reached at the Computer Science Department, University of Wollongong, New South Wales 2522, Australia; jon@cs.uow.edu.au.

Innes Jelly is a senior research fellow in the Computing Research Centre at Sheffield Hallam University. She has undertaken research in parallel processing and has recently focused on parallel software engineering techniques and tools. Her research interests also include performance evaluation of large database systems and hardware/software codesign. She has been closely involved in technology transfer to Eastern European countries, and has teaching and research connections with a number of European countries. She received an MSc in 1986 and a PhD in 1990 from Sheffield Hallam University. She is a member of the IEEE. She can be contacted at the Computing Research Centre, Sheffield Hallam University, 100 Napier St., Sheffield 811, UK; ijelly@shu.ac.uk.

Summer 1995