Proceedings of the 1999 Winter Simulation Conference
P. A. Farrington, H. B. Nembhard, D. T. Sturrock, and G. W. Evans, eds.

COMPONENT-BASED SIMULATION ON THE WEB?

Michael Pidd
Noelia Oses
Roger J. Brooks

Department of Management Science
Lancaster University,
Lancaster LA1 4YX, U.K.

ABSTRACT

Various forms of distributed simulation are possible over the world-wide web, including simple multiple replications of the same model, client-server architectures for one or more simultaneously running models and the distributed operation of one or more linked models. Like all web-based operations, these simulations are slow due to current bandwidth limitations, but that could change in the next few years. Languages such as Java make this distributed work possible within standard web-browsers such as Internet Explorer and Netscape, though security considerations mean that this is not always straightforward. Component-based simulation stems from the ideas of object-orientation, which enable libraries of simulation based components to be developed for re-use. The development of the world-wide-web means that distributed component, discrete simulation libraries in Java are now feasible. This paper reviews some of these developments and considers requirements for such distributed libraries, drawing on our experience at Lancaster.

1 INTRODUCTION

1.1 Distributed Computing

It is a commonplace that the world of computing has changed from one in which most work was done on isolated computers to one in which individual machines can be seen as processing nodes on a world-wide network. Of course, no one knows where this shift to networked computing will lead in the longer term, but its impact is already obvious in the explosive growth of the Internet. In parallel with this, the massive growth in the use of cellular phones and the increasing sophistication of peripheral devices, suggests that global networking will have a major impact on all of our lives. The combination of networked computing and mobile devices suggests that intelligent mobile devices, some of them recognisable as computers in today's terms, may become commonplace before too long. It also means that the ability of computer programs to communicate and co-operate across space and time may become a fundamental requirement in future, rather than an esoteric domain of expertise that requires enormous and specialised computing power.

As well as the growth of networked computing, two other threads lie behind the development of distributed component-based libraries for discrete simulation. The first is that component-based work, primarily based on object-oriented approaches is known to be feasible and underlies many current commercial software products outside the domain of discrete simulation. Object-oriented approaches, properly applied, enable components to be represented as classes that communicate by message passing through defined interfaces. Thus, in theory at least, some of the ideas suggested by people such as Zeigler (1976, 1984, 1990) for modelling in the large can now be implemented using commonly available programming software. It is also possible, as is evident in approaches such as the HLA (U.S. DoD, 1998a, 1998b, 1998c) to take existing computer programs that are not object-oriented and to wrap them inside an object-oriented shell that provides a defined interface across which messages may be passed to other such wrapped programs.

1.2 Increased Complexity

The successful use of discrete simulation for relatively small applications has emboldened developers to consider much larger and more complex applications. This is true of many application sectors such as air transportation, road traffic systems management, telecommunications network design and, perhaps especially, the military sphere. With such large scale applications it would be perverse to ignore some of the lessons learned in software engineering for the development of large scale systems. One such lesson is that well-tested and re-usable code is preferable to ad-hoc programs that will be used just once (Sommerville 1995).

That is, an approach in which programs are developed (in part at least) from well-tested and re-usable components provides a sensible way to develop large scale applications. Using components in this way should greatly decrease the time needed for program development and ought to lead to simulation programs that are rather more robust.

Putting these aspects together suggests that software developers should be working on components with well-defined interfaces that can be assembled, with other special-purpose code, for particular applications. Further, these components can be distributed across the Internet and accessed as required. This implies that there will be directory services which will enable application developers to find the components that they may need. In some instances these components will be assembled and run on a single computer, in others they will be run as communicating applications across a network.

2 SOFTWARE COMPONENTS – LEGO BRICKS

2.1 Software Components in Software Engineering

It is important to be clear about what is meant by a software component and by component-based software development. The idea of software components has been a major theme of software engineering for some time and general definitions can be found in this literature. One such is, "a unit of composition with contractually specified interfaces and explicit context dependencies only" (Szyperski 1998). That is, components are independent of one another and communicate only by defined means. In one sense, of course, simple components have been used in computer programming for many years in all languages that permit the definition of functions with their own local variables. A well defined function for, say, generating random numbers entirely meets Syzperski's definition as quoted above.

In a similar vein, Kara (1996) discusses component-based development as "the process of building systems by combining and integrating pre-tested and pre-engineered software objects… Components are thus encapsulated software objects, providing some type of known service, that can be used in combination with other components to build systems". This adds the notion that, to be useful, software components must have clearly defined functionality that has been properly and thoroughly tested.

Batory and O'Malley (1992) discuss software components in the light of other concepts in computer programming. They write that, "A type is a set of values. An Abstract Data Type (ADT) is a type plus operations on the values of the type. A class is an ADT that belongs to an inheritance lattice. A component is a closely-knit cluster of classes that act as a unit". This implies that a component can consist of a set of classes which, with Kara's views above, have a well defined function. If object orientation is a step beyond the use of ADTs, Miller et al. (1998) argue that, for simulation, components are a step beyond object orientation.

Taylor (1998) points out the close link between object orientation and the idea of component-based development, though he argues that software components need not be based upon models of inheritance. Malak (1998) makes a similar observation when writing about the HLA. However, it is hard to reconcile this view with the careful definition of Batory & O'Malley (op cit) since, if any class stems from an abstract data type, then this can only be achieved via inheritance.

2.2 Software Components in Computer Simulation

This conference and others have seen a number of papers in recent years that discuss the role of software components in discrete simulation. Fukunari et al. (1998) introduce a Java-based methodology to ease web-based simulation development: component-based development, which permits use of drag and drop development, and so can reduce development time significantly since it requires no coding. They use Java and Java's component technology, JavaBeans, which enhance reusability, and they define 'component' as "a self-contained element of software that can be controlled dynamically and assembled to form applications." They point out that component-based simulation has many advantages, such as reusability of components and simplicity of development due to visual programming. Writing about continuous simulation models, Küçük and Zobel (1998) argue that "…submodels can help in the simulation of complex models by providing a natural basis for splitting the computation into manageable pieces. Each of these software entities corresponds to a subsystem simulator and will be referred to as a component."

Zeigler's work on DEVS (Zeigler 1976, 1984, 1990) is well-known and has been successfully implemented in a number of applications. In DEVS, a large model is built from a series of independent components known as atomic models that communicate only via defined set of input and output ports. Pidd and Bayer Castro (1998) discuss how DEVS can be made more object oriented so as to bring it more into line with current software developments. In DEVS, to "componentise" a coupled model means to encapsulate it in a form that enables it to be treated in the same way as an atomic model (Zeigler and Sarjoughian, 1999). It should be noted, however, that in DEVS an atomic model and any composite models that are closed under coupling really are simulation models, in their own right, and are not just general purpose components that may become part of another model.

2.3 A Definition of a Software Component for Discrete Simulation

The preceding discussion suggests that, to be regarded as a software component to be used in a discrete simulation, a program should meet the following criteria.

- Its functionality should be entirely defined: that is, its stated function should be fully implemented and there should be no possibility of unintended side effects. This implies encapsulation in which all variables of the components are defined as local or private.
- All communication with any other program fragments or components should be through a fully defined interface. This implies that computation will be organised via message passing as in conventional object orientation.
- The interface should be wholly unambiguous: that is, it should be entirely clear what inputs are required and what outputs are produced by the component and this must include error handling should incorrect inputs be provided in use. The interface definition will include the messages to which the component can respond and that it may itself produce.
- Its functionality and interface should be defined in a 'directory service' to which other components have access and from which components may be selected and used.
- Such components will, in general, need to abide by whatever rules are defined in an overall architecture that defines in detail how components will be linked and used.

3 SOME EXISTING COMPONENT-BASED ARCHITECTURES FOR DISCRETE SIMULATION

This section discusses three approaches to the development of discrete simulation models from componentware. The examples are chosen to illustrate approaches that range from systems in which any language is used (DEVS), an architecture intended to enhance the interoperability of models written in almost any language (the HLA), one in which a general purpose language is used (Silk, using Java) and a dedicated simulation environment and scripting language (VSE).

3.1 DEVS

The Discrete Event Systems Specification (DEVS) was proposed by Zeigler (1976, 1984) and has since been implemented in a number of systems using a range of languages; for example in C++ (Praehofer 1996; Cho and Cho 1997a), Java (Cho and Cho 1997b) and in the LISP-based framework of DEVS-Scheme (Zeigler 1987, 1990). The fundamental component of DEVS is known as an atomic model, which can be defined in set-theoretic terms as follows.

$$AM = <S; X; Y; \delta_{ext}; \delta_{nt}; \lambda; ta>$$

where S, X, Y are sets containing the sequential states, input and output events; δ_{ext} and δ_{int} are the external and internal transition functions; λ is the output function and *ta* is the time advance function of the model.

Zeigler insists that a modular model must be regarded as a 'black box' which receives messages (external events) through its input port(s) and which sends messages (information about changes in its internal state) through its output port(s). The description of the 'black box' must be such that it is contextually independent. That is, its internal description must make no assumptions whatsoever about the origin of messages on its input port(s) nor about the destination of messages from its output port(s). (Pidd and Bayer Castro 1998). Thus a DEVS atomic model is a wholly self-contained simulation model in its own right.

In DEVS, larger models are built by coupling atomic models in a coupling scheme that links the input ports of atomic models to the output ports of other atomic models. If this coupling is done correctly, the resulting coupled model is regarded as closed under coupling, which means that it can be treated as if it were an atomic model with its own DEVS specification. Thus, to build a model using DEVS, the modeller must define atomic models and their associated simulators and then link these in coupling schemes, such that the resulting overall model, built from the components, is itself a DEVS model.

As is made clear in Pidd and Bayer Castro (op cit) and Kim and Kim (1998) an important issue to be faced when using DEVS is the complexity of the coupling that may result since it may be necessary to couple each atomic model with every other such model. If the coupling is direct, which it should be to satisfy the strict requirements of DEVS, then this results in very messy conceptual wiring between the input and output ports and a resulting simulation that may run very slowly. To improve things, Kim and Kim (op cit) suggest that atomic models could communicate via a bus rather than directly with one another, which should result is much simpler conceptual wiring and could enable DEVS to be used within the HLA (see below). Pidd and Bayer Castro (op cit) suggest a similar concept, known as selective external modularity.

3.2 HLA

The High Level Architecture (HLA) was developed to enable the inter-operability and re-use of simulations and

real-time activity. A definition of its main requirements can be found at http://hla.dmso.mil/ and it is not restricted to simulation models, whether discrete or continuous. It specifies how dynamic models and real-time activities may communicate with one another even though they may not have originally been designed to do so. It may be used to provide a language and computer independent architecture for developing discrete simulation models. In the parlance of the HLA, component models or activities are referred to as federates and the job of the HLA is to ensure that these federates interact safely and without breaching causality conditions. A collection of federates linked together during a run is referred to as a federation.

Each federate must provide an interface through which messages will be passed and received. Thus, to run under the HLA, a new model must have an interface that meets the HLA interface specification. Models and activities that were not originally developed to run under the HLA must be wrapped inside an interface specially written to comply with the same specification. Federates do not communicate directly with one another but do so via the Run-Time Infrastructure (RTI) which acts like an operating system and communication bus. Thus federates communicate through their interfaces via the RTI.

There are three main parts to the HLA definition, as follows.

1. The HLA rules: these specify the responsibilities that are devolved to the federates and which must be implemented if the federate is to be HLA compliant.
2. The HLA interface specification: this defines the standard services and interfaces to be used by the federates in order to support efficient information exchange when participating in a distributed federation execution. It includes 6 classes of service offered by the RTI to federates and vice versa. These cover aspects such as time management, data distribution management and object management.
3. The HLA object model template (OMT): which defines how each federate and each federation must document its object model. The OMT prescribes the format and syntax for recording the information in HLA object models, to include objects, attributes, interactions, and parameters, but it does not define the specific data (e.g., vehicles, unit types) that will appear in the object models. The HLA requires that federations and individual federates be described by an object model which identifies the data exchanged at runtime in order to achieve federation objectives.

HLA federates can either be purpose written so that they comply with the federate interface specification or can be produced by taking an existing model and wrapping it. In effect, the wrapping takes an existing model and restricts input and output from that model to that defined in a purpose written interface that does comply with that required by the HLA.

It is important to realise that any HLA federates that are simulation models are wholly self-contained except for the communication permitted across their HLA compliant interfaces. Thus, each such simulation federate will have its own simulation clock and the simulation federates proceed asynchronously. Time management within a simulation federate may be optimistic or pessimistic, and in the latter case a message retraction system is employed in the time management provided by the RTI.

3.3 VSE

The Visual Simulation Environment (VSE) (Balci et al. 1997, 1998a, 1998b) is an architecture for discrete-event model simulation. It is fully object-oriented and thus supports inheritance, instantiation, message passing, encapsulation and polymorphism. VSE models are divided in static and dynamic parts. The static architecture, which generally represents the structure of a model, is composed of hierarchically decomposed components. The dynamic parts consists of dynamic objects, which are entities that move from one point to another of the model. VSE's own components are made available via the VSLibrary, all VSE models are developed by subclassing (extending) classes in VSLibrary, which has VSObject as its root class. Anyone wishing to add new components to a library must do in the VSE scripting language, which is designed to support discrete simulation. A VSE Editor is provided to ease the process of component production.

In its execution, VSE employs a TCP/IP stack to enable communication between its simulation components as the model runs. This means that it can execute on a single computer, on a multi-processor compute-box or across a network of individual machines. It was originally written for the NextStep™ environment and requires appropriate emulation software to be pre-installed if it is to run under Windows NT4.

Balci et al. (1997) define a component as "a part of the model architecture or a part of the structure of a dynamic object". Thus components may be models of the behaviour of an entity class (a dynamic object) or may be concerned with the interaction of those classes (the model architecture). Components may also be viewed as shallow or deep. A shallow component cannot be further decomposed into underlying components. A deep component may be decomposed into other sub-components, which themselves may be shallow or deep.

Deep components may have graphical attributes and layouts amongst their own constituent components.

At least four approaches to model development in VSE can be envisaged. The first is by the reuse of model components from a library with no programming, which VSE supports by a drag and drop approach. Of course, this will often be impossible since the library may be incomplete for the task at hand, and the second approaches build on this by some programming, to extend, modify or add new methods to a component by subclassing). A third way is to re-use model components created during the model development process (components created in the Templates window). Finally a user may build a VSE model with no reuse where each model component is engineered from scratch.

VSE is designed to support the development of reusable components and regards a library (a set of reusable model components) as a form of component in itself. Reusable components are created in the Templates window, and can be created, manipulated, and destroyed during model execution. Thus, a modeller could make use of VSE components developed by others to build a model for a particular application.

3.4 Silk

Silk™ (Healy and Kilgore 1998) is a general purpose discrete simulation system, written in Java, and based around a process-interaction approach. Being written in Java, it is wholly object oriented, and therefore well-placed for component-based development. There is no Silk simulation language as such, programs are written in Java and make use of software components written in Java that meet the standards expected of Java beans. The *java.beans* package was first defined in the API of Java 1.1 and it provides a framework for re-usable software components. Its classes and interfaces can be used at three levels (Flanagan 1998).

1. To create application builder tools (for example Silk) that can be used by programmers and non-programmers to develop applications out of individual Java beans.
2. To develop more Java beans for use in such application builders.
3. In the development of Java applications, using Java beans, but without using an application builder tool.

The JavaBeans specification defines a bean as “a re-usable software components that can be manipulated visually in a builder tool”, which could cover simple uses of the Java AWT package or complex uses of embedded components such as a graphics editor. The interface for a Java bean declares properties, events and methods. A property is usually a private state variable of some kind whose value can be queried or modified through *get* and *set* methods. A bean may also generate *events* much as can other Java components. The methods are public and are, therefore, usable by other components.

Since Java provides support for multi-threaded execution, Silk is able to manage its process interaction in this way. Simulation entities in an application are instances of entity classes that descend from the *Entity* base class and each such instance can run as a separate thread. Each application entity class must have its own private data and methods to implement the unique behaviour of the entity it represents. In addition, each entity class needs a *process* method to start the process thread in the simulation. A simulation executive thread, which must include a *run* method, acts as a time manager and also co-ordinates the resumption of suspended threads – which is crucial to avoid deadlock between suspended threads. There remains, however, as in all process interaction approaches, the basic issue of process/thread management which grows increasingly important and increasingly difficult as interaction between threads increases.

Given the basic Silk architecture, it should be possible for third party developers to write entity and utility classes that conform to its requirements in simulation terms and as Java beans. Thus, it should be possible for a library of such beans to be developed for a range of simulation applications. There remains, however, the need (as identified above) to ensure that thread management and prioritisation are carefully implemented, otherwise causation errors and deadlock are possible, even in single-processor simulations.

4 COMPONENT-BASED SIMULATION ON THE WEB?

Like others, for example, Joines et al. (1992), Buss and Stork (1996), Fukunari et al. (1998), Unger (1986), Bezvizin (1987) and Fishwick (1992), we have developed at Lancaster a number of discrete simulation systems based on object oriented languages such as C++ and Java (Pidd 1995; Pidd and Cassel 1998, 1999). Our most recent such work has been with Java in which we have developed discrete simulation systems, using the three phase approach of Tocher (1963) and this has been extended into Java Beans.. The systems permit simulations to be run in several ways. Options include: a single computer (rather like Silk),; as multiple replications of the same model on a number of networked computers; as a client/server system in which the server provides simulation services to multiple three phase clients running different applications on other computers, and as a fully distributed system using conservative protocols.

None of our systems is intended for commercial use and all could be much improved to make them easier to use and faster in execution. However, they do demonstrate proof of concept by showing that object oriented approaches to discrete simulation form a useful basis for component-based development and that Java (or something like it) enables the components to be distributed across the world-wide-web. For such distributed components to be successfully implemented requires an architecture that defines how the components will interact as well as defining the components themselves. It is possible that the HLA may form the basis for such distributed components when there is a need for fully-fledged models to inter-operate. However, its requirements are rather cumbersome for simulations in which the need is to link smaller components from across the web into a single application. Thus, the main challenge to be faced in achieving distributed component-based simulation on the web is the development of an architecture that will enable this.

ACKNOWLEDGEMENTS

Noelia Oses is supported by the Departamento de Educacion, Universidades e Investigacion, of the Government of the Basque Country, Spain.

REFERENCES

Balci O., Bertelrud A.I., Esterbrook C.M. and Nance R.E. (1997) Developing a library of reusable model components by using the Visual Simulation Environment. In *Proceedings of the 1997 Summer Computer Simulation Conference* Institute of Electrical and Elecltronics Engineering, Piscataway, New Jersey.

Balci O., Bertelrud A.I., Esterbrook C.M. and Nance R.E. (1998a) Visual Simulation Environment. In *Proceedings of the 1998 Winter Simulation Conference*. ed D.J. Medeiros, E. Watson, J. Carson, and M.S. Manivannan. Institute of Electrical and Elecltronics Engineering, Piscataway, New Jersey.

Balci O., Ulusaraç C., Shah P. and Fox E.A. (1998b). A library of reusable model components for visual simulation of the NCSTRL system. In *Proceedings of the 1998 Winter Simulation Conference*. ed. D.J. Medeiros, E. Watson, J. Carson, and M.S. Manivannan. Institute of Electrical and Elecltronics Engineering, Piscataway, New Jersey.

Batory D.and O'Malley S. (1992) The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology*. Vol. 1, No. 4, October 1992, Pages 355-398.

Bezvizin J. (1987) Some experiments in object-oriented simulation. OOPSLA.

Buss A. H. and, Stork K.A. (1996) Discrete event simulation on the world wide web using Java. In *Proceedings of the 1996 Winter Simulation Conference*. Ed. J.M. Charnes, D.J. Morrice, D.T. Brunner, and J.J. Swain. Institute of Electrical and Elecltronics Engineering, Piscataway, New Jersey.

Cho H.J. and Cho Y.K. (1997a) DEVS - C++ Reference Guide, URL:http://cactus.ece.arizona.edu/~ykcho/doc/devs_v41.doc

Cho H.J. and Cho Y.K. (1997b) DEVS - Java Reference Guide, URL:http://cactus.ece.arizona.edu/~ykcho/doc/devs_java.doc

Fishwick P.A. (1992) SIMPACK: Getting started with simulation programming in C and C++. In *Proceedings of the 1992 Winter Simulation Conference*, ed. J.J. Swain, D. Goldsman, R.C. Crain, and J.R. Wilson. Institute of Electrical and Elecltronics Engineering, Piscataway, New Jersey.

Flanagan D. (1998) *Java in a nutshell: a desktop quick reference*. O'Reilly & Associates, Sebastopol, CA.

Fukunari M., Chi Y. and Wolfe P.M. (1998) JavaBean-based simulation with a decision making bean. In *Proceedings of the 1998 Winter Simulation Conference*. ed D.J. Medeiros, E.F. Watson, J.S. Carson and M.S. Manivannan. Institute of Electrical and Elecltronics Engineering, Piscataway, New Jersey.

Healy K.J. and, Kilgore R.A. (1998). Introduction to Silk™ and Java-based simulation. In *Proceedings of the 1998 Winter Simulation Conference*. ed D.J. Medeiros, E.F. Watson, J.S. Carson and M.S. Manivannan Institute of Electrical and Elecltronics Engineering, Piscataway, New Jersey.

Joines J.A., Powell K.A. and Roberts S.D. (1992) Object-oriented modelling and simulation in C++. In *Proceedings of the 1992 Winter Simulation Conference*, ed. J.J. Swain, D. Goldsman, R.C. Crain, and J.R. Wilson. Institute of Electrical and Elecltronics Engineering, Piscataway, New Jersey.

Kara D. (1996) Components defined. *Application Development Trends*, June 1996.

Kim Y.K. and Kim T.G. (1998) A heterogeneous simulation framework based on the DEVS bus and the high level architecture. In *Proceedings of the 1998 Winter Simulation Conference*, ed D.J. Medeiros, E. Watson, J. Carson, and M.S. Manivannan. Institute of Electrical and Elecltronics Engineering, Piscataway, New Jersey

Kucuk B. and Zobel R.N. (1998). Component-oriented continuous-time simulation. In the 12th European Simulation Multi-conference. Manchester, UK. June 16-19, 1998.

Malak M. (1998) A software component framework for HLA tools. 1998 Fall Simulation Interoperability Workshop. Sept 14-18.

Miller J.A., Yogfe G. and Junxiu T. (1998) Component – based simulation environments: JSIM as a case study using Java Beans. In *Proceedings of the 1998 Winter Simulation Conference*, ed. D.J. Medeiros, E.F. Watson, J.S. Carson and M.S. Manivannan. Institute of Electrical and Elecltronics Engineering, Piscataway, New Jersey

Pidd M. (1995) Object orientation, discrete simulation and the three-phase approach. *Jnl Opl Res Soc*., 46, pp362-374

Pidd M. and Cassel R.A. (1998) Three phase simulation in Java. In *Proceedings of the 1998 Winter Simulation Conference*, ed D.J. Medeiros, E.F. Watson, J.S. Carson and M.S. Manivannan. Institute of Electrical and Elecltronics Engineering, Piscataway, New Jersey.

Pidd M. and Cassel R.A. (1999) Using Java to develop discrete event simulations. Accepted for publication in *Jnl Opl Res Soc*.

Pidd M. and Bayer Castro (1998) Hierarchical modular modelling in discrete simulation. In *Proceedings of the 1998 Winter Simulation Conference,* ed D.J. Medeiros, E.F. Watson, J.S. Carson and M.S. Manivannan. Institute of Electrical and Elecltronics Engineering, Piscataway, New Jersey.

Praehofer H. (1996) An environment for DEVS-based multi-formalism modeling and simulation in C++. In *Proceedings of the Sixth Annual Conference on AI, Simulation and Planning in High Autonomy Systems*, La Jolla, CA, March 25-27.

Sommerville I. (1995) *Software Engineering* (5th Ed). Addison-Wesley, Reading, Mass.

Szyperski C (1998) *Component software – Beyond Object-Oriented programming*. Addison-Wesley, Reading, Mass.

Taylor D. (1998) Are objects obsolete? *Component Strategies*. July, 1998.

Tocher K.D. (1963) *The Art of Simulation*. English Univ Press, London

U.S. Department of Defense (1998a) High Level Architecture Interface Specification, Version 1.3 (Draft 9) February. http://hla.dmso.mil/hla/tech/ifspec/if1-3d9b.doc

U.S. Department of Defense (1998b) High Level Architecture Object Model Template, Version 1.3, February. http://hla.dmso.mil/hla/tech/omtspec/omt1-3d4.doc

U.S. Department of Defense (1998c) High Level Architecture Rules, Version 1.3 (Draft 2) February. http://hla.dmso.mil/hla/tech/rules/rules1-3d2b.doc

Unger B.W. (1986) Object-oriented simulation - Ada, C++, Simula. In *Proceedings of the 1986 Winter Simulation Conference*, ed James O. Henriksen, Stephen D. Roberts, James R. Wilson. Institute of Electrical and Elecltronics Engineering, Piscataway, New Jersey.

Zeigler B.P. (1976) *Theory of Modelling and Simulation*. John Wiley & Sons Inc, NY.

Zeigler B.P. (1984) *Multi-facetted Modelling and Discrete Event Simulation*. Academic Press, NY.

Zeigler B.P. (1987) Hierarchical, Modular Discrete-Event Modelling in an Object-Oriented Environment. *Simulation*, 50, pp 219-230.

Zeigler B.P. (1990) *Object Oriented Simulation with Hierarchical, Modular Models - Intelligent Agents and Endomorphic Systems*, Academic Press, Boston.

Zeigler B.P., and Sarjoughian H. (1999) Support for Hierarchical Modular Component-based Model Construction in DEVS/HLA. 1999 Spring Simulation Interoperability Workshop. March 14-19, 1999.

AUTHOR BIOGRAPHIES

MIKE PIDD is Head of the Department of Management Science and Professor of Management Science at Lancaster University in the UK. He is best known for two books, *Computer Simulation in Management Science* (now in its fourth edition) and *Tools for Thinking: Modelling in Management Science*; both published by John Wiley. He is active in researching simulation methods related to modularity and object orientation. He acts as consultant to a number of private and public sector organisations in Europe.

NOELIA OSES is a PhD student in the Department of Management Science at Lancaster University. She completed her first degree in mathematics from the University of the Basque Country, Bilbao, Spain. Her research focuses on component-based approaches to discrete simulation.

ROGER BROOKS is a lecturer in the Department of Management Science at Lancaster University, which he joined about a year ago after developing simulation models of crop yields in large-scale climate change, working at the University of Bristol. He has a mathematics degree from the University of Oxford, a PhD from the University of Birmingham and is also a chartered accountant.