in 1st Workshop on Distributed Object Trends (WTOD 2001). pages 157–169. Nov. 2011

Current Trends in Rigorous Distributed Object Based Software Development

Carlos Henrique Cabral Duarte
carlos.duarte@computer.org

Banco Nacional de Desenvolvimento Econômico e Social
Av. Chile 100, Centro, Rio de Janeiro, RJ, Brazil, 20001-970

Universidade Estácio de Sá
Rua do Bispo 83, Rio Comprido, Rio de Janeiro, RJ, Brazil, 20261-902

Abstract

We present in this paper a brief survey of the current research and development trends related to rigorous distributed object based software development. The topics covered here range from programming language design and implementation to software development methods, techniques and tools.

Keywords: Object-based Approach, Distributed Systems, Software Engineering.

Resumo

Este trabalho apresenta um breve apanhado das perspectivas atuais em pesquisa e desenvolvimento sobre o desenvolvimento rigoroso de *software* baseado em objetos distribuídos. Os tópicos aqui abordados vão desde o desenho e a implementação de linguagens de programação até métodos, técnicas e ferramentas de desenvolvimento de *software*.

1 Introduction

Computing has evolved at such an amazing speed over the past few years that the technologies available today could barely be foreseen a decade ago. It is astonishing to realise that we already have massive processing power at each computing terminal due to the continuing development of microprocessor design and manufacture. With the support of networking solutions such as IP over optical networks [31], comprising not only protocols but also equipment, it is now possible to develop distributed computing infrastructures offering virtually unlimited bandwidth. In addition, the new computing devices which are about to reach the marketplace will offer native ubiquity support, maybe complying with Wireless IP [29], for instance. This new technological reality makes it possible to satisfy currently unusual requirements such as those having to do with extending the other human senses than sight and hearing: touch, taste and smell.

So far, software engineers have been competent enough to deliver computing systems satisfying most user requirements while taking advantage of the existing technology. Facing an exponential increase in the number of required technological design decisions and obliged to deploy in less time reduced cost solutions for problems of ever higher complexity and size, it is hardly guaranteed that all their current practices will remain appropriate for a long time. So, which are the software engineering methods, techniques and tools that will allow them to continue performing a good job?

One central issue related to this question is productivity. Adopting controlled, predictable and scalable practices, it becomes less challenging to deal with many distinct technologies, perform corresponding design decisions and treat the size of specific problems. The unexplored complexity remains as the real puzzle to be solved in each case. Object based approaches have been adopted with success to treat these requirements.

Another issue that appears to be related to the question above is reliability. Dealing with environments of greater complexity and heterogeneity, it is more important than ever to ensure that each software system presents a controlled and predictable behaviour. For that to be feasible, the engineer not only has to provide beforehand a specification of system and perhaps environment stating some of their properties but must also guarantee that these properties are preserved throughout the development process. A naive approach to this problem is not recommended, however, as some software development flaws causing substantial financial losses have shown, such as the explosion of an Ariane rocket in 1996 even with the replication of all systems on board [27]. Many software development methods, techniques and tools that are systematic or rigorous (or even formal) have been proposed to enable manual or automated verification of such properties.

Distribution is yet another issue that appears to be relevant here, either as a user requirement or as an implementation decision to increase reliability. Software components can be distributed over computer networks not only to ensure continuous availability of a system in case of failures but also because it may be necessary for the system to make its functionality available in many geographic locations, perhaps behaving differently at each site precisely due to its distinct location. Many distributed system models have been proposed to represent these aspects.

We address in the present paper the current research and development trends related to rigorous distributed object based software development. After presenting a few background notions, we begin to discuss concrete topics such as the design and implementation of programming languages and conclude with more abstract ones like software development methods, techniques and tools. We present here a rather brief survey and hope to provide in the future a more extensive and deeper account of this important subject area.

2 Distributed Object Models

The notion of object has already been extensively discussed in the Computer Science literature [42]. An *object* is an identified unit of computation, possibly having an internal state and means of interaction with other similar objects. The current state of an object may be partitioned into *attributes*, each of which denoting some of the current properties of that object. Object states are *encapsulated*, meaning that a state can only be modified by the computations of the corresponding object. Whenever we try to understand a problem or propose a solution in terms of objects as defined above, we say that an *object based approach* is being adopted.

In the definition of *object oriented approaches*, the notion of class is also required. A *class* is a collection of objects which obey the same definition, be it an abstract specification or a computer program. This notion is normally coupled to a technique which permits easy statement and automated reuse of definition parts, *inheritance*, and also to a relation between properties satisfied by class elements, *subtyping*. Inheritance and subtyping are supported in many modern object oriented programming languages.

The behaviour of object communities may be quite intricate. Objects receive *messages* and perform corresponding *methods* as a result. A method may carry out a local computation, which in turn may cause the dispatch of other messages. In some object based models, such as in the actor model [1], it is also possible to create new objects while performing a local computation. The semantics of object interaction is not fixed.

It is precisely in the semantics of interaction and in the assumption that *locations* are relevant that distributed systems differ from centralised ones. In a distributed system, the interaction of two units of computation does not assume that they are co-located. Distributed computing units can be grouped into logical entities called *components*, which have an explicit interface with their environment [40]. The definition of components facilitates not only the modularisation of large software systems but also the study if system behaviour is appropriate in environments with little or perhaps no constraints, i.e. to assess whether or not the system is *open* [15].

Many distributed system interaction modes which are defined in terms of resource sharing violate inherent assumptions of object based approaches such as encapsulation. Among other reasons, this is why message based interaction has been prefered in the definition of *distributed object based approaches*. Messages may be transmitted point-to-point [1], addressed to groups (multicasting) [5] or to all objects of a system (broadcasting) [25]. For point-to-point communication, messages may be transmitted in synchrony, asynchrony or even partial synchrony between sender and receiver [9, 26]. Asynchrony here means that it is not possible to place internal bounds on communication delays nor on the relative speed of each object and partial synchrony means that a strict partial order on the computations of objects can be derived from their interactions. The delivery of each message may be guaranteed or not and the configuration of the objects of each system may be static or dynamic, in this last case perhaps varying at some points in their behaviour. A *model* of distributed systems has to spell out all these characteristics. The development of such models is an active basic research area (see [12] for an example).

It is important to mention that performing object based computation without the support of object based operating environments is possible, much in the way that some such environments allow us to bypass assumptions of an object based approach. Concerning distribution, however, the situation is the opposite: in order to do any sort of distributed computation, a distributed operating environment is required, which may be a distributed programming language, database management system or operating system. As a corollary, we infer that one can adopt a software development approach based on a distributed object model just with the support to distributed computing in mind.

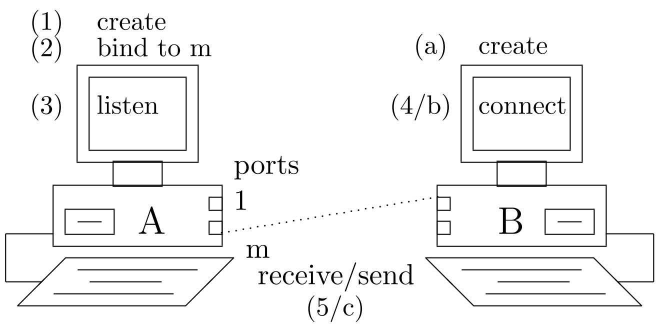

Figure 1: Socket based distributed computing.

3 Distributed Object Languages

The C language [24] was perhaps the first to be used in the implementation of distributed software systems, based on the built in support of TCP/IP and multitasking offered by UNIX workstations. For practical purposes, each independent process performing the local computations and interactions with other processes specified in a C program could be considered as a distributed object implementation. The linguistic support for this behaviour are function calls for process forking and socket manipulation.

The *socket* mechanism is rather simple, generalising the notion of pipe present in some operating systems. The basic *modus operandi* is illustrated in Figure 1. Once there is interest in exchanging data synchronously through a service provided at a specific host, a socket is created therein (1) and bound to a communication port (2), where a process will be listening for attempts of connection (3). At the client side, knowing the server address and operating port, a local socket is created (a), a connection to the server socket initiated (b/4) and data exchanged in either way (c/5) until an attempt at closing the connection happens or a fatal error occurs.

Despite their effectiveness, the adoption of sockets and process forking has limitations. The programmer is obliged to replicate the same low level code for each interaction point and control flow source, treating explicitly all the relevant operating environment details and possible error conditions. In addition, the distinction between server and client may vanish once a socket connection is established. Essentially, these problems happen because there are no specific language constructs nor a simple model that could transparently support in C object creation and distributed interaction.

In order to alleviate some of these limitations, the client server model of distributed computing [35] and a transparent method for distributed interaction based on remote procedure calls (RPCs) [6] were proposed. With the help of a compilation tool that isolates network details and repeatable patterns of code, any functionality can be provided without knowledge of a distributed execution, as if in a local procedure or function call. This enforces the client and server roles in each interaction. Instead of invoking the remote functionality explicitly, the client and server rely on stub code generated automatically by the compilation tool, which in turn relies on publically available runtime support provided by the operating environment. This is illustrated in Figure 2.

Some solutions to the problems of originating a new control flow and supporting

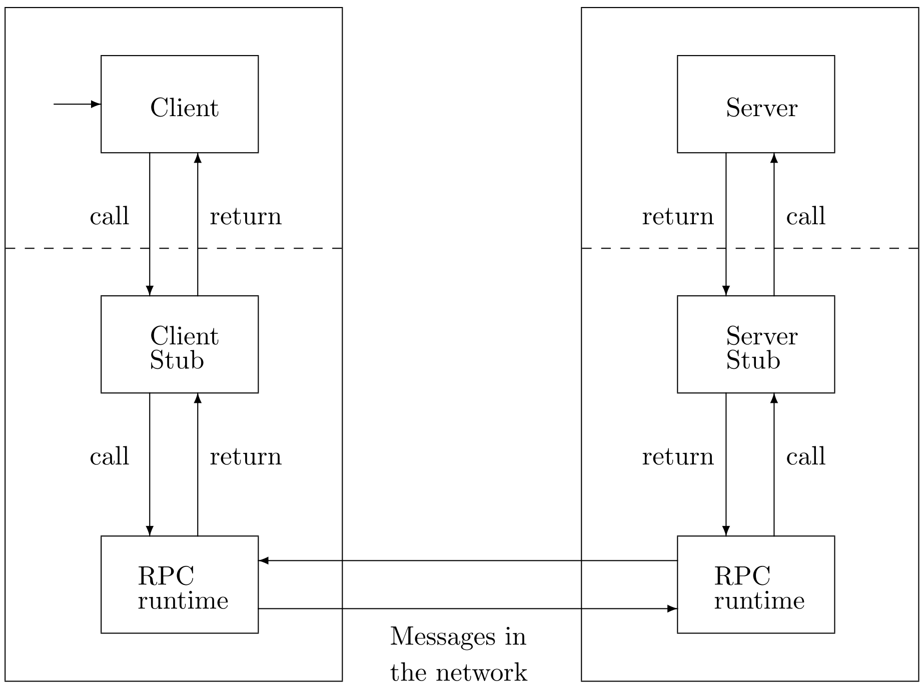

Figure 2: Remote procedure calls.

distributed object models natively appeared in object oriented languages such as C++ [39] and Java [20]. Using these languages and their constructs, new uniquely identified objects may be created, sockets can be represented and used as any kind of object and remote methods may be invoked in a RPC like manner.

It is important to mention that until recently most existing distributed object languages would address only distributed object computing, neglecting the data involved in distributed interactions. This is reflected, for instance, in the necessity of eventually marshaling and unmarshalling the data objects provided as arguments in remote invocations. In turn, these operations force such objects to be serializable, passive of serial transmission through data streams between separate hosts. Recently, though, some attempts at providing data description and structuring languages with an object based interpretation have been reported, which potentially can make distributed interactions more transparent. The most prominent of these efforts seems to be that of the World Wide Web Consortium (W3C) in defining an extensible markup language (XML) [11] with its associated document object model (DOM) [10]. XML and DOM have been used in connection with many high level object based programming languages.

The development of new distributed object based languages with their runtime support appears to be directed mainly towards two promising (not disjoint) directions:

- Domain specific languages and dialects: In this category are those attempts at providing better support for security, mobility, new human interfaces and others;
- Timed and real-time languages: In this category are those attempts at providing better support for quality of service and other timing constraints required in multimedia systems, groupware and other areas.

$$\begin{tikzcd}
 & \textbf{Architecture} & \\
\textbf{Pattern} \arrow[ur, "\begin{array}{c}\text{increase}\\ \text{scale}\end{array}"] & & \textbf{Framework} \arrow[ul, "\begin{array}{c}\text{increase}\\ \text{abstraction}\end{array}"'] \\
 & \begin{array}{c}\textbf{concrete}\\ \textbf{software}\\ \textbf{artifact}\\ \textbf{in the small}\end{array} \arrow[ul, "\begin{array}{c}\text{increase}\\ \text{abstraction}\end{array}"] \arrow[ur, "\begin{array}{c}\text{increase}\\ \text{scale}\end{array}"'] &
\end{tikzcd}$$

Figure 3: Relationships between distributed object based software engineering notions.

The challenge here appears to be the development of new language constructs in ways that do not disturb the supported distributed object models nor other properties of interest such as modularity. For each of these to be a rigorous effort, the new linguistic constructs have to be proposed together with a rationale based on the language semantics.

On the other hand, the current basic research concerns in this area appear to be focused on developing a better understanding of such languages and designing software tools that exploit these improved visions. Understanding any formal language is connected to developing an associated logical or mathematical semantics for such a language, which in turn allows us to create formally verifiable software tools. For instance, the semantics of Java defined in [7] using abstract state machines can be adopted to guide the implementation of a correct fully operational virtual machine for this language. Other examples concerning Java abound in the literature.

4 Architectures, Frameworks and Patterns

Developing large scale distributed object based systems has given rise to many original ideas concerning how to make this task more effective. Beginning at a low level of abstraction with small scale software artifacts, such as object based programs, we may be tempted to develop a new notion to perform distributed object based programming in the large, much in the way that module structuring mechanisms were considered as an attempt at increasing the scale of structured programming. Although most current object based programming languages do offer support for programming in the large, pursuing this idea a bit further has already led to the development of *frameworks* [17]. Starting at the same point but exploring a distinct direction, we may wish to increase slightly the level of abstraction of our implementations, but likewise this has already led to the development of *distributed object design patterns* [19]. The conjunction of these two efforts has already originated a notion of *distributed object based architectures* [34]. The relationships between all these notions are illustrated in Figure 3.

Frameworks are collections of concrete software artifacts that programmers can instantiate, compose or customise for specific purposes. Programmers develop such frameworks to facilitate reuse by grouping together entities that encapsulate the knowledge concerning an application domain (domain frameworks), best-practice implementations (application frameworks) or software infrastructures (utility frameworks). For instance, a set of component specifications for file system handling, written in an object based programming language, defines a utility framework.

Patterns are generic linguistic constructions written in terms of particular object based languages. They represent an object based attempt at supporting generic programming and should be proposed to solve specific well defined problems of repetitive nature. There are factory method patterns that describe the instantiation of object descriptions and software patterns may also be used to represent specialised modes of interaction [2]. Neither patterns nor frameworks necessarily describe complete object based behaviours. While it may be necessary to provide proper connections to make the instances of a framework operational, a pattern can only give rise to runtime behaviour after particularisation and instantiation.

Architecture descriptions classify software system families according to their high level structure and configuration: the gross decomposition of each system in components, interfaces and connectors. All these notions can be formulated in distributed object based terms and actually the use of patterns and frameworks is an effective technique in architectural description activities. The study of software architectures aims to promote reuse and facilitate the development of specific software development tools. This effort identified some usual architectural styles, such as *client-server* [35] and *meta-level* [41] architectures in which a clear classification of component roles exists: components that provide (server) or request (client) functionality, in the first case; that perform (base level) or support (meta level) computation and interaction, in the second one. Architectures following the same style share the same vocabulary and configuration rules.

The research and development efforts regarding architectures, frameworks and patterns have been quite fruitful. From the standardisation efforts of the Object Management Group (OMG) resulted CORBA [21], a reference architecture for distributed object based computing. The rationale for proposing a common architecture for this purpose is that continuously distinct distributed object based programming languages will be adopted for software development and, instead of suggesting the use of a single language and operating environment, it seems to be more appropriate to provide means for these diverse heterogeneous environments to cooperate effectively. Many distributed computing utility frameworks called *object request brokers* (ORBs) already comply with the common architectural style suggested by the OMG. Frameworks satisfying this or similar styles are generically classified as distributed *middleware*. Particular examples of middleware are *application servers* which bring to the distributed world the notion of production environments adopted in the realm of centralised software platforms.

The object request broker architecture is defined in terms a separation between client and server roles. Clients may detect the occurrence of events and generate messages as a result, which are considered as requests of distributed functionality. These are treated by brokers that take care of associating to each request a server object capable of supplying the desired functionality. In order to adopt an object request broker, designers and programmers are forced to rely on a method similar to RPCs. There is an interface description language (IDL) compiler which generates stubs (client side) and skeletons

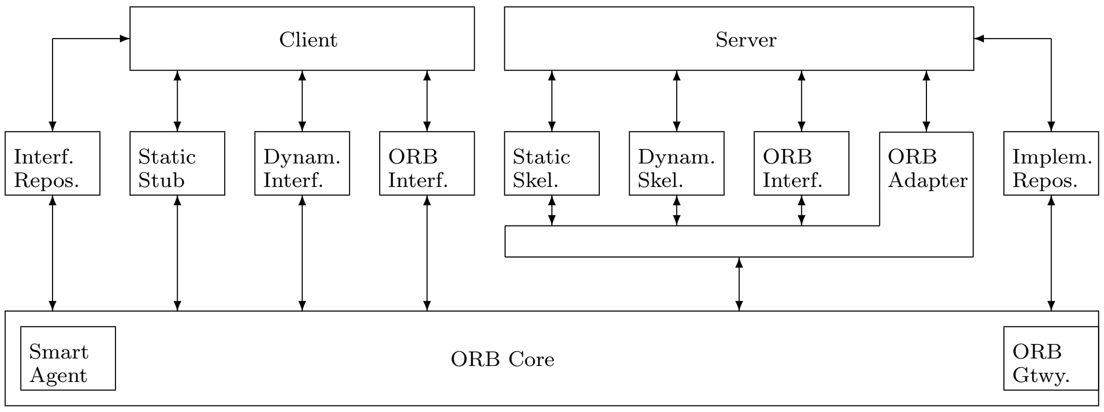

Figure 4: Common Object Request Broker Architecture.

(server side) for each object type specification. Smart agents are objects local to each architecture node responsible for providing runtime services such as distributed directory, load balancing, crash recovery and remote cooperation. ORB gateways treat the interaction with other brokers. This architectural style is illustrated in Figure 4.

Research and development trends in this area are strictly connected to those listed in the previous section. For instance, it is an active research area to develop object brokers providing support for specific features such as real time computing [32]. Concerning basic research, so far results on the rigorous treatment of patterns, frameworks and architectures have been concentrated on this last notion [18], contributing to clarify the nature of architectural styles and the meaning of architectural representations. An analogous study appears to be needed concerning patterns and frameworks.

5 Distributed Object Software Development

The use of distributed object models, languages and other notions underlies a broader activity which we call distributed object based software development. Although we have already mentioned many techniques and tools that are adopted in specific tasks of this activity, software development may also be seen as an evolutionary process comprising stages of decreasing level of abstraction such as requirement analysis, software design and implementation. In this section, we focus on methods, techniques and tools that are helpful at stages and transitions following object based development processes.

In addition to the definition of CORBA, the standardisation efforts of the OMG also resulted in a modelling technique that assists software engineers in designing and implementing distributed object based systems. The OMG defined a Unified Modelling Language (UML) [28] that either supersedes or incorporates many previously proposed best practice modelling techniques. The adopted graphical notations for software design are the following:

- use case diagrams: can be used to specify external views of systems or components in particular utilisation contexts, with the involved actors and functionalities made explicit. They are particularly appropriate for capturing user requirements;

- class diagrams: can be used to specify object classes with their attributes and methods as well as static structural relationships between classes, objects and interfaces;
- state transition diagrams: can be used to specify discrete object behaviour. They are multi-layer object oriented state transition diagrams inspired by the Statechart notation [22];
- activity diagrams: they are useful for specifying the sequences and conditions of actions within transactions. They are essentially transition diagrams inspired by Petri nets [30];
- sequence diagrams: another kind of transition diagram, usefully for specifying complex interactions between objects where the actors, states and exchanged messages are explicitly represented;
- collaboration diagrams: structural descriptions of the participants in collaborative activities together with their communication patterns.

The adopted graphical notations for software implementation are the following:

- component diagrams: describe dependencies between software artifacts specifying components, which possibly belong to distinct abstraction levels;
- deployment diagrams: describe software system runtime configurations, possibly also addressing their association to distributed operating locations.

All these different notations allow us to develop multiple perspective views of a problem and possible solutions. There are specific guidelines to help software engineers move from an abstract specification of a system created with these notations to a concrete implementation in a distributed object based programming language.

Many basic research efforts on rigorous software development are connected to obtaining a better understanding of modelling languages like UML and its notations. Diagrammatic notations such as Statecharts have been defined as textual languages for which many formally defined semantics exist. This has been an active research subject since this notation was proposed [23]. Other graphical notations such as the specification diagrams proposed in [37] are defined precisely in terms of their underlying object based formal semantics. In particular, specification diagrams have been proposed as an alternative to the sequence diagrams of UML.

Another basic research area that has attracted substantial attention is the definition of formal object based specification languages. These languages are defined in terms of logical systems comprising a formal language, a model theory defining the language semantics, and a proof theory that provides axioms and proof rules to enable the verification of some properties. Object based logical systems may be generic as OSL [33], in which case they do not fully define a model of distributed computing, or may be specific tailored to one particular model, such as the work reported in [14] based on the actor model. Notations, techniques and guidelines to support the rigorous step by step development of distributed object based software systems are still needed.

The adoption of a distributed object based approach for software development is particularly well suited to mechanisation [36]. Informal or semi-formal notations such as those adopted in UML are already supported by software development environments such as Rational Rose [38], which takes care of the whole development process. The existence of a meta-level definition of UML facilitates not only the implementation of corresponding interoperable software development environments but also the extensibility of the language itself.

At the present moment, just specific tools are available to help developing distributed object based systems in a rigorous way. One of the techniques that has been adopted with great success is *model checking* [16], which consists in taking advantage of a specification language with a logical semantics to verify the satisfaction of interesting properties of software systems. In [8], for instance, an automated translation of UML architectural models into the language of a model checker is described, showing how some temporal properties can be verified and simulated with automated support.

Another technique that is gaining attention nowadays is *theorem proving*, which relies on the implementation of a logical system in some automated reasoning tool [4]. In [13], for example, the application of the meta-logical framework Maude to analyse active networks and communication protocols is reported. Although this appears to be an incipient basic research area, there is some evidence that the verification rules useful in software development are among those derivable from the adopted underlying logical system [3, 15].

6 Final Remarks

We have surveyed some current research and development trends related to rigorous distributed object based software development. This area covers not only standard subjects in Computer Science such as the design and implementation of programming languages as well as software development methods, techniques and tools, but also innovative notions such as frameworks, patterns and software architectures. We hope that our personal account to these subjects and notions, in addition to the organisation of the presented topics, could somehow contribute to the advancement of this interesting and active subject area.

Acknowledgements
The remarks of the three anonymous referees contributed to improve the quality of this paper. The reported work has been partially supported by CNPq research grant number 301037/00-0.

References

[1] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[2] Gul Agha. Abstracting interaction patterns: A programming paradigm for open distributed systems. In Elie Najm and Jean-Bertran Stefani, editors, *Proc. 1st*

IFIP Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'96), pages 135–153. Chapman and Hall, 1997.

[3] Penny Anderson and David Basin. Program development schemata as derived rules. *Journal of Symbolic Computation*, 30(1):5–36, July 2000.

[4] David Basin. Logical framework based program development. *ACM Computing Surveys*, 30(3):1–4, 1998.

[5] Ken Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):37–53, December 1993.

[6] Andrew D. Birrel and Bruce J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computing Systems*, 21(1):39–59, 1984.

[7] Egon Börger and Wolfram Schulte. A programmer friendly modular definition of the semantics of Java. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java(tm)*, volume 1523 of *Lecture Notes in Computer Science*, pages 353–404. Springer Verlag, 1998.

[8] Prasanta Bose. Automated translation of UML models of architectures for verification and simulation using Spin. In *Proc. 14th Conference on Automated Software Engineering*, pages 102–109. IEEE Computer Society, October 1999.

[9] Bernardette Charron-Bost, Friedemann Mattern, and Gerard Tel. Synchronous, asynchronous and causally ordered communication. *Distributed Computing*, 9:173–191, 1996.

[10] World Wide Web Consortium. Document object model (DOM) level 1 specification. W3C recommendation, World Wide Web Consortium, October 1998.

[11] World Wide Web Consortium. Extensible markup language (XML) 1.0 (second edition). W3C recommendation, World Wide Web Consortium, October 2000.

[12] Flaviu Cristian and Christof Fetzer. The timed asynchronous distributed system model. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):642–657, June 1999.

[13] Grit Denker, José Meseguer, and Carolyn Talcott. Formal specification and analysis of active networks and communication protocols: The Maude experience. In *Proc. DARPA Information Survivability Conference and Exposition*, volume I, pages 207–221. IEEE Computer Society, 2000.

[14] Carlos Henrique C. Duarte. Proof-theoretic foundations for the design of actor systems. *Mathematical Structures in Computer Science*, 9(2):227–252, 1999.

[15] Carlos Henrique C. Duarte and Tom Maibaum. A rely-guarantee discipline for open distributed systems design. *Information Processing Letters*, 74(1–2):55–63, April 2000.

[16] E. Allen Emerson, A. Prashad Sistla, and Edmund Clarke. Automatic verification of finite state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[17] Mohamed Fayad and Douglas C. Schmidt. Object-oriented application frameworks. *Communications of the ACM*, 40(10):85–87, October 1997.

[18] José L. Fiadeiro and Tom Maibaum. A mathematical toolbox for the software architect. In Jeff Kramer and Alexander Wolf, editors, *Proc. 8th Workshop on Software Specification and Design*, pages 46–55. IEEE Computer Society Press, 1996.

[19] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, 1995.

[20] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison-Wesley, 1996.

[21] Object Management Group. CORBA: Architecture and specification. Technical report, Object Management Group, February 1998. Revision 2.2.

[22] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[23] David Harel, Amir Pnueli, Jeanette P. Schmidt, and Richard Sherman. On the formal semantics of Statecharts. In *Proc. 2nd Symposium on Logic in Computer Science*, pages 54–64. IEEE Computer Society Press, 1987.

[24] Brian Kernighan and Dennis M. Richie. *The C Programming Language.* Prentice-Hall, 1978.

[25] Robert M. Metcalfe and David R. Boggs. Ethernet: Distributed packet switching for local computer networks. *Communications of the ACM*, 19(7):395–404, 1976.

[26] Venkatesh Murty and Vijay K. Garg. Characterization of message ordering specifications and protocols. In *Proc. 17th Conference on Distributed Computing Systems (ICDCS '97).* IEEE Computer Society, May 1997.

[27] Bashar Nuseibeh. Ariane 5: Who dunnit? *IEEE Software*, 14(3):15–16, 1997.

[28] Object Management Group OMG. *Unified Modelling Language Specification.* Object Management Group — OMG, June 1999. Version 1.3.

[29] Charles Perkins. IP mobility support. RFC 2002, October 1996.

[30] Carl A. Petri. Fundamentals of a theory of asynchronous information flow. In Cicely M. Popplewell, editor, *Information Processing 62: Proc. 1st IFIP World Computer Congress*, pages 386–390. North Holland Publishing Company, 1962.

[31] Bala Rajagopalan, James Luciani, Daniel Anduche, Brad Cain, Bilel Jamoussi, and Debanan Saha. IP over optical networks: A framework. IETF draft, Internet Eletronic Task Force, March 2001.

[32] Douglas C. Schmidt, David Levine, and Sumedh Mungee. The design of the TAO real-time object request broker. *Computer Communications Special Issue on Building Quality of Service into Distributed Systems*, 21(4):294–324, April 1998.

[33] Amílcar Sernadas, Cristina Sernadas, and José Félix Costa. Object specification logic. *Journal of Logic and Computation*, 5(5):603–630, October 1995.

[34] Mary Shaw and David Garlan. *Software Architecture: Perspectives of an Emerging Discipline.* Prentice Hall, 1996.

[35] Alok Sinha. Client-server computing. *Communications of the ACM*, 35(7):78–98, July 1992.

[36] Douglas R. Smith. Mechanizing the development of software. In Manfred Broy and Ralf Steinbrueggen, editors, *Calculational System Design*, pages 251–292. IOS Press, 1999.

[37] Scott Smith. Specification diagrams for actor systems. In Andrew Gordon, Andrew Pitts, and Carolyn Talcott, editors, *Proc. Higher Order Operational Techniques in Semantics II*, volume 10 of *Electronic Notes in Theoretical Computer Science.* Elsevier Science Publishers, 2000.

[38] Rational Software. *A Rational Approach to Software Development using Rational Rose 4.0.* 2001.

[39] Bjarne Stroustrup. *The C++ Programming Language.* Prentice-Hall, 2^{nd} edition, 1992.

[40] Carolyn Talcott. Composable semantic models for actor theories. *Higher-Order and Symbolic Computation*, 11(3):281–343, 1998.

[41] Nalini Venkatasubramanian and Carolyn Talcott. A meta architecture for distributed resource management. In *Proc. Hawaii Conference on System Sciences*, pages 124–133. IEEE Computer Society Press, January 1993.

[42] Peter Wegner. Dimensions of object-based language design. In *Proc. Object Oriented Programming Systems, Languages and Applications (OOPSLA'87)*, pages 168–182. ACM Press, October 1987.