Applying MDE Methodologies to Design Communication Protocols for Distributed Systems

Jesús Martínez, Pedro Merino, Alberto Salmerón
Dpto. de Lenguajes y Ciencias de la Computación
University of Málaga
29071 Málaga, Spain
{jmcruz,pedro,salmeron}@lcc.uma.es

Abstract

Traditionally, protocol engineers have to deal with the design and implementation of complex network services, spending considerable time and effort on creating robust and reliable final source code. Although approaches exist to assist engineers in the development of communication protocols which use several object-oriented frameworks, these do not benefit from new modelling guidelines developed in MDE *and* UML *to exploit automatic code generation from graphical models. This paper introduces a new* UML*2 Profile for Communications which guides the construction of software for communications following the Client-Server architecture. The* MDE *process is then used to design suitable platform-specific models for the well-known Adaptive Communications Environment (*ACE*), a high performance C++ toolkit for implementing concurrent and network applications which relies heavily on architectural patterns.* [1]

1. Introduction

The design and implementation of high performance network services for distributed and embedded systems has traditionally implied the construction of communication protocols. In order to build protocols and related software, protocol engineers follow standard reference models such as ISO's Open Systems Interconnection model (OSI) [2] and the well-known Client-Server architecture model for Internet applications [3]. Both of these standards describe communications in terms of stacks of layered processes for which each layer (or entity) provides a service to its upper layer relying on the services provided by the layer below. In this context, protocols are defined as the rules for communicating two remote entities at the same level in the stack.

Implementing protocols is a complex and error-prone task for which two approaches currently exist. The first consists of developing with classic programming languages (usually C), using the facilities offered by operating systems and communication libraries, e.g the Socket API. Although widely used, this approach is traditionally dependent on low-level APIs and platforms, producing non-portable code which is difficult to extend and reuse. This effect may be mitigated by the use of object-oriented frameworks for communications [14]. These frameworks are oriented to wrap all low-level communication concerns, achieving large-scale designs and code reuse. Nevertheless, they imply a high initial learning curve with many classes and many levels of abstraction.

The second approach consists of using modern (visual) modelling languages. In the context of protocols two parallel lines of work exist. The first employs the SDL formal notation [9]. This language is supported by the International Telecommunication Union (ITU) and the main companies in the telecommunications sector. SDL has been the traditional way to design complete protocol stacks. Supported by visual environments, SDL designs benefit from their formal semantics to obtain executable code and to perform rigorous analysis of protocol designs. The second area of work is the use of a widely adopted modelling language such us UML to design object-oriented software. UML is supported by the Object Management Group (OMG) and the software engineering community.

Unfortunately, the convergence of these areas of work with the traditional implementation of networked services for the Internet is still missing. In this paper we argue that the two approaches presented above are complementary, in the sense that object-oriented frameworks for communications could be instantiated by using modelling languages. However, in order to avoid the complexity of us-

[1]Work partially supported by projects TIN2004-7943-C04 and TIN2005-09405-C0201

Proceedings of the First International Conference
on Complex, Intelligent and Software Intensive Systems (CISIS'07)
0-7695-2823-6/07 $20.00 © 2007 IEEE

IEEE COMPUTER SOCIETY

ing these frameworks in a visual language such as UML, we should assist protocol designers with clear and easy mechanisms which hide the difficulties around communication details and other peculiarities in the Client-Server architecture. These objectives may be achieved by applying Model Driven Engineering (MDE) concepts [10].

The model-driven approach to system development uses models to direct its design, construction, maintenance, testing and modification. In MDE different kinds of models exist [12]. The so-called platform-independent models (PIMs) are those which do not contain specific details about the final implementation platform. Those details are added in so-called platform-specific models (PSMs), which are better oriented to guide the (automatic) final code generation.

Surprisingly, very little work has been done to exploit the benefits of MDE methodologies to design software for communications. The objective of this paper is to apply MDE concepts in order to simplify the design of network services for the Internet. We advocate the use of the recent UML2 to describe the structure and behavior of protocols by: i) defining a UML Communications Profile to describe high-level concepts dealing with communications in the Client-Server architecture, and ii) demonstrating its viability to automatically obtain a model including a valid configuration for the frameworks available in the Adaptive Communications Environment (ACE), a high performance C++ toolkit for implementing concurrent and network applications which relies heavily on architectural patterns. According to the MDE guidelines, it is advisable to design a general model for communications (a PIM model) and then find automatic ways to generate a specific model (PSM) which will include the customized functionality needed to instantiate the appropriate ACE frameworks.

The paper is organized as follows. Section 2 gives a brief background on protocol design and UML. In Section 3 we describe the features available in the Communications Profile. Section 4 discusses the MDE transformation from a (general) communications model to an ACE-oriented one which will guide the final C++ code generation. Finally, we outline our conclusions.

2 Background

The programming library most used to develop network applications for the Internet is the Socket API. Sockets were developed to provide an application-level interface to the TCP/IP protocol suite [16]. Therefore, applications may use the C functions in the API to create and manage local endpoints of communication (sockets). Each socket is bound to a local and a remote address. These addresses define the association between communicating peers. The native Socket API has severe limitations, and is considered a complex, non-portable and error-prone library [14]. Fortunately, modern object-oriented frameworks address these limitations and offer other interesting features, such as support for developing layered/modular services and protocols.

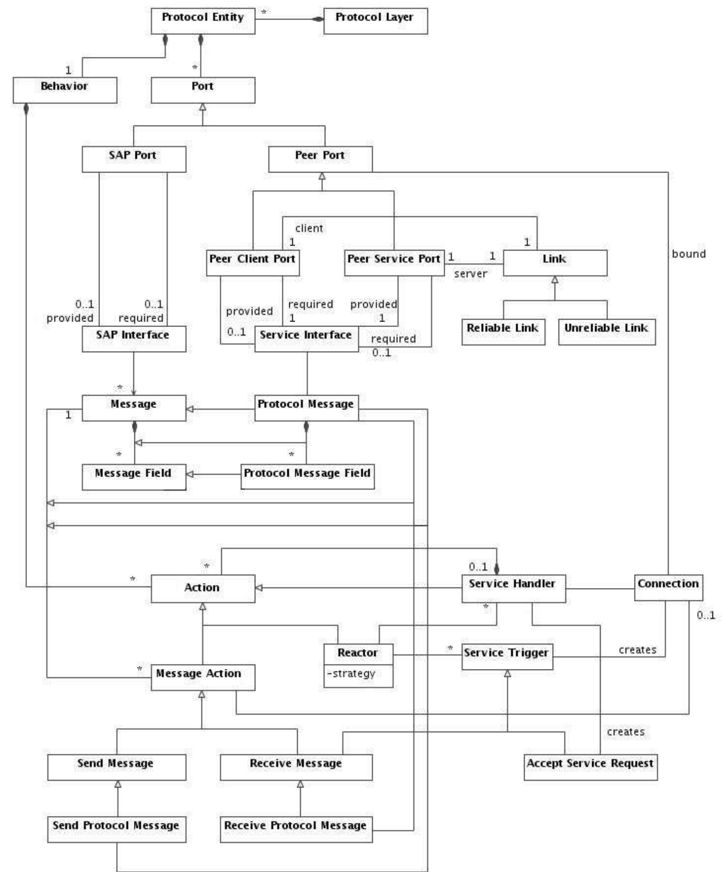

Figure 1. Communication concepts to be covered by the Profile

For instance, ACE is a highly portable object-oriented toolkit composed of frameworks which can be instantiated and customized to provide high performance networked applications. The capabilities of ACE cover the session, presentation and application layers in the OSI model. The toolkit is a combination of an operating system adaptation layer and C++ wrappers, which together encapsulate core network programming mechanisms available in common platforms [14]. The ACE frameworks are an integrated set of classes implementing design and architectural patterns for communications [6, 15], such as Reactors, Proactors, Acceptors or Connectors, among others. Regarding the Socket API, ACE defines a set of C++ wrappers which encapsulate its functions and data in order to enhance type-safety, to ensure platform independency and to reduce development effort spent on lower-lever network programming details.

Some other examples of previous work on object-

Proceedings of the First International Conference
on Complex, Intelligent and Software Intensive Systems (CISIS'07)
0-7695-2823-6/07 $20.00 © 2007 IEEE

IEEE
COMPUTER
SOCIETY

oriented frameworks for communications are x-kernel [8] and Conduits+ [7]. Both of these proposals are oriented towards creating protocol layers from scratch, it being possible to develop complete transport facilities such as the TCP/IP stack. The elements in these frameworks support basic tasks needed by protocol layers: input and output queues, event (de)multiplexing/analysis/filtering or service processing. ACE also supports this modular (layered) design through its Streams framework [14].

As mentioned before, there are only a few approaches which employ UML and MDE guidelines to support the design of communication protocols, although none of them focus on the Client-Server architecture and do not make use of object-oriented frameworks.

The proposal in [13] explores the benefits of designing a UML (1.4) Profile for describing communication protocols. The approach is close to the Conduits architecture. It introduces so-called protocol system structure diagrams where communication layers may be defined with UML class diagrams. The Profile only defines semantics for protocols, peers and communication gates or interfaces (some of them as the substitutes of current UML2 interfaces, ports or connectors). However, the behavior of the protocol is specified in state diagrams, in which transitions have to be specified in a graphical protocol description language, introduced by the authors to solve, once again, the lack of action semantics in the UML version used. Ultimately they provide a tool which translates all their descriptions to SDL.

In contrast, the more recent PROSPEX tool [4] exploits some UML2 features to build protocol models. Their objective is the incorporation of performance parameters to their descriptions (delays, scheduling, time constraints) in order to obtain suitable representations of the communication system for a network discrete-event simulator. The authors use Telelogic Tau G2, which combines UML and SDL. Therefore, composite structure diagrams and state charts define the architecture and the behavior of communication software. Unfortunately, the method selected to specify those performance parameters is not a Profile, but simple UML comments.

The proposal in [1] introduces a model driven method applicable to the specification and design of embedded systems, especially for protocol processing applications. It was employed to design and implement an IPv6 router, and the authors use some of the MDE concepts to build models and to transform them in order to incorporate specific-domain and platform-dependent information. They start with a combined set of UML diagrams and the data flow paradigm, which constitute the platform-independent models. Then they introduce the requirements specification and the domain knowledge (operations regarding communication protocols) to obtain so-called domain-dependent platform-independent models. These models guide the final code generation to the router processor.

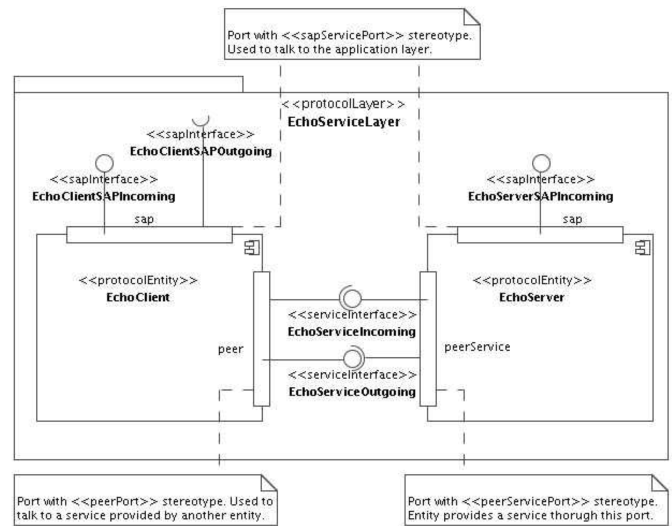

Figure 2. Example protocol entities

3 The UML Communications Profile

The concepts covered by the Communications Profile include structural modelling and behavior. This section shows how these concepts (see Fig. 1) have been translated to a UML2 Profile.

3.1 Structural modelling

The normal way to organize software for communications is in layers, each of which defines a protocol. It is worth noting that our Profile will focus on modelling protocols following the Client-Server approach, so our layers will operate on top of the TCP/IP stack. For each new layer provided we will map the ProtocolLayer concept in Fig. 1 to a UML2 package with the `<<protocolLayer>>` stereotype.

Entities are the distributed parts of the system which communicate following some protocol, and will be contained in the `<<protocolLayer>>` package. We will identify an entity as a component in the layer with the `<<protocolEntity>>` stereotype, which is consistent with the definition given by OSI. Fig. 2 shows an example of an echo service with two `<<protocolEntity>>` components, EchoServer and EchoClient, contained within the EchoServiceLayer.

Entities interact through ports described with interfaces. A single entity can have multiple ports and interfaces. The Profile will distinguish between ports for communicating with remote entities (peers) and those communicating with the application layer, marking them with the

IEEE COMPUTER SOCIETY

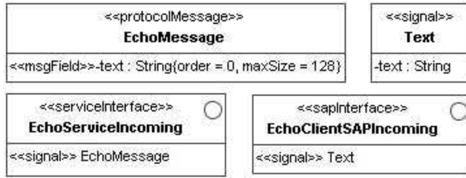

Figure 3. Example interfaces and messages

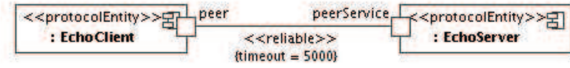

Figure 4. The link description between the echo client and server

<<peerPort>> and <<sapPort>> stereotypes, respectively. Each port has at most one provided and one required interface, which declare the messages exchanged. The provided interface lists the messages a port expects, and the required interfaces the ones it can send back. The <<serviceInterface>> stereotype is applied to interfaces that describe a service, while interfaces with the application layer have the <<sapInterface>> stereotype. The two entities of our example are connected by EchoServiceIncoming and EchoServiceOutgoing interfaces, as shown in Fig. 2. EchoService is providing the service, so it needs a <<servicePort>>, while EchoClient uses a <<peerPort>> to communicate with the remote service port. EchoClient offers its functionality to the application layer through a <<sapPort>>, described by both EchoClientSAPIncoming and EchoClientSAPOutgoing <<sapInterface>>s. EchoServer also accepts stop messages from the application layer, thus only needing an EchoServerSAPIncoming interface.

The server or client role is not always well-defined, because some applications may be simultaneously servers and clients. Therefore we have moved the Client-Server role discussion to the ports: we can decide whether a port is used to provide a service (with the <<servicePort>> stereotype), or to access a service provided by another entity (with the <<peerPort>> stereotype).

After defining the communication interfaces, we have to describe the messages they will interchange, a message being the abstraction of a protocol data unit (PDU). Fig. 3 shows the class diagram corresponding to the messages (and interfaces) existing in our example. Communications considered here are always asynchronous, so the messages are modelled as UML signals. For instance, the Stop signal in Fig. 3 corresponds to a message received by the EchoServerSAPIncoming interface of the EchoServer.

The <<protocolMessage>> stereotype is a specialization of a signal which is used to model messages between peer entities in the same layer. For each message, a property represents a field (stereotyped <<msgField>>). The order of these fields is important to marshal/demarshal the PDU at origin/destination correctly. This order will be indicated with the tagged value *order*. It is worth noting that nowadays protocols may have quite complicated structures in their message fields, such as XML or ASN.1. Although we can expect the definition of new user-defined data types to deal with their complex features (multiplicity, optional tags,...), this is an interesting problem that deserves further research.

The class diagram in Fig. 3 describes some of the interfaces used in our example. They declare the messages they accept as UML2 receptions. This avoids ambiguities when specifying signal routes on send or receive operations. Fig. 3 also shows some of the messages used by the echo service. We will describe its use in the following subsections.

The last step will consist of modelling the links between two entities. Although previously we have used ports and interfaces to describe the messages to be exchanged, links will be abstractions of a real network channel. Therefore, links are drawn in UML2 composite structure diagrams (see Fig. 4). As always, the Communications Profile hides low-level communication aspects and shows only some configurable attributes. We will apply the <<reliable>> stereotype to a link when the communication is established through a protocol stack that guarantees reliable data transport (such as TCP). As reliable transport protocols usually establish initial connection between clients and servers, the *timeout* tagged value specifies the time (in milliseconds) in which the connection has to be completed before aborting (see Fig. 4). When reliable transport is not guaranteed, we will use the <<unreliable>> stereotype.

3.2 Behavior modelling

The behavior of protocols in the Communications Profile will be described using UML2 activity diagrams. We are considering them semantically equivalent to finite state machines, similarly to [5]. This results in fewer diagrams per protocol entity, thereby making the network code generation process more effective.

Fig. 5 depicts the behavior of our EchoClient example. This entity interacts with two other entities: its application layer, which will benefit from the echo service, and the EchoServer, which provides the service. To state the port or link through which the messages will flow, Fig. 5 splits the activity into two vertical swimlanes, separating the interactions within each entity. The left swimlane references the <<sapPort>> of the entity, while the right swimlane references its <<peerPort>>. The client starts when it

Proceedings of the First International Conference
on Complex, Intelligent and Software Intensive Systems (CISIS'07)
0-7695-2823-6/07 $20.00 © 2007 IEEE
IEEE COMPUTER SOCIETY

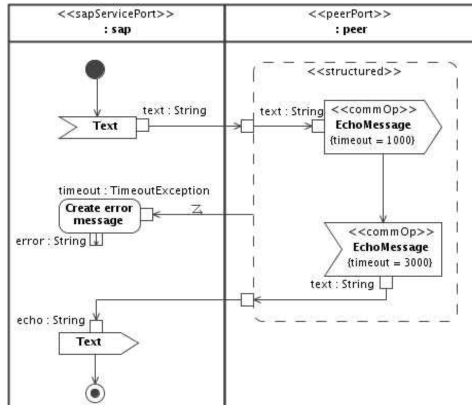

Figure 5. Activity diagram for a client entity

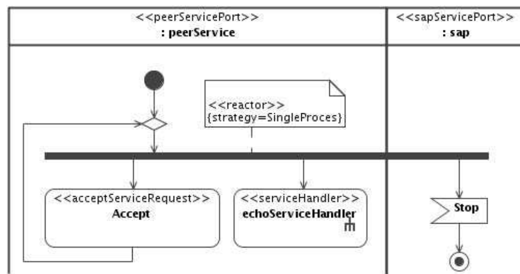

Figure 6. Activity diagram for a server entity

receives a Text message through its SAP. This signal contains the text that will be sent next to the echo service encapsulated within an EchoMessage. Then the client waits for a response from the server. Upon reception of the expected EchoMessage, the client forwards the text received to the application layer. Both `<<commOp>>` stereotyped actions in Fig. 5 are used to send to/receive from peer entities. They may throw a TimeoutException after a timeout, which will interrupt the execution flow.

Fig. 6 shows the activity diagram for the server. The point where an incoming service request from a client is accepted must be explicitly stated with an activity node stereotyped with `<<acceptServiceRequest>>`. The service itself will be provided in the following actions, using the newly opened connection. However, the profile offers support for the most common strategies that servers employ to provide their services: to attend one client at a time or to attend several clients simultaneously. The latter implies the use of concurrency mechanisms. Our Profile will use fork nodes at the point where the desired strategy is applied. We will mark them with the `<<reactor>>` stereotype, and the strategy will be specified with the strategy tagged value.

The logic behind a service managed by a `<<reactor>>` will be encapsulated in a new activity diagram stereotyped with `<<serviceHandler>>` (in our example, to send the echo). The same `<<reactor>>` node may be used for more than one service. In the figure, the Stop signal is dispatched through the `<<reactor>>` node.

4 Obtaining a communications PSM for ACE

After adding communication semantics to our UML model, we may decide how to transform it to a refined PSM model. The resulting UML model will contain all complex details dealing with the Client-Server communications through TCP/IP. We will use the ACE Socket wrappers of ACE [14] in order to obtain multi-platform object-oriented robust code. Those entities providing a service will be enforced using the ACE frameworks needed to implement the selected (concurrent) strategy to handle the service.

The UML elements that will correspond directly to ACE classes are the service and peer ports. The former uses the ACE Reactor framework, which implements the Reactor pattern [15] to allow event-driven applications to demultiplex and dispatch service requests that are delivered to an application from one or more clients. The peer port implements part of the ACE Acceptor-Connector framework, which implements the Acceptor-Connector pattern [15] to decouple the connection and initialization of cooperating peer services in a networked system from the processing they perform once connected and initialized.

Sending and receiving messages requires special manipulation of the data transported, since networked applications run in heterogeneous platforms with different byte-order rules. We also have to consider the inclusion of non-primitive data, such as arrays. Therefore, messages in our Profile will now be represented in CORBA Common Data Representation (CDR) format [11]. The motivation for this is the availability of the ACE_InputCDR and ACE_OutputCDR streaming classes [14] which provide an optimized and portable means to marshal and demarshal message data.

Regarding network links, reliable ones will be established over the TCP layer using an ACE_SOCK_Stream object. Unreliable links will use UDP with an ACE_SOCK_Dgram object.

The `<<commOp>>` signals will specify a send or receive operation with a socket. It is worth noting that connections between peer entities are implicit in the Communications Profile, which implies a previous connection when using a TCP socket (reliable links). Therefore, the first time an entity sends/receives a message through a port in our example, the PSM will automatically incorporate a connection with its peer.

The action by which servers will wait for new

IEEE COMPUTER SOCIETY

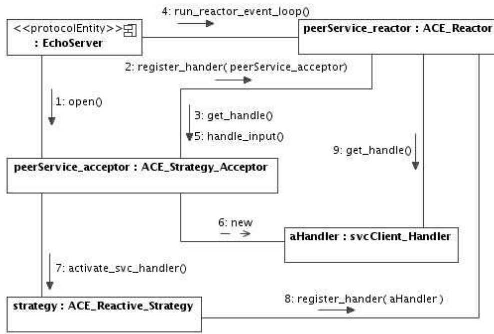

Figure 7. The behavior of the ACE objects implementing a Reactive Strategy

service requests from clients is stereotyped as <<acceptServiceRequest>> action. This will correspond to an ACE object implementing the Acceptor pattern: ACE_Acceptor. The combination of this object and the service model strategy is one of the most powerful features available in ACE. In servers, our PSM for ACE will implement the service strategy selected in the Profile. As mentioned before, different approaches exist to handle a service. If each service handler maintains separate state information for each client, then allocating a service handler per client is generally the most straightforward approach. However, if each service handler does not maintain a separate state for each client, then a server that allocates one service handler for all clients can potentially use less space and operate faster than if it dynamically allocates a handler for each client. The former approach implies the creation of a new thread (or process) per client.

The Communications Profile provides information on how to handle a service in the <<reactor>> fork nodes, which use the tagged value strategy to indicate whether a single process/thread or a multi process/thread design is preferred. The implementation of the selected strategy will make use of the ACE Acceptor/Reactor frameworks. Fig. 7 shows the interaction among objects in both frameworks. The ACE_Strategy_Acceptor (containing the ACE_Acceptor mentioned previously) will register in the ACE_Reactor object, which acts as a network-event demultiplexer. After a new service request, the acceptor will be asked to create a new service handler (the svcClient_Handler object). In order to implement the selected service strategy, the ACE_Strategy_Acceptor object delegates to a specific ACE_Reactive_Strategy. This object will not spawn any new threads, as requested (the opposite operation would be performed by an ACE_Thread_Strategy object). The last step is the registration of the new service handler in the reactor, in order to receive available network events (messages).

5 Conclusions

This paper has introduced a new MDE-based methodology to support the inclusion of ACE's high performance communication features in UML2 models. We start with a platform-independent model for communications which uses a new UML Communications Profile to describe the Client-Server architecture. Then we obtain an ACE platform-specific model. Our method exploits new features available in UML2 diagrams and is general enough to be independent of any UML tool. In the future we also plan to obtain platform-specific models for analysis tools, such as model checkers.

References

[1] M. Alanen, J. Lilius, I. Porres, and D. Truscan. MDE Support in a Protocol Processing Design Method. In *Proceedings of Model-Driven Architecture: Foundations and Applications* , pages 234–247, 2004.

[2] U. Black. *OSI: A Model for Computer Communications Standards*. Prentice Hall, 1991.

[3] D. E. Comer and D. L. Stevens. *Internetworking with TCP/IP Volume III: Client-Server Programming and Applications*. Prentice Hall, 1992.

[4] N. de Wet and P. Kritzinger. Using UML Models for the Performance Analysis of Network Systems. In *Proceedings of the Workshop on Integrated-reliability with Telecommunications and UML Languages*, 2004.

[5] R. Eshuis. Symbolic model checking of uml activity diagrams. *ACM Trans. Softw. Eng. Methodol.*, 15(1):1–38, 2006.

[6] E. Gamma, H. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley Pub Co., 1995.

[7] H. Hueni, R. Johnson, and R. Engel. A framework for network protocol software. In *Proceedings of OOPSLA'95. Austin, TX. ACM*, 1995.

[8] N. Hutchinson and L. Peterson. The x-kernel: an architecture for implementing network protocols. *IEEE Transactions on Software Engineering 17(1)*, pages 64–76, Jan. 1991.

[9] ITU-T Z.100. Specification and Description Language (SDL), 2000.

[10] S. Kent. Model Driven Engineering. In *Proceedings of IFM 2002*, LNCS 2335, pages 286–298. Springer-Verlag, 2002.

[11] Object Management Group. *The Common Object Request Broker: Architecture and Specification, 2.5 edition*. 2001.

[12] Object Management Group. MDA guide version 1.0.1. omg/2003-06-01, June 2003.

[13] J. Parssinen and N. von Knorring. UML for Protocol engineering - Extensions and Experiences. In *Proceedings of TOOLS EUROPE'2000* , 2000.

[14] D. C. Schmidt and S. Huston. *C++ Network Programming Volume 1: Mastering Complexity with ACE and Patterns*. Addison-Wesley, 2002.

[15] D. C. Schmidt and S. Huston. *C++ Network Programming Volume 2: Systematic Reuse with ACE and Frameworks*. Addison-Wesley, 2003.

[16] W. R. Stevens. *TCP/IP Illustrated, Volume 1*. Addison-Wesley, 1993.

IEEE COMPUTER SOCIETY