In the Proc. of Int. Conf. on Communications, Internet and Information Tech. (CIIT2003), Scottsdale, AZ, 17-19 Nov. 03

PROTOCOL SPECIFICATION AND AUTOMATIC IMPLEMENTATION USING XML AND CBSE

Ibrahim S. Abdullah Daniel A. Menascé
Department of Computer Science, MS 4A5
George Mason University
Fairfax, VA 22030-4444, USA
iabdulla@gmu.edu menasce@cs.gmu.edu

Abstract

A communication protocol is an agreement among two or more parties on the sequence of operations and the format of messages to be exchanged. Standardization organizations define protocols in the form of recommendations (e.g., RFC) written in technical English, which requires a manual translation of the specification into the protocol implementation. This human translation is error-prone due in part to the ambiguities of natural language and in part due to the complexity of some protocols. We propose an XML-based language for protocol specification along with a process, based in XSLT stylesheets, for automatic code generation. The code generated uses components from a library of special purpose components that perform frequently executed functions (e.g., encryption). Our approach was validated on three different protocols: Needham-Schroeder's authentication protocol, TCP's three-way handshake, and SSL's handshake. We developed a Java XSLT stylesheet along with an XML Schema for validation of the XML specifications.

Keywords: communications protocol, automatic protocol implementation, XML, CBSE.

1. Introduction

A protocol is a set of rules shared by two or more communicating parties to facilitate their data communication. These rules have two parts: 1) syntax, i.e., the format of the messages to be exchanged and 2) semantic, i.e., the sequence of operations to be performed by each party. The traditional process of implementing a protocol is tedious and error-prone. Specifications in natural language are often ambiguous and may lead to defective implementations. Moreover, implementations have to be thoroughly tested, a time-consuming effort due to the timing dependencies of events processed by protocols.

In this paper, we propose a new model for sharing protocol specifications to produce protocol implementations automatically. The proposed approach is based the eXtensible Markup Language (XML), extensible Stylesheet Language for Transformations (XSLT), and on Component-based Software Engineering (CBSE). XML is used to specify protocols described through Finite State Machines (FSM). XSLT is used to transform the specification description into actual code. XSLT stylesheets can be designed to produce code in different programming languages (e.g., C++ or Java). CBSE is used to build the set of operations needed by the protocol.

Therefore, our approach replaces the use of natural language (e.g., English) for protocol specification with a special purpose language based on XML elements. This approach has four advantages: 1) the specification can be read by machines to produce implementations, 2) the potential for human errors is reduced, 3) XML is a neutral language that is not associated with any specific programming language, and 4) protocol specifications in XML can be easily communicated over a network so that code is automatically generated at a remote network node. This approach is useful for distributing new versions of a protocol. Each node would download the new XML specification and generate local code from it. Moreover, the use of XSLT offers the flexibility of producing code in different flavors to address systems specific needs.

Related work that includes the use of XML to produce program code is found in [1]. In that work, an instrument designer produces an XML document that describes the instrument services. This XML document can be used later to produce a user information document in HTML format and source code for the instrument embedded services. The XML description is just a listing of the services. Each service is associated with a piece of C code that is loaded from a library by an XSLT transformer. Our work extends that work to the area of protocol implementations in conjunction with the use of CBSE. Our XML specification is based on FSMs to provide a detailed description of the flow of control, handling of data objects, and the set of operations.

In [2], FSMs were used to generate code for protocols; a C++ code skeleton for the flow of control of the protocol was generated directly out of FSM diagrams. However, in our work we encode FSMs in XML allowing for automatic code generation in any programming language. A more detailed discussion on related work is presented at the end of the paper.

The rest of the paper is organized as follows. Section two describes the use of FSMs in protocol design. Section three presents approaches to protocol specification. Section four describes our proposed XML-based specification language. Section five presents the mechanism for automatic code generation. Section six discusses related work and section seven presents some concluding remarks.

2. Use of FSMs in Protocol Design

Finite state machines (FSM) are commonly used for describing protocols. The graphical representation of an FSM is a visual aid to the designer of the protocol. This representation helps in testing and verifying the correctness of the design. An FSM consists of a set of *states* (represented by circles) and a set of *transitions* between these states (represented by directed arrows connecting the states). At each state, a set of *events* may occur. Each event triggers a specific transition out of a state. Before a transition occurs, a set of actions (possibly an empty set) is taken. One of the states in an FSM is designated the *initial state*. Figure 1 shows an FSM for the *Needham-Schroeder* authentication protocol *(NSAP)*, a two- way authentication protocol based on public key cryptography. A client starts the process by using the public key of a server to encrypt his identity and a random number, Na. The server receives the message, generates a random number Nb, encrypts the concatenation of Na and Nb using the client's public key, and sends the message back to the client. The authenticity of the server is verified if the client successfully retrieves his Na. Then, the client returns Nb encrypted with the server's public key to prove his identity. Figure 2 shows the steps of the protocol.

A pair of FSMs, one for each communicating party, specifies a protocol. State activities and state transitions are coordinated between these two FSMs based on message exchanges. Each state machine keeps track of its internal shared state. The machine shared state holds common information needed by individual states such as: public or shared keys, user identities, and access permissions. The termination of the FSM or the final state is implicitly defined according to the sequence of execution.

We consider three types of actions carried out at a state before a transition to another state occurs: 1) execution of standard components (e.g., encryption, random number generation), 2) execution of user defined components (e.g., composing specific messages), and 3) conditional transition statements used to decide what state to go next based on the value of variables local to the state or variables global to the FSM.

3. Approaches to Protocol Specification

An essential concept in our work is the use of a domain-specific specification language in lieu of technical natural language for the specification of protocols, as in IETF RFCs. This approach facilitates automatic protocol implementation from specification.

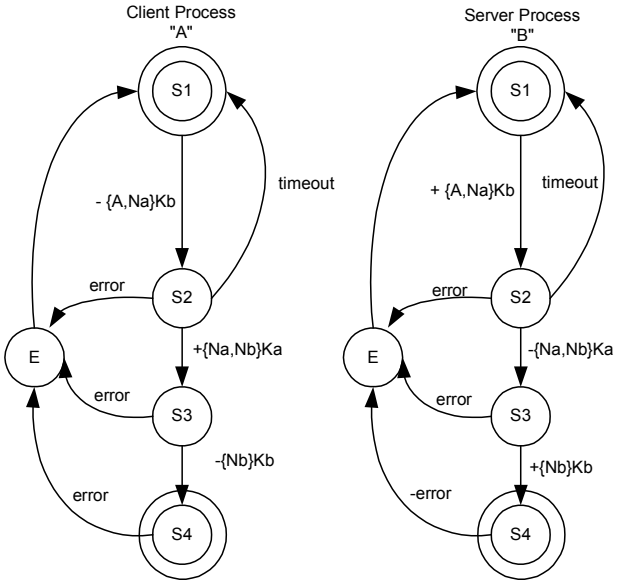

Figure 1. FSM for the Needham-Schroeder authentication protocol.

Needham-Schroeder Authentication:
A,B: Users
Na, Nb: Random numbers
Ka,Kb: Public Keys.

A->B: {A,Na}Kb
B->A: {Na,Nb}Ka
A->B:{Nb}Kb

Figure 2 Needham Schroeder authentication protocol.

Figure 3 shows the traditional manual production process for protocols. The process passes through three stages: design, verification, and implementation. The first two stages produce a specification document. Standard protocols (e.g., TCP/IP, FTP, SSL, and IPsec) specifications are written in natural English (e.g., RFC). Programmers must translate these documents into code.

In our approach (see Figure 4) the manual design and verification process is almost the same as in the traditional approach. However, the design process uses FSMs to produce protocol specifications in XML. These specification documents have to be well-formed XML documents and have to comply with the rules specified in the protocol XML schema. If both of these conditions are satisfied, the XSLT stylesheet automatically produces implementation for the protocol without human intervention.

Many different implementations in several programming languages can be developed for the same protocol specification.

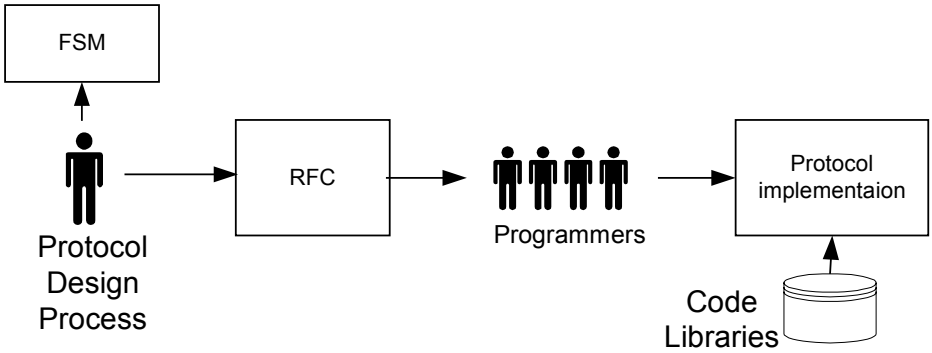

Figure 3. Manual protocol production process.

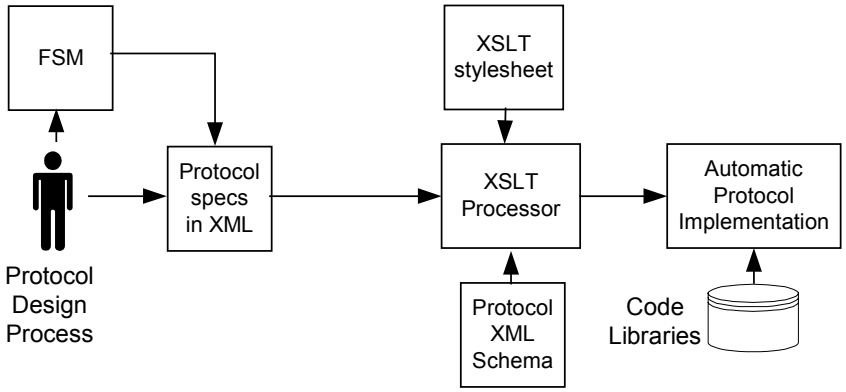

Figure 4. Proposed automated protocol production process.

4. XML Protocol Specifications

We chose XML as the language to express the specification of protocols for several reasons: first, it is an open standard; second, the tools for handling and processing XML documents and its related technologies are becoming very popular and available; third, to provide interoperability among systems, we decided to use a neutral language that is not associated with any specific programming language; fourth, the availability of XSLT technology makes it easy to transform an XML document into any user-preferred programming language.

Five types of XML elements are required to produce a protocol specification: name, first-state, object, instance, and state elements. Name is an optional single element used to identify the protocol; the XSLT processor can ignore or replace it with any other user-supplied identifier. All other elements are mandatory. The first-state, used to identify the starting state, is a single occurrence element because any protocol has a single point of entry for execution. All other elements may have multiple occurrences.

The *object* element (see Figure 5) is used to define data needed during protocol operations such as: messages, keys, constants, and random numbers. Every object consists of a name and a set of fields. Every field has a type attribute associated with its name type. Data types are predefined types, such as: Keys, Strings, Integers, and Booleans. As needed in the operations, these objects are transferred from one action to another as parameter passing. For example, to maintain the shared information among states a special object is used. In the example provided here, an object called session-state is used to hold this shared information. In other words, there is no public information unless it is defined as an object and passed as a parameter to the actions of a state.

```
<object>
  <name>Session_State</name>
  <field type="String">id</field>
  <field type="int">Na</field>
  <field type="int">Nb</field>
  <field type="key">PrivateKey</field>
  <field type="key">PublicKey</field>
</object>
```

Figure 5 – Object element example.

The *instance* element is similar to the concept of instance in object-oriented programming; objects represent definitions of classes and instances are the actual

realization of the definition. An object may have several instances, as needed, but every instance represents one object and there is no shared information among instances. For example, an object called error-message can be used to create several instances of error messages, as needed in the protocol states. Every error message instance can hold different error codes and an identification of the component that generated the message. The following listing shows two of the NSAP objects.

The *state* element is used to describe the different states of the protocol. Every state consists of a set of action elements and every action element corresponds to a component that is to be executed. Every action is a call to the interface of the required operation. In addition to the set of actions, a state has a *moveto* element. This element is similar to a switch-case statement in a procedural programming language. This element provides the mechanism that controls the transitions from one state to another. The *moveto* element consists of a logical expression and a list of cases. The logical expression is evaluated to produce a set of predefined values. Corresponding to each value, there is a *case-element* to point to the next state. Figure 6 shows the XML syntax for the first state in NSAP.

```
<state>
  <name>state3</name>
    <arg name="SS">
      <type>SessionState</type>
    </arg>
    <arg name="e">
     <type>CompLib.Error_message</type>
    </arg>
    <action>CompLib.ComposeClientMsg
           (SS.Nb,"ClientConfirm.txt")
    </action>
    <action>CompLib.Encrypt
           ("ClientConfirm.txt",
       "ClientConfirm.enc","myPubl")
    </action>
    <action>CompLib.SendMsg(2)
    </action>
    <action>e.code=1</action>
    <moveto>
    <expression>e.code</expression>
      <case test="1">Successful_end
         </case>
      <case test="2">ErrorState(e)
         </case>
    </moveto>
</state>
```

Figure 6. XML specification of the first state for NSAP.

Forcing XML protocol rules is the job of the XML schema, which is the mechanism to validate the syntax of the protocol specification. For example, the schema will prevent developers of protocol specification from introducing illegal elements. The schema also helps developers find any missing mandatory elements, such as the *first-state* element, before they run the transformation.

5. Automatic Code Generation

We used XSLT to transform protocol specifications, given as an XML document, into implementation code. We developed an XSLT stylesheet to produce Java code for any protocol. Every element in the XML document causes the XSLT processor to produce the required code. In our experiments we used Sun XML Pack, which includes an XSLT transformer and XML Schema checker. XSLT rules and filters are used to identify different XML elements, convert them to their correspondent program code, and place them in the proper XSLT output tree. Every programming language requires the design of its own XSLT transformation sheet. Once that sheet is available, any protocol represented as an XML document can be transformed to that language and compiled at the appropriate system.

As a result, any system on the Internet will be able to automatically generate protocol code as and when needed. Every system that has the proper XSLT stylesheet will be able to download the XML specification and use the stylesheet to produce the code in the preferred programming language, compile it, and run it. In addition to the XML specification for NSAP, we also developed XML specifications for SSL's handshake, and TCP's three-way handshake. The same Java XSLT stylesheet was used to automatically generate code for these three protocols.

This code was compiled and thoroughly tested to demonstrate the effectiveness of the approach described in this paper. The XML specifications, the Java XSLT stylesheet, and the XML Schema that was developed for validation of the XML specifications are at http://mason.gmu.edu/~iabdulla/.

The model presented in this paper depends highly on CBSE. Therefore, we have developed a library that comprises all the operations needed to run these experiments. The library contains about 50 components and 9 data objects. All are written in Java. The user does not have to know the internal details of the design of these components. The interfaces of these components are listed along with a brief description on each component at our web site (http://mason.gmu.edu/~iabdulla/.)

6. Related Work

Protocol implementations can be derived automatically from abstract specifications in languages such as SDL [3], Esterel [4, 5], Estelle [6, 7], LOTOS [8, 9], Promela++ [10], SMURPH [11], and Cicero [12]. However, these languages follow a low level procedural programming paradigm. Therefore, protocol developers need to work

out all the details of the operations of the protocols [13]. Our approach, on the other hand, capitalizes on CBSE to produce a high level specification. Therefore, most of the details of the operations of a protocol are hidden inside the components.

In the context of protocols, the term Automatic Protocol Generation (APG) means designing the protocol in an automated way [14, 15, 16, 17]. The process of APG focuses on formal methods. It takes as input a set of requirements and produces a set of proposed protocol designs. Our approach is different from APGs; we take a design (e.g., the output of the APG process) and convert it into a specification that generates an executable code for protocols.

The research in [18, 19, 20, 21, 22, 23, 24, 25, 26] is similar to our research in their mechanisms for producing implementations for network communication protocols from components. Their objective is the flexibility of configuring and customizing generic communication protocols. There are three differences between our approach and the work just mentioned. First, the aforementioned research does not provide a mechanism to share the configuration information of a newly implemented protocol. Second, these approaches do not provide a precise method to express the specification of a protocol. Finally, the proposed solutions are either associated with a single layer of the network stack as in [19, 20, 24, 25], or they replace the entire network stack, as in [18, 21, 22, 23, 26].

The Interface Definition Language (IDL) also provides a specification language that describes the requirements of accessing objects. The IDL forms a contract among the client, the object, and runtime environment with respect to the number of methods, their order, and the number, type, order, and direction of the method parameters. The language used in IDL is neutral but limited to producing code that facilitates accessing objects over networks. IDL also requires special purpose compilers to produce code [27].

7. Concluding Remarks

Automatic code generation is not an easy process. It requires enough knowledge about the domain of the problem, enough knowledge about the detailed design, and enough knowledge about the implementation [28]. Our approach was successful because we limited the domain of the problem to the area of protocols. In addition to that, we separated the implementation from the design of the protocol to further reduce the complexity of the problem.

In this paper, we have focused on the specification aspect of a protocol and on the transformation process of the specification into an implementation. We have successfully developed a language based on XML and FMS rules to describe any protocol. We also successfully developed an XSLT stylesheet that transforms protocol specifications given as an XML document to Java code. Developing additional XSLT stylesheets to produce other programming language code is straightforward.

Code that is produced by automatic means usually is considered less efficient than that produced manually. However, in our case, this effect is limited because our approach is based on component-based technologies. XSLT stylesheets are used to produce the control flow (skeleton) of the implementation; actions are produced through CBSE. Therefore, the efficiency of the code produced does not suffer much from automation. Further code optimization can be accomplished by fine-tuning the code of the components.

Finally, we are currently looking at the integration of ASN.1, a language often used in IETF RFCs to describe protocol messages and how they appear over the wire, into our XML specification.

Acknowledgment

The research of Ibrahim Abdullah was sponsored by King Abdulaziz University and by the Ministry of Higher Education, Saudi Arabia.

References

[1] S. Perrin, E. Benoit, and L. Foulloy, Automatic Code Generation based on Generic Description of Intelligent Instrument, *IEEE Int'l Conf. Systems, Man, and Cybernetics*, Vol. 6, 2002, pp. 569-574.

[2] C. Liu and K. Su, An FSM-Based Program Generator for Communication Protocol Software, *IEEE Proceeding of the 18th Annual International Computer and Application Conference*, Taipie, Taiwan, 1994, 181-187.

[3] F. Belina and D. Hogrefe, The CCITT Specification and Description Language SDL, *Computer Networks and ISDN Systems*, 16, 1989, 311-341.

[4] G. Berry and G. Gonthier, The Esterel Synchronous Programming Language: Design, Semantics, and Implementation, *J. of Science of Computer Programming*, 19, 1992, 87-152.

[5] G. Berry, The Semantics of Pure Esterel, http://citeseer.nj.nec.com/berry93semantics.html.

[6] J. Thees, Protocol implementation with Estelle-from prototypes to efficient implementations, *Proc. Int'l. Workshop on the Formal Description Technique Estelle*, France, 1998.

[7] A. Jirachiefpattana, and R. Lai, A Rapid Protocol Prototyping Development System, *Proc. IEEE Int'l Workshop on Rapid System Prototyping*, Chapel Hill, NC , USA 1995, 118-124.

[8] G. Leduc and F. Germeau, Verification of Security Protocols Using LOTOS-method and Application,

Computer Communication- Elsevier Science, 23, 2000, 1089-1103.

[9] T. Bolognesi and E. Brinksmaa, Introduction to the ISO Specification Language LOTOS, *Computer Networks and ISDN Systems*, 14, 1987, 25-59.

[10] A. Basu, G. Morrisett, and T. Von Eicken, Promela++: a language for constructing correct and efficient protocols, *Proc. INFOCOM 98 17th Annual Joint Conf. IEEE Computer and Communications Societies,* San Francisco, USA, Mar-Apr 1998, 455-462.

[11] W. Dobosiewicz and P. Gburzynski, Protocol Design in SMURPH, http://citeseer.nj.nec.com/138853.html

[12] Y. Huang, and C. Ravishankar, Cicero: A Protocol Construction Language, Tech. rep. CSE-TR-171-93, Dept. EECS, The U. of Michigan Ann Arbor, Michigan, 1993, 1-39.

[13] C. Castelluccia, W. Dabbous, and S. O'Malley, *Generating Efficient Protocol Code from an Abstract Specification, ACM Tr. Networking*, 5(4), Aug 1997, 514-524.

[14] A. Perrig and D. Song, Looking for diamonds in the desert-Extending Automatic Protocol Generation to Three-Party Authentication and Key Agreement Protocols, *Proc. 13th IEEE Computer Security Foundation Workshop,* Cambridge , UK 2000, 64-76.

[15] A. Perrig and D. Song, A First Step toward Automatic Generation of Security Protocols, http://citeseer.nj.nec.com/284940, 2000.

[16] D. Song, S. Berezin, and A. Perrig, Athena: a novel approach to efficient automatic security, *J. of Computer Security*, 9(1/2), 2001, 47-74.

[17] R. De Silva, L. Dairaine, and A. Seneviratne, A Comparison of Automatic Protocol Generation Techniques, *Australian Computer Journal*, 28(2), 1996, 55-60.

[18] S. O'Malley and L. Peterson, A Dynamic Network Architecture, *ACM Tr. Computer Systems*, 10(2), May 1992, 110-143.

[19] M. Jung and E. Biersack, A Component-Based Architecture for Software Communication Systems, *Proc. IEEE ECBS*, Edinburgh, UK, April 2000, 36-44.

[20] H. Huni, R. Johnson, and R. Engel, A framework for Network Protocol Software, *Proc.* of the tenth annual conference on Object-oriented programming systems, languages, and applications, New York, USA, 1995, 1-14.

[21] N. Hutchinson and L. Peterson, The x-kernel: an architecture for implementing network protocols, *IEEE Tr. Software Enginering*, 17(1), 1991, 64-76.

[22] N.T. Bhatti and R. Schlichting, A System for Constructing Configurable High-Level Protocols, *Proc. Conf. Application, Technologies, Architectures, and Protocols for Computer Communications*, 1995, 138-150.

[23] N.T. Bhatti, A. Hiltunen, R. Schlichting, and W. Chiu, COYOTE A System for Constructing Fine-Grain Configurable Communication Services, *ACM Tr. Computer Systems*, 16(4), Nov. 1998, 321-366.

[24] P. McDaniel, A. Prakash, and P. Honeyman, Antigone: A Flexible Framework for Secure Group Communication, *Proc. 8th USENIX UNIX Security Symposium*, Washington DC, USA, August 1999, 99-114.

[25] P. McDaniel and A. Prakash, Ismene: Provisioning and Policy Reconciliation in Secure Group Communication, *Electrical Engineering and Computer Science, U. of Michigan*, Technical Report CSE-TR-438-00, Dec. 2000.

[26] G. Wong, M. Hiltunen, and R. Schlichting, A configurable and Extensible Transport Protocol, *Proc. IEEE INFOCOM 20th Annual Joint Conf. IEEE Computer and Communications Societies*, Anchorage, AK , USA, 2001, 319-328.

[27] M. Gudgin, *Essential IDL interface Design for Com* (Boston, Addison Wesley, 2001).

[28] R. Glass, Some thoughts on Automatic Code Generation, *ACM SIGMIS Database*, 27(2), Spring 1996, 16-18.