Title: The Meta-Protocol Framework

Author: Ibrahim S. Abdullah<!–<query id="Q1">Please confirm that given names and surnames have been identified correctly.</query>–> Daniel A. Menascé

PII: S0164-1212(13)00138-6
DOI: http://dx.doi.org/doi:10.1016/j.jss.2013.05.096
Reference: JSS 9176

To appear in:

Received date: 15-8-2012
Revised date: 26-5-2013
Accepted date: 26-5-2013

Please cite this article as: Abdullah, I.S., Menascé, D.A., The Meta-Protocol Framework, *The Journal of Systems & Software* (2013), http://dx.doi.org/10.1016/j.jss.2013.05.096.

Highlight - The Meta-Protocol Framework

- The Meta-Protocol is framework for the negotiation, distribution, and automatic code generation of communication protocols.
- The goal of the Meta-Protocol framework is to provide a high level of flexibility to handle evolution of communication protocols
- The Meta-Protocol uses FSM-based description of a protocol to dynamically negotiate and generate code for communication protocols.
- Areas of applications of the Meta-Protocol are: QoS, security protocols, Web-services, B2B protocols and overlay networks.

The Meta-Protocol Framework[☆]

Ibrahim S. Abdullah[a], Daniel A. Menascé[b]

[a]*Department of MIS, King Abdulaziz University, Jeddah, Saudi Arabia, iabdullah@kau.edu.sa*
[b]*Department of Computer Science, George Mason University, Fairfax, VA 22030, USA, menasce@gmu.edu*

Abstract

A communication protocol is a set of rules shared by two or more communicating 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. To mitigate these problems, we divided the expression of a protocol specification into two parts. First, we designed an XML-based protocol specification language (XPSL) that allows for the high-level specification of a protocol—expressed as a Finite State Machine (FSM)—using Component-Based Software Engineering (CBSE) principles. Then, the components required by the protocol are specified in any suitable technical language (formal or informal). In addition, we developed the multi-layer Meta-Protocol framework, which allows for on-the-fly protocol discovery and negotiation, distribution of protocol specifications and components, and automatic protocol implementation in any programming language.

Keywords: Meta-Protocol, XPSL, automatic code generation, security, communication protocol, XML, XSLT, CBSE, UDDI, ISAKMP

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 when messages are received or when other events (e.g., timeouts) occur. 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, which is a time-consuming effort due to the timing dependencies of events processed by protocols.

TCP/IP, the widely used Internet protocol stack, was developed in the mid 1970's with the support of the Defense Advanced Research Projects Agency (DARPA) of the

[☆]The Meta-protocol framework is described in U.S. patent number US 8,086,744 B2, issued on 27 December 2011

U.S Department of Defense. The first full implementation of TCP/IP in Berkeley (BSD) UNIX was also funded by DARPA in 1983. The protocol has several features that led to its widespread adoption: open standard, freely available for developers, and independent of any specific physical network hardware. The first two features led to TCP/IP's wide acceptance and the latter feature made it easy for different kinds of networks to interoperate. With the adoption of TCP/IP the Internet has seen an exponentially growth.

The TCP/IP stack consists of four layers: the network access layer, the Internet protocol layer (IP), the transport layer, and the application layer. The network access layer is located at the bottom of the architecture. It consists of many protocols that provide access to physical networks. Compared to the Open System Interconnection (OSI) seven-layer reference model, the network access layer comprises two layers: data link and physical. The IP layer is responsible for defining the Internet addressing scheme, composing datagrams, and routing them to their destination. IP is a connectionless protocol. It does not provide handshake or reliability mechanisms. Therefore, it depends on other layers to provide such services. The transport layer consists of two protocols Transmission Control Protocol (TCP) and User Datagram Protocol UDP. TCP provides connection management reliability, flow control, and sequencing. On the other hand, UDP serves as a simple interface between applications and IP. UDP does not provide reliability, flow-control, or error-recovery. The application layer is at the top of the stack. It includes protocols that use the transport layer to deliver data to the network. Many protocols run at the application layer to provide services such TELNET, HTTP, SMTP, and FTP.

We developed a framework, called the *Meta-Protocol*, by which two communicating parties can exchange machine-readable protocol specifications and have the protocol implementation automatically generated just-in-time for the communication. To mitigate the problems of depending on natural languages to specify protocols, we divided the expression of a protocol specification into two parts. First, we designed the *XML-based Protocol Specification Language (XPSL)* to express, at a high-level, protocols that can be specified through Finite State Machines (FSM). The specification is also based on Component-Based Software Engineering (CBSE) principles. Then, the components required by the protocol are specified in any suitable technical language (formal or informal).

Therefore, our approach replaces the use of natural language for protocol specification with XPSL. This approach has the following advantages: 1) the specification can be read by machines to produce implementations in any programming language through the use of Extensible Stylesheet Language Transformations (XSLT), 2) the potential for human errors is reduced, 3) XPSL is based on XML, which is a neutral language that is not associated with any specific programming language, and 4) protocol specifications in XPSL 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 and for quickly generating protocol implementations for evaluation by standardization bodies. The XPSL specification of the new version is downloaded and code is generated locally using XSLT in the preferred local language. Our approach requires that the executable code of the Meta-Protocol framework be pre-installed on the machines of the communicating parties that need to use it to negotiate and agree upon a protocol. It is however possible for communicating parties to download a version of the Meta-Protocol expressed in XPSL and generate the code for it automatically using the

XSLT transformation described here. The validation section of this paper shows a partial demonstration of this capability when it describes experiments that automatically generate the ISAKMP base exchange protocol.

The rest of the paper is organized as follows. Section two presents some motivating examples. Section three presents the Meta-Protocol framework. Section four describes our approach to protocol specification. Section five presents the mechanism for automatic code generation. The next section describes the proof-of-concept experiments. Section seven describes related work. Finally, section eight presents concluding remarks.

2. Motivation

The goal of the Meta-Protocol framework is to free protocols from being dependent on the monolithic implementation of a limited number of capabilities. Communication protocols in general, and security protocols in particular, need a higher level of flexibility to handle evolution. Our approach increases the level of code sharing and reusability.

A motivating example in the arena of communication protocols is in Quality of Service (QoS). Several studies tried to improve QoS by developing highly configurable transport protocols that support multimedia and collaborative traffic [9, 11, 25, 37]. An exclusive reliance on UDP or TCP does not match the requirements of all types of applications [9]. Therefore, selecting a protocol specification that fits application needs and generates code automatically provides multimedia and collaborative systems a wide range of opportunities to improve QoS.

B2B transactions are based on protocols such as the ebXML standard, which consists of an architecture, message formats, and business processes [18]. Business process protocols need to be shared by the parties conducting business transactions. Since business requirements and business partnerships evolve over time, these protocols need to change as well. Our framework builds a flexible infrastructure for businesses to change protocol specifications, share them, and automatically generate code. Similarly, in collaborative environments, different groups may have different communication requirements such as different security levels. According to our framework, each group may share a different protocol specification and communicate according to their specific needs.

Another motivating example can be found in the implementation of security protocols. Virtually every protocol provides a limited set of services. For instance, TLS v.1 has 32 modes of operation [27]. Adding new modes of operation requires releasing a new version of the protocol. Another class of limitations is related to the sequence of operations. For instance, TLS performs MAC before encryption. This design choice has no significant effect on the strength of the protocol. In fact, IPsec does the opposite. This choice is hard-coded in the design of TLS as well as in its implementations. Such design choices should be left to the applications that use the protocols.

The Meta-Protocol framework can also be applied to overlay networks and Web Services. In overlay networks, the Meta-Protocol framework allows hosts to negotiate and run new communication protocols other than the standards. Similarly, for Web Services, the Meta-Protocol framework provides a mechanism to control the sequence of Web Services that need to be composed to complete a task. An example would be a sequence of Web Services starting with a purchase order followed by a payment order followed by a shipping order.

3. The Meta-Protocol Framework

The Meta-Protool framework is applicable to most kinds of communication protocols. However, it is more useful for end-to-end protocols. This is due to the fact that end-to-end protocols typically execute in environments with more computational resources and privileges to access wider Internet resources than other lower level protocols such as packet routing and forwarding protocols.

The objective of the framework is twofold: first, it provides a systematic process of coordination between communicating parties in order to reach an agreement on the specification of the protocol they want to adopt during the course of communication. Second, it manages the process of generating the executable code of the chosen protocol. The Meta-Protocol framework was designed to be simple and straightforward because it is the root of all other protocols that it produces. In addition, the framework has to be lightweight and robust in order to minimize the performance hit that might result from producing protocols on-the-fly as needed.

The framework is based on three technologies: FSM, XSLT and CBSE. A Finite State Machine (FSM) is a computational model used in many software and hardware applications [10]. It is used to model many types of problems in the fields of electronics, computing, and communications. An FSM is a directed graph that represents the behavior of a machine that can move from a state to another based on certain triggers or conditions. The Extensible Markup Language (XML) is a set of rules for creating new markup languages. Originally, XML is a subset of the Standard Generalized Markup Language (SGML), which specifies syntactic and semantic rules for creating new markup languages such as the Hypertext Markup Language (HTML). SGML is an ISO standard (ISO 8879). XML 1.0 was announced as a World Wide Web Consortium (W3C) recommendation on Feb 10, 1998 [39]. The goal of XML is to provide a language for expressing metadata. Metadata refers to information about data. Metadata is important in searches, filtering, and document management. XML is intended to be human-readable and machine-readable. A document that conforms to the XML specification is called well-formed. XML editors and parsers typically provide the capability to check if an XML document is well-formed.

The XML specification consists of two parts: one specifies the rules for constructing XML documents and the other for Document Type Definitions (DTD). A DTD specifies how tags are used in a class of XML documents, the order in which they should appear, the nesting structure of a document, and the attributes of XML elements for that class of documents. A DTD describes constraints on XML elements that are used to validate the correctness of an XML document. A DTD has two major disadvantages. First, it is not written in XML, it has a special syntax. Second, a DTD has a limited expressive syntax to address needs such as different kinds of data types and cardinality constraints. XML Schemas were introduced to overcome the limitations of DTDs.

The W3C released the Schema standard in May 2001 [40]. An XML schema specifies valid elements and attributes in an XML document. It also specifies XML elements order, attribute constraints, accepted data types, and accepted value ranges. A schema helps applications determine whether an XML document complies with the requirements of a system. The Extensible Stylesheet Language (XSL) is a language that allows XML document users to specify how an XML document should be transformed into a resulting output document (e.g., HTML, SVG, PS, PDF, plain text, or any other format). The

XSLT 1.0 recommendation was released on November 16, 1999. XSLT is attractive to developers because it is not a programming language. An XSLT stylesheet is written down as a set of rules. This set of rules is applied to the XML document during the transformation to produce the required output [41].

The idea of modular or component-based design approach is common in many engineering fields and industries. In software engineering, Component-Based Software Engineering (CBSE) improves the efficiency of the software development process, reduces maintenance costs, and enhances the quality of the resulting product [42]. CBSE has been adopted by many vendors in technologies such as CORBA, COM+, JavaBeans, and Software Agents.

A software component is defined as a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard. The component model mentioned in this definition refers to the interaction and composition standard. The interaction aspect of the model clearly defines the interface of the component. The composition aspect defines the deployment of the component in terms of installation, configuration, and instantiating [42]. Other definitions have been discussed in [43]. Each definition emphasizes some aspects and leaves out others. In any case, the important characteristics identified by these definitions are: independency, interface, context, relationships, and the architecture of the component.

The interest in CBSE has permeated numerous projects across all application domains. Examples include IRIDIUM (Motorola's global cellular communication system), MCI's SONET network management system, AT&T's "ANGEL" wireless nation-wide fixed-wireless network, and the CUI direct nation-wide customer service support system [44]. Other signs of interest in CBSE are reflected by the growing standardization effort of CBSE technologies (e.g., TMN, CORBA, EJB, and COM). CBSE is becoming a dominating topic at software engineering conferences and workshops. There is a continuing effort in developing CBSE technologies such as design patterns, design languages, software architectures, and implementations such as ActiveX, COM, and JavaBeans [43, 44].

The Meta-Protocol framework consists of four layers: protocol negotiation, distribution of protocol specifications and components, automatic implementation of protocols, and protocol execution (see Fig. 1) In the negotiation layer, the communicating parties agree upon a protocol specification. In the distribution layer, the responding party locates the specification, written in XPSL, as well as any components needed for the protocol implementation. The specification may exist locally or at a remote site. The responder retrieves the specification and verifies its authenticity. In the implementation layer, the protocol specification is automatically converted into executable code through a transformation process that uses eXtensible Stylesheet Language Transformations (XSLT). In the execution layer, the code is loaded and executed as needed.

In the first two layers, negotiation and distribution, the framework focuses on coordination between communicating parties on the specification of the protocol they want to use. At this stage, they establish a trust. If they cannot not reach an agreement, they do not have a trust relationship and the communication stops. The parties involved in the distribution layer may need to access a third party to collect the specification of the protocol or some of its components if they do not exist locally. Another trust relationship has to be established between the third party and the communicating parties. Otherwise, the communication stops. These two aspects of trust help improve the security of the

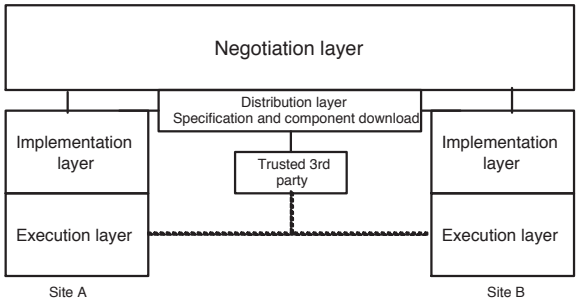

Figure 1: Layered structure of the Meta Protocol framework.

communication.

The other two layers, implementation and execution, perform local site tasks. The function of the implementation layer is to transform the specification into executable code. The specification of a communication protocol is written in a high level language (e.g., the XML-based XPSL language) and based on components. Components are hosted locally or remotely. In either case, the system is able to locate the required components and use them to build the executable code of the protocol. The role of the execution layer is to manage the execution of the code generated by the framework.

The operation of the Meta-Protocol framework is triggered by a set of messages exchanged between the communicating parties as described in Fig. 2. The initiator sends a proposal message. This message contains among other things the name of the proposed protocol and its location. A naming convention has to be strictly adhered to; protocol names should be considered unique similarly to that of web sites. Each responder has a policy to decide on the trusted protocols, sites, and parties. According to this policy,

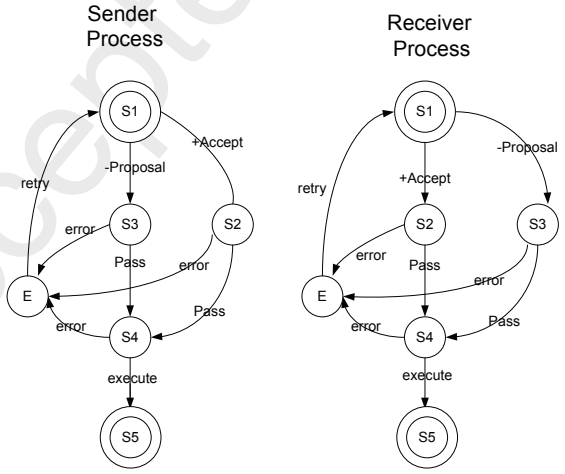

Figure 2: The Meta-Protocol operation.

the responder may accept one of the proposed protocols, reject the communication and produce an error message, or propose his own preferred set of protocols to the initiator. If one of the proposals is accepted by both parties, the trust relationship is established and the process continues to the next state. Otherwise, the communication stops between the parties and an error is generated. In the next state (either S2 or S3) the system checks the proposal and collects the specification of the proposed protocol (an XML file) and generates an executable code for the communication protocol. The process of generating a protocol's executable code involves collecting any missing components from trusted sites. If the process of generating the executable code is successful, the process moves to state S4. In that state, the executable code is run and the process moves to the final state.

4. The Meta-Protocol Framework Implementation

The four layers of the framework discussed above can be implemented in many ways. We have adopted as building blocks several standard technologies in order to benefit from the experience of standardization committees. The following sections describe the requirements of the framework and the technologies chosen in its implementation.

4.1. Framework Requirements

The basic requirements that make the Meta-Protocol framework achieve its objectives are: a generic negotiation mechanism to determine a protocol specification, a delivery mechanism for retrieving the protocol specification and any missing components, a searchable repository system to store specifications and components, a mechanism to transform a protocol specification into executable code, a system that manages executable code by loading it and running it as needed, and security measures against attacks such as identity theft, Denial of Service (DoS), anti-replay, and connection hijacking.

The layered approach shown in Fig. 1 is necessary to maintain flexibility. We found natural boundaries between the layers according to the technologies chosen at each layer. In fact, each layer may encompass multiple choices of technologies that fit the requirements of that layer. For example, off-line or online approaches may be used at the negotiation layer. Moreover, the security measures needed at the negotiation layer are different from those needed at the distribution layer.

4.2. The Negotiation Layer

The goal of this layer is to establish an agreement between two parties on the specification of a communication protocol. The initiator of the communication provides a set of proposed protocol specifications. The responder selects one according to its preferences and notifies the initiator. The goal of this layer can be achieved in several ways: manual, secure channel (e.g., SSL or IPsec), or using the Internet Security Association and Key Management Protocol (ISAKMP). The manual approach while feasible for local areas is not appropriate for long distances. Negotiation mechanisms would need to be added if secure channels are used. Therefore, a customized version of ISAKMP was chosen for this layer because it provides a negotiation mechanism in addition to several important security features.

ISAKMP provides packet formats and negotiation procedures for establishing security associations (SA) and key management [28]. In ISAKMP, for the purpose of maintaining interoperability between systems, security association management is cleanly separated from key exchange. In addition, the ISAKMP standard is independent of any specific security protocol or key exchange algorithm. Therefore, the standard is highly flexible and interoperable with other protocols and algorithms. We customized IPsec's ISAKMP Domain of Interpretation (DOI) to save time and effort. Therefore, we use the IPsec format with types of information tailored to the needs of our framework. For example, IPsec uses ISAKMP proposals to deliver information related to the IPsec type of operation and transforms (e.g., AH or ESP). In our framework, ISAKMP proposals are used to deliver information related to protocol name, location, and versions.

Figure 3 shows an example of an ISAKMP packet sent by an initiator to deliver a protocol proposal composed of a protocol specification name, its location, and version. The name and location of the protocol specification are mandatory and the protocol version is optional. In this example, the name (*Protocol_X.*) and versions (`MjVer` and `MinVer`) are placed in the first payload while the location goes in the next payload. An initiator may send multiple proposals linked to a main ISAKMP header using the Next Payload field. The list is terminated by a null value in the Next Payload field.

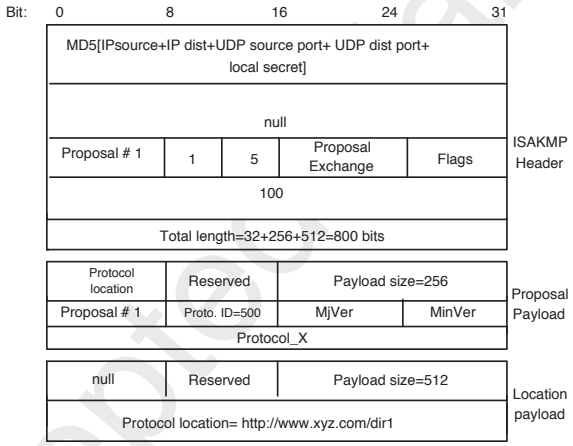

Figure 3: Example of an ISAKMP header carrying a proposal payload

The initiator also has the option to append to the packet a key exchange payload carrying information about the proposed key exchange technique (e.g., Oakley or Diffie-Hellman) [33]. The key exchange also carries the required data to generate session keys.

It is important to note that an XPSL specification of ISAKMP is the bootstrap of the Meta-Protocol framework. In other words, this is what is required from any network node to use the framework, since the code for ISAKMP can be generated from its XPSL specification.

4.3. The Distribution Layer

XPSL specifications and components do not need to be stored locally. Communicating parties may exchange them directly or obtain them from a trusted third party. Specifi-

cations and components that are only needed a short period of time may be deleted after their use to save space. To protect XPSL specifications and components from malicious alteration they have to be digitally signed by their authors and verified by the receiving party. Trusted providers may place their specifications and components at public repositories on the Internet so users may retrieve them when needed. Newer versions may also be pushed from these providers to subscribed users to keep their systems up-to-date.

The distribution layer uses UDDI, which is basically a directory service that provides advertising and discovery services [3]. Communicating parties in the Meta-Protocol framework use UDDI to advertise protocol names, the location of their specifications, the location of the components needed by the specification, and the preferred transport mechanism to download the specification and its components.

Using UDDI at the distribution layer also serves those users who are planning to start a negotiation but do not have specifications to propose to the other party. These users can consult UDDI directories to discover protocol specifications, their providers, the location of the specifications, and their components for download.

4.4. The Implementation Layer

This layer includes our novel approach to produce protocol implementations automatically from protocol specifications written in XPSL. The proposed approach is based on the eXtensible Markup Language (XML), on the eXtensible Stylesheet Language for Transformations (XSLT), and on Component-based Software Engineering (CBSE). XPSL is used to specify protocols described through Finite State Machines (FSM). XSLT is used to transform the specification 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 (e.g., encryption) needed by the protocol. Figure 4 shows the proposed automated protocol production process [1].

Every component used in an XPSL specification is an executable program designed based on CBSE principles. A component may be shared by more than one XPSL specification. For example, an RSA encryption algorithm is a single component. Such component may be used in a single protocol several times and may be shared by many protocols. On the other hand, a component may also comprise several subcomponents. For instance, a security envelope component consists of a header processing subcomponent, a MAC subcomponent, and an encryption algorithm subcomponent.

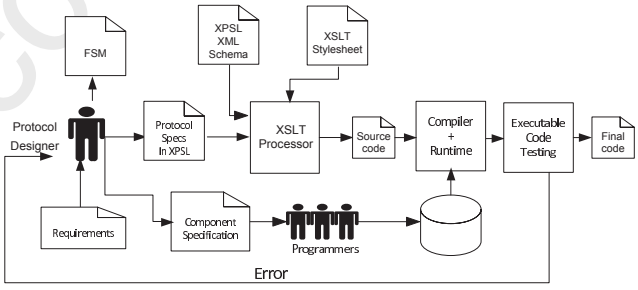

Figure 4: Automated protocol production process

The detailed technical description of the proposed technique is presented in sections 5 and 6, which explains how a protocol specification is written in XPSL, how the schema and the XSLT stylesheet are developed, and how they are used to produce a protocol implementation.

4.5. The Execution Layer

The execution layer plays a central and integrative role in the framework. This layer consists of a set of interfaces and control operations. The interfaces are responsible for delivering requests and feedback between the external world and the execution layer to, for example, deliver user requests to start a protocol or to terminate it. Control operations manage the internal affairs of the execution layer. Examples include authenticating a user to access a protocol, checking the availability of a protocol, loading or terminating a protocol.

5. Specifying Protocols in XPSL

This section describes our model for sharing protocol specifications to produce protocol implementations automatically.

5.1. Use of FSMs in Protocol Design

Finite state machines (FSM) are commonly used for describing protocols. 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 state transition. Before a transition occurs, a (possibly empty) set of actions are executed. One of the states in an FSM is designated the *initial state*. Figure 5 shows an FSM for the *Needham-Schroeder* authentication protocol *(NSAP)*, a two-way authentication protocol based on public key cryptography [30]. 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.

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 execution sequence.

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.

The traditional manual process for producing protocols (see Fig. 6) consists of three stages: design, verification, and implementation. The first two stages produce a specification document. The specification of standard protocols (e.g., TCP/IP, FTP, SSL, and

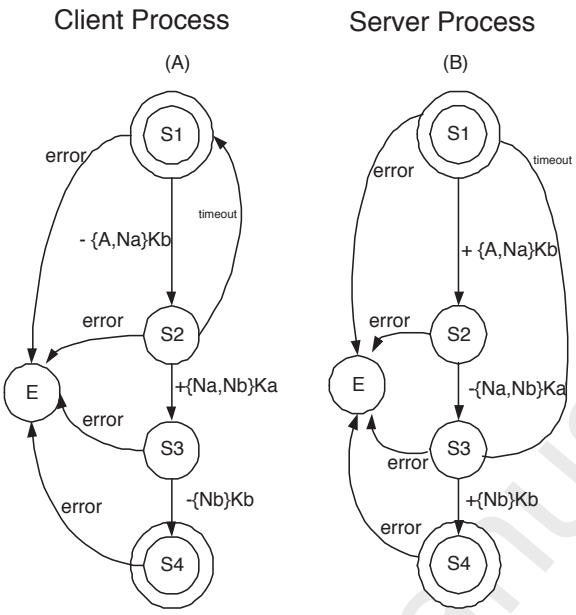

Figure 5: FSM for the Needham-Schroeder authentication protocol

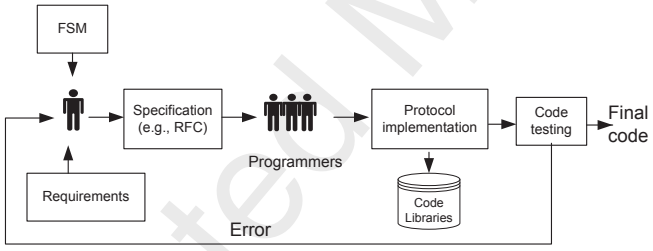

Figure 6: Manual protocol production process

IPsec) is written in natural language (e.g., English) and must be translated into code by programmers.

In our approach (see Fig. 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 XPSL. These specifications have to be well-formed XML documents and have to comply with the rules specified in the XML schema we designed (see section 5.3) for XPSL. If both of these conditions are satisfied, an XSLT stylesheet can be used to automatically produce an implementation of the protocol without human intervention. Implementations in several programming languages can be developed for the same protocol specification.

We chose XML as the basis for our language to express the specification of protocols because it is an open standard, XML tools for handling and processing XML documents and its related technologies are mature and widely available, XML is pro-

gramming language neutral and XSLT technology makes it easy to transform an XML document into any user-preferred programming language. Eight types of XML elements are required to produce a protocol specification in XPSL: <protocol>, <name>, <first-state>, <object>, <instance>, <moveto>, <state>, and <action>. A tree structure of these XML elements indicating the relationship among them is shown in Fig. 7. XML elements are shown in angular brackets and their attributes in square brackets.

An FSM is manually mapped into an XPSL specification (see Figure 8).

The process of writing down XPSL specifications follows the sequence:

1. Place the <protocol> element as the root of the XPSL specification document.
2. Add a <name> element as a child of the <protocol> element to specify the name of the protocol.
3. If needed, add a group of elements that prepare the environment (e.g., define public objects as <objects> or <instances>)
4. Add the <first-state> element to specify the starting state of the FSM.
5. For each state in the FSM insert a <state> element.
    - Inside each <state> element, add a group of <action> elements to perform the actions required by the state. See Fig. 9 for an example.
    - At the end of each state, add a <moveto> element to indicate the next state.

We have set up a Web site that contains techinical details about the XPSL specification and some experimental examples at www.metaprotocol.net. The root element is the <protocol> of the XML document and it is a mandatory element according to the XPSL definition. This element has an optional child element <name>, which can be used to identify the protocol. The <protocol> element is also the parent element of the following elements: <object>, <instance>, <first-state>, <action>, and <state>. The first two elements have a global scope and are used to prepare objects shared by the states of the protocol.

The <first-state> element is another mandatory element that identifies the starting state of the FSM and can occur only once in an XPSL specification given that an FSM can only have one initial state. All other elements may have multiple occurrences.

The <object> element is used to define data, such as messages, keys, constants, and random numbers, needed during the operation of a protocol. Every object consists of a name and a set of fields. Every field has a type attribute associated with the <name> element. Data types are predefined types, such as key, string, integer, and boolean. Objects are passed as parameters from one action to another. In the example of Fig. 9, an object called *SessionState* is used to hold information shared among states. 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.

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.

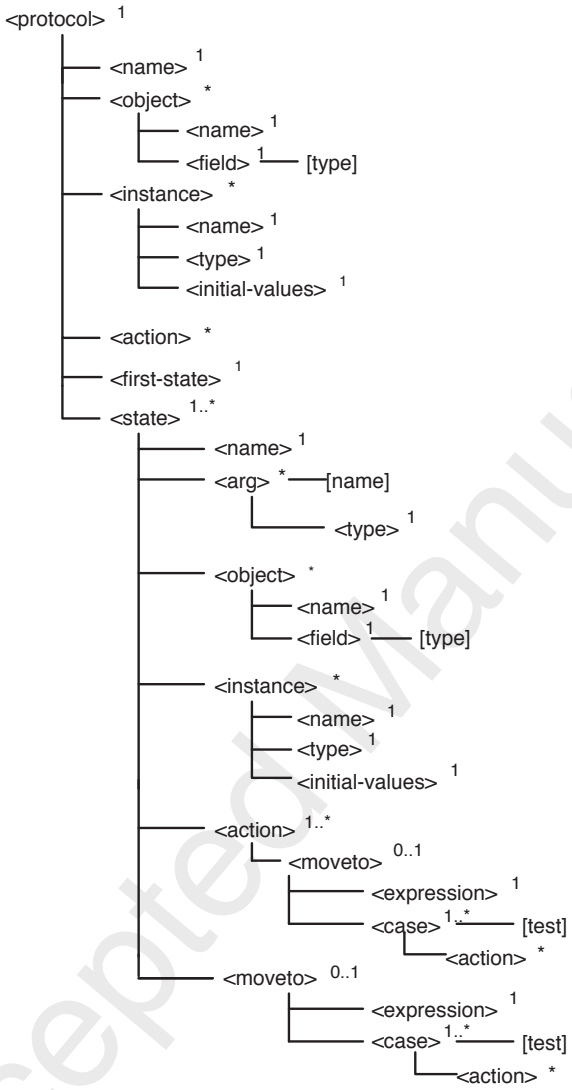

Figure 7: XML elements used to express protocol specification

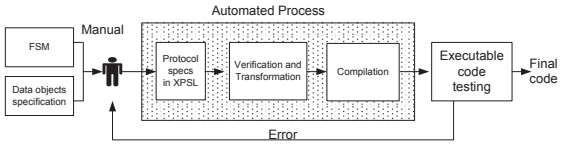

Figure 8: Protocol XML specification and implementation steps

```
<object><name>SessionState</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 9: Example of an <object> element

The <moveto> element is similar to a switch-case statement in a procedural programming language and 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. A <case> element points to the next state for each possible value of the expression. Figure 10 shows an example of the XML syntax for a <moveto> element.

The <action> element defines the actions (e.g., calls to components in a library) that take place inside a state. In some cases, elements such as <instance> and <moveto> may be nested inside an <action> element. These situations occur when a component needs an instantiation of a data object that was not passed to the state for temporary local usage.

The <state> element describes the different states of a protocol. Figure 11 shows the XPSL syntax for state 3 of the NSAP client. A <state> element cannot be nested inside another <state> element and every state in the FSM has to be described by exactly one <state> element in XPSL. Each <state> element has an optional child element called <arg>, which describes the external objects that are passed to the state. The <arg> element has a `name` attribute, the name of the object to be passed to the state, and must be followed by a <type> element, describing the type of the object.

A <state> element can have one of the following child elements: <object>, <instance>, <action>, and <moveto>. These elements express the operations that are performed inside every state of the FSM.

5.2. Sufficiency of the Specification Language

This section shows that the set of XML elements used in XPSL is *sufficient* to describe any FSM. We show below that all elements of an FSM are mapped into the XPSL specification and that the XPSL specification captures the behavior of an FSM in terms of state transitions. An FSM F is a tuple $(\mathcal{S}, s_0, T)$, where $\mathcal{S}$ is a set of states, s_0 is the initial state, and T is a set of state transitions defined as $T = \{(s, v, c, A) | s, v \in \mathcal{S}, c \in$

```
<moveto> <expression> logical-expression </expression>
            <case test="1">Successful_end </case>
            <case test="2">ErrorState(var)</case>
</moveto>
```

Figure 10: Example of a <moveto> element

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

Figure 11: XML specification for NSAP's client state 3

$\Omega, A \subseteq \Theta\}$. In a transition $(s, v, c, A) \in T$, s is the "from" state, v is the "to" state, c is a condition from the set of conditions Ω, and A is a set of actions from the set of possible actions Θ. The condition c is a boolean expression whose terms may include occurrence of events (e.g., arrival of a message and timeout) and comparisons including variables (e.g., window size) and constants. Examples of actions include composing and sending messages.

The behavior of an FSM is defined by the set of rules that determine how the FSM moves from state to state starting from the initial state s_0. This behavior can be described by the following algorithm:

$s \leftarrow s_0$
repeat
  if $\exists\ (s, v, c, A)\ \in T | (v \neq \text{null})\ \wedge\ (c = true)$
  then execute actions in A;
    $s \leftarrow v$;
  else stop
forever

This algorithm starts the execution from the initial state s_0, the current state s at the beginning. Then, all transitions (s, v, c, A) out of the current state are examined. The one for which the condition c is true triggers a state transition to state v. Before moving to the next state, all actions in the set A are executed. The next state then becomes the current state s. The process repeats itself until a terminal state (i.e., a state with no outgoing transitions) is reached.

Figure 12(a) shows the common pictorial view of an FSM, indicating the possible

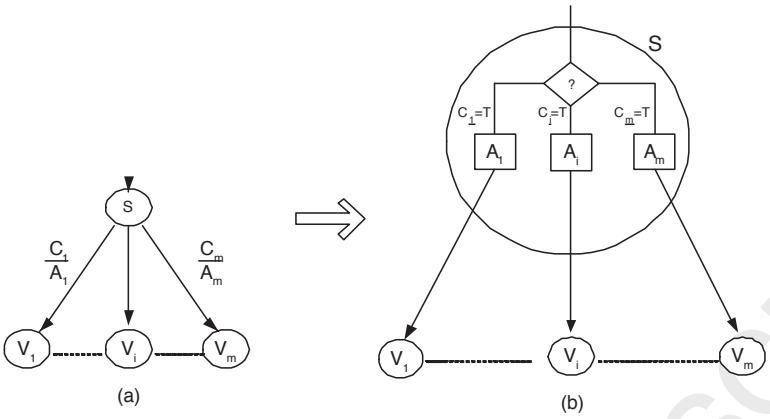

Figure 12: Transformation of an FSM to a behaviorally equivalent representation

transitions between state s and states $V_1, \ldots, V_m$. The labels alongside each transition show the condition c_i that triggers the transition and the set of actions A_i carried out when the transition takes place. Figure 12(b) shows a behaviorally equivalent pictorial representation. Here, the decision on what state to move to and the corresponding actions to execute are shown "inside" the state circle and not along the transition arc. The behavior is the same though. The design of XPSL is based on the pictorial representation of Fig. 12(b), which is equivalent to the FSM representation of Fig. 12(a).

Figure 13 shows the tree of XML elements designed to describe an FSM. The initial state s_0 of an FSM is mapped to the <first-state> element and every state in the FSM is mapped to a <state> element. Every <state> element has two child elements: a <name> element to identify the state and a <moveto> element that specifies the process of evaluating the conditions and executing the actions associated with each state. The <moveto> element has two acceptable types of child elements: a single <expression> element and a set of <case> elements. Each <case> element corresponds to a transition in the set T in the FSM in which the "from" state is the state identified by the <name> element.

An FSM has limitations for describing the behavior of programs [10]. Therefore, the XML tree shown in Fig. 13 needs to be extended to address the requirements imposed by the general XML specification and the process of generating executable code. To address these requirements, five elements were added to XPSL.

First, every XML document must have a root element. Therefore, the <protocol> element was added to the specification as the parent of all other elements. Second, an executable code needs to be identified by a name. Therefore, a <name> element was added to the specification to identify the generated executable code. This <name> element is placed as a child of the <protocol> element. Third, since an FSM does not explicitly express data objects, the <object> element was added to express the types and structure of the data objects involved in the FSM. Fourth, according to the object-oriented programming approach, instances are created out of data objects during the execution to hold the actual data. Therefore, an <instance> element was added to address this requirement. Fifth, to control the scope of access of objects in the different

states, an <arg> element was added as a child of the <state> element. The purpose of the <arg> element is to facilitate passing data objects to states. Figure 7 shows the entire XML tree for XPSL. This figure shows that <action> elements are allowed under the root element before the <first-state> element and also under the <state> element before the <moveto> element. Actions placed in these positions are common actions required by the protocol or by a state and result in reducing the repetition of shared code.

5.3. The XML Schema for XPSL

We designed an XML schema to define XML documents compliant with XPSL. This schema can be used to validate the syntax of the XPSL specifications. For example, a schema prevents developers of protocol specifications 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.

For each element or attribute in XPSL, there is a corresponding definition or constraint that controls the content of the specification. Figure 14 shows the main segment of XPSL's XML schema. The first two lines declare the XML and schema versions followed by the definition of the protocol root element. The root element contains a <name> element and five references. Each reference defines the allowed content and its cardinality. The `minOccurs` and `maxOccurs` constraints define how many times an element can appear in a protocol specification.

The definitions in Fig. 14 show that a protocol specification may have zero or more objects. Similarly, instances and actions may occur zero or more times. However, the <first-state> element has to appear exactly once. The <state> element must appear at least once, otherwise an error is signaled.

Figure 15 shows the definition of an <object> element. This definition indicates that if an <object> element exists, it must have a <name> element and at least one <field> element. The value in the field element indicates the name of the field. The field also has a required attribute that indicates its type. If the value or attributes are missing from the XPSL specification, the XML Schema checker raises an error. The permitted data types (e.g., string, int, byte) for a field are enforced by XML Schema checkers.

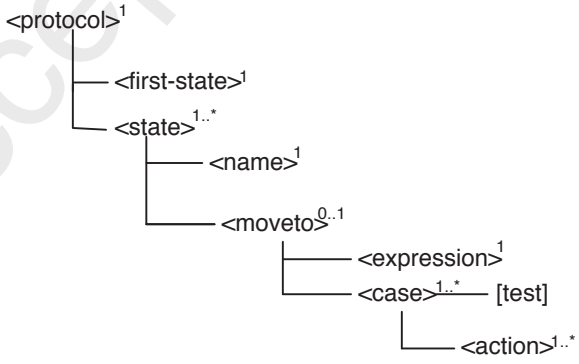

Figure 13: Tree of XML elements corresponding to an FSM

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd=
     "http://www.w3.org/2001/XMLSchema">
<xsd:element name="protocol"> <xsd:complexType>
    <xsd:sequence>
        <xsd:element name="name" type="xsd:string"/>
        <xsd:element ref="object" minOccurs="0"
                maxOccurs="unbounded"/>
        <xsd:element ref="instance" minOccurs="0"
                maxOccurs="unbounded"/>
        <xsd:element ref="action" minOccurs="0"
                maxOccurs="unbounded"/>
        <xsd:element ref="first-state" minOccurs="1"
                maxOccurs="1" />
        <xsd:element ref="state" minOccurs="1"
                maxOccurs="unbounded"/>
    </xsd:sequence>
</xsd:complexType> </xsd:element>
```

Figure 14: The main segment of the XML Schema for XPSL specifications

```
<xsd:element name="object">
<xsd:complexType> <xsd:sequence>
<xsd:element name="name" type="xsd:string"/>
<xsd:element ref="field" maxOccurs="unbounded"/>
</xsd:sequence> </xsd:complexType>
</xsd:element>
<xsd:element name="field">
<xsd:complexType><xsd:simpleContent>
<xsd:extension base="xsd:string">
<xsd:attribute name="type" use="required"/>
</xsd:extension>
</xsd:simpleContent></xsd:complexType>
</xsd:element>
```

Figure 15: The Schema definition of the <object> element

6. Automatic Code Generation

We used XSLT to transform protocol specifications written in XPSL into code. Every element in the XPSL specification causes the XSLT processor to produce the required code. We developed an XSLT stylesheet to produce Java code from any XPSL protocol specification. This stylesheet was successfully used on several XPSL protocol specifications. The resulting code was successfully compiled and thoroughly tested. In our experiments, we used Sun's XML Pack, which includes an XSLT transformer and an XML Schema checker. XSLT rules and filters are used to identify different XML elements, convert them to their corresponding program code, and place them in the proper

XSLT output tree. Every programming language requires its own XSLT transformation sheet. Once that sheet is available, any protocol specification written in XPSL can be transformed to that language and compiled in the appropriate system.

Figure 16 shows the steps required to generate a protocol implementation from an XPSL specification. Using this approach, any system on the Internet with the proper XSLT stylesheet can download an XPSL specification and use the stylesheet to produce code in the preferred programming language, compile it, and run it.

In the Java XSLT stylesheet, 16 templates were used to make the transformation from an XML specification to Java code. Some of these templates are illustrated here.

Testing the correctness of this stylesheet is similar to testing any software product. The popular use of XSLT stylesheets is for presentation purposes. Our use of XSLT for code generation explores the power of XSLT to generate source code.

6.1. The stylesheet structure

Producing Java code requires writing a main program and a set of classes. The main program is responsible for starting the execution. Classes correspond to either objects or states. Therefore, the first few lines in the stylesheet describe the XML version, the stylesheet version, the XSL name space, and the type of output (see Fig. 17). Then, the main template determines the structure of the stylesheet. This main template pushes to the output a header for the main Java program and then calls five other templates: /protocol/name, action, first-state, object, and state template. Figure 17 shows the declarative instructions and the main template of the stylesheet. Notice that Java statements are embedded inside the template as needed. The /protocol/name template is responsible for producing Java code for the identifier of the public class of the program which comprises all other classes. This section presents three examples of templates for the Java stylesheet: first-state, object, and instance.

6.2. The <first-state> template

This is a trivial template that produces the name of the entry point of the state machine. This template is called once from the main template of the stylesheet. Therefore, the main Java program only has one class to call, which is determined by this template (see Fig. 18).

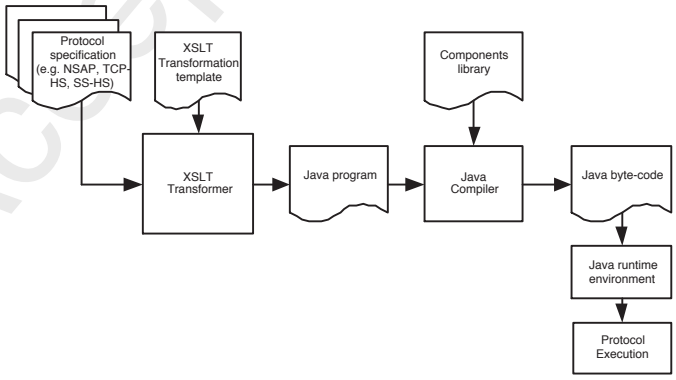

Figure 16: Steps to generate a protocol implementation from an XPSL specification

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" ?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html" indent="yes"/>
<xsl:template match="/protocol">
        <xsl:apply-templates select="/protocol/name"/>
        public static void main(String[] args)
        <xsl:apply-templates select="action" />
        <xsl:apply-templates select="first-state"/> }
        <xsl:apply-templates select="object" />
        <xsl:apply-templates select="state"/>
        private void vpGen() throws Exception {}}
</xsl:template>
```

Figure 17: XSLT stylesheet main template

```
<xsl:template match="first-state">
        <xsl:value-of select="."/> ;
</xsl:template>
```

Figure 18: The <first-state> template

6.3. The <object> template

This template produces code for data objects. As shown in Fig. 19, this template produces a Java class that defines an object interface and a construct for the instantiation of the data members. This template may be called from the main template or from any state. If it is called from the main template, the object becomes a public object. However, if it is called from within a state, it becomes local to that state.

```
<xsl:template match="object" >
        public static class
        <xsl:value-of select="name"/> {
        <xsl:apply-templates select="field"/>
        public <xsl:value-of select="name"/>
        ( <xsl:apply-templates select="field"
        mode="argdef"/>){
        <xsl:apply-templates select="field"
        mode="construct"/> }}
</xsl:template>
```

Figure 19: The <object> template

6.4. The <instance> template

In this template, the transformer pushes Java code that corresponds to instantiating an object. The data object name, type, and initial values are picked up from the XPSL

specification and arranged into the appropriate Java code. See Fig. 20 for an example.

```
<xsl:template match="instance" >
        <xsl:value-of select="type"/>
        <xsl:text>   </xsl:text>
        <xsl:apply-templates select="name"/> = new
        <xsl:value-of select="type"/>
        <xsl:apply-templates select="initial-values"/>;
< /xsl:template>
```

Figure 20: The <instance> template

6.5. Protocol Components

Every component has an interface that specifies its parameters and their types. This interface should be published with a description of the component's operations so that protocol designers can understand it and use it. Developing interfaces and defining conventions for component names and functions is, however, essential for compatibility and interoperability among components in the framework. This can be achieved in two ways: either by standards development committees or by the fact that a component becomes widely available because a major vendor with global reach supports it. A component should not have different names on the same or different machines because this may prevent the execution layer of the Meta-Protocol from locating the correct component. This issue, which is beyond the scope of this paper, is one of the most challenging issues in software engineering and in CBSE. No easy solution is available for these problems except through standardization (e.g., IANA).

Data objects are also considered components. Such data objects are used during the process of writing protocol specifications. Such data objects are often used to hold values that are common among several components such as (time, security keys, sequence and random numbers, and header information).

6.6. Performance

Our work is primarily targeted at building more reliable and more flexible protocol implementations. However, performance is also a concern. While actual performance depends on the implementation of the framework, some architectural issues may be relevant to performance. In general, we believe that an automatically generated implementation should have a performance similar to that of a manually generated one. Some performance degradation should be tolerated given that the Meta-Protocol framework provides an extra level of flexibility not available in manually generated implementations (e.g., on-the-fly code generation for new protocols). We believe that most performance concerns can be handled if the components used in the implementations generated by the Meta-Protocol are optimized for performance. Further performance optimizations can be obtained by caching already generated code for reuse.

7. Validation Experiments

We conducted several experiments to demonstrate the feasibility of the Meta-Protocol approach. We used XPSL to specify the following four protocols: Needham-Schroeder

Authentication Protocol (NSAP) [30], TCP's three-way handshake (see RFC 793 [29]), ISAKMP (see RFC 2408 [28]), and SSL cipher suite number 16 (referred heretofore as SSL#16) [27]. This choice of protocols provides diversity in terms of protocol purpose (e.g., authentication, transport, negotiation), location in the network stack, and complexity. Except for NSAP, which is a simple theoretical security protocol that has no practical use unless it is combined with other security mechanisms (e.g., anti-replay), all other protocols are widely used in the Internet.

Each experiment has two parts: 1) specification and automatic code generation and 2) execution and verification of the resulting code. The XPSL specification for TCP, ISAKMP, and SSL followed the specifications provided in the corresponding IETF RFCs. The NSAP specification is based on [30]. Details about the component library used in the experiments can be found in section 7.5.

The number of lines of XPSL specification and Java code generated for the four experiments is shown in Table 1. The Java code mentioned in the table does not include the code for the components, which are contained in a separate library. The total number of lines of Java code in the components library is 2150. The source code for the experiments can be downloaded from www.metaprotocol.net.

Table 1: Number of lines of the XPSL specification and Java code

	NSAP	TCP Handshake	ISAKMP	SSL#16
XPSL Specification	119	469	405	397
Java Code	87	260	227	208

These experiments are designed to test four qualities of the framework: completeness, consistency, correctness, and flexibility. A complete sequence of operations consists of: negotiation, delivery, automatic code generation, and execution. To demonstrate the consistency of the framework, the four experiments were developed according to the Meta-Protocol framework. An XPSL specification was written for these protocols. This specification is then shared between two remote parties and they successfully generate code. The code for ISAKMP's Base Exchange is used for the negotiation to determine the required subsequent protocol (e.g., the SSL#16).

The correctness of the framework is validated by verifying the correctness of the XPSL specification, the XSLT stylesheet, and the generated code. The correctness of the generated code is verified by compiling and running the code and checking its output (e.g., values, formats, and sequence of messages). Several test cases were designed to provide adequate coverage of the main paths of the protocol operation. The results of these tests showed that the four protocol implementations compiled correctly and performed according to their specifications.

The flexibility aspect of the framework was demonstrated by introducing changes into the XPSL specification such as replacing some components with others that perform different functionality, changing the order of operations, or inserting additional operations into the sequence. In the ISAKMP Base Exchange (BX) experiment, we replaced MD5 secure hash with SHA. In the SSL experiment, we replaced DES encryption with 3DES.

These changes required modifying a single line in the XPSL specification. The code generated through the Java XSLT stylesheet was tested and was found to run correctly.

7.1. The NSAP Experiment

NSAP was described in section 5 and its FSM is shown in Figure 5. Two processes are needed: one for the client and one for the server.

Figure 21 shows partial Java code for NSAP. This code shows Java output of the transformation of the object `SessionState` shown in Figure 9.

```
public static class SessionState {
    public String id ; public String Na ;
    public String Nb ;
    public String ClientPrivKey ;
    public String ServerPublKey ;
    public SessionState
    (String id , String Na , String Nb ,
    String ClientPrivKey , String ServerPublKey )
    { this.id = id ; this.Na = Na ;
    this.Nb = Nb ; this.ClientPrivKey =
    ClientPrivKey;
    this.ServerPublKey = ServerPublKey ; } }
```

Figure 21: Java Code for the SessionState object for NSAP

7.2. The TCP Handshake Experiment

This experiment requires a single process because the FSM for TCP given in RFC 793 [29] recommends the use of a single FSM that combines the client and server operations. Therefore, the XPSL protocol specification and the XSLT transformation produce code that works on behalf of either a client or a server.

7.3. ISAKMP Experiment

ISAKMP was introduced in section 4.2. This protocol is used to implement the negotiation layer of the Meta-Protocol. Figure 22 shows the FSM diagram of ISAKMP. Based on this diagram, we developed two XPSL specifications: one for an initiator and the other for a responder. In addition, we developed all the components and objects needed for the actions taking place inside each of the FMS states. Fourteen new components and five new objects were added to our component library to implement ISAKMP.

In the first state of an ISAKMP session, the initiator receives some input parameters from an application: destination IP and port, a secret value to be used in generating the cookies, and the initiator private and public RSA keys. These values are passed inside an object called `ISAKMPFSM`. This object holds all the variables that are publicly required by all the states during a session. Figure 23 shows the data structure of the ISAKMPFSM object.

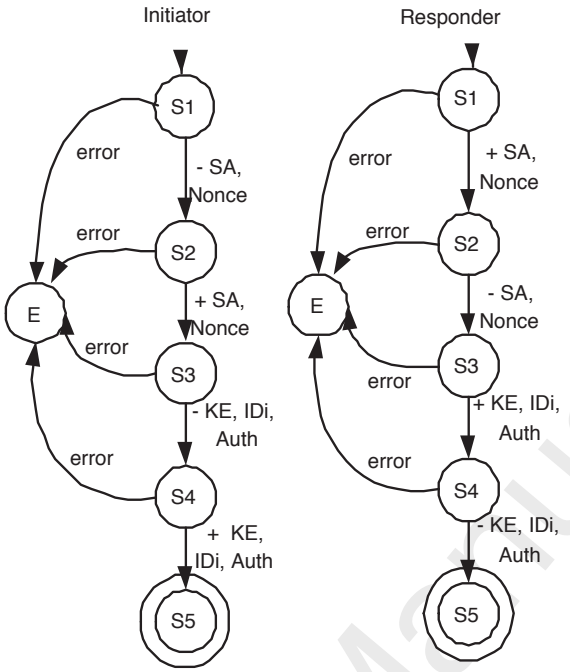

Figure 22: FSM for ISAKMP's Base Exchange.

```
public static class ISAKMPFSM {
        public int SourcePort, DistPort;
        public String SourceIP, DistIP, Secret;
        public String SourceCookie, DistCookie;
        public String certPublic, certPrivate;
        public int Mjrver, Minver, Flags,
        MessageID, Totallength;
        public String InitNonce, RespNonce;
        public String ProtName, ProtLocation;
        public int SelectedProtID,
        ProtMjrver, ProtMinver;}
```

Figure 23: ISAKMPFSM data structure

The goal of the initiator, in the first state, is to prepare a proposal and send it to the responder. This proposal consists of three parts: payload header, proposed protocol name, and location. Five components are used to build the client first message in the first state: HashingStringMD5, BXcomposeMainHeader, CompLib.BXcomposeProposals, GenerateIntRandomNumber, BXcomposePayload. The names used for these components indicate their functionality.

The responder XML specification is the dual of the initiator's. For example, when the initiator sends a message, the responder receives that message. Most of the action

components are shared by both parties. However, there are two main differences: the values they process and some context-specific operations. For example cookies and nonces are prepared by the same components (e.g., MD5, and random number), but their values vary because they are generated by two different processes with different input values (e.g., IP address and port numbers for cookies). Examples of context-specific operations include 1) preparation of a proposal by an initiator, and 2) acceptance of a proposal by a responder.

7.4. The SSL Experiment

In this experiment, again, two processes are used to represent a client and a server. The SSL record layer protocol has to use the negotiated set of keys in the handshake phase to apply the crypto operations and add the SSL header. The SSL specification provides 32 options. Each option represents a set of cryptographic algorithms. In this experiment, we limited the operation to a single option: *TLS_RSA_with_DES_CBC_SHA*. In this experiment 44 components including 9 data objects were developed to carry out the SSL#16 operations. The calculation of the keys is done according to the specification given in RFC 2246, TLS protocol Version 1.0 [27].

This experiment implements transfer of files from a client to a server. Implementing the other way around is straightforward and can be achieved in two ways: either by switching the XPSL specification or adding the components that are responsible for sending files to the XPSL server specification. After the handshake phase, the secure session starts the process of protecting application data by receiving a file name from the user of the experiment. The client process loads the file, produces an SHA1 hash value, appends the hash value to the file, adds the SSL header, encrypts the file using DES, and sends the encrypted file to the server.

7.5. The Components Library

The framework presented here depends heavily on CBSE. Therefore, we developed a library that comprises all the operations needed to run these experiments. The library contains 86 components and 24 data objects for a total of 2150 lines of Java code. The user does not need to know the details of the internal design of these components.

Another important point is that the implementations in these experiments follow the objected-oriented programming paradigm. Therefore, messages and data structures are designed as objects that are passed to the components for processing. Moreover, objects that are shared by more than one protocol (e.g., error message) are implemented as part of the component library for easier accessibility. Similarly, objects that are shared by more than one component are also included as part of the component library (e.g., StringObject, BytesObject, and TCP datagram).

Most of the components have been developed by the authors for the purpose of these experiments. However, some are just the result of wrapping of standard Java crypto functions or modified pieces of borrowed code.

8. Related Work

Related work in the area of communication protocols can be found in several research projects: x-kernel [16], Cactus [8, 36] Ensemble [12], and Conduit [15, 14]. The x-kernel

is an operating system communication kernel designed to provide configurable communication services in which a communication protocol represents a unit of composition; [7] extends the x-kernel architecture. Horus and Coyote are extensions and applications of the x-kernel architecture to the area of group communications [6, 13, 20, 21, 26]. [17] proposes a component-based architecture for software communication systems.

Our approach differs from the research work just discussed as follows. The first distinction is in the way configuration information of a newly designed protocol is shared. In the projects mentioned above, the configuration information of a newly designed protocol has to be shared manually among the parties involved. However, in our work, we provide support for secure online negotiation and distribution of protocol specifications. Second, previous work does not provide a language, as we do, to express the specification of a protocol configuration. Finally, previous proposed solutions are either associated with a single layer of the network stack [14, 17] or they replace the entire network stack [6, 7, 16]. Our work, on the other hand, is applicable to any layer of the network stack.

Click router [22] is a software architecture for building routing software. It is similar to our work in proposing a language for building software from components. However, our work differs from the Click router in several aspects. First, the Click router is limited to the configuration of a router's software. On the other hand, our work applies to all types of communication protocols. Second, the Click router does not provide a mechanism to exchange configurations. Third, the language proposed by the Click router is low level and cannot be validated automatically. On the other hand, the language we developed is XML-based and it has a validation schema that reduces some human coding errors.

Software-defined network (SDN) is one of the most recent development in network architecture. Its objective is to provide separatation between data plan access and control plan access in order to simplify network operation and management. The OpenFlow is a protocol developed to build an SDN environment [2]. Our Meta-protocol approach complements the SDN approach by providing a mechanism to define the OpenFlow protocol itself , some parts of it, or other flavors of it, in XSLT specification to provide higher flexibility in protocol specification and deployment.

In security protocols, Cactus is a framework used to implement a security system called SecComm [8]. SecComm differs from our approach in two aspects. The first aspect is that the specification of the required configuration is transmitted through the header information of the messages. In our approach, the specification is coded in a separate document so that it can be transmitted independently of the messages in a secure fashion. Second, the event-driven approach in [8] to activate components at run time adds a layer of delay to the system. In our approach, the sequence of operations is predetermined before runtime based on the specification of the protocol leading to efficient execution of the selected components.

In Automatic Protocol Generation (APG), protocol implementations can be derived automatically from abstract specifications in languages such as SDL [5], Estelle [34], Promela++ [4], and [38]. 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. Our approach, on the other hand, capitalizes on CBSE to produce a high level specification. Therefore, most of the details of the operation of a protocol are hidden inside the components.

In the context of protocols, APG means designing the protocol in an automated

way [24, 31, 35]. The APG process takes as input a set of requirements and produces a set of proposed protocol designs. Our approach takes a design (i.e., the output of the APG process) and converts it into a specification used to automatically generate executable code for a protocol.

Related work that includes the use of XML to produce program code can be found in [23]. In that work, an instrument designer produces an XML document that describes the instrument services. This XML document is used 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.

In [19], 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. In our work we encode FSMs in XML allowing for automatic code generation in any programming language.

9. Concluding Remarks

The traditional approach for designing security protocols assumes an environment that does not change over time. This is not a valid assumption because no single solution will be able to accommodate the rapid evolution in communication protocols in general and specifically in security protocols for Internet applications. It is difficult to make changes to a monolithic implementation of a protocol. Users often do not have access to source code and even in the case of open source, it is not trivial to change large pieces of code. In contrast, the Meta-Protocol approach assumes a dynamic environment and provides enough flexibility so that applications can select the specification that fits their exact needs. Users can also transmit their preferred protocol specification to their corresponding parties to generate the code.

The design of the framework and the supporting systems are based on CBSE. The advantages of this approach are similar to that of computer manufacturing: personal computers are designed for incremental evolution of processor, memory, and other parts of the system [32]. Similarly, communication and security components (e.g., functions and algorithms) can be produced by many different manufacturers and then assembled into one protocol very rapidly. The assemblers then just need to design a good architecture and they do not need to be experts in the internals of each assembled component.

The basic difference between the traditional protocol approach and the one proposed here is that the latter provides flexibility for change, while earlier fractionated methods lock the developer into a rigid process. Having flexibility does not mean that every developer will be able to choose the right set of components, but certainly experts can, and should therefore not be restricted to use a set of limited protocols. Standardization committees will also play a role in advising and approving XPSL specification documents after testing them. This additional flexibility is likely to reduce the time spent in producing new protocol standards and implementations.

References

[1] I.S. Abdullah, and D.A. Menascé, "Protocol Specification and Automatic Implementation Using XML and CBSE," Proc. *IASTED Intl. Conf. Communications, Internet and Information Technology* (CIIT2003), Scottsdale, Arizona, USA, Nov 03. pp. 191-196.

[2] "SDN Resources Overview, https://www.opennetworking.org/sdn-resources/sdn-resources-overview

[3] N. Apte, and T. Metha, *UDDI: Building Registry-Based Web Services Solutions*, Prentice Hall, New Jersey, 2003.

[4] 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*, vol. 2, Mar-Apr 1998, pp. 455–462.

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

[6] N. Bhatti, A. Hiltunen, R. Schlichting, and W. Chiu, "COYOTE A System for Constructing Fine-Grain Configurable Communication Services," *ACM Trans. Computer Systems*, vol. 16, Issue 4, Nov. 1998, pp. 321–366.

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

[8] *The Cactus Project*, University of Arizona, Computer Science Department, www.cs.arizona.edu/cactus/.

[9] A. Caro et al. "SCTP: A Proposed Standard for Robust Internet Data Transport," *IEEE Computer*, Nov. 2003, pp. 56–63.

[10] H. Hamburger, and D. Richards, *Logic and Language Models for Computer Science*, Prentice Hall, New Jersey, USA, 2002.

[11] A. Harris, and R. Kravets, "The design of a transport protocol for on-demand graphical rendering," *Proc. ACM 12th Intl. Workshop on Network and Operating Systems Support for Digital Audio and Video*, Miami, FL, 2002, pp. 43–49.

[12] M. Hayden, *The Ensemble System,* Ph.D. dissertation, Dept. Computer Sciences, Cornell University, 1998.

[13] M. Hiltunen, and R. Schlichting, "The Cactus Approach to Building Configurable Middleware Services," *Proc. of the Workshop on Dependable System Middleware and Group Comm.* (DSMGC 2000), Nuremberg, Germany, Oct. 2000.

[14] H. Huni, R. Johnson, and R. Engel, "A framework for Network Protocol Software," *Proc. OOPSLA95 ACM SIGPLAN Notices*, 1995, pp.1-14.

[15] J. Zweig and R. E. Johnson, "The Conduit: a Communication Abstraction in C++," *Proc. USENIX C++ Conf.*, USENIX Association, April 1990, pp. 191-203.

[16] N. Hutchinson, and L. Peterson, "The x-kernel: an architecture for implementing network protocols," *IEEE Tr. Software Eng.*, vol. 17, Issue 1, 1991, pp. 64-76.

[17] M. Jung, and E. Biersack, "A Component-Based Architecture for Software Communication Systems," *Proc. IEEE ECBS*, Edinburgh, Scotland, April 2000, pp. 36–44.

[18] A. Kotok, and D. Webber, *ebXML: The New Global Standard for Doing Business over the Internet*, New Riders Publishing, 2002.

[19] C. Liu, and K. Su, "An FSM-Based Program Generator for Communication Protocol Software," *Proc. IEEE 18th Annual Int'l Computer and Application Conf.*, Taipie, Taiwan, 1994, pp. 181–187.

[20] P. McDaniel, and A. Prakash, *Ismene: Provisioning and Policy Reconciliation in Secure Group Communication*, Tech. Report CSE-TR-438-00, Electrical Eng. and Computer Science Dept., Univ. of Michigan, Dec. 2000.

[21] P. McDaniel, A. Prakash, and P. Honeyman, "Antigone: A Flexible Framework for Secure Group Communication," *Proc. 8th USENIX UNIX Security Symp.*, Aug. 1999, pp. 99–114.

[22] R. Morris et al. "The Click router," *Proc. 17th ACM Symp. Operating Systems Principles*, Dec. 1999, pp. 217–231.

[23] S. Perrin, E. Benoit, and L. Foulloy, "Automatic Code Generation based on Generic Description of Intelligent Instrument," *IEEE Intl. Conf. Systems, Man, and Cybernetics*, Vol. 6, 2002, pp. 569–574.

[24] A. Perrig, 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*, 2000, pp. 64–76.

[25] S. Raman, H. Balakrishnan, and M. Srinivasan, "ITP: an Image Transport Protocol for the Internet," *IEEE/ACM Trans. on Networking*, Vol. 10, Issue 3, June 2002, pp. 297–307.

[26] R. Rensesse et al. "A Framework for Protocol Composition in Horus," *Proc. Annual ACM Symp. Principles of Distributed Computing*, 1995, pp. 80–89.
[27] T. Dierks, and C. Allen, "The TLS Protocol, Version 1.0," IETF RFC 2246, Jan. 1999, www.ietf.org/rfc/rfc2246.txt.
[28] D. Maughan et al. "Internet Security Association and Key Management Protocol (ISAKMP)", IETF RFC 2408, Nov. 1998, www.ietf.org/rfc/rfc2408.txt.
[29] Transmission Control Protocol, IETF RFC 793, Sept. 1981. www.ietf.org/rfc/rfc0793.txt
[30] B. Schneier, *Applied Criptography: protocols, algorithms, and source code in C*, 2nd. ed., John Wiley & Sons, NY, 1996.
[31] D. Song, S. Berezin, and A. Perrig, "Athena: a novel approach to efficient automatic security," *J. of Computer Security*, vol. 9, no. 1/2, 2001, pp. 47–74.
[32] D. Sportt, "Componentization the Enterprise Application Packages," *Comm. of the ACM*, vol. 43, no. 4, April 2000, pp. 63-69.
[33] W. Stallings, *Network Security Essentials: Applications and Standards*, Prentice-Hall Inc, NJ, 2000.
[34] J. Thees, "Protocol implementation with Estelle- from prototypes to efficient implementations," *Proc. Intl. Workshop on the Formal Description Technique Estelle*, France, 1998.
[35] B. Vassall, *A Look at Automatic Protocol Generation & Security Protocols*, July 16, 2001, www.sans.com.
[36] G. Wong, M. Hiltunen, and R. Schlichting, "A Configurable and Extensible Transport Protocol," *Proc. IEEE INFOCOM 20th Annual Joint Conf. IEEE Computer and Comm. Societies*, vol. 1, 2001, pp. 319–328.
[37] H. Zheng, and J. Boyce, "An Improved UDP Protocol for Video Transmission Over Internet-to-Wireless Networks," *IEEE Tr. Multimedia*, Vol. 3, Issue 3, Sept. 2001, pp. 356–365.
[38] J. Rodrigues, J. Ventura, A. M. Campos, and L. Rodrigues , "Implementation and analysis of real-time communication protocol compositions," *J. Real-Time Systems* , Kluwer Academic Publishers ,Vol. 37, Issue 1, Oct. 2007, pp. 45–76.
[39] M. Daconate, and A. Saganich, "XML Development with Java 2," Sams Publishing, Indiana, 2000.
[40] C. Binstock, D. Peterson, M. Smith, M. Wooding, C. Dix, and C. Galtenberg, "The XML Schema Complete Reference," Addison Wesley Pearson Education, NJ, 2003.
[41] S. Holzner, "Inside XSLT, New Riders," Indiana, 2002.
[42] G. Heineman, and W. Councill, "Component-Based Software Engineering," Addison-Wesley, 2001.
[43] A. Brown, and K. Wallnau, "The Current State of CBSE," *IEEE Software*, Vol. 15, Issue 5, Sep-Oct. 1998, pp.37–46.
[44] V. Tran, and D. Liu, "Application of CBSE to Projects with Evolving Requirements- A lesson-learned," *Proc. Sixth Asia Pacific Software Engineering Conf.*, Japan, Dec. 1999 pp. 28–37.