Executable Specification and Prototyping of Network Protocols Using UML and Java

K.Chandra Sekaran
Dept. of Computer Engineering,
NITK Surathkal, India
kchnitk@gmail.com

R.K.Gnanamurthy,
Vivekananda College of Engineering for Women,
Trichengodu, India,
rkgnanam@yahoo.co.in

Abstract

Network protocols are often implemented in software and / or hardware, and, it becomes essential to design and test them in an efficient manner. This paper explores a dual phase approach for developing network protocols: in the first phase protocols are modeled using UML (Unified Modeling Language) as the formalism, and, in the second phase, use of executable specification and prototyping of protocols based on Java is supported. The prototyping of a protocol is useful for further investigations such as verification of protocol properties, test case generation etc. Once the second phase provides a satisfied result, the developers can go ahead in developing and deploying the protocol in the real environment. Yet another objective in this work is to design executable constructs in Java to specify protocols and prototyping them. The protocols designed using this approach ensures sustenance of the models already developed. Illustration of using executable constructs in Java to specify and prototyping of protocols, and comparison with native implementations is presented in this paper.

Keywords: *Protocol specification, prototyping, Object Orientation, UML and Java, Executable constructs in Java*

1. Introduction

Executable specifications have been advocated as a promising method for understanding the requirements specification of systems, and for developing systems incrementally. An executable specification serves as a prototype of the final implementation [1]. The main advantage of prototyping systems is that it gives the designer early feedback that what is being specified is indeed what is desired. In addition, the prototype can be checked for functional and timing correctness before moving to the final implementation. The development of Formal Descriptive Techniques (FDTs) such as LOTOS, ESTELLE and SDL [2] to a large extent, is triggered by the need to overcome the so-called software crisis: the inability to control the complexity and correctness of large software designs (e.g., network protocols), which is particularly harmful for the development of large and modern distributed systems where hidden errors in the high level design and specification may cause great difficulties in their operation and maintenance. FDTs originally aimed at controlling these complexity and correctness, along with unambiguousness of designs. Majority of these FDTs are still in the research world, and, are not popular in the industry world [3] as there are inherent problems experienced with these formalisms while putting them in practice. In this context, it is worth to recall that most of the software industries have accepted UML [4] as an alternative formalism for their developmental work. Thus, there is a need for a new look at the usability of FDTs while incorporating all the features which are feasible to produce a prototype of the final product [3]. In other words, any formal method should essentially allow the designer to specify the protocols as well as enable to have a prototype model in order to inspect.

1.1 Prototyping of Protocols

Prototype [5] is an approximation of a system that exhibits the essential feature of the final version of that system. Prototyping is used in many areas, including engineering and information systems development. In the latter area, prototyping (which is also known as *rapid prototyping* when suitable tools are used to support the rapid creation of design elements) addresses some of the problems of traditional systems analysis, in particular, the complaint that users see their final product / system only at the last stage of development cycle (i.e., implementation time) and felt it was too late to make changes. In that case, the risk of failure (of that product) because of user dissatisfaction, including outright user rejection, is significant while

978-1-4244-5757-1/10/$26.00 ©2009 IEEE

making the product (in our case, it is communication service). On the other hand, by implementing a prototype first, the developer can show the users something tangible, inputs, intermediary stages, and outputs, before finally committing the user to the new design. These prototypes are commercially cannot be diagrammatic approximations, but actual output on work station screens, and, the formats can be changed quickly, as per the suggestions of users. Further, it may only be by using this approach that the users discover exactly what they want from the system, as well as what is feasible. It is also possible to try out a run using real data, perhaps generated by the users themselves [6].

Rapid prototyping of protocols specified in the Formal Description Technique (FDT) LOTOS has been presented in [2], and, however, the approach in [2] has used the programming language C, without aiming at executable specifications. The work in [7], uses Java for implementing the protocols, and, in Cicero library of constructs [8], again, C programming language is being used. These approaches do not make use of object orientation and thereby do not address the issues of reusability, portability (platform independence) and interoperability in the development stage of complex products such as protocols. The use of 'objects' has also been attempted in ESTELLE [9] and in SDL [10] FDTs without prototyping. However, the use of these standard FDTs in the industry environment is poor [2].

1.2 UML and Java

This paper explores a dual phase approach to the development of network protocols. In the first phase, the protocols are modeled in UML. In the second phase, the UML model is translated into a Java based executable specification (of protocols) and then in an run time environment, the prototype of the model is made available for further investigation such as verification of protocol properties, test case generation etc. In this way, developers exploit the advantages of using formal approach UML while using executable specifications in Java for prototyping purpose. Once this second stage provides a satisfied result, the developers can go ahead in developing and deploying the protocol in the real environment in a full-fledged way. Discussion on this dual approach is given section 2 of this paper.

The use of object oriented approach in designing of a library for protocol specification ensures reusability, portability as well as interoperability. Thus the protocols designed using the dual phase approach ensures sustenance of the models already developed, as it uses object orientation in its library of executable constructs using the target language Java. The choice of UML as formalism for the modeling the protocols has been on many folds. UML's strength is its expressiveness without using any mathematical formula and using only graphical representations. The target language has been chosen to be Java, a powerful object oriented language which has all the features needed for object oriented implementation of constructive specification and preferred by programmers worldwide for a wide range of applications as the language for implementation. Since, Java is platform-independent; portability and inter-operability are inherently ensured. Also, Java provides support for multiple thread execution so that parallelism in protocol implementation can be fully exploited. The library is developed based on event driven approach, which is a popularly used one in real time applications. As protocols of modern / Internet era require to work in this style, the executable specification constructs (or, library of constructs) make use of this approach. Details of these constructs are provided in section 3.

2. Development of protocols – the Dual phase approach

The proposed dual phase approach for developing network protocols consists of two paths – P1 and P2 - in it. Path P1 with two phases guide the protocol developers (a) to use UML to model the protocol and, (b) to use the executable constructs (library) for specifying protocols in Java and then for prototyping purpose. On successful completion of the second phase the developers can follow the either continuation of path P1 or second path P2. These paths enable the developer to proceed with the actual development and deployment of the protocols. The path P2 also enables to come back to UML model, in case there is any problem in the deployment / implementation of the product. Figure 1 gives these two paths and the approach.

2.1 Role of UML and Java in path P1

As shown in Figure1, the path P1 is the main path in this work. It consists of (a) UML modeling and (b) Protocol Specification and Prototyping. Prior to UML modeling step, it is envisaged that the conventional description of the product (i.e., protocols) has been completed.

Protocol Description: This is the initial step of any methodology where a protocol or, communication service is refined from the initial concept into a careful product definition and a comprehensible description. The description of the protocol is presented in a structured way and includes all the important characteristics of it, such as basic service features, available service extensions, alternative design options, and existing standards. These service characteristics take (if applicable) the form of a service profile, which has a layered structure and comprises the following layers (bottom-up) [11]: (a) Infrastructure Layer: the necessary infrastructure (wireless network and end-computing) characteristics for the provision of the service, (b)Distributed Processing Layer: service characteristics related to the division of the required processing work into separate tasks able to be executed a number of wireless devices over a wireless network in a distributed manner, (c) Layer service characteristics – the interactions between the service features, and, (d) User Interface Layer: service characteristics facilitating the user to interact with the service.

UML model: The UML model step reveals the functionality of the protocol based on the requirements and proposed profile. It creates a logical model (design) of the protocol and it defines the technical options related to it. The technical options will be implementation specific, because there are many alternative hardware, software, communications and development strategies possible. Protocol specifications are normally broken into four layers: user interface (describing the service and user interactions), service architecture (describing service features and primitives and the way they interact with each other), physical network architecture (describing the topology and the interconnection of network elements), and communications drivers (describing the low level protocols to be used between network elements). For creation of specifications of the protocols, as we noted earlier, formal methods are preferably used. Formal specification, which includes semantics, can be used to analyze not only for completeness and consistency, but also for the product's behavior. In this work, we have used UML as the formal method.

Executable Specification and Prototyping: In this step, the executable specifications using the constructs in Java (details are in section 3) are developed followed by prototyping of the protocol under examination. As noted earlier, it is necessary to determine initially the logical design of the protocol on hand, which reflects the service specifications that will be prototyped. This is then 'mapped' onto a particular physical environment in the deployment stage. It should be noted that the 'prototyping' can occur repeatedly. The protocol designer enriches the parts of the logical design of the protocol or network service that are prototyped, until they decide that the prototype includes all the necessary functionality to proceed to the development of it, which is widely known as *operational prototyping* [8]. In this work, the concept of operational prototyping is taken care by means of providing executable constructs for specifications of protocols and prototyping.

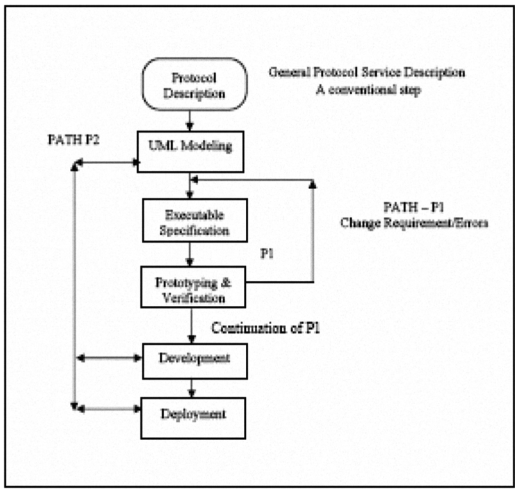

Figure 1: Protocols Development Methodology (Dual phase approach)

3. Design Details of Executable Constructs in Java

3.1 Design Rationale

Standard FDTs are declarative specification languages, and the design of a protocol should be centered around it by the use of appropriate tools and packages. On the other hand, as our work uses Java as a means to specify the protocols using executable constructs of Java for specifications purpose and thereafter prototyping, there is no need of any additional tools or software. Thus, this work differs fundamentally from such languages in a way that it allows designers to specify in Java and synchronize events within a protocol using the executable constructs in order to have ordering of events and control over them in a better way.

Our work has been derived from Cicero [8]. Cicero is not a full language, but is a language veneer extending existing languages. Cicero model is based on the notion of events, event instances and event patterns. Here, events are unbounded sequences of event instances, and combinations of events is called event patterns. Cicero is a small set of high-level control constructs for handling these event driven approach of constructing protocols: (a) *emit* construct is used to construct new event instances (b) *when* construct controls the execution of the associated code for the events, (c) *cond* construct implements conditional branches and helps to distinguish between active events and event patterns, (d) *bunde* provides grouping of events and encourages modular programming, and, (e) *escape* construct allows programmers to include statements in the base language. Cicero uses an existing programming language and the users of it need to learn only these five Cicero constructs. Following drawbacks of this Cicero have been observed: Multithreaded execution is important to increase throughput, which is not feasible in it. A natural abstraction for specifying synchrony, asynchrony and concurrency in protocol execution is an event driven paradigm, where a protocol is viewed as a machine reacting to internal/external events or messages. Use of these too, is limited in Cicero. Implementation Cicero for modern network protocols while using Java as the target language with event handling mechanisms has also certain limitations: *Emit* is not possible directly in Java because each *event listener* has to have a copy of the *event;* and, in the exception handling once the exception is consumed it is not available for later use. Similar to the AWT package, we need to have methods for adding *Action Event* generators and appropriate listeners. *Cond* is superfluous and hence is not a concern in further design process. The same applies for *escape* construct also. Without proper design for *emit*, it is pointless to proceed into the designing of more complex constructs such as *bundle*. Considering the above drawbacks of Cicero, we proceeded towards a different pragmatic design.

3.2 Revised Design

In order to overcome the drawbacks mentioned in section 3.1, and to develop an event generation and handling mechanism with a new approach using Java as the target language for network protocols, it is required to handle user-defined events. The AWT library of JDK does not support user defined events; i.e., it supports events are of pre-defined types only, such as mouse click, key press, cursor movement etc. Hence, design process headed in the direction of developing event generation and handling mechanisms for user defined events. Thus, *event* was developed as a separate class named ***Jevent*** with attributes namely,

- Event name-String signifying the name of the event
- Instance number-reflects the ordering among events
- Priority-to facilitate exception events which require immediate action
- Generic object—to associate a value or data structure with the event

Each call to the *emit* must now contain an instance of the class *Jevent*. This is equivalent to generating a new event whose details are contained in the corresponding instance of *Jevent*. Since many events can be generated at once, it is required to have a data structure which can temporarily hold the event instances, which are then given to *waiting when* constructs. Queue is the data structure that has been used for this purpose. But, a simple FIFO (First-In-First-Out) queue is insufficient. The reason being, we have to facilitate exception events, which require immediate action. A priority queue, thus, is found to be most suitable to our problem. The priority of this queue called is based on priority field of *Jevent* objects. The typical priority queue (***Event Queue***) operations such as enqueue, dequeue are provided for the effective use of it. Next task is to find a mechanism for the construct ***emit***. *Emit* is designed as a method in the ***Jicero*** class which accepts as input parameter an instance of *Jevent*. The sequence of operations that occur following an *emit* call are as follows:

- Accept the *Jevent* instance
- *Enqueue* the event in the event queue using the *enqueue* method of the *event queue*
- Notify the process which passes the events to the waiting when constructs. (This process is called *Dispatcher*, which will be dealt in detail later).

The ***when*** construct consists of two parts, namely

- Event – the event whose occurrence which will trigger the actions
- List of target statements – the actions to be performed.

Each *when* construct needs a separate thread of execution and control. Hence for each and every *when* construct we create a separate Java Thread using *Runnable Interface* [7]. Since many *when* constructs may name the same event, one event instance may trigger many *when* constructs. These threads will all run concurrently. But Java does not allow attachment of independent code to a thread. Instead, a thread can only be associated with an object of a particular type i.e. all threads of objects of a particular class will have the same code. Hence the only option remaining was to implement each *when* construct in a class of its own. To simplify structure, each *when* construct is inherited from an abstract base class called `when`.

As required by Java the target statements are included in the *run* ( ) method of the class. Since the *when* constructs may be executed several times during the course of protocol execution, it is necessary that the actions be put in an infinite loop using *while* (true) { }. At the beginning of this loop we invoke the thread suspension mechanism using *Thread.wait* ( ). It is the job of the other method, namely ***updateJe*** of the *when* construct to awaken the thread on the occurrence of the correct event. The *updateJe* ( ) method is used to update the copy of the event to the currently passed event. Since parallelism is involved, each when construct will have its own copy of the event. The passing of the event will be done by an event dispatcher. It is the duty of the *updateJe* ( ) method to check for the correct triggering event. This is done by a simple String compare operation on the event name field of the event instance. If the correct instance is found, the method *Thread.notify* ( ) is used to notify and awaken the *run*( ) method to perform the necessary actions. Care is also taken to implement concurrent executions in our approach.

4. Experiments and Results

Experiments have been carried out for fairly complex protocols such as at-least-once semantics, RPC, cryptographic handshake protocol, client server message passing, video server, etc. These protocols were written using our specification language. They were executed under environments like Windows 98, Linux, etc. We present here, only a single example, the RPC implemented using our package.

4.1 RPC

Remote procedure calls (RPC) form a major class of protocols. They are concerned with the concepts of distributed systems. Here we demonstrate the implementation of an RPC with at least once semantics. In other words the protocol ensures that the procedure is executed at least once. It may be executed more than once, but that is not the concern of the protocol. The first step in protocol development is its description; a finite state machine based modeling the RPC is chosen here in order to describe its working. Since RPC is a client server protocol we developed two FSMs, one each of client and server. The FSM of the client: In this client FSM, after initialization, a request is sent once (event *send_data*) to the other party. Then the FSM goes to the Wait state where the FSM waits for a certain time before a time out occurs. After the time out another request is sent and the FSM enters the wait state again. This proceeds until a response from server is obtained (event *recv_data*) or until the maximum number of retries is reached, whereby failure is reported. The FSM of the server: The FSM of the server is quite simple. After initialization, the FSM waits for a request to arrive (send_data). On receiving such a request, it will proceed to another state, where the procedure is actually carried out or called. After calling the procedure, the FSM return to the original state.
Based on these FSMs, the specifications using the Java constructs (for client and server), i.e., the executable specifications were written and the prototype of the protocol was developed and tested. The actual work was implemented in a Linux workstation. Typical test results are tabulated below:

Table 1: Results

Protocol	Native Implementation	Our Implementation
RPC	Time for RPC execution 430 ms	Execution time 640 ms

It has been observed that there is a time difference between our implementation and the native implementation of most of the protocols. It is due to the fact that our implementations do use Java as target language and the native implementation uses C/C++ language. This difference can be eliminated by using advanced compilers including JIT compilers for Java; the work towards this task is an on going one in our lab. Because of the space problem, complete details could not be provided here.

5. Conclusion and Future work

The protocols like at-least-once semantics, RPC, cryptographic handshake protocol, client server message passing, video server, etc were written using our specification language. They were executed under environments like Windows 98, Linux, etc. The protocols behaved as per expectation. There was no visible difference when the two ends were on different platforms (i.e. Windows 98 & Linux). Hence the objective of inter-operability as also the objective of portability, both of which were among the primary concerns have been successfully achieved This corroborates the correctness and validity of our design decisions such as using local dispatcher coupled with message passing techniques. Also, since we could experiment fairly complex protocols without fallacy using our specification language which is based on Java our decision to omit certain constructs of Cicero, and make other optimal modifications of certain concepts has been justified. We chose Lex and Yacc to speed up the implementation process in Linux environment. Our project is an attempt, one of the first of its kind, to provide a user friendly tool for prototyping (and if required, implementing) protocols directly from the design stage without having to undergo the pains of coding the lower levels details of protocol implementation. One of the improvements possible, is that the parser can also be made to be platform independent. With suitable natural language processing capabilities and ability to convert plaintext to code, one can also specify the actions to be performed in natural languages rather than the present underlying language specification. With computers of better capabilities viz. multiprocessor computers a better degree of parallelism can be achieved to bring the performance of the implementations as close as possible to theoretical concepts.

10. References

[1] Sitaram C.V., Raju and Alan C.Shaw, A Prototyping Environment for Specifying, Executing and Checking Communicating Real-Time State Machines, Software-Practice and Experience, vol.24(2), Feb.1994, pp.175-95

[2] A.Valenzano, R.Sisto and L.Ciminiera, Rapid Prototyping of Protocols from LOTOS specification, Software- Practice and Experience, vol.23(1), Jan.1993, pp.31-54.

[3] Marc Zimmerman et.al, Making Formal Methods Practical, paper in MIT, Cambridge 2000.

[4] F.Dietrich and J.P Hubaux, Formal methods for communication services: meeting the industry expectations, Jl. Of Computer Networks, 38 (2002) pp.99-120.

5] R. Budde, K.Kautz, K.Kuhlenkamp, H.Zullighoven , prototyping. As Approach to Evolutionary System Development, Springer. Berlin 1991.

[6] D.E. Avison, G. Fitzgerald. Information Systems Development methodologies, Techniques and Tools. McGraw-Hill, London.1997.

[7] Bobby Krupczak, Kenneth L.Calvert and Mostafa H.Ammar, Implementing Communication Protocols in Java, IEEE Communications Magazine, Oct.1998, pp.93-99.

[8] C.V. Ravi Shankar and Yen-Min Huang, *CICERO: A Constructive Protocol Specification Language.* IEEE Transactions on Software Engineering 13(9):257-266, June 1992.

[9] B.Prabhakaran and S.V.Raghavan, Object Oriented Extensions to ESTELLE, proc. Of ICCC, 1990. pp.750-758

[10] Rick Reed, Notes on SDL-2000 for the New Millennium, Computer Networks 35(2001), pp.709-720, Elsevier publications.

[11] D. Lewis, T. Tiropanis, A. McEwan, Inter-domain integration of services and service management, Lectrure Notes in Computer service No 1238 (intelligence in service and Networks: Technology for Co-operative Competition), Springer Verlag. 1997, pp. 283-292

[12] www.java.sun.com:- the main Java site which has in detail documentation on Java and Java Threads.

K.Chandra Sekaran is Professor in the Dept. of Computer Engg at NITK Surathkal, India. He has 22 years of teaching and research in the areas of networks, distributed computing, and has over 95 publications in reputed journals / conferences.

R.K.Gnanamurthy is Professor in Electronics and communication and currently working as Principal at Vivekananda College of Engineering for Women, Trichengodu (TN). He has over 50 reputed publications and specialized in wireless networks.