Abstract: This paper presents an object-oriented approach to describe an embedded communication protocol in UML. We build on top of existing work based on object-oriented patterns. We follow the belief that there should exist a generalized structure or pattern on which protocols should be formed. We feel that the system structure is of importance for promoting reuse. These protocol patterns would provide flexibility for the cost of additional complexity. This complexity could be later simplified for a flexibility tradeoff. We will describe our protocol pattern, and use it to structure our protocol in our embedded communication system. Our ultimate goal is to look for a describe-and-synthesize technique for communication protocols.

1 Introduction

Embedded Communication protocols describe the legal sequence of process that process can exhibit. Essentially, they provide the “rules” of communication. At the highest level of abstraction the protocol may actually describe the purpose of the system. The refinements of the highest layer bring about objects that are required to implement the higher layer. These objects then need coordination or protocols. This process could repeat many times to evolve into the communication protocol for the embedded system.

Protocol design is becoming an increasing complex and cost-sensitive process. For this reason we wish to find a good describe-and-synthesize path. Therefore our first attempt is to describe our protocol cleanly, and to identify the boundaries of our system. Thus our primary goal is to find a novel protocol structure that fits all and use UML to describe it. Our secondary goal is to clearly describe the semantics of UML and find its place in our describe-and-synthesize path. We chose to describe our model in UML because it has roots in programming languages, it is object-oriented, and it was a new tool.

1.1 What is UML?

Booch's definition of UML is a graphical language for visualizing, constructing, and documenting the artifacts of a software intensive system [Boo98]. It is a language that captures the object-oriented model. Some of the items this model stresses are data encapsulation, information hiding, and inheritance. UML has mainly been targeted for software projects where it has met the most success. UML’s intent is to capture the object-oriented (OO) model, and many OO practitioners have praised this model. We also hope to reap some of the OO benefits, and believe we are not limited by the software nature of UML. We believe in philosophy that software and hardware can be abstracted away so we do not see UML’s software nature as a big hindrance. We also hope to use object-oriented properties to ultimately promote reuse.

Actually there already exists an object-oriented language for protocols called SDL. SDL is more formal with its communication, assumptions, and it is a Formal Description Technique (FDT). SDL is based on an extended finite state machines (EFSM) framework that have asynchronous processes with zero-delay intra-process signals [Tru98] and unbounded delay between process blocks. We see UML not as a replacement for SDL but as a front-end for it. SDL was not at the right level of abstraction to begin designing protocols so we had hoped UML could remedy this. UML is really very general (and still incomplete) but this gives us freedom to explore structures without worrying too much about the details. Thus the main focus of this paper is the exploration of next generation protocol structure, but first let us finish describing the semantics of UML.

UML is still under development and is mainly a notation rather than a formal language right now. UML does not clearly define the communication between objects very rigidly. It uses the notion of message passing, but they can be rendezvous or unsynchronized or different shades of both. Also the existence of queue is implied but the exact type is not specified. Each object in a UML diagram has a class which generalizes that object, and within the class there is a state diagram that describes the behavior. These state diagrams are very much like Harel's STATEMATE for statecharts [HN96]. UML allows states to be hierarchical, concurrent, and nondeterministic defined by \{name, entry/exit actions, internal transition, substrates, deferred events\}. State transitions are defined by \{source state, event trigger, guard condition (on event trigger), action (atomic, local object only), target state\}. States are reactive, but not synchronous and process that occur on transitions or in states have some unknown bounded delay. Actually UML carries no notion of any time, real or abstract.

All this expressive power tends to causes trouble in completely specifying a system for the real-time systems. Its practitioners have recognized these problems and have added some constraints to synthesize usable code. These constraints include fully deterministic state machines and some partial ordering of signals [SL98]. However, it is not

---

1 I use formal loosely. It is however more formal than UML.
2 [LSS98] definition of rendezvous
3 [LSS98] definition of unsynchronized
4 A more detailed description found in [BJR98] pp.291-3
5 [LSS98] definition of synchronous
our intent to construct some underlying MoC\(^6\) for UML. We use UML as tool to visualize object-oriented protocol structures, and in this aspect we found it useful. The message sequence charts helped to describe dynamic behavior, state charts helped describe some internal behavior, class diagrams helped describe the structure.

The remaining outline of the paper is as follows: a section on object-oriented protocol design, a UML description of a communication protocol using our protocol structure, a few wrap-up sections towards the end of the paper.

2 Object-Oriented Protocol Design

Research in object-oriented structures for protocols has had some success. Most of the papers describe a protocol that existed and then model the structure for it. Yet what most of these papers lacked was an insight on what the generalized object-oriented pattern that the protocols took.

We believe that there should be a generalized structure that holds for all protocol layers, and that they should not be constrained to be rigidly layered as in the OSI model. In fact, the designer should design the protocol stack so that it could change dramatically with only local changes. We stress that layering may not be the most effective implementation contrary to popular belief\(^7\). The layering of the protocol stack constrains services performed because it enforces a serial order on the services provided. The whole chain of the protocol stack must be traversed even if some services provided by the stack are not required. The OSI model slightly alleviates this problem by allowing the application to skip to a stack layer, but after the initial jump, the rest of the stack must be traversed. Also sometimes in order to realize real-time packet transfer, error handling, and misordered data it is useful for the application to control frame boundaries. Our idea of the general protocol stack is that the link\(^8\), and physical layers are serial, but the upper layers are unrestricted.

Our first position is that the protocol structure should be modeled as an interface object, and an actual implementation object that handles the actual requests. We arrived at this structure from the merger of a "Interface Pattern" found in the Java context [Doug98], and OSI's\(^9\) service access points (SAP). The "interface pattern" provided a structure for reuse of an implementation by multiple clients, and the SAP's provided a successful framework. In our protocol structure the SAP's have become the interface objects but they no longer reside on the boundaries of only two layers, but two or more layers. In addition, the interface objects have become complex, and are no longer a "dummy" class as with Java interfaces.

Some additional constraints we have added are that the interface contains knowledge to understand either high-level message to primitive message translation or vice versa, but not both. Second, the interface can access any number of implementation objects. Third, interface objects reside in multiple layers and interface-to-interface communication is allowed. Finally, objects that control the same resource will belong in the same layer, and classes that share the same data are really one class that should be merged.

The interface object's purpose is to provide a translation between layers. The specific interface that a client talks to determines a particular service. If a client object requires a different type of service it may use another interface. Figure 1 shows a general structure of our protocol.

3 UML Description of Intercom System

Finally, we get the description of the intercom system in UML. The Intercom is a wireless embedded system designed to be a testbench for designing new radios, and most importantly new protocols. Currently it has been geared towards supporting conferences with other Intercom users. A precondition is that the user must be

---


\(^7\) We advocate that new generation protocols will incorporate Application Level Framing and Integrated Layer Processing, [CT90] for more info.

\(^8\) There are serially linked if all possible protocols traverse a common path, which is usually true for link and physical layers.

\(^9\) Additional Information [Zimm80]
logged-in and have proper privileges for receiving and setting up connections. This high level purpose outlines actually describes most of the control protocol. We used a object-oriented technique defined by [SL98] to derive our classes. Basically this begins by identifying objects in the system, and then the data needed to describe the purpose of the system. This will lead to the evolution of classes and their attributes. Next, operations are defined by identifying what transformations are needed.

Figure 5 shows the top level specification in UML. We found that UML was good tool in describing the general purpose of the system. Looking at the SDL specification proved to be a laborious task because it required sorting through too much detail. With UML a model could be described more compactly. The refinements of the connections between the user and the connection manager are shown in detail at our Website10. The refinements are not the exact translation of the SDL specification. They are mainly our result in defining the protocol stack cleanly.

Using UML, we realized that basically the protocol constructed connection between users. We came about this by using object-oriented analysis. The steps were as follow identify the UML actors, limiting resources, and information needed to function (or use case in UML terminology). Identify methods needed to transform data, and responsibilities if associations exist. Basically these were the core steps that led us to figure 5. From there we refined the model such as assuming the conferences exist in a frequency medium. Therefore new resources, methods, data were needed which led to more objects.

3.1 Example

The message sequence charts (Figures 3 & 4) show an example scenario of how the protocol structure would work. Let’s run through figure 3 first. It starts by having some object (Connection Manager Mobile here but this could change) sending data to an interface IMAC-client-TDMA. The interface translates the high level message to a primitives for the actual implementation of the TDMA MAC. This would correspond to the set tx.slots.active() & set data.to.send(). Next the MAC access another interface (IFront-End Radio-Basic) to transmit the data, and then it modulated by the radio. Now, lets analyze the scenario. The interfaces add another level of indirection and hence a possible cost in reaction time. Yet if we were to replace the radio with an optical fiber, the designer would only need to change the interface IFront-End Radio-Basic (a local change). The new interface would only have to guarantee that it was backward compatible (still provide methods that other client objects invoked). Otherwise the change here can ripple through the whole system. Standardizing interfaces help to achieve smoother interconnections but are not absolutely necessary in the protocol structure. The key point is that bringing in new components will cause a modification of the system, but designer should aim for localizing rather than minimizing change. Figure 2 shows the physical and link layer object in our communication protocol.

The second scenario involves the setup of a FDMA-CDMA channel. Again we start out with a client object requesting the IMAC-CDMA-FDMA interface to send data. This now involves manipulating two implementation objects, FDMA MAC & CDMA MAC, to accomplish its purpose. Figure 2 show that the IMAC-CDMA-FDMA interface object servers two implementation objects. The IMAC interface knows that it must send messages to both MAC’s before it can transmit the data to the radio. Notice now this requires our radio to have a new interface (IFront-End Radio-FDMA) so it can set the oscillator frequency.

---

10 Files are available at http://infopad.eecs.berkeley.edu/~dpatel/UML/intercom/
3.1.1 Results

These scenarios brought up a few interesting points which we are still trying to resolve. First, when an interface sends the primitive messages to a implementation object should it return the thread of control back to the interface? Our example in figure 3 did, but figure 2 did not. So in general it should return the thread of control but this may require many levels of indirection which may lead to a slowly reacting system. Yet, we believe that this could be optimized later. We hold a position that constructing a flexible, if slower, structure in the beginning can later be optimized as needed. Of course, both scenarios could be further optimized by removing the interface objects, and inlining the logic and calls needed from client to implementation. This would require less message passing, but flexibility would be lost. In addition, the boundaries of the protocol may begin to blur. The layers start to depend on each other since they need to be aware of the correct method calls.

4 Achievements

In this paper, we stress that there should exist a general protocol structure which applies to all layers. In addition we stress that the optimum protocol stack is not serially linked. Only the link and physical layer should be serially linked in an embedded system since this is shared by all. The remaining upper layers except for the application layer may be arranged in any configuration. We believe that this leads to the most reuse and flexibility. From here, the designer can streamline the protocol stack and serialize it in order to optimize it.

We defined a protocol pattern that is simple and allows for some flexibility. This pattern consists of a client object, interface object, and implementation object. We have done only analysis, so some experiments on generated protocol stacks is next.

We used UML to describe these protocol structures and use our Intercom project as a working example. We use message sequence charts, state diagrams, and class diagrams in UML to describe our system.

Finally we studied UML use as a descriptive tool, and took a look at the underlying assumptions and properties.

5 Future Work

We plan to translate our description in UML to SDL. This is to see how well we can go to SDL if we decide to use UML as a front-end. In addition, it will allow us an environment to verify our design.

We plan to explore another possible pattern that is similar to the "interface pattern" but with additional objects for resource management and thread control.

6 Acknowledgements

The author would like to thank Fred Burghardt and Marco Sgroi for their mentorship and for numerous insightful discussions.

7 Bibliography


