On the Description of Communications Between Software Components with UML

Zhiwei An Dennis Peters

Faculty of Engineering and Applied Science
Memorial University of Newfoundland
St. John's NL A1B 3X5

zhiwei@engr.mun.ca *dpeters@engr.mun.ca*

November 12, 2003

Abstract

For the purpose of analysis and verification, in software design, architecture of the software system and communications between software components should be specified. Unified Modeling Language (UML) is a standard software design notation that includes Sequence diagrams and Collaboration diagrams, which describe the interaction between objects. They also can be used to describe communications between components. In this paper, we discuss what should be modeled in the communication and how the elements in UML can be adopted to model the communication. A formalism of UML design models that can be used for design analysis is also proposed.

1 Introduction

Software systems are often composed of several components each of which is a computational entity that realizes a particular function. Components interact with each other by communications between them. Documenting communications is one part of software design and the design models should be verifiable to ensure the correctness of the design.

In software systems, synchronous and asynchronous communication could exist. To describe different types of communications between components, several methodologies have been proposed. Two kinds of methods are 1) Architectural Description Languages (ADL) [1] and 2) UML in architecture description [3, 4].

The Unified Modeling Language (UML) [6] is a standard modeling language with rich diagrams to model static and dynamic aspects of a system. Sequence diagrams and collaboration diagrams are two types of UML diagrams widely used in communication description.

In a sequence diagram, the horizontal dimension represents different objects and the vertical dimension represents time. Each object has a lifeline with activation bars. The bar begins with the invocation of a method and stops when the method ends. Arrows represent messages transmitted between the objects. The life line could have a branch at a time point and the two or more lines could merge at a later time. The branch means conditional branch or concurrency.

Sequence diagrams specify time explicitly. In a sequence diagram, objects interact with each other via messages. A message may specify several different time points such as sending time and receiving time. One message links two events and the order of events is specified.

A collaboration diagram presents a set of roles to be played by instances as well as required relationships between them. It also presents a set of messages specifying the interaction between the instances playing the roles to achieve the desired result. In collaboration diagrams, message order is described by adding numbers to arrow labels.

To verify the UML design models, the idea of model checking [2] is proposed. Typically, a model checking algorithm checks properties of a system description based on a (finite) state machine model with parallel composition.

To generate the automata based model, sequence diagrams should be analyzed and translated into state machines. A sequence diagram describes time explicitly so the timing relation of the events can be derived from it. In UML design models for a software system there are usually several sequence diagrams. Analysis of all of these diagrams could generate a state machine based model for model checking.

The rest of this paper is organized as follows: In Section 2, we discuss types of communications between software components and what should be modeled in the communication. Section 3 uses an simplified elevator example to show how to use UML diagrams to describe interactions. In Section 4, we propose one behavior model of the software system and illustrate how information in Sequence diagrams could be mapped to the behavior model. In Section 5, we draw some conclusions.

2 Types of Communications

Communications between software components are either asynchronous or synchronous. The difference between these two classes of communications is that synchronous communications involves blocking operations in the communication. In synchronous communication, the component is suspended after the send operation until it is unblocked by the other partner in the communication. In asynchronous communication, nonblocking operations are used which means that the components will proceed without waiting for the completion of the communication.

The following classes of communications are possible.

Shared Variable A variable that can be accessed by more than one component is a means of communication. The basic operations on a shared variable are read and write so mutual exclusion is the main problem in this type.

Asynchronous Message Passing (AMP) In this type of communication, there are two events 1) the sender sends out the message and continues running, 2) the receiver receives the message. If the receiver is available and the processes are co-located, 1 and 2 happen at essentially the same time and could be considered as the same event. If the receiver is not available, the message is stored in a buffer until the receiver is available. The sender is not blocked at any time.

Synchronous Message Passing (SMP) In Synchronous Message Passing, the sender cannot send the message until the receiver is available to receive it. There is no buffer in this type of communication.

Procedure Call In Procedure Call, there are four events 1) the caller calls an access program in the callee, 2) the access program is invoked, 3) the access program finishes, and 4) the caller knows that the callee finishes. In Procedure Call 1 and 2 happen at the same time and they could be considered as one event. 3 and 4 are also the same event. Between the events of 2 and 3, the caller is blocked.

Remote Procedure Call When there are more than one process and a process calls a function in another process, this type of communication is called Remote Procedure Call (RPC). The mechanism of RPC is almost the same as procedure call except that the function in another process may be unavailable because that function is called by another component and it can not be called twice at the same time.

Asynchronous communication and synchronous communication have the similar semantics and can be modeled in a similar way. For example, synchronous message passing is a special case of asynchronous message passing without a buffer and asynchronous message passing between two components could be modeled as two synchronous messages: from one component to a buffer and then from the buffer to the other component.

Table 1: Concepts in Communication and UML

Communication Concepts	**UML Notations**
Component	Object
Component's Life	Lifeline
Running Access Program	Activation Bar
Messages or Calls (Operations)	Message Arrow
Message Name or Call Name	Arrow Label
Event	Two Ends of an Arrow Two ends of an Activation Bar

3 Description of Communications in UML

The techniques for denoting communication types in UML are defined in UML 1.4. To describe communications with UML, the first step is to map concepts in communication to the elements in UML diagram. Table 1 illustrates the relations of UML notations and concepts of communication in this work.

Operations and events should be distinguished here: operations have time duration and they are often composed of several events which are points in time.

Some concepts in communication cannot be represented by UML notations. For example, the data state is not in UML at all and control state may be represented implicitly.

After mapping concepts of communications to elements of UML, we need to model communications with the semantics of collaboration diagrams and sequence diagrams. To describe the whole system, the first step is to use collaboration diagrams to describe relations between objects, the second step is to use sequence diagrams to describe interactions.

3.1 Collaboration Diagrams in Communication

Because a collaboration diagram presents a collection of instances and their relationships, we can use it to describe the relations between components, as illustrated in Figure 1. The arrows between components represent messages and calls. Since this diagram only illustrates the static relations between components no number is used in the arrow label.

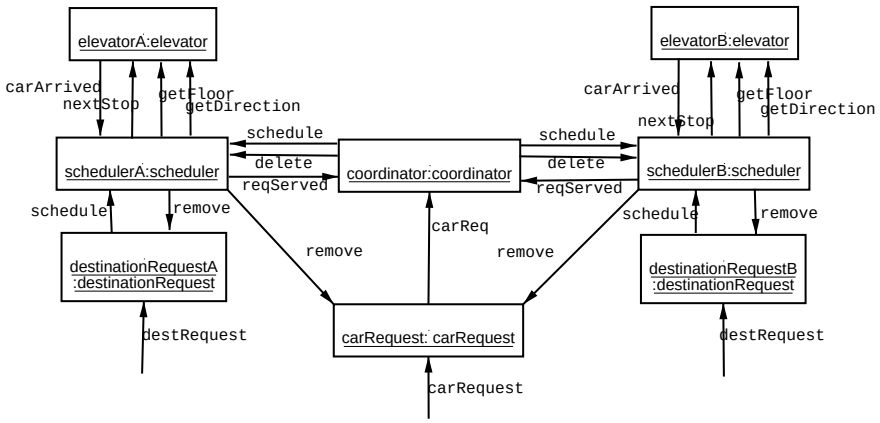

Figure 1: Elevator System in Collaboration Diagram

3.2 Sequence Diagrams in Communication

A sequence diagrams describes several aspects of the communication. First, it can describe the phenomenon of when one access program is invoked, what other events could be generated during the run time of the

access programs. Second, it can illustrate the mechanism of communication.

When one access program is invoked, it may send messages to or call access programs in other components. Messages and calls are distinguished by different types of arrows. For example, Figure 2 shows that when the access program *schedule* is invoked, three operations, *getFloor*, *getDirection* and *nextStop* happen. Figure 3 illustrates the order of the events in the case of AMP.

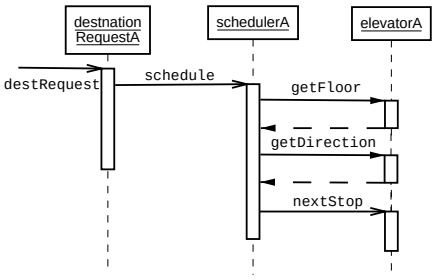

Figure 2: Stimulus in UML, I

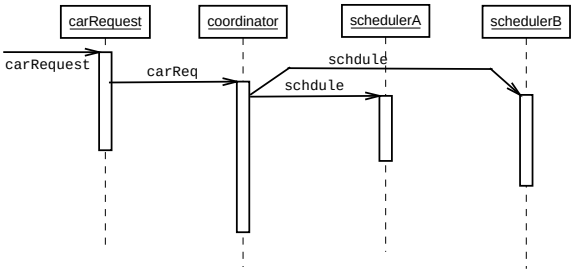

Figure 3: Stimulus in UML, II

Since there are a finite number of access programs in the system and we can draw one sequence diagram for each access program, the number of sequence diagrams is equal to the number of access programs and hence the number of sequence diagrams will not grow too rapidly as the system size increases.

Figure 2 and Figure 3 only illustrate the behaviour when one access program is invoked. More sequence diagrams are needed to completely describe the communications between components, including conflict resolution.

Figure 4 illustrates all types of communications discussed in Section 2. In I, a procedure call is illustrated: component P calls an access program in component Q (operation a). The access program in Q is invoked at the same time as the component P calls it and P is blocked. After the access program in Q ends, P is unblocked. In II, P calls Q (operation a) first and it is blocked until the operation b. If Q is unavailable and R wants to call Q, R is blocked and should wait until the operation b is over. R will be unblocked only after the operation d. In III, P sends a message (operation a) to Q and P is not blocked. If Q is not available before it finishes serving the message form P and R sends a message to it (operation b), a buffer is used to store the message and re-send the message to Q (operation c) after Q is available. In this type of communication, no component is blocked. In IV, P sends a message to Q (operation a), R sends a message to Q (operation b) when Q is not available and R is blocked, this operation can finish only after Q finishes serving the message from P. When the operation b ends, R is unblocked and Q begins to serve the message from R. In V, *read* and *write* are two basic operations and they should obey the rules of mutual exclusion.

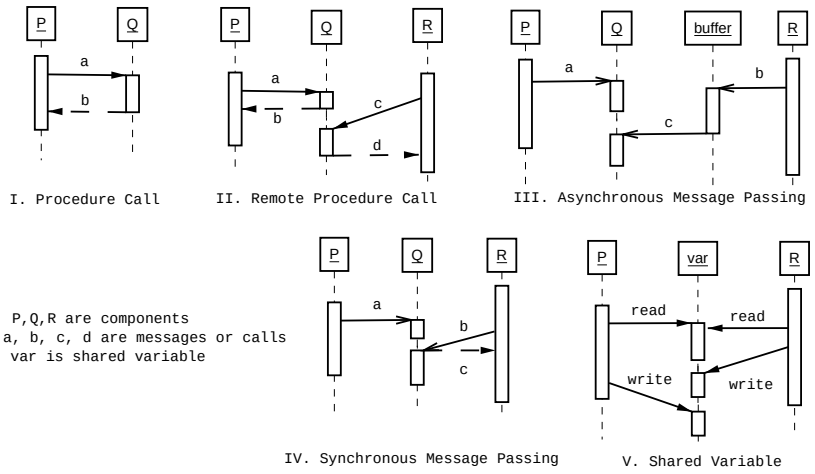

Figure 4: Types of Communication in UML

Figure 4 only illustrates some basic types of communications. In a realistic system mixtures and variants of these communications exist. In Figure 5, the communication are AMP. In this example, the coordinator can access two messages from two schedulers so we can draw two lifelines for *coordinator*. A buffer is necessary when message *delete* cannot be processed immediately after being sent.

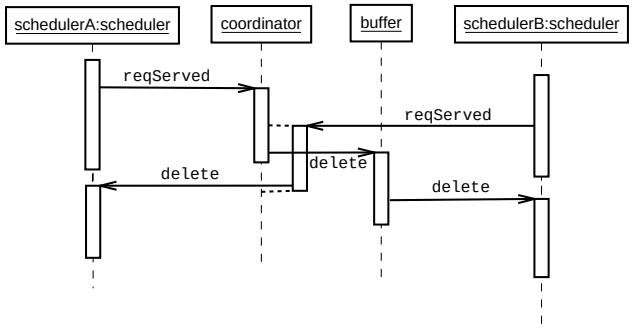

Figure 5: Communication Mechanism

4 Analysis Technique

To verify UML design models, we need a formalism to accept all sequences of events described by the model. In software systems, components are modeled as state machines and most model checking algorithms accept automata based specification as the input, so parallel composition of state machines is the model in this research work.

4.1 From UML to State Machine Model

Sequence diagrams describe the relative order of events. In communications, there are several possible orders of events so the events are not totally ordered. The best description of the relationship of events is a *partial order* $\preceq$.

In [5], relations of messages in Message Sequence Charts (MSC) are translated into a partial order. Because sequence diagrams come from MSCs with extension, a similar process can be used to extract the partial order from sequence diagrams.

UML Sequence diagrams do not have state variables in them so they do not represent component state precisely. UML Statecharts have the ability to describe the behavior of the component but Module Interface Specifications (MIS)[7] make better use of abstraction and are more amenable to machine processing. A discussion of the process for generating a formal behaviour model from component MIS is beyond the scope of this paper.

From the discussion above, we can propose that the problem of synthesizing concurrent automata from Sequence diagrams can be divided into two steps. 1)Describe partial order relation in sequence diagrams formally and 2)synthesizing concurrent automata model from the partial order relations and component MIS.

5 Conclusion

UML interaction diagrams have the ability to describe the communication between software components. Collaboration diagrams describe the relations between components and sequence diagrams describe two aspects of the interaction: 1) when one event happens, what other events could happen, and 2) what communications mechanisms represent the communication types. To verify software design, an automata based behavior model could be derived from the sequence diagrams and used for model checking.

References

[1] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Trans. Software Eng. and Methodology*, July 1997.

[2] E. M. Clark, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[3] Hassan Gomaa. *Designing Concurrent, Distributed, and Real-Time Applications with UML*. Addison-Wesley, 2000.

[4] Christine Hofmeister, Robert Nord, and Dilip Soni. *Applied Software Architecture*. Addison-Wesley, 2000.

[5] Madhavan Mukund, K. Narayan Kumar, and Milind Sohoni. Synthesizing distributed finite-state systems from MSCs. In *Proc. Int'l Conf. Concurrency Theory (CONCUR)*, number 1877 in Lecture Notes in Computer Science, pages 521–535, University Park, PA, 2000. Springer-Verlag.

[6] Rational Software Inc., *et al*. *OMG Unified Modelling Language Specification*, version 1.5 edition, March 2003.

[7] Yabo Wang. Formal and abstract software module specifications—a survey. CRL Report 238, Communications Research Laboratory, Hamilton, Ontario, Canada, November 1991.