Extended Sequence Model for Mobile Embedded Software Execution

Haeng-Kon Kim[1], Eun-Ju Park[1]

[1]Dept. of Computer information & Communication Engineering
Catholic Univ. of Daegu, Korea
{hangkon, ejpark}@cu.ac.kr

Abstract. Mobile embedded software applications are typically more difficult to design and build, both because of the problem domain and the constraints placed on them by the target environment. Embedded systems usually have limited memory and CPU processing power; these constraints are likely due to size, cost, power efficiency, and heat generation limitations. Embedded system design needs to model together application and hardware architecture. For that a huge number of models are available, each one proposing its own abstraction level associated to its own software platform for simulation or synthesis. Mobile and multiplatform solutions even increase this complexity. Testing and managing mobile software applications are important way of handling these complex systems, describing it to different actors inside software business and for analyzing and improving system performance.
In this paper, we present an approach to generate and extend mobile test sequences model using UML. Although sequence diagrams are used for modeling the dynamic aspects of the system, they can also be used for model based testing. Existing work does not encompass certain important features of sequence diagrams like the various interaction operators of combined fragment.

1 Introduction

Modeling is frequently used in other engineering domains as a way to verifying design decisions before committing large resources to constructing or manufacturing a product. Models allow you to perform thought experiments on a design, and to try different approaches to solving the problem at a higher level of abstraction than the raw medium of the constructed product. In mobile embedded software is not typically the case: although modeling tools have been around for decades, the desire, capability, and modeling tool maturity have not been sufficient. This is no longer the case, and using models -- not only to think about the problem but also to drive the design and construction of software -- is a reality. MDD(Model Driven Development) tools can take models at a high level of abstraction and generate executable and efficient source code. Models (in particular visual ones) provide rich information on the structure and behavior of the system. Sharing and working with models on a software development team is now more efficient and less error-prone than working only in source code[1], [2].

This article discusses the use of extended sequence model for mobile embedded software on various resource-constrained embedded systems, and looks at the trade-offs required in order to make and generate message sequences from existing modeling tools. Since sequence diagrams give a macro level view of the implementation under test, both readability and understandability of the model are high [6]. Further UML supports many new features like the "Execution Occurrence for mobile" that are best suited for message sequence generation. The sequence diagrams are more commonly used in practice than communication diagrams. Common practices identify sequence diagrams as the major modeling diagram that captures the detailed behavior of the mobile in the embedded system. In addition, errors in the sequence diagrams are one of the major sources of defects in the final products.

2 Related Work

2.1 Mobile Sequence with UML

Though there are many testing works based on UML models, most of the work deals with UML state diagrams [9], [10], [11], [12]. Hartmann et al. [13] describe an approach for generating and executing system tests. Their approach models the system behavior using UML use cases and activity diagrams. The coverage criteria being applied is transition coverage. Offut and Abdurazik in [11] developed a technique to generate test cases from UML state diagrams, which helps performing class-level testing. The input of this method is a state transition table and the method generates test cases for full predicate coverage criteria. Kansomkeat in [12] described a technique that can automatically generate and select test cases from UML statechart diagrams. Firstly the method transforms this diagram into an intermediate diagram called Testing Flow Graph (TFG) and explicitly identifies flows of UML state chart diagrams. Secondly from TFG, test cases are generated using the testing criteria that are the coverage of the state and transition of diagrams. Briand and Labiche [14] describe the TOTEM (Testing Object –oriented systems with the Unified Modeling Language) system test methodology. Functional system test requirements are derived from UML analysis artifacts such as use cases, their corresponding sequence and collaboration diagrams, class diagrams and from OCL used in all these artifacts. Test sequences are generated from UML sequence diagrams but the work deals with UML 1.3 sequence and hence they do not deal with features like alt, loop, break and opt.
To support control flow analysis an extended activity diagram metamodel, referred to as Concurrent Control Flow Graph (CCFG) is generated. An OCL - based mapping is defined in a formal and verifiable form as consistency rules between a SD and a CCFG, so as to ensure the completeness of the rules and the CCFG metamodel with respect to our control flow analysis purpose and allow their verification.

2.2 An overview of UML sequence diagrams

A sequence diagram shows a set of interacting objects and the sequences of messages exchanged among them [4]. Other than the objects and the messages that are passed among them, sequence diagrams may also contain additional information about the flow of control during the interaction among objects, such as if-else ("if a is true send message x else send message y") or if-then ("if a is true send message x) and iteration (send message n many times)[5][6]. Now we shall discuss some of the features of UML 2.0 sequence diagrams that are relevant in our work.

Object lifeline: A lifeline represents an individual participant in an interaction [3][7]. A lifeline will usually have a rectangle followed by a vertical line (dashed) that represents the lifetime of a participant [7].

Message: A message defines a particular communication between two lifelines of an interaction [7]. The messages may involve calling an operation, sending a signal, creating or destroying an instance [3].

Event occurrence: Event occurrence represents moments in time to which actions are associated (see Figure 1). Event occurrences are ordered along a lifeline [7]. A message has two types of event occurrences, send event occurrence and receive event occurrence. The send event is at the base of message arrow where the message departs from lifeline of the sending object, while receive event is at the point of the message arrow where the arrow hits the lifeline of the receiving object [8].

Guard conditions: Guard condition determines whether their operands execute. The operands execute if, and only if, the expression evaluates to true [3].

Execution occurrence: Execution occurrences represent the period of time during which an instance is active. An execution occurrence is displayed as a thin vertical rectangle that overlaps the dashed line of a lifeline (see Figure 1). Because the execution occurrence has duration, it is represented by two event occurrences, the start event occurrence and the finish event occurrence [7].

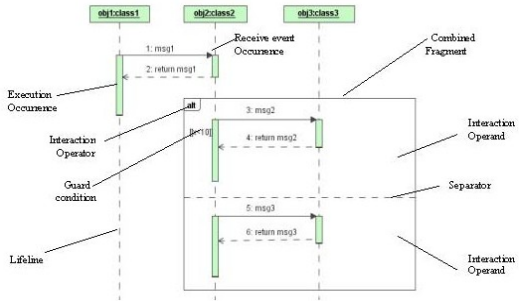

Fig. 1. An example showing some features of UML 2.0 sequence

Combined fragment: Combined fragments divide a sequence diagram into different areas with different Behavior [3]. Each combined fragment has one operator called interaction operator; one or more operands called interaction operands and zero or more guard conditions [3]. Depending on the guard condition, the decision is made on what all operands need to be processed.

Interaction operator: An interaction operator defines how to execute the interaction operands within the context of the combined fragment [8]. Some important interaction operands are alt, opt, loop and break. Alternative (denoted "alt") models if…then…else construct. Opt models if...then constructs. In the Loop (denoted "loop") combined fragment, the enclosed series of messages is repeated until the condition is false. A break (denoted by "break") fragment breaks out of the immediately surrounding fragment.

3. Mobile dependency analysis

Message sequence in our work is a sequence of dependent messages that are guaranteed to execute together. Identifying message sequence helps grouping of dependent messages. To identify message sequence from the code is difficult, particularly when there are many lines of code. The reason is that different methods may be defined in different classes and it can be difficult to trace the methods that are dependent by going through the lines of code. So traceability is difficult and it consumes lots of time too. The advantage of sequence diagrams comes here. Since they give a macro level view of the implementation under test, it is easier to derive message sequence from sequence diagrams than the code. Further, developing it during the design phase will help the developers to check for the correctness even while developing the product and the workload of the testers will be considerably reduced as the developers will be scanning it, prior to the start of test phase. To generate message sequences, first we identify the types of relationship that can exist among the messages. We define four types of relationship USED_BY, IS_PART_OF, IS_FOLLOWED_BY and IS_SUCCEEDED_BY that can exist between two messages mi and m_{i+1}. Based on this, four types of dependencies among the messages are identified in our work, indirect message dependency, direct message dependency, simple indirect message dependency and simple direct message dependency.

3.1 Indirect message dependency

Consider a simple sequence diagram in Figure 2. Here the message *getTotalMarks(rollNo)* return *marks* which is used by the method *getPercentage(marks)* to calculate the percentage. So, there exists a USED_BY relationship between *getTotalMarks(rollNo)* and Consider three consecutive messages m_i, m_{ri} and m_j where m_i is a synchronous message and m_{ri} the return message of m_i. We say that m_i USED_BY m_j if m_j requires the presence of m_i because mi provides the resource (which is the return message m_{ri}) that m_j needs to accomplish the task. If there is a USED_BY relationship that exists between the messages m_i and m_j, then we say that there is *indirect message dependency* between m_i and m_j. If m_{rj} is the return message of the message m_j, which is used by another message say m_k, then we say there is indirect message dependencies among m_i, m_j and m_k. In our work, m_i, m_j and m_k form a message sequence and is represented (using dot operator)as $m_i . m_j . m_k$.

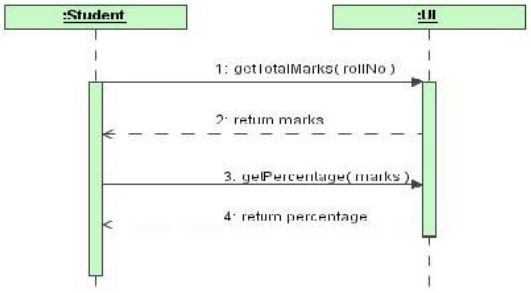

Fig. 2. An example sequence diagram to show indirect message dependency

3.2 Direct message dependency

Let m_i and m_j be two consecutive messages (which are not return messages) in figure 3. We say m_j IS_PART_OF m_i means that m_i is realized by aggregating several messages, one of them being m_j. If there is an IS_PART_OF relationship that exists between the messages m_i and m_j then we say that there is direct message dependency between m_i and m_j. In our work, m_i and m_j form a message sequence and is represented (using dot operator) as m_i. m_j. Consider the Sequence Diagram in Figure 3. Here the message *Dept.Verify Username and Passwd()* IS_PART_OF message *Enter Username and Passwd()* and the message *Dept.SearchResults (RegNo)* IS_PART_OF the message *ShowResults(RegNo)*. So the messages *Dept.Verify Username and Passwd()* and *Enter Username and Passwd()* form one message sequence.

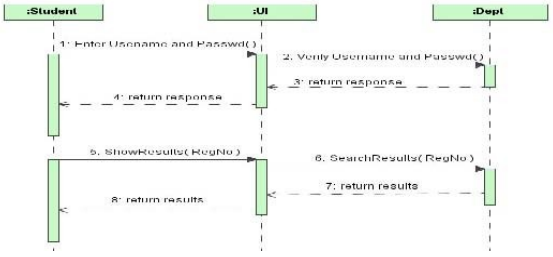

Fig. 3. An example sequence diagram to show direct message dependency

3.3 Simple indirect message dependency

Consider a sequence diagram given in Figure 4. Here the return message opt of the message *enterOption()* is not used by the message *enterNumbers()*. But it is obvious from the sequence diagram that the message *enterNumbers()* always follows the message enterOption() and that if the message enterOption() is executed, then the message *enterNumbers()* will definitely get executed. Since the message enterOption() if

followed by the message enterNumbers(), we say that there is a IS_FOLLOWED_BY relationship between these two messages. We call such type of dependency between the messages as simple indirect message dependency. The only difference between IS_FOLLOWED_BY relationship and USED_BY relationship is that in the case of USED_BY relationship the return message m_{ri} of the message m_i is used by the message m_j immediately following m_{ri} whereas, in the case of IS_FOLLOWED_BY relationship the return message m_{ri} of the message m_i is not used by the message m_j for further processing although the processing of m_i needs to be completed for m_j to start processing. So we can say that IS_FOLLOWED_BY relationship is a variation of USED_BY relationship.

Fig. 4. An example sequence diagram to show simple indirect message dependency

3.4 Simple direct message dependency

Consider two classes class A and class B. If class A calls a method msg1 in class B and that method in class B calls a method msg2 in class A then we say that msg1 IS_SUCCEEDED_BY msg2 and the dependency between such messages are called simple direct message dependency. Consider the sequence diagram in Figure 5. Here the class *num* calls the messages enterOption() in class *processing* which in turn calls the message printOption() in the class *num* which calls the message enterNumbers() in the class *processing* which calls the message printNumbers() in the class num. Thus there is IS_SUCCEEDED_BY relationship between the messages enterOption() and printOption(), the messages printOption() and enterNumbers(), and the messages enterNumbers() and printNumbers().

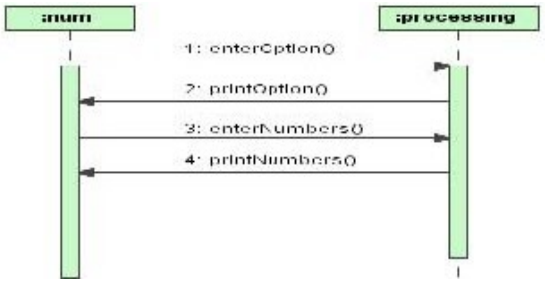

Fig. 5. An example sequence diagram to show simple message dependency

4. Generation of Mobile message sequence

The feature of the sequence diagram that mainly supports the generation of message sequence is the execution occurrence. Execution occurrence is used in sequence diagrams to show the period of time during which an object performs an action. Let us now first see how a message sequence is identified from sequence diagrams. Let msg1 and msg2 be two consecutive messages. If the receive event of msg1 and send event of msg2 sent immediately after msg1 lies between the start event occurrence and finish event occurrence of a particular execution occurrence, then these two messages form a message sequence. The message sequences identified can be direct message sequence, indirect message sequence, simple direct message sequence or simple indirect message sequences. We represent a message sequence comprising two messages m1() and m2() as m1().m2().

The sequence diagram in Figure 6 describes a scenario in which, in order to get access to the student's marks, the student needs to get registered. Once registered, the student will be provided with a student ID. By entering name and ID, he can get his marks. Our task is to develop message sequence directly from this diagram. In Figure 6, we have named each execution occurrence. Execution occurrence is abbreviated as 'Eo' for ease of representation (remember these are not part of sequence diagrams. For ease of explanation this naming is done).

Execution occurrence 'Eo1' begins when a message *newRegistration(details)* is sent from the class *Student* to the class *Registrar*. A new execution occurrence 'Eo2' begins in the lifeline of the instance, *theRegistrar*. From execution occurrence 'Eo2', the message *verifyRegistrationDetails(details)* is sent, which instantiates another execution occurrence 'Eo3'. No messages are sent during the execution occurrence 'Eo3'. The execution occurrence 'Eo3' of the message *VerifyRegistrationDetails()* ends once the message is executed and the control returns to 'Eo2', where the message *generateStudIDandPasswd()* is sent which instantiates an 'Eo4'. After that the return messages are sent and finally the control comes back to execution occurrence 'Eo1' with the *return* message studID, passwd. Remember 'Eo1' is still active. Next *enterStudIDandPasswd(studID,passwd)* instantiates the execution occurrence Eo5'. During its execution occurrence, *verifyStudentNameandID(studID,passwd)* is invoked, which results in an execution occurrence 'Eo6'. Once that message is completed, the control returns to 'Eo5', which is still running. It invokes a message *Enter(studID,passwd)* that results in the instantiation of execution occurrence 'Eo7'. The execution occurrence of the message ends with the return message return marks and finally the control is back to 'Eo1'.

There are two message sequences in this sequence diagram.

Messagesequence#1: newRegistration(details).verifyRegistrationDetails(details)

Messagesequence#2: generateStudIDandPasswd().enterStudIDandPasswd(StudID,passwd).verifyStudentNameandID (studID, passwd) .Enter (studID, passwd). (remember return messages are also considered to check for dependency though they are not included in the message sequence)

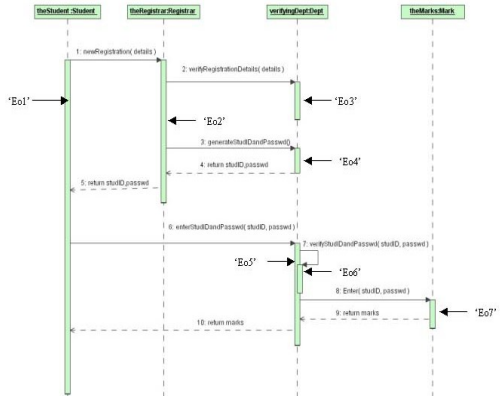

Fig. 6. An example mobile sequence diagram to show Message-Sequence generation

4.1 Generating test sequences from sequence dependency graph

We define a path as a finite sequence of nodes (start, n_1, n_2…n_k, end), k>=1, such that there is an edge (n_i , n_{i+1}) for all nodes n_i in the sequence, except the end node (remember a path in our graph can contain a single node also). To generate test sequences, we need to generate all possible paths from start node to end node. The pseudo code given below generates all possible paths from start node to end node.

```
Start=current;
  Traverse(nodeArray [], current)
  {
    nodeArray [] = Add (nodeArray[ ],current);
  If(current==end)
    {Print (nodeArray []);}
  Else
    For each current.next
       Traverse(nodeArray [], current.next)
}
```

Initially the nodeArray[] is empty and the current node contains the start node. When the method Traverse (*nodeArray [], current*) is called, the current node is added to the nodeArray by calling the function *Add* (*nodeArray, current*).

5. Implementation

Test sequences are generated by using a prototype tool named ESMME, (*Extended Sequence Model for Mobile Execution*) which generates test sequences automatically. This tool is implemented by using java and uses the API of Java for parsing the XML

files. The Figure 8 shows mobile sequence executive environment, and Figure 9 shows mobile sequence execution diagram.

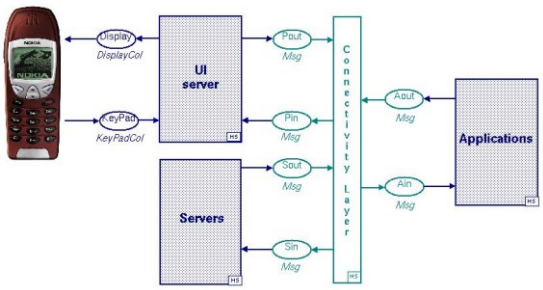

Fig. 8. Mobile sequence execution environment

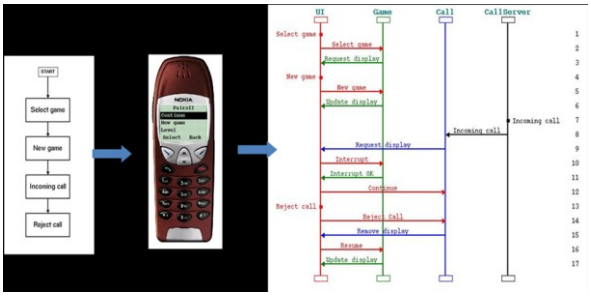

Fig. 9. Mobile sequence execution diagram

The important classes used in our implementation are as follows. The getSD class is responsible for accepting a sequence diagram in XML format. This class displays a prompt to user and asks for a filename. The file name is the name of the XML file to be processed. The class GetElement uses the getElementsByTagName method of the Element object to obtain the list of nodes inside parent node. The Element is an interface of the package org.w3c.dom. It represents an element in an HTML or XML document. The Element mainly extracted is mdElement. The element mdElement has many attributes. The most important attributes of the element that are used in our work are *CombinedFragmentView*, *SeqMessageView*, *OperationView* and *Activation*. The class ProcessDetails is responsible for processing the data obtained from the class. The class Generate SDG generates the sequency dependency graph. The class TraverseGraph performs a depth-first exploration of the graph and generates the test sequences. The class FormatSequence is responsible for formatting the output. The GUI was developed using the swing component of Java. From an input XML file representing a sequence diagram, ESMME generates the test sequences with no human intervention.

6. Conclusion

We have proposed a method to generate mobile test sequences from UML 2.0 sequence diagram. The method first transforms sequence diagram to an intermediate form called the sequence dependency graph from which test sequences are generated. We have proposed a method to identify message sequences from UML 2.0 sequence diagrams. For generating message sequences, first we defined the types of relationship that can exist among the messages. Based on this, four types of dependencies are identified in our work. The dependencies are indirect message dependency, direct message dependency, simple indirect message dependency and simple direct message dependency. The message sequence is identified using an important feature of UML 2.0 sequence diagrams called the "Execution Occurrence". To our understanding, there are no approaches that use execution occurrence in order to find the dependency among the messages. Our method can be used earlier in the software development life cycle when the UML design of a system becomes available. Since the sequence diagrams are used for use case realization, the test sequence generated from sequence diagram can be used to support functional system testing. In UML, sequence diagrams are used to illustrate use case diagrams. Hence they contain scenarios that depict the sequence of executions of the system, and tests can be derived from the use-case model and it's corresponding UML diagrams. Hence our work also supports scenario based testing.

7. References

1. Dobing, B., Parsons, J.: How UML is Used. In: Communications of the ACM, Vol. 49, No. 5, pp. 109--113 (2006)
2. Introduction to OMG's Unified Modeling Language, Version 2.0. Object Management Group, http://www.omg.org/gettingstarted/what_is_uml.htm
3. Arlow, J., Neustad, I.: UML2 and the Unified Process: Practical Object-Oriented Analysis and design. Pearson Education (2005)
4. Quatrani, T.: Visual Modeling with Rational Rose 2002 and UML. Pearson Education (2005)
5. UML 2.1.1 Infrastructure Specification, Object management group, www.omg.org
6. Rumbaugh, J., Jacobson, I., Booch, G.: UML Reference manual. Addison-Wesley (1999)
7. UML2.1.1 Superstructure Specification, Object management group, www.omg.org
8. Garousi, V., Briand, L., Labiche, Y.: Control flow analysis on UML 2.0 Sequence diagram. In: MDAFA. LNCS, vol. 3748, pp. 160--174 (2005)
9. Cavarra, A., Crichton, C., Davies, J.: A method for the automatic generation of test suites from object models. In: Information and Software Technology, vol. 46, no. 5, pp. 309--314. (2004)
10. Kim, Y. G., Hong, H. S., Bae, D. H., Cha, S. D.: Test cases generation from UML state diagrams. In: Software, IEEE Proceedings, vol. 146, issue. 4, pp. 187--192 (1999)
11. Offutt, J., Abdurazik, A.: Generating tests from UML specifications. In: Proceeding of the UML'99, LNCS, vol. 1723, pp. 416--429 (1999)
12. Kansomkeat, S., Rivepiboon, W.: Automated-generating test case using UML statechart diagrams. In: Proceedings of SAICSIT 2003, ACM, pp. 296--300 (2003)

13. Hartmann, J., Imoberdorf, C., Meisinger, Michael.: UML-Based integration testing. In: Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis, pp. 60--70 (2000)
14. Briand, L., Labiche, Y.: A UML-based approach to system testing. In: Journal of Software and Systems modeling, vol. 1, no. 1, pp. 10--42 (2002)