CHAPTER VI

MESSAGE SENDING SEQUENCE & VERIFICATION

6.1 Overview

Message sending is a way to express object communication in an interaction. A sender object sends a message to request a receiver object to fulfill a specific service [11]. "Message" is a design term and is usually in higher level and conceptual; hence, it can be interpreted into several different meanings in implementation. "An object sends a message to another object" can be interpreted as "an object calls an operation on another object", "an object posts an event which causes another object, as an event handler, to be executed", or even "an object asynchronously calls an operation on another object". Although there are a lot of possible types of messages, only synchronous operation call messages are considered in this research.

A message sending sequence is a sequence of messages flying between objects in an interaction under a specific scenario. It can be used for representing both an object interaction, which is illustrated in an interaction diagram, UML sequence diagram in particular, and an object interaction, which actually takes place in program execution. From now on, they are called expected message sending sequence and actual message sending sequence respectively.

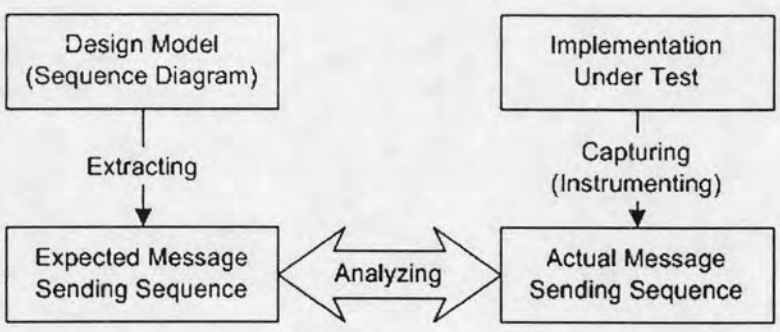

Figure 6.1 Overview of Message Sending Sequence Verification

Verification of message sending sequence is a part of this software testing approach. As stated earlier, the approach aims to detect any difference between interactions in the design model and the implementation. Any differences between the message sending sequences are considered as a defect. As shown in figure 6.1, the expected message sending sequence extracted from the design model is analyzed against the actual message sending sequence instrumented from the execution of the implementation.

6.2 Concept of Message Sending Sequence Model

The basic goal is to have both expected message sending sequences and actual message sending sequences represented in the same model to ease the sequence verification. The proposed model must consist of all common features of both message sending sequences, while must also avoid contradiction or confusion from both message sending sequences.

There are two views of requirements that influence the design of message sending sequence model. Instrumentation model requirements describe the requirements of representing actual message sending sequences, while the requirements of expected message sending sequence is discussed as design model requirements. Each of them is discussed in the next subsections.

6.2.1 Instrumentation Model Requirements

The basic requirement of the instrumentation model is that the model must represent message sending sequence between objects which occurs in an execution of object-oriented software. The model is significantly different from instrumentation models which capture only function entry/exit in structural programs, since object-oriented software makes uses of objects. An object is a cohesive piece of data and operations that manipulate the data. It is different from a structure type in structural programming which contains only data; the manipulation operations live apart. Moreover, each object has an identity of its own which distinguishes itself from other objects. Two objects, although contain identical values of data members, have different identities; hence, they are not identical. This difference is the major requirement of the model.

As polymorphism is a specific feature of object-oriented paradigm, discovering faults related to polymorphism is the main purpose of an object-oriented testing technique. As a consequence, the model must be capable of representing polymorphic operations.

Beside the issues addressed above, the requirements are similar to usual function entry/exit instrumentation models. The model must be able to address call hierarchy, called function name and arguments (to identify the specific function), and context of the call (to identify where the function is called).

6.2.2 Design Model Requirements

The model, which satisfies requirements in the previous subsection, should also be capable of representing message sending sequences from design models, as the general information about the message sending sequence (message sender, message receiver, message action, and information about the operation and class where the execution occurs) are similar. Although for now we design the model to support message sending sequence comparison between the execution and the design model, the model should be extensible to support further analysis, for example return condition (usual return or thrown exception), and returned value verification. Since this part of the requirements is not in the scope of the research, this part of the model must be open for possible future extension.

6.2.3 Discussion

There are two assumptions about the model. First regarding the interpretation of message sending as described earlier, only the message sending which is an operation invocation is supported. Another assumption is about thread of execution. It is allowed to present concurrent execution in a single UML sequence diagram, by using notations like asynchronous call, return call, and object lifeline. However, testing for concurrent execution is complicated. Specific test techniques and instrumentation techniques are required to support concurrency. As testing asynchronous operations is beyond the scope of the research, the current model supports only a single thread of execution for both the actual execution and the design model.

As UML sequence diagram allows a guard condition to be attached to a message in a diagram, a single sequence diagram can represent a main scenario with several alternatives. An instance of the model is a representative of only one scenario; therefore, guard conditions are not currently considered. Further extension of this model, however, might include guard conditions for analysis purpose.

There may be a question why UML semantic is not appropriate for the requirements discussed so far. As a matter of fact, it suits the requirements to a degree. However, its structure is not exactly suitable for representing message sending sequence in this case. For UML sequence diagram, all messages in an interaction are contained in a single list. They are not arranged as a call hierarchy. This is necessary for UML sequence diagrams, since they can also represent concurrent operations which do not fit into a call hierarchy. In this research, only synchronous calls are considered, as a result, a model with call-hierarchy-like structure is more appropriate for representing message sending sequence. Moreover, the model should have a structure similar to dynamic call graph [40], which does not share a node when there is more than one call to an operation.

6.3 Message Sending Sequence Model

The model of message sending sequence is shown as a class diagram in figure 6.2. In subsequent subsections, each element in the class diagram is explained.

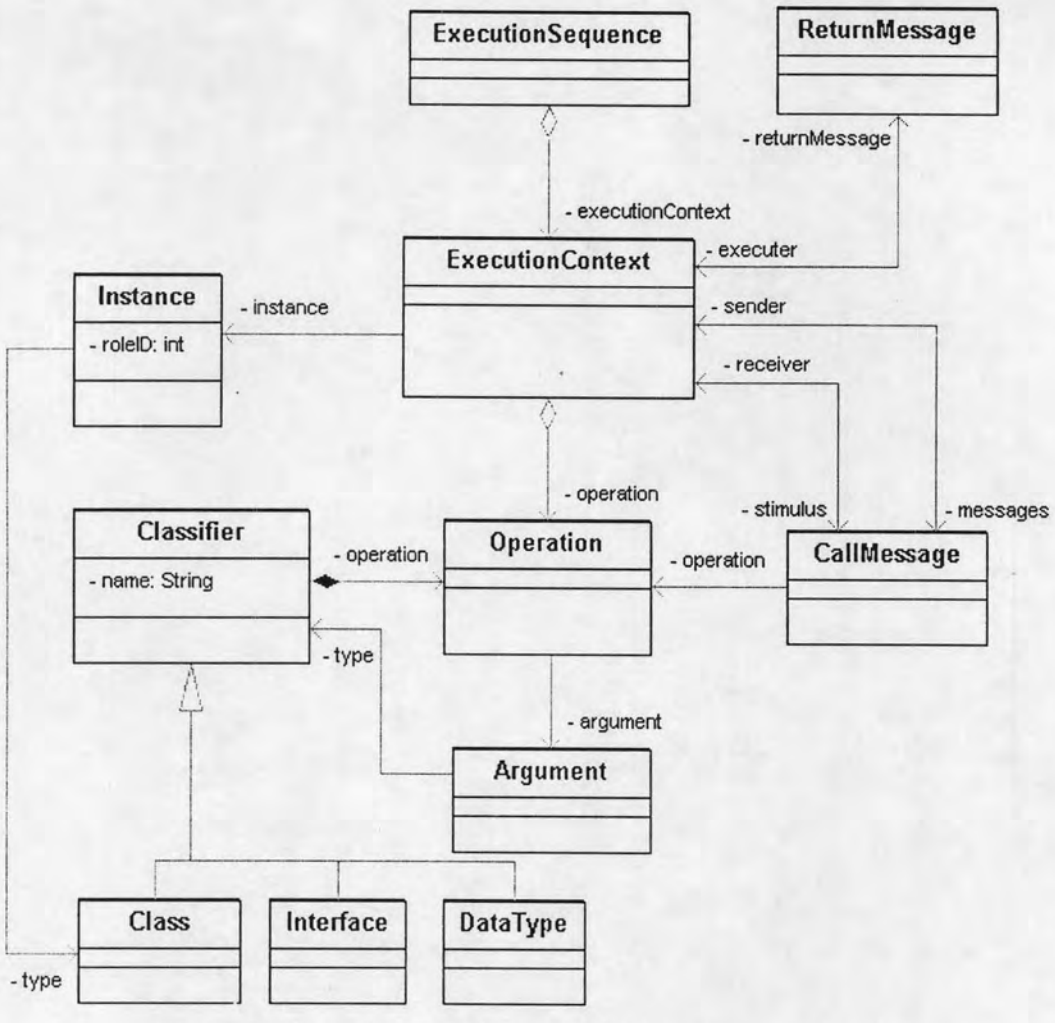

Figure 6.2 Class diagram of message sending sequence model

6.3.1 ExecutionContext

An instance of ExecutionContext represents an execution of an operation on a particular object under a specific circumstance. An execution may send messages to other objects and results in executions in other instances of ExecutionContext. Therefore, an instance of ExecutionContext can be a message sender, a message receiver, or both at the same.

As a message sender, a context has a reference to an ordered list of messages it sends. The messages are sorted in chronological order. Each message represents call and implicit return of the call. Using depth first traversal on the context hierarchy yields the sequence of message sending for entire interaction chronological order.

For the execution model, there are 2 aspects of operation attached to ExecutionContext: a direct associated operation in the execution context as an operation actually executed, and an operation associated to the stimulus message as an operation the sender intends to invoke. This is for representation of polymorphism in the execution. For design models, where polymorphism is usually not explicitly presented, both operations are always identical.

Unless the execution is on class operation, an execution context must be associated to an instance of Instance. The instance represents an object of the class where the operation of the ExecutionContext and the operation of the stimulus message are defined or inherited. This is not an issue in some object-oriented programming languages (i.e. Java, C++), since they perform static type checking which already prevents calling an invalid operation on an object.

6.3.2 Instance

An Instance element is a representative of an object in an execution of an object-oriented program. It is similar to ClassifierRole in UML semantic. The major difference is the association with type information. ClassfierRole can have more than one base Classifier as its types, because an object in the collaboration can play roles of several Classifiers. However, an object in the implementation usually has only one concrete class as its type.

In an interaction, several instances of the same class possibly play different roles. It is important for the model to identify instances in an execution, as the verification needs to compare the instances of the sender and the receiver of each message as well as the message itself. In the design model, instances are identified by assigning a role to each instance. The instrumentation model must also support instance identification by, for example, using generated object runtime ID. Although mapping between roles in the design and the execution is not possible, improper instance usage is recognizable.

6.3.3 CallMessage

A CallMessage element is a representative of a message sent from a sender object to a receiver object. Comparing to UML semantic, CallMessage is similar to

CallAction, which also represents a message of an operation call. An instance of CallMessage has references to its sender and its receiver, which are both ExecutionContext elements. The ExecutionContext of the sender is the context where the message is issued, while the ExecutionContext of the receiver execution context is the context that the receiver instance executes as the effect of the message. As an ExecutionContext element has a reference to an operation it is associated to, it is possible to trace from a message to the operation which is the origin of the message and the operation which is the target of the execution.

In addition to the context of the sender and the receiver, a CallMessage element also has a reference, named "call", to an operation as the target of the call. The purpose of the reference is to accommodate polymorphic interaction. Along with the operation associated to the ExecutionContext of the receiver, the call association on the call message is the key to evaluating adequacy criteria according to polymorphism.

6.3.4 ReturnMessage

A ReturnMessage element contains information about return value or return condition. As stated earlier, the return information from both the design model and the instrumentation model are important if we want to perform analysis on the execution result or condition, in addition to message sending sequence. For design model, this element may contain information specified in the model as return message of the execution, if any. For the execution model, it may represent actual return value or condition captured from the execution, probably the thrown exception. However, we do not have particular requirements on this issue yet, so the element is simply provided for further extension.

6.3.5 ExecutionSequence

An ExecutionSequence element is a container element which wraps all other elements. It has an association to an execution context, which is the context of the actor of the interaction. The actor, as the stimulus, initiates the message sequence by sending messages to other objects. This element may contain other information which is applied to all elements in the interaction as global attributes of the interaction; for example, an

identifier which is assigned to a particular execution scenario for further reference or the condition of the scenario collected from guard conditions of the sequence diagram.

6.3.6 Classifier and Operation

Though sharing the same name, the Classifier element in this model is different from Classifier in UML semantic. In UML, a Classifier element is a base element for others, like Class, Interface, Components etc., providing general attributes to support different aspects of modeling. In our model, Classifier, as a base class for Class, Interface, and DataType elements, only provides the "name" attribute common to all of its subtypes. The only purpose of Classifier element in our model is to provide a common structure for its subtypes.

An operation is identified with its signature which includes its name, the list of its arguments along with their types, and the class where it is declared.

6.3.7 Class, Interface, and DataType

As described in the previous subsection, Class, Interface, and DataType elements are subtypes of Classifier element. They inherit an attribute, name, from Classifier element. The attribute represents name of class, interface, and data type respectively.

We designed these elements based on well-known object-oriented programming languages, Java and C++. Class element is apparently required as these languages are class-based. These languages, while provide support for most of object-oriented features like inheritance and dynamic binding, are classified as hybrid between procedural programming and object-oriented programming. They do not require everything to be an object, so there are data elements that are defined as primitive types, which are represented by DataType elements in our model. In addition, Java has an additional construct, interface, which is considerably different from class in many senses; as a consequence, Interface element is designed to support this construct.

6.4 Verification Issues

Basically, elementary characters such as operation signature, order of messages, sender objects, and receiver objects are of concern to verification criteria of

message sending sequence. Correctness of these characters is the majority of the correctness of overall message sending sequence. Incorrect method signature, invalid order of messages, and faulty sender or receiver object possibly exhibit a malformation in implementation. These characters must be verified in order to achieve basic conformance of the implementation under test.

However, the verification procedure of message sending sequence is not straightforward, as polymorphism and refinement can cause the actual message sending sequence to be different in some degree from the expected message sending sequence. If an interaction is a polymorphic interaction, it is possible that the polymorphic object is of any subclass of the class explicitly defined in the code. So the behavior of a polymorphic interaction under actual execution is unpredictable prior to execution. Verification of such interaction must take into account the fact that the object can be of any class in the inheritance hierarchy.

Additionally refinement is a very important issue to be considered. Interactions defined in design model are usually implemented with more details in the implementation; therefore, the expected message sending sequence extracted from design model contains interactions in coarser level than the actual message sending sequence instrumented from the implementation under test. Although this poses problems in testing, it is not uncommon in practice to do so. This issue complicates verification of message sending sequence. It is impossible for the verification procedure to comprehend refinement context without additional information from developers or testers. Even with this kind of information, verification may become imprecise if the implementation is applied with a high degree of refinement. In this case, human decision is more reliable in terms of accuracy.

6.4.1 Consideration of Defects

Before discussing about the procedure, an insight of what is considered as a defect is required. Essentially, an interaction is considered to be defective if there is any significant difference between the interaction in the design model and the implementation, although the difference is not necessarily a defect for every case. Since the main purpose of the test approach is to ensure conformance between the design

model and the implementation, the verification procedure must seek for differences between them.

For example, two operations, defined to be called in an order in design model, are called by the implementation in the reversed order. If the operations are designed to be called in a particular order, violating this is undoubtedly a defect. On a contrary, if the operations are independent to each other in terms of calling order, changing the calling order does not affect overall behavior of the interaction; hence, it is not a defect in a sense of software correctness. The verification procedure, however, always considers this difference in the interaction to be defective, even though the latter case is essentially not. For the latter case, although the difference is harmless, there is no proper reason to allow it.

6.4.2 Polymorphism

For a polymorphic interaction, a call to a method can result in an execution on an instance of any subclass of the declared class; therefore, the verification procedure must allow the class of the receiver object to be any subclass of the declared class. However, there are two views of the class of an object in an interaction. The first view is the class of the object reference defined in design model or source code, and the other view is, as addressed above, the class of the actual object under execution. Figure 6.3 shows an example which illustrates the difference of both views. The upper section shows declaration of class A and B. Class B is a subclass of class A, and it overrides "perform" method from class A. The lower section shows a snippet of code which makes a method call to an instance of class A. As "doSomething" method takes a reference to an instance of class A as its argument, it is possible that an instance of any subclass of class A is passed as the argument. Therefore, the class of the receiver object in the source code is class A, while the class of receiver object in the actual execution is either class A or class B. The verification procedure must take into account the possibility that an interaction can become polymorphic which results in variance of the class of the receiver object.

```
class A {
      public void perform() {…}
}

class B extends A {
      public void perform() {…}
}
```

```
public doSomething(A obj) {
      obj.perform();
}
```

Figure 6.3 Example source code of polymorphic interaction

However, note that this is only important in the case that the test scenario does not enforce testing of a particular subclass. If a particular subclass is required to be tested by the test scenario, the verification procedure must strictly check that the class in the execution matches the class defined in the test scenario.

6.4.3 Refinement

As stated earlier, refinement is a significant issue that prevents us from directly comparing the actual message sending sequence to the expected message sending sequence. In this subsection, issues about refinement are discussed along with approaches to overcome them. Considering these issues, verification of message sending sequence can verify message sending sequence in the context where refinement is applied.

The most common refinement is to introduce interactions with common classes in implementation, while they are usually omitted from design model. Example of common classes, in this case, are utility classes like collection classes and elementary classes, e.g. String and Integer in Java, which provide primitive services to other application-specific classes. However, there is no standard judgment whether a class is a common class or not. Sometimes its realm is extended to include architectural classes and some application-specific classes as well. In different situations, designers seem to have different criteria whether to treat a class as common class and exclude them from

design model. In some situations, a class, says java.io.PrintWriter in Java, is too elementary to include in design model. For example, in a case where the design model represents an interaction of a business-related activity, including PrintWriter class is too detailed. But in other situations, it may become essential. For instance, modeling a logging mechanism essentially requires PrintWriter to be included in the interaction.

This kind of refinement can be handled in the verification procedure with a supplied list of common classes. Interactions with these classes in the actual message sending sequence are skipped during verification of message sending sequence. If the list of common classes is complete and the implementation strictly follows the design model, the expected message sending sequence will be identical to the actual message sending sequence. The issue of disagreement whether a class is common is not a problem, as long as the list of common class is adjustable. In addition, for each project, the agreement of whether to include a particular class in the interactions in design model or not is usually required to be established earlier in the design phase, in order to have every design model written under the same convention. Since it is a good practice to have such a design standard, the list of common classes is considerably sufficient in solving this kind of refinement.

Another kind of refinement is an introduction of new operations, usually private ones, to the classes in the interactions. There are several purposes for this kind of refinement. New operations may be introduced to promote reusability of the code. For instance, several operations in a class, which involve in several interactions, may comprise some functions in common; therefore, new operations are introduced to remove redundant code. This practice helps in improving reusability and also eases code maintenance. Another purpose of this kind of refinement is for breaking down a long operation. It is always a best practice in procedure-oriented programming to break a large procedure into several manageable-size sub-procedures. This practice is also applied well in object-oriented approach, generally, in a form of additional private operations as described above. However, this kind of refinement may not be planned beforehand, as programmers may find it appropriate to introduce new operations during coding. It is acceptable, since it does neither significantly alter the design nor have any effect to the overall program.

Handling this kind of refinement is rather harder than the previous one, because the list of common classes (or operations) is insufficient to identify refinement. As state earlier that this kind of refinement is usually not planned beforehand, providing the list of additional classes or operations is not possible. However, a simple implication about the refinement can be made based on several rules. For example, a called to a private operation not defined in the interaction in design model is classified as refinement. Other rules can be established in similar fashion using naming convention, e.g. an operation whose name begins with an underscore is a refinement operation, or scope modifier, e.g. an additional private operation is a refinement operation. An interesting option is to consider all operations not defined in class diagram during design to be refinement operations and skip them during the verification of message sending sequence.

Despite various choices of rules, verification of message sending sequence where this kind of refinement is employed is not completely precise, as rules are usually not followed strictly, and there are some exceptions. Precision depends on the rules selected to be applied and the degree of the refinement applied in the implementation.

6.5 Verification Procedure

There are three assumptions for the verification procedure presented in this section. These assumptions are essentially required to resolve verification issues discussed in the previous section. The first assumption is that inheritance hierarchies of the classes in the interactions are available in one form or another, as they are necessary for verification of polymorphic interactions. Also note that this assumption is only required when the test scenario does not enforce testing of a particular subclass.

Other assumptions are for identification of refinement. One of the assumptions is that a list of common classes is established. This assumption is important to overcome the problem where designers exclude interactions with some classes, considered as common classes, from the design model as discussed in the previous section. The other assumption is for the cases where developers introduce new operations in the implementation. Although there are many possible rules to identify additional operations for refinement, it is assumed that any additional private operation is an additional refined operation, in order to simplify the verification procedure.

Based on the issues in the previous section, the assumptions discussed in the previous paragraph demonstrate how the verification procedure deals with the issues. Referring to the overview of verification of message sending sequence in figure 6.1, the verification procedure is elaborated in figure 6.4. Following subsequent subsections describe verification procedure on each element of message sending sequence, beginning with ExecutionSequence.

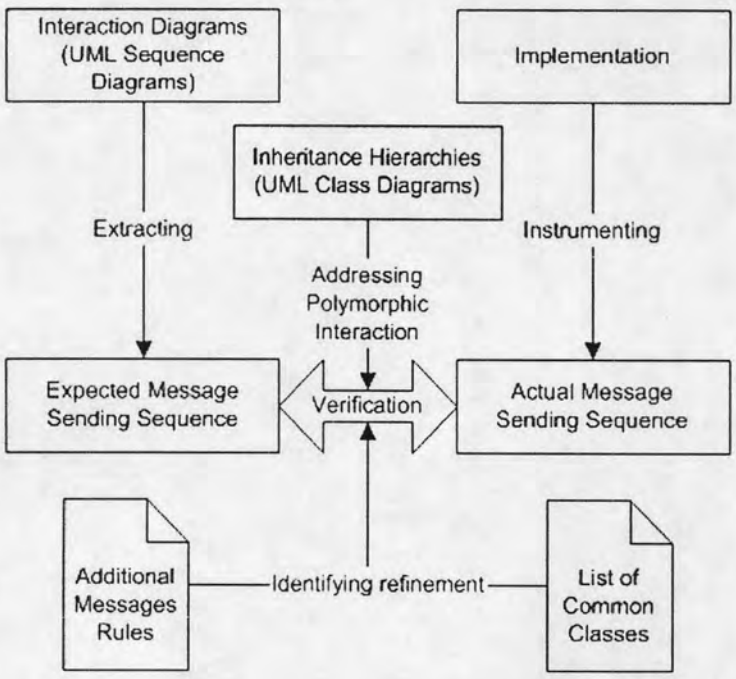

Figure 6.4 Overview of verification procedure

6.5.1 ExecutionSequence Verification

Starting from the root ExecutionSequence element, each associated ExecutionContext as an actor from each message sending sequence is the starting point for the analysis. It is verified as an ExecutionContext as explained in the next subsection.

6.5.2 ExecutionContext Verification

For an ExecutionContext, each of its attributes is analyzed. Regarding the associated Instances in both message sending sequences, the Instance in the actual message sending sequence must have its classifier as one of the classes under the

inheritance hierarchy of the classifier of the Instance in the design model. Next the role of the Instance must be examined. In the expected message sending sequence, roles are identified from the design model. During instrumentation, when an Instance in the actual message sending sequence is reached for the first time, a unique identifier is assigned to it, and the identifier is used to match to the role of the corresponding Instance from the expected message sending sequence during verification. For the entire verification, roles from both message sending sequences must match to each other consistently.

Operations from both message sending sequences must have exactly matching signatures (operation name and argument type list). The procedure is summarized as pseudo code in figure 6.5. The list of sent messages is analyzed in the next subsection.

```
verifyExecutionContext(expectedCtx,actualCtx: ExecutionContext)
begin

# verify classifier of instances
# if polymorphism is not allowed, the classes must matched exactly
if not (actualCtx.instance.classifier is or is a subclass of
expectedCtx.instance.classifier)
      report defect "Object is not of one of expected classifier"
end if

# verify role usage
if expectedCtx.instance.role and actualCtx.instance.role are
encountered for the first time
      put a map entry of expectedCtx.instance.role and
      actualCtx.instance.role in rolemap
else if expectedCtx.instance.role and actualCtx.instance.role are
encountered before
      if a map entry of expectedCtx.instance.role and
      actualCtx.instance.role is not in rolemap
            report defect "Invalid role usage"
      end if
else
      report defect "Invalid role usage"
end if

# verify operation
if expectedCtx.operation not match to actualCtx.operation
      report defect "Mismatched operation signature"
end if

# verify call message
if expectedCtx.sentMessages or actualCtx.sentMessages are not empty
      verifyCallMessages(expectedCtx.sentMessages,
      actualCtx.sentMessages)
end if

end
```

Figure 6.5 ExecutionContext verification

6.5.3 CallMessage Verification

For an ExecutionContext, its list of sent messages (as CallMessage) is analyzed. Issues on refinement significantly influence this step, as there are additional messages in the actual message sending sequences as refined interactions. Basically the message from the actual message sending sequence, which is not in the expected message sending sequence, is examined whether it is a refined message. If a message is a refined message, the verification for the message and its enclosed messages are skipped. Criteria to examine refined message are as described earlier in this section about the assumptions.

Checking whether a CallMessage is equivalent to another CallMessage includes checking its receiver (ExecutionContext) and checking Operation of the CallMessage itself. Unlike the Operation of the ExecutionContext, the Operations associated to CallMessages must match exactly for both their signatures (name and list of argument type) and their classifiers. These Operations from both message sending sequences are required to be identical, since each Operation reflects the operation the sender sees as the target of its sent message. Verification of receiver of CallMessage follows the criteria in previous subsection, which may involve recursively checking the lists of sent messages using the criteria defined in this subsection. The pseudo code of the verification is shown in figure 6.6. The verification goes through the hierarchy of message sending until an ExecutionContext element without sent message is reached.

```
verifyCallMessages(expectedMsgs,actualMsgs: CallMessage[])
begin

# verify each call message in expected call message list
foreach exptMsg in expectedMsgs
begin foreach
      # fetch call message from actual call message,
      # skipping refinement message
      actlMsg = next actualMsgs element
      while exptMsg.operation not match to actlMsg.operation
            if actlMsg is refinement message
                  actlMsg = next actualMsgs element
            else
                  report defect "Mismatched call message"

      # verify receivers of the messages
      verifyExecutionContext(exptMsg.receiver, actlMsg.receiver)
end foreach

end
```

Figure 6.6 CallMessage verification

6.6 Example

In this section, verification of message sending sequence is demonstrated. To capture actual message sending sequence, a Java program instrumentation tool is implemented, utilizing bytecode manipulation provided by AspectJ [41]. The tool is just for convenience in capturing actual message sending sequence from Java program execution.

A simple version of bank account withdrawal operation is selected as an example to demonstrate appropriateness of the approach. Shown in figure 6.7 is a UML sequence diagram of the example. An instance of BankApp class responds to a message to its "withdraw" operation by fetching an instance of Account class, according to the given account id as a parameter of the "withdraw" operation; then messages to "withdraw" and "getBalance" operations are sent to the fetched instance of Account. The value returned from the instance of BankApp is the return value from

"getBalance" operation of the instance of Account, which is the remaining balance after withdrawal.

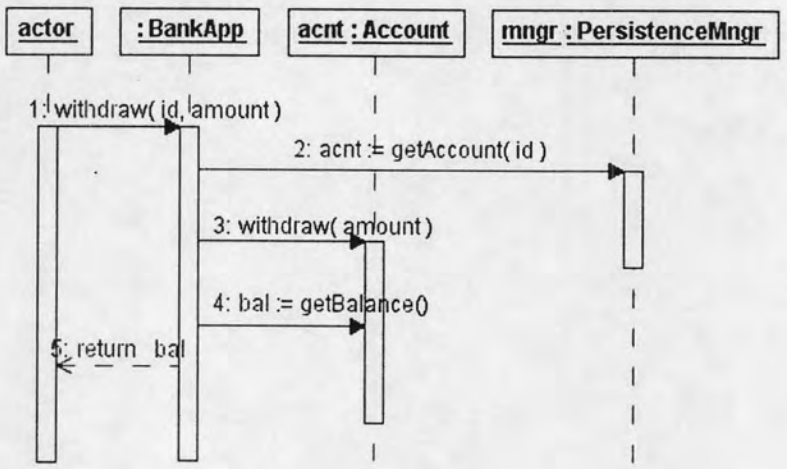

Figure 6.7 Example of a UML sequence diagram

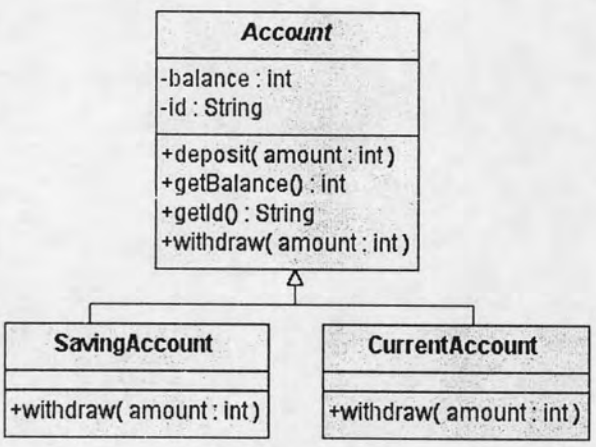

Figure 6.8 UML class diagram showing the inheritance hierarchy from the example

In addition to the classes in the interaction, two additional classes, SavingAccount and CurrentAccount, are introduced in the class diagram in figure 6.8. They are representative of two types of accounts, saving accounts and current accounts respectively. In the example, it is required that distinct withdrawal criteria are assigned for the account types (e.g. each account type requires the different minimum amount of balance that must remain in the account). Therefore, both SavingAccount and

CurrentAccount classes override the "withdraw" method from Account class. Since the "getAccount" operation of PersistenceMngr class returns an instance of either SavingAcccount or CurrentAccount according to the account type of the given account id, the interaction in figure 6.7 is an illustration of withdrawal process for both types of accounts.

The implementation according to the interaction is instrumented and verified against the expected message sending sequence. Figure 6.9 and figure 6.10 show the expected message sending sequence and an example of actual message sending sequences respectively. The message sending sequences are shown in graphical format for the convenience of presentation regardless of their actual persistence format. An ExecutionContext element and a CallMessage element are represented by a rectangle and an arrow respectively. The direction of an arrow identifies message sender and receiver; an arrow goes from a sender to a receiver. For each of either ExecutionContext or CallMessage, its label presents associated operation and classifier. The other information in message sending sequence model (i.e. Instance), which is unnecessary for the discussion in this section, is left off.

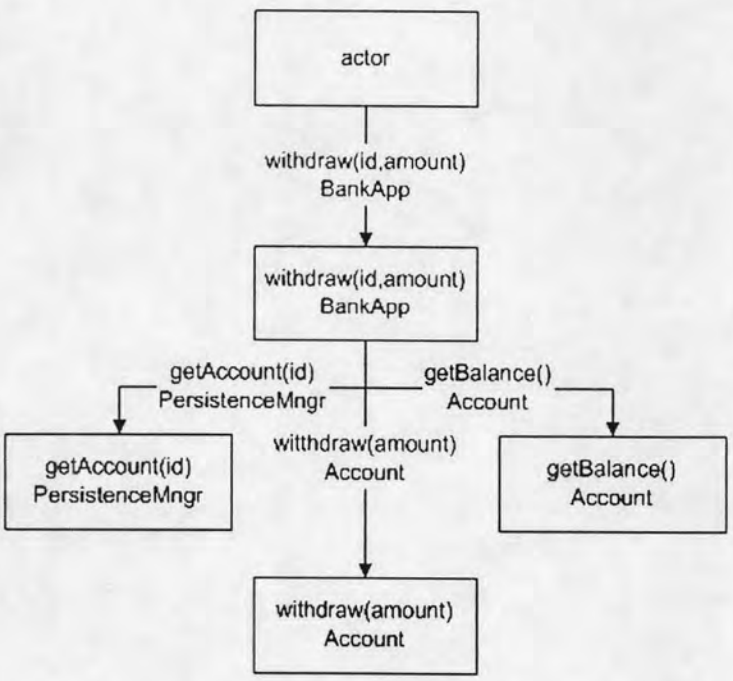

Figure 6.9 Expected message sending sequence from the example

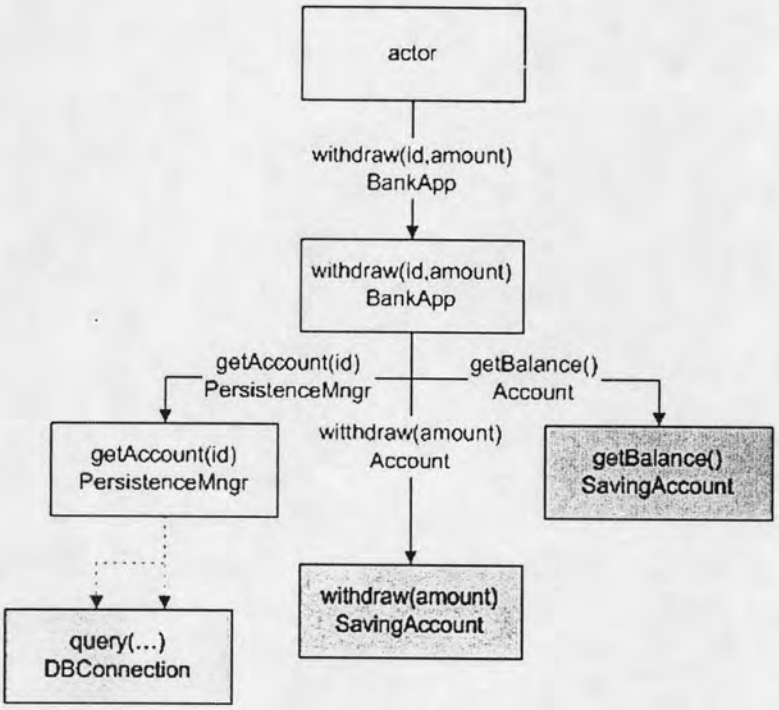

Figure 6.10 Actual message sending sequence from the example

The interaction in figure 6.7 is represented in the message sending sequence model as shown in figure 6.9. The ExecutionContext of the actor is the starting point of the interaction. The actor sends a message, represented by an arrow, to an instance of BankApp class, whose "withdraw" operation is executed as a result. The execution of the operation is represented by an ExecutionContext associated to the arrow head of the message. In turn, the operation issues three messages to other three ExecutionContexts, according to the interaction in figure 6.7.

An example of actual message sending sequences shown in figure 6.10 is from the situation where an instance of SavingAccount class is returned from "getAccount" operation on PersistenceMngr class. The part that is different from the expected message sending sequence is highlighted. The ExecutionContexts of the execution of account associates to SavingAccount class, instead of Account class. Dot arrows on the bottom-left corner of the figure are additional messages, which are discussed later.

Verification of this example follows steps as explained in the previous section. The actor from each sequence sends an exactly identical message to an instance of BankApp class. The verification proceeds to the ExecutionContexts of the instances; still are the elements from both sequences identical. Three identical messages sent from the ExecutionContexts fork another three ExecutionContexts. As shown in the figure 6.10, two of them have the classifier as SavingAccount, while the classifier of the corresponding ExecutionContexts in the expected message sending sequence is Account. Applying the verification rule regarding polymorphism, these ExecutionContexts are considered as valid, since SavingAccount class is a subclass of Account class. Note that the operations of the matched ExecutionContexts are identical, which satisfies the requirement of the verification procedure.

As stated earlier, there are additional messages sent from an instance of PersistenceMngr class in the implementation. These messages are, for example, calls to data access classes (e.g. file access, database access etc.) to perform account query. These messages do not exist in the expected message sending sequence. Applying the concept of a list of common classes described earlier, these messages can be identified as refinement, if the data access classes are included in the list of common classes.

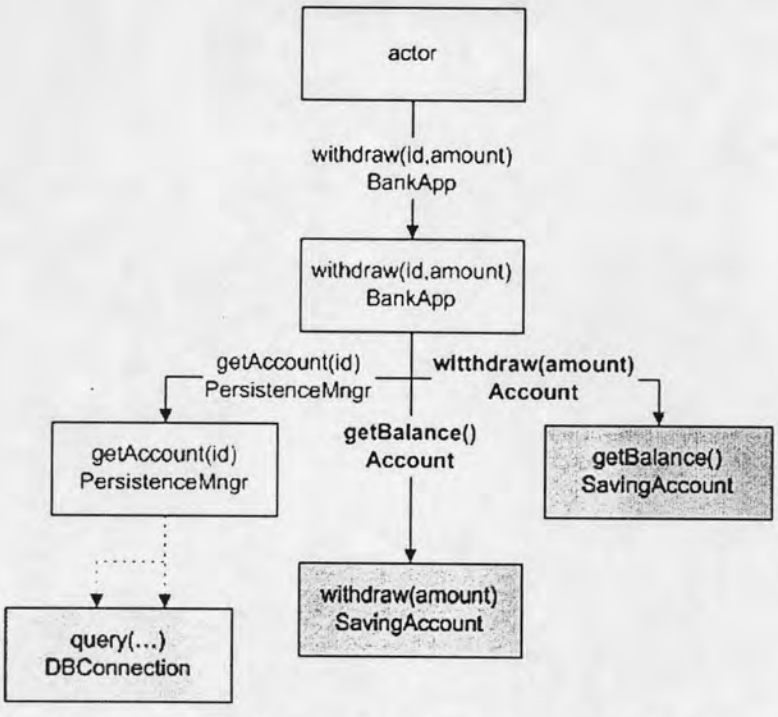

Figure 6.11 Example of invalid actual message sending sequence

Figure 6.11 shows an example of invalid actual message sending sequence. Notice that the messages with bold texts are not in the correct order according to the sequence diagram in figure 6.7 and the expected message sending sequence in figure 6.9. Other examples of invalid actual message sending sequence include mismatched method name/arguments and mismatched instance role.

The verification procedure we discuss in the previous sections is capable of recognizing and verifying polymorphic interactions, as shown in the case study. However, there exists a condition where polymorphism verification is not required as stated earlier. So far, the verification procedure is discussed regardless of test case generation. In order to achieve thorough testing, a test case generation approach possibly requires a specific test scenario for each subclass substitution according to one of the adequacy criteria discussed in chapter 4. One way to achieve this requirement is to generate a specific expected message sending sequence for each test scenario; consequently, the verification of the message sending sequences ignores the polymorphism issue and demands an exact match of both the classifiers of

Instances and the classifiers of Operations of Execution. This alteration is simple and can be considered as an optional mode of the verification procedure.