An Architectural View for Test

Ivan Murray
Motorola Ireland Ltd.
ivan.murray@motorola.com

David Russell
Motorola Ireland Ltd.
drusse01@email.mot.com

ABSTRACT

This paper presents a description of the benefits to the Test organisation to be gained in developing a 'test' view of a software system. The paper describes how Use Cases within UML are inherently structured in favour of testers. It will compare textual with diagrammatic representation of a Use Case and will detail why the diagrammatic form is favourable to the tester.

The viewpoint can be described as – applying a specific 'rule set' to the UML Activity Diagram to present the data within the Use Case. This viewpoint can be utilised to effectively address such test concerns as traceability, testability, and requirement coverage verification.

The paper will then describe how with further additions to these 'rules', the authors were able to automate test case identification and test code generation and the benefits to be gained from this.

1.0 INTRODUCTION

The concepts discussed in this paper first came about when the software development group, within which we form the test team, began utilising the Unified Modelling Language (UML). "The UML is a graphical language for specifying, describing, constructing and documenting the artefacts of a software system." [1] UML specifies a particular way of documenting the requirements of a system through 'Use Cases'. "A Use Case is a description of a set of sequence of actions that a system performs that yields an observable result of value to a particular user." [1]

As testers, we had never been exposed to this form of requirements before and looked to The UML User Guide [1] and The Rational Unified Process (RUP) [2] for instruction on how to identify valid tests from these Use Cases with the purpose of verifying the systems requirements. Although the need for a test model was pointed-out, the process for identifying and extracting test cases from the requirements was not documented in either. We looked further afield but did not find direction on how best to tackle requirements in Use Case form.

We concluded that we would need to devise our own solution to this problem. The approach we took was to try and define our own test view of the requirements with the goal of being able to methodically extract test cases from that view.

1.1 Use Cases and Testers

We found that requirements documented in the form of 'behaviors' and 'uses' are inherently suitable to testers. One of the primary concerns of a tester is to attempt to force the system into all of its possible states and ensure that the system acts as documented in the requirements in these states. Forcing the system into all of these states is not trivial. If the requirements are documented as 'behaviors' or 'uses', then every 'statement' within the requirements will have been arrived at by following a behavior and thus all states into which the system needs to be forced can be met by following those behaviors. Testers no longer need to struggle with designing ways of forcing the system into particular states to satisfy requirement verification, the author of the requirements has already done this work.

There is nothing new in this, nor in the conclusion that the Unified Modeling Language provides the structures and guidelines to describe the 'behaviors' and "uses" of the system through its Use Cases. As testers within the telecommunications industry, who have worked for years with "shall" statement requirements, the change to Use Cases in the last couple of years have proved hugely beneficial. It allows us to see very quickly, what the nature of our testing will be, and provides a structure with which to build a test set from.

1.2 Scenarios and Testers

The Use Case describes a "Use" of the system in terms of a set of interactions with the system and its actors. This

set of interactions is called the ‘Flow of Events’. The common means for organizing the flow of events is by

- Main (Sunny Day) Flow - The flow of events that is most likely to occur.
- Alternate Flows - A set of events, outside the “Main Flow”, that can also occur during the specific “Use” of the System (e.g. an error condition)

By documenting one flow of events of a Use Case, from a Use Case Start point to a Use Case End point, a complete behavior of the system has been identified. This is what UML defines a ‘scenario’, “a specific sequence of actions that defines a behavior ... basically one instance of a Use Case” [1]. Simplistically, identifying system test cases can be reduced to the task of identifying scenarios from the Use Case.

“The complete collection of Use Cases describes all possible ways in which the System can be used”[3]. It follows that if every scenario can be identified and translated into a test case, then complete verification of the system requirements can be achieved by successfully exercising these test cases. Of course, there will be further testing necessary to capture what is not documented in the requirements, but the ability to ascertain that complete requirement verification has been achieved, for any set of requirements, in a consistent, methodical manner, is a very desirable goal.

2.0 TEXT V'S DIAGRAM

Typically, the Use Case description is provided using natural language in a narrative style. This textual representation is not standardised as part of UML, and is thus open to interpretation. We will argue why an Activity Diagram Representation of 'Flow of Events' can be a preferable representation. An Activity Diagram is defined in UML as "A diagram that shows flow from activity to activity; activity diagrams address the dynamic view of a system. A special case of a state diagram in which all or most of the states are activity states and in which all or most of the transistions are triggered by completion of activities in the source states." [1]

2.1 Problems with Textual Representation

There are a number of problems with the textual approach to representing a Use Case:

- Text representation is ambiguous with regard to the flow of control. Positioning and activation of alternates is not clear.
- When tracing or visualising flow of control, jumping between alternate flows and the main flow can be disorientating & confusing
- Completeness of alternate flows is not readily apparent.
- It is difficult to present concurrent flow of control with a textual description.
- Typical telephony use cases are long and complex, and this complexity is not readily apparent from viewing a textual description.

The non-standard format of a Use Case gives rise to inconsistent specifications being presented to the consumers.

2.2 Why a Diagram ?

The UML standard allows other representations than text for the flow of events in a use case. In particular, a use case may have an associated UML Activity Diagram, which completely defines the scenarios of that Use Case. Figure 1 provides a good example of a simple Use Case with 3 alternate flow's identified by 3 decision activities.

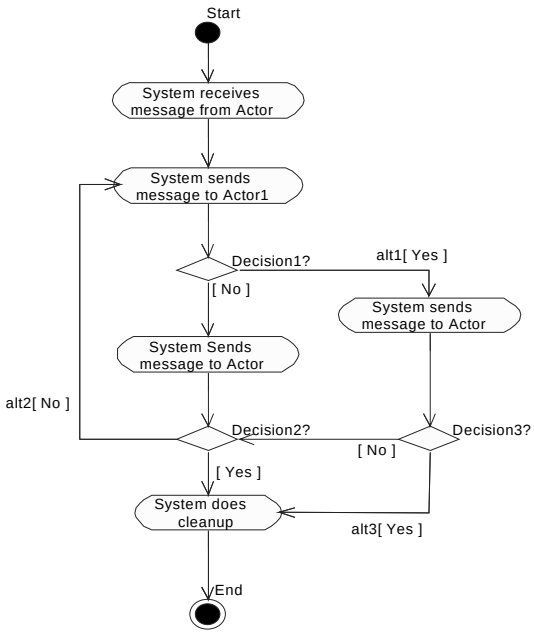


Figure 1: A Sample Activity Diagram

- The representation of Flow of Control is not ambiguous.
- Decision points are clearly defined in the Activity Diagram, an important aspect for the test consumer.
- Modelling the Use Case as a UML Activity Diagram makes visualising the flow of events through the Use Case easier.
- Scenario's can be readily identified. One particular path through the Activity Diagram (from begin to end) represents one possible flow of events that could occur if this Use Case is exercised.
- Complexity can be immediately judged.
- Concurrency can be represented without difficulty

This can be extremely useful; it allows the tester to see clearly the complexity of the Use Case and to validate the flow of events through it. Identifying valid tests to cover the requirements documented in the Use Case is then a matter of picking out scenarios through the activity diagram.

Figure 2 provides an example scenario through the activity diagram, and Figure 3 shows the particular path extracted to its own Diagram.

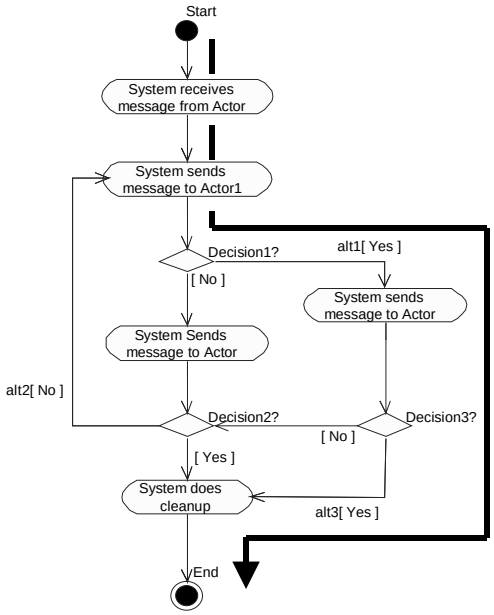


Figure 2. One Particular Path through the Activity Diagram

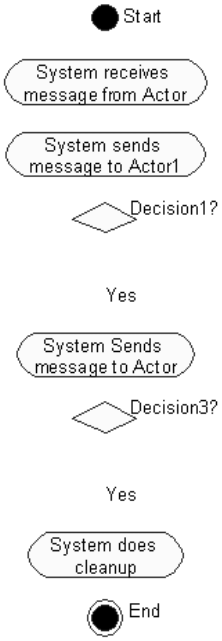


Figure 3. A 'subset' activity diagram representing the path through the Activity Diagram indicated in figure 2. Flow of control is inherent thus the Transition arrows are not drawn on the extracted diagram.

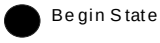
3.0 A VIEW NEEDS RULES

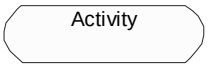
In the previous section, the merits of using a UML Activity Diagram, as a means of documenting the Use Case was discussed. It would be the preferred presentation of the test view of the system. The UML standard is again open to interpretation with its rules that govern the generation of the Activity Diagram. These problems are resolved by using a more precise, unambiguous notation for the flow of events. For the Test view a more stringent set of rules is required.

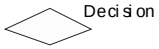
A **bold** font will identify the comment on the rules as distinct from the rules themselves.

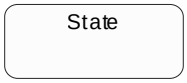
3.1 Rules

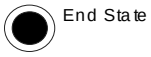
In the development of the UC model, we only use 5 different activity model entities. These are...

1. Start State - 

2. Activity - 

3. Decision - 

4. State - 

5. End State - 

The goal is to produce a model that will exactly reflect the flow of control through the UC.

1) There can only be one "Start State" in the model. This start state must have one outgoing transition and no incoming transitions. **By definition a Use Case should be describing how the system reacts to one specific action from an actor.**

2) There can be any number of "End State" entities in the model. The "End State" will have no outward transitions. **The goal of the exercise is for easier identification of paths from Begin state to End state of the Use Case.**

3) The "State" entity is used especially to specify "Unanchored Alternates". These are alternate flows of the Use Case that can happen at any number of locations throughout the Use Case. The "Stereotype" of the state should be set to "unanchored" & the name of the State should be set to the name of the "unanchored alternate" that it is representing. There are 3 types of "unanchored alternates" that can be specified.

- One that can do a number of steps & completes at an "End State".

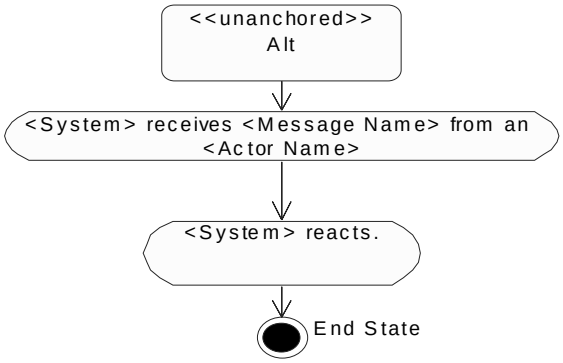


Figure 5. Unanchored Alternate that Terminates with End State

- One that can do a number of steps, and return to the same point that it occurred.

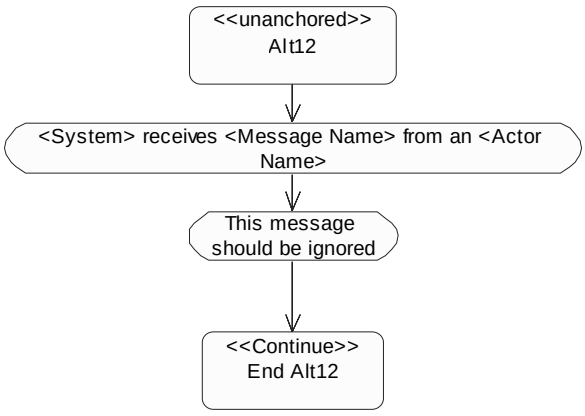


Figure 6. Unanchored Alternate that Returns to the Same State it was Invoked.

This type "unanchored alternate" needs to complete with a

State of Stereotype "Continue", as show in the diagram.

- One that does a number of steps, and returns to a specific location in the flow.

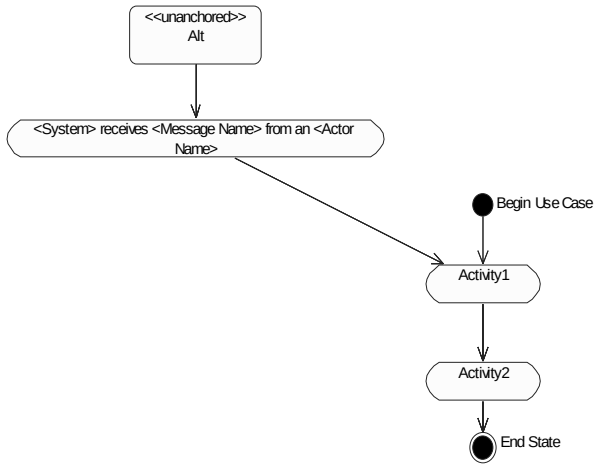


Figure 7. Unanchored Alternate that Returns to a Specific Event in the Flow.

4) There can be any number of Activity entities in the model. Each activity can have any number of incoming transitions, but must only have one outgoing transition.

5) If an Activity is specifying a message being sent/receive to/from System the Activity should be describer as follows....

<System> receives <Message Name> from an <Actor Name>

<System> send <Message Name> to an <Actor Name>

This facilitates identifying message flows between system and Actors.

6) If an activity specifies another Use Case, then the Activity is given a stereotype "includes". An Example of this would be as follows.

<<includes>>
Action handled by another Use Case.

This test view will highlight Use Case interactions immediately to the tester.

7) Decisions must be labelled as a question. Decisions can have any number of incoming transitions, but can only have exactly 2 outgoing transitions. **In the test view the decision points are the most important activity. They drive the flow of events. The tester needs to be given the information in clear unambiguous form. A two-answer question is the easiest form of decision point**.

8) Decision outward transitions must be labelled with a guard condition that is the answer to the Decision question. Example...

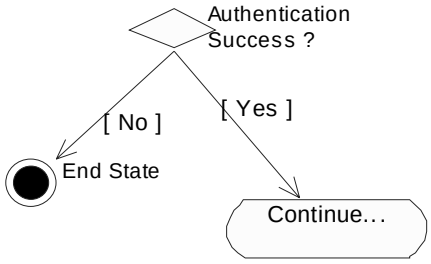


Figure 8. The Decision Activity

9) Of the 2 decision outward transitions, one must be given the name of the Alternate flow it is signifying. If there are a number of decision points signifying the same alternate flow, delineation with numbers can be used.

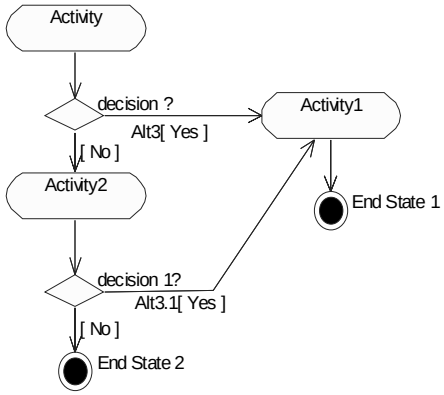


Figure 9. Alternate Flow Labelling.

Each scenario within the use case needs to be delineated with a naming convention. The Test View uses the names of the alternates through which the scenario traverses. UseCaseName_N_altx_alty_altz

- **N = unique integer between 0 and the maximum number of paths**

- ***x, y* and *z* = the numbers assigned to alternate flows in the use case.**

This naming convention inherently provides a good degree of traceability between Test Scenario and Use Case.

4.0 TAKING THE TEST VIEW A STAGE FURTHER

4.1 Automatic Scenario Extraction

In formalising the requirements to suit the needs of the testers (i.e. clearly defined behaviours, no ambiguous wording, consistent presentation, structured flow of events) we have armed testers with the ability to methodically identify test scenarios. A stringent rule set is the basis for a case tool algorithm, and we saw that with a few additions to the rules, we could automatically generate all possible paths through the activity diagram.

The output of the case tool can be defined as a set of 'one-dimensional' Activity Diagrams (example in Figure 3) representing all scenarios that could possible happen for that 'Use' of the system, and is the absolute set from which test cases could be chosen.

4.2 Automatic Test Script Generation

From these 'one-dimensional' Activity Diagrams it was possible to automatically create corresponding sequence diagrams, the message source and destination being determined from the actors identified in every 'message' activity. The sequence diagram in Figure 10 was generated from the Activity Diagram in Figure 3.

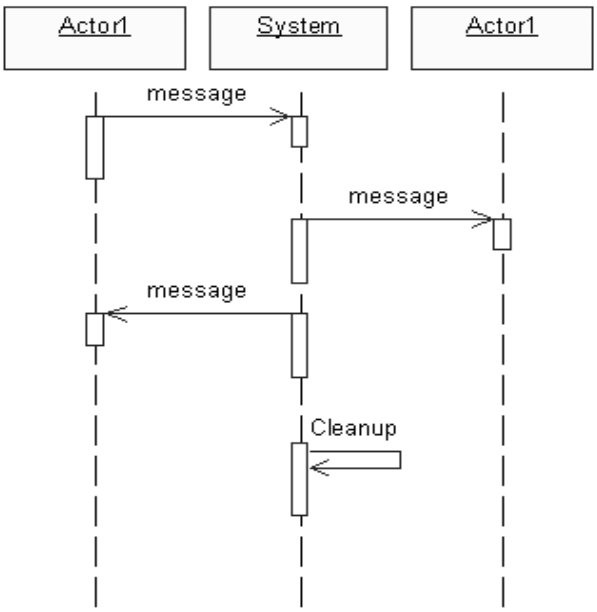


Figure 10. A Sequence Diagrm of a Scenario.

The final piece in the jigsaw was to produce test code to access the proprietary test environment utilised in our labs. This was achieved by translating every message, sent or received, within a sequence diagram, into an 'API' call on the test harness.

5.0 CONCLUSION

What we have presented in this paper is a means to producing a view of a system, which is specifically aimed at improving the testability of that system and thus the quality of the finished product. Although the extra layer of definition this view demands is specific to the testers needs, it is not contradictory or unhelpful to other system stakeholders. There is no reason why, for example, the design team couldn't work directly from this view. Although some extra effort is required up-front in defining this view, the reward is achieved at each stage of test development further down the line.

The benefits of this test view, as we found, are…

- enforces unambiguous requirements
  - We found within the test organisation, that the Use Cases were not returning as many validation errors as with the textual representation.

- a very quick estimate of the complexity of the test effort for requirements coverage is possible
  - After seeing a number of diagrams, we were finding that judging the complexity was intuitive. Examining the number of branches, decision points, and backward direction transitions, we could get a feel for the size and complexity of the test effort.

- with a methodical approach to identifying the test cases, inexperienced testers can work as effectively as experienced testers in identifying tests for requirements coverage
  - This was applied first on a new project, with 4 new test engineers. All could see the benefit, and all were applied easily to the task of requirements analysis & test extraction.

- Use Case requirements coverage is guaranteed
  - With the nature of a Use Case and it's scenarios, we can guarantee that if all scenarios of the Use Case are run as test cases, then we have 100% Use Case Requirements Coverage. As part of a Use Case model, there is often a Supplementary Specification that captures requirements that are not maintained in Use Cases i.e. performance. We do not say that these requirements are verifiable within this view.

The further benefits of automating the test identification and test code generation tasks are

- cycle time reduction
  - We saw a 4x-cycle time reduction with test case selection and documentation with this method.

- test cases scripted in a uniform manner
  - Because test scripts are auto generated from the Sequence Diagrams (as in Figure 10), they have a uniform pattern of Sends and Expects of particular messages.

In closing, this paper has centered around one particular test view, based on the UML. However, the over-riding message of the paper is that, with any system, investing in a test view of the architecture, based on the specific needs of the test organizations procedures and tools, a very powerful means of methodically generating test cases and improving requirements coverage is possible.

REFERENCES

[1] Grady Booch, James Rumbaugh, Ivar Jacobson, *The Unified Modelling Language User Guide,* Addison –Wesley, Oct 99

[2] Phillippe Kruchten, *The Rational Unified Process An Introduction Second Edition,* Addison –Wesley, March 2000

[3] Don Leffingwell, Don Widrig, *Managing Software Requirements, A Unified Approach,* Addison –Wesley, Oct 99.