

# Using State Diagrams to Generate Unit Tests for Object-Oriented Systems

Florentin Ipate<sup>1</sup> and Mike Holcombe<sup>2</sup>

<sup>1</sup> IFSOft, Romania  
fipate@ifsoft.ro  
www.ifsoft.ro

<sup>2</sup> Department of Computer Science, University of Sheffield, UK  
m.holcombe@dcs.shef.ac.uk

**Abstract.** Traditionally, finite state machines and their extensions, such as stream X-machines, have been used for modelling and testing of graphical user interfaces (GUI) and for acceptance testing. This paper shows how these testing techniques can be successfully extended to unit test generation for object-oriented systems and integrated into Extreme Programming in a simple and designer-friendly way. The approach has been used by MSc students in Computer Science at the Pitesti University to write JUnit tests for XP projects and the effectiveness of these tests has been compared with that of tests produced using ad-hoc and traditional functional methods. The conclusions show that over 90 % of the faults found by other methods have also been found by the stream X-machine based approach, whereas less than 75 % (in many instances less than half) of the faults uncovered by the stream X-machine based testing have been found by other methods. As the finite state machine based test generation has been automated, the time spent using the two testing strategies was roughly equal.

**Keywords:** unit testing, functional testing, state diagrams, finite state machines, stream X-machines

## 1 A Simple Example

Suppose that we are building a simple computer system for a library. We might identify a number of stories such as the following: Borrow book (a book can be borrowed if the customer does not have the maximum permitted number of books on loan), Return book, Reserve book (a book that is currently on loan can be reserved) Extend loan (the loan can be extended if the borrowed book has not been reserved by another customer).

From the above user stories we can identify two obvious class candidates, *Book* and *Customer*, and their operations; for *Book*, these are *borrow*, *return*, *extend*, *reserve*. Three states of a book can also be identified: *Available*, *Borrowed*, *Reserved*. Having identified the operations and the states, we can now proceed to drawing a state diagram for the *Book* class. As it turns out, the state diagram (Fig. 1) has actually 4 states, since, once a book has been reserved,

it has to be known whether the book is still on loan by the current customer (*Borrowed&Reserved*) or has been returned (*Reserved*) and the new customer can proceed with the borrowing. Furthermore, there has to be a way of progressing from the *Reserved* state, so we can decide that the reservation has to be cancelled before the new customer can borrow the book. If the book is no longer of interest for the customer who has made the reservation, s/he can just *cancel* the reservation, either from the *Reserved* state or from the *Borrowed&Reserved* state. The state diagram in Fig. 1 describes the possible sequences of operations that the class can perform in correct use, that must be obeyed by any class users or clients. On the other hand, the class cannot control these users, so it is never known when an operation will be called. Thus, in order to insure the correctness of the system, programmers use suitable error handling to deal with incorrect or unexpected use. This situation can be modelled by adding to the diagram erroneous transitions to an *Error* state.

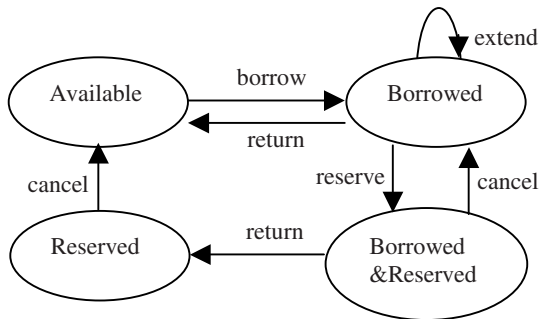


Fig. 1. The state diagram for *Book*

We can now proceed in a similar way with the *Customer* class and identify 3 potential states: *Empty* – when the customer has no books on loan, *Full* – when the customer has the maximum permitted number of books on loan, and *Partial* – when neither of these situations apply. There is, however, an important difference in comparison with the diagram for *Book*, since, in this case, the transitions from the *Partial* state will be *conditioned* by predicates.

As classes form associations, it is not sufficient to test them independently and some testing has to be done for some small groups of classes with high coupling between them. This kind of testing cannot be left until system integration and has to be performed during the unit testing phase. When an object in a class *A* can send messages to an object in a class *B*, we can use a state diagram to show the effect of the operations of *A* on the object in *B*, i.e. apply these operations in the states of the class *B* diagram. Obviously, only those operations of *A* that can affect the state of *B* (a message from *A* to *B* is sent as a result of the operation) need to be considered, the others may be omitted. When the association between the two classes is bi-directional, two diagrams (one for each

direction) will normally be needed. We can safely assume that a *Customer* object will send messages to a *Book* object, but not the other way around, so the association between these two classes can be modelled by a single state diagram.

## 2 Finite State Machine and X-Machine Based Testing

Once we have produced the states diagrams, we can use them to generate test cases in a rigorous manner and to automate the testing process, by applying existing finite state machine based strategies. One of the most general approaches is the *W*-method [2], that generates sequences of symbols to reach every state in the diagram, check all the transitions from that state and identify all destination states to ensure that their counterparts in the implementation are correct.

It is straightforward to apply the *W*-method to a finite state machine, but the state diagrams that describe the behaviour of a class or a class association are not, strictly speaking, finite state machines, as transition labels are not mere symbols from an abstract alphabet, but have some functionality attached to them – they can be represented by *mathematical functions*. This more general model, in which the labels of the transitions are mathematical functions is called a *stream X-machine* [1]. Each such function is driven by some input (operation name, input parameters), performs some processing on the object data and may produce some output (output parameters, display messages). Thus, the following information is associated with each transition label:

- **Input:** the *name* of the operation performed, the *input parameters* (if any) and their domains.
- **Data domain:** the domain of the data values for which the operation is valid.
- **New data:** the updated data values.
- **Output:** the *output parameters* (if any) of the operation and their values and any other *observable outputs*. If the state diagram shows the effect of the operations of a class *A* in the states of a class *B*, the object in *A* will also be assimilated as input parameter for each operation.

For each diagram, the four components for each label will be identified and the results will be placed into a table. Each user story will correspond to one or more rows in this table. The table can then be used to identify the sequence of inputs (operation names and input parameters) that drives a sequence of transitions so that each sequence of transition labels generated by the *W*-method can be translated into a sequence of program statements and an appropriate test program can be written. Furthermore, the expected outputs can also be derived from the table and these can be compared with the outputs produced by the test program. However, this testing method is only effective if the diagram (stream X-machine) satisfies some *design for test* conditions: observability and controllability [1]. In unit testing, these requirements can be achieved by splitting a transition into two or more transitions (observability) and by designing special operations to set up the appropriate context for the conditioned transitions (controllability).

### 3 Conclusions

State diagrams are intuitive and easy to use. They require little formal training but, at the same time, are rigorous means of describing the behaviour of a class or a system, since they are based on mathematical models such as finite state machines and their extensions. State diagrams can help to clarify and refine design details (in our example, a new method, *cancel*, that did not come out directly from the user stories, was identified when the *Book* class diagram was drawn up). Finite state machines and stream X-machines provide the basis for rigorous testing, without any other kind of (semi-) formal specification being necessary, which is an important advantage in the context of Extreme Programming.

Testing must be automated as much as possible in Extreme Programming and functional tests themselves are written as computer programs [3]. It is straightforward to convert a finite state machine into a computer program and this process can be easily automated. Furthermore, the process of generating test sequences from a finite state machine can also be automated and appropriate tools exist. Obviously, the tester will have to look up in the table which stores the transition label details and produce appropriate sequences of code statements to drive the sequences of labels that come out of the finite state machine test generation tool, but, on the other hand, functional tests cannot be fully automated unless a complete (formal) specification and tools for writing and executing it are available, which is not the case in Extreme Programming, nor in most development approaches in the core software industry.

As the method thoroughly tests a class or a system, it usually produces a larger number of test cases in comparison with traditional functional methods, such as such as category-partition. However, the test cases produced are easier to run in comparison with these methods, since there is no need to explicitly establish the context (the state) before actually running the tests (the test sequences reach the state and identify it before checking all the transitions that come out from it).

The key benefit of the stream X-machine based testing method [1] is that sets generated *fully tests* the class, as all possible transitions, including the error handling part of the operations, will be checked in every possible context (state of the diagram). Furthermore, the method does not only test the operations individually, it also tests their coordination within the class.

### References

1. Holcombe, M. and Ipate, F. 1998. *Correct Systems: Building a Business Process Solution*. Springer Verlag: Berlin.
2. Ipate, F. and Holcombe M. 1997. An Integration Testing Method That is Proved to Find all Faults. *Intern. J. Computer Math.* **63**: 159-178.
3. Jeffries, R., Anderson, A., Hendrickson, C. 2000. *Extreme programming installed*, Addison-Wesley.