INFO TO GO

- Using actions to express models allows testers to design the tests and programmers to automate them.
- A tabular format makes the correspondence between test and model easier to grasp.
- Tests that are expressed in actions are easier for non-technical people to understand.

It can be complicated to automate model-based testing. Here's how to employ action words to get the job done.

BY HANS BUWALDA

Action Figures

YOU CAN MAKE A MODEL FOR JUST ABOUT ANYTHING: systems, business processes, organizations, the universe, and so on. A model is an abstraction; it usually serves to represent only the aspects of interest, while leaving out everything else. A model can be a picture, a set of mathematical expressions, a simulation program in a language like Modula, or even a toy car.

Models can be an invaluable testing tool. Unfortunately, they are often difficult to automate—requiring specialized model-based software and advanced programming skills. One solution is to have testers use action words, or action-based testing, to express models, enabling them to design action-based models that can then be easily automated by a programming expert. When a tester designs the test and a programmer creates the automation, each person is performing the role he is most comfortable with, resulting in well-designed, automated tests.

Model-Based Testing

Models used for testing usually describe either the structure (the elements and their relationships) or the behavior (how the system will respond to outside events) of a system. Testers will mostly focus on models for behavior: the desired behavior from the system under test and/or the possible behavior of the environment towards that system. In fact, one could argue that a set of test cases in itself is a model of the environment behavior combined with the expected behavior from the system under test.

I personally will use elaborate models for testing only if there are specific benefits; the fact that models are interesting from a scientific point of view is not necessarily one of them. For many users, models are quite abstract and can be hard to conceive of and understand. Good old-fashioned test cases or scenarios might just as well do the trick. The time saved by letting a model generate tests should balance with the time needed to set up the model.

We sometimes use models when tests are too lengthy or complex to set up "by hand," such as communication protocols or performance tests. Models allow us to describe the system in a generic, compact format and automatically generate all the detailed actions and verifications. This not only saves typing, but also clarifies what is actually being tested. Models can also help when implementing a standard, well-proven testing technique, such as decision tables. By automating the model to generate the test cases, one can be more confident that the test technique is being applied correctly (assuming that the models are implemented correctly, of course).

Models can also be helpful when testing systems with non-deterministic behavior (behavior that can't be fully predicted by the test). For instance, on a recent project, I used a model approach to automatically test gaming software. Because of the game's non-deterministic behavior, the model turned out to be a better solution than the more regular test automation I had done before. The model approach made it possible to express the test more clearly and efficiently than was possible otherwise.

Action Based Testing

Action Based Testing (ABT) is an extension of the action words approach. I first used action words in a project in 1994 to make testing and test automation easier to do. What I didn't fully realize at that time was that the approach would turn out to be fairly flexible and generic, even making it suitable for something as specialized as model-based testing.

When using action words, tests are divided into **test clusters**. Each cluster contains tests with a similar scope. Some clusters will contain tests of the user interface of a system (do the buttons work?), while others might test premium calculations or contain high-level business cases.

The main idea is that the testers develop the test clusters—without any need for automation skills.

The test clusters are physically developed in a spreadsheet. Each row of the spreadsheet can start with an action in the first cell, while the other cells of that row can contain any number of arguments. For example, a row might have a login action in the first cell. The two arguments for login, the user's name and password, are in the next two cells. Many of the actions are reused both within the same test cluster and in any number of the other test clusters. Actions form an ad hoc test language. This language can be tailored relatively easily towards any model we need.

An automation engineer (a separate person, with a different set of skills than the tester) uses these test clusters to form an automation scheme. This scheme focuses on the automation of individual actions. For instance, an engineer would automate the login action by writing code to make a GUI tool press the right buttons and type the user's name and password to the right text fields. Instead of automating an entire test case, only the individual actions are automated, giving the approach some of its more important advantages. Maintenance can be performed quickly and easily, and tests, expressed in actions, are quite readable even for non-technical people.

To illustrate how ABT can be used to express models, we'll examine two simple models: a decision table and a state machine.

ACCIDENTS	GENDER	DECISION
0 or 1	Irrelevant	Accept
2	Female	Accept
2	Male	Refuse
3 or more	Irrelevant	Refuse

Figure 1: A decision table is useful in describing rules.

Decision Tables and ABT

Decision tables allow one to quickly describe combinations of events and expected responses from a system under test. A variation or extension of the decision table technique is called "action tables," proposed by David Gelperin in his workshop at the STAR testing conference in 1998. In action tables, the cells can contain more details than just an X marker, like the original technique, thus greatly enhancing their power to express situations.

For example, suppose we're testing a program that decides whether to accept applications for car insurance. Two criteria are relevant for the acceptance. First, there is the number of accidents in the past five years. If that amount is zero or one, the applicant is accepted; if it is three or more, the applicant is refused. If the number of accidents is two, the decision is based on gender. If the applicant is female, she is still accepted, since this insurance company happens to think that female drivers are less risky than male ones. These rules can be described in a decision table (see Figure 1).

The test of this application should verify that it makes the right decision in each of the cases (rows). In those cases where the applicant *is* accepted, the test should also check to see if the premium was calculated correctly.

In model-based testing, the tests themselves are created from a decision table. Figure 2 shows how you could create the tests using ABT instead.

In Figure 2 (on page 46), the blue text in the spreadsheet is the **decision table**, expressed in ABT format. The "table" action states that the next set of lines should be kept in memory as a table, which will be used later by the action "run action table." The numbers in the first column are the starting values of the ranges given in the decision table in Figure 1. The end of the range is the number in the first column of the next row. So the first entry means 0 or 1, and the next two entries describe two cases with 2 accidents. The last row, with the value 3, means all values from 3 and up. It is up to the automation to try all kinds of values, like 0, 1, 2, 3, 10, and so on.

The green text following the decision table in Figure 2 is a generic ABT test **scenario** that can be executed repeatedly for each row of the table. Look first at the third line, action "enter car value." That action will cause the test to tell the program that Chris's car is worth $20,000. The automation engineer will program "enter car value" to place the contents of the second and third cells into the appropriate fields on the insurance application's screen.

The second line, for action "enter customer" looks slightly different. The cells contain "#accidents" and "#gender." The pound sign signals that these are placeholders. Each time the scenario is used, they are given values from the correspondingly named columns in the decision table. So, the first time the scenario is run, Chris's number of accidents

Table	**decision table**		
accidents	gender	situation	
0	—	accept	
2	f	accept	
2	m	refuse	
3	—	refuse	
end			
Scenario	**test insurance**		
enter customer	Chris	# accidents	# gender
enter car value	Chris	20000	
accept	1000		
refuse			
end			
situation	**accept**	value	
check premium	# value		
end			
situation	**refuse**		
check message	Not eligible for insurance		
end			
run action table	decision table	test insurance	

Figure 2: Use Action Based Testing to create automated tests from decision tables.

is 0 or 1 and gender is "—" (because it is irrelevant to the test).

Another difference from ordinary ABT tests is that the actions "accept" and "refuse" are executed only when they apply. So for the first two decision table rows "accept 1000" is performed, meaning that the test will check if the premium was indeed calculated and the value was indeed 1000. The "refuse" line is skipped for those two rows. Then, for the next two table rows, the "accept" line is skipped and the automation will check that the application was indeed refused.

The actions "accept" and "refuse" are small ABT test cases or **situations** (see the red text in Figure 2). Whenever "accept" is evaluated, it gets a parameter "value" denoted by the pound sign and stating an expected value for the premium. The result of the program's premium calculation will be checked against that expected value. In the "refuse" situation, we will just go ahead and check whether there is a message on the screen stating that the applicant is not eligible.

At the bottom of the spreadsheet in Figure 2, in black, is the "run action table." This kicks off the actual evaluation of the table. It will loop through the rows of the table and run the generic scenario for each with the proper values for gender and damages set, executing either "accept" or "refuse" depending on the third column of the table. Its implementation is relatively straightforward, but goes beyond the scope of this article, since much of it is dependent on the technical specifics of our ABT toolset.

State Machines and ABT

Models can also be used for tests of systems with behaviors that can't be predicted easily by the test. For instance, suppose you are testing a quiz game, where the test can't know in advance which question the computer will ask. You need to verify whether wrong and right answers are reacted upon correctly. The quiz has only two questions. If they are answered correctly, the contestant wins the grand prize—a subscription to *STQE* magazine. (This example has been kept as simple as possible.)

When testing this quiz, you'd be interested in checking cases like these:

- If two questions in a row are answered correctly, is the prize awarded?
- If a question isn't answered, is a hint given?
- If a person fails twice at a question, is the quiz over?

How would you produce an automated test for this application, given that you don't know what the questions will be in advance?

The quiz program can be modeled with what is technically called a **state machine** model, in which events cause transitions between states. Since there were non-technical people involved in my project, we chose terminology that we believed would be more accessible: situations and moves. A sample situation is "the first question has been answered incorrectly and a hint has been given." Two moves out of that situation are "answer the question correctly" and "answer the question incorrectly."

The actual questions and their answers are in a table, which we will assume is present where we can reach it from the automation. The table is assumed to have the following columns:

1. the text of the question
2. the answer
3. a hint towards the answer

For simplicity, we assume that the answers are in a format that the system can recognize easily (e.g., "the name of an animal" or "a city in the US"). Figure 3 shows the state machine model that can be created to automatically test this quiz game.

In Figure 3, every situation starts with the action "situation":

start—start of the game, the first question is on the screen (in black text)

hint 1—first hint for the first question (in blue text)

round 2—the second question is on the screen (in red text)

winner—we are congratulated (in green text)

loser—the game stops, we are invited to play the game again (in orange text)

The first actions in situation "start" and "check prompt" simply verify that the text on the screen is "Welcome to our quiz. Here is your first question." The "question" action tells us that there should be a question displayed. You can read that action as follows: "Check that the text displayed matches one of the questions in column 1 of the 'questions' table, and store the matching row for

situation	start			
check prompt	Welcome to our quiz. Here is your first question.			
question	q1	questions	1	
	table	record	from	to
move	questions	#q1[2]	start	round 2
move	questions	wrong	start	hint 1
situation	hint 1			
check prompt	That wasn't right. Here is a hint...			
check table response	questions	q1	3	
	table	record	from	to
move	questions	#q1[2]	hint 1	round 2
move	questions	wrong	hint 1	loser
situation	round 2			
check prompt	Very good, now for the big prize...			
question	q2	questions	1	
	table	record	from	to
move	questions	#q2[2]	round 2	winner
move	questions	wrong	round 2	hint 2
situation	winner			
check prompt	YES!! Congratulations, you have won a subscription to *STQE* magazine!!			
situation	loser			
check prompt	Sorry, wrong again. Thanks for playing, feel free to try again.			
play game	**start**	**questions**		

Figure 3: Using a state machine model to produce an automated test.

later use in 'q1.'"

The moves, which bring us from situation to situation, are given as well. The first possible move is to enter a correct answer. The correct answer is found in cell 2 of the stored row "q1." If the test gives that answer, it expects the quiz to continue with round 2. The other possible move is to enter a wrong answer, the text "wrong" (which is known not to be the answer to any question). If the test gives that answer, it expects the quiz to continue with hint 1.

The test determines that the game has made the right move by checking the prompt. What if the game makes the wrong move in response to an answer? When an expected situation is not encountered, we assume for simplicity's sake that we record a failure, reset the game (in the automation), and start another round. That way, the test can go on, even if there are failures. If that assumption is not met in a practical situation, an alternative might be to find a path from whatever situation we are in towards some end situation and play that path (after printing a warning in the report that we are doing this).

The "play game" at the end of Figure 3 is a kick-off action that sets the test in motion. Basically every possible path of the game is walked through, until no more paths are available. However, for more complex models, a more selective approach might be necessary. For example, one could take a random walk (making sure no paths are done twice) until a certain percentage of paths have been traversed and each situation with all its actions has been visited at least once.

A Word of Caution

To make model-based testing successful, a number of skills need to be present in the team, whether through training or hiring. Team members need to understand the problem area and the model technique(s) involved. The automation engineer(s) must have a substantial technical background—preferably one that includes models. Although ABT can make model techniques more accessible, implementing actions requires programming experience. Finally, if no one has experience with ABT, I would recommend building some experience with that technique on "normal" test cases, prior to applying it to models.

ABT is not the only way to deal with models, of course, but it can help you to quickly get some models in place when needed. I hope that the examples in this article, although simplified, give an idea of how to go about this. STQE

Hans Buwalda, now Chief Architect at LogiGear in California (and former Project Director at LogicaCMG in Europe), is the original architect of the action words approach for testing and test automation, and related concepts like testing with "soap operas." You can reach him through www.logigear.com.