Functional Test Generation for Extreme Programming

Mike Holcombe
University of Sheffield,
Regent Court, Portobello Street,
Sheffield, S1 4DP, UK,
+44 114 222 1802
m.holcombe@dcs.shef.ac.uk

Kirill Bogdanov
University of Sheffield,
Regent Court, Portobello Street,
Sheffield, S1 4DP, UK,
+44 114 222 1837
k.bogdanov@dcs.shef.ac.uk

Marian Gheorghe,
University of Sheffield,
Regent Court, Portobello Street,
Sheffield, S1 4DP, UK,
+44 114 222 1843
marian@dcs.shef.ac.uk

ABSTRACT

Test generation and the engineering, including maintenance, of the set of test cases are a key part of the Extreme Programming approach. Since so much depends on the viability of these test sets it is therefore important that methods for constructing them make use of the best available techniques. Total testing provides a mechanism whereby test sets are created which can detect ALL possible faults in an implementation, provided that a number of key conditions are satisfied. This paper describes how total testing can be used in Extreme Programming and illustrates the concepts with a simple case study. The methods proposed here are being used in a number of industrial projects and some interim conclusions from these are presented.

Keywords

Extreme programming, test set generation, general (X-) machines, total testing, test refinement.

1 INTRODUCTION

All software systems are subject to testing - for some of them testing is the major activity in the project. In Extreme Programming [4], [5] the production of test cases is now a vital part of the initial phases of a project. Testing, however, rarely gets the attention it deserves from researchers and developers, partly because its foundations are very weak and ill-understood. The principal purpose of testing is to detect (and then remove) faults in a software system. A number of techniques for carrying out testing, and in particular, for the generation of test sets exist. Many sophisticated (and expensive) tools are available on the market and many developers look to these to provide a solution to the problems of building fault-free systems. We consider the problem of fault detection and note that few, if any, of the existing methods really address the real issues. In particular no methods allow us to make any statement about the *type* or *number of faults* that remain undetected after testing is completed. Thus we cannot really measure the effectiveness of our testing activities in any rigorous way.

Emphasis is usually placed on coverage measures which really indicate effort rather than effectiveness. However, by considering testing from a straightforward, theoretical point of view we demonstrate that a new method for generating test cases can provide a more convincing approach to the problem of detecting ALL faults and allows us to make sensible claims about the level and type of faults remaining after the testing process is complete. We can then integrate this approach into XP in a simple and designer-friendly way. The approach is outlined in the following section, illustrated by an example in section 3 and conclusions made in section 4.

2 THE FUNDAMENTALS OF TOTAL TESTING

The basis of the testing method is that of computational modelling with X-machines (these date from the mid 70s) which are a simple and elegant way of visualising the dynamics of a software system.

The model identifies the set of events and *inputs* that produce observable change, these are mouse clicks, data entry, sensor inputs etc. and the observable *outputs* such as screen displays, commands to peripheral devices etc. Alongside these are a model of - essentially a global - *memory* which describes what the system knows and needs to know in order to permit effective operation of the key functions of the system, examples might be a database or various internal variables.

The concept of a key or *basic* function is one that takes a pair of parameters consisting of the current input and the current state of the memory and produces an output whilst updating the memory. We call these basic functions *business processes* in [1]. These business processes are then *integrated* into a state-based machine model, the *stream X-machine*. This then provides us with a mechanism for building extremely powerful test strategies, [1], [2], [3]. Note that the machine constructed above could itself be regarded as a basic function for a higher level system thus providing the hierarchical leap that makes the method work so well. Essentially the approach allows us to manage the test process, we start at the bottom and test the lowest level basic functions. The method then lets us test the integration of these into the lowest level machine in such a way that, under suitable assumptions (see below) we can detect ALL faults in an implementation. Now we can repeat the process at a

higher level until we are able to test the full integration of the system.

Design for test.

Not many software developers realise that the design can affect the effectiveness of testing and the issue of *design for test* is one that should be considered more. Organising the system in a particular way can make an enormous difference, some systems are almost impossible to test properly if this is not done.

The two key issues are *controllability* and *observability.* By controllability we mean putting in enough functionality so that we can drive the system to any state and apply any basic function from that state under all the necessary conditions needed. This is usually done by designing in special test inputs that can set up the system in a suitable way, these would not be used in normal operation. Observability simply means arranging for enough information to be output that we can distinguish between the various basic functions that might have fired. We achieve this by making suitable data available as outputs, for example printing out appropriate variable values at appropriate times, these need to be disabled at delivery. This is a process that can be a source of error but one that, if done in a controlled and careful way, can significantly increase the effectiveness of testing.

We need to assume that the system satisfies these design for test requirements. There are a couple of other conditions, firstly we need to be sure that the basic functions are correct, this can be done by a separate functional testing method, using category partition and boundary values is an effective way of doing this, or maybe you are using tried and trusted components, for example, functions that take keyboard input and echo it to a screen or put it in a register or perhaps a function that accesses a cell in a database table

The final condition is some sort of estimate of how many extra states there might be in the implementation, compared to the model, usually there are few extra states but one can be pessimistic at the cost of larger test sets.

The test generation algorithm, itself, is best described using an example.

3 A SIMPLE EXAMPLE

Suppose that we are building a simple customer and orders database. We might identify a number of stories such as the following:

1. Customer details are entered customer by customer.

2. Customer details can be edited.

3. Orders are entered by customer

4. Orders can be edited when necessary.

The details of the structure of the customer and orders details are left until later, we try to build an abstract model of the user interface and then refine it. The test approach permits us to generate an abstract high level test strategy and to refine the test cases *in parallel* with the design [1], [2], thus saving enormously in test case size for large examples - a recent case study, involving 3 million transitions, demonstrated this, [6].

story	function	input	current memory	output	updated memory	change risk
1	click(customer)	customer button click	-	new customer screen	-	low
1	enter(customer)	customer details entered	current customer database	confirmation details screen	-	medium (nature of details liable to change)
1	confirm(customer)	customer confirm button clicked	(current customer database)	OK message and start screen button	updated customer database	low
3	click(order)	orders button clicked	-	new orders screen	-	low
3	enter(order)	new order details entered	current orders database	confirmation orders screen	-	high (nature of details of orders liable to change)
3	confirm(order)	orders confirm button clicked	(current orders database)	Ok message and start screen button	updated orders database	low
3	quit()	click on return to start button	-	start screen	-	low

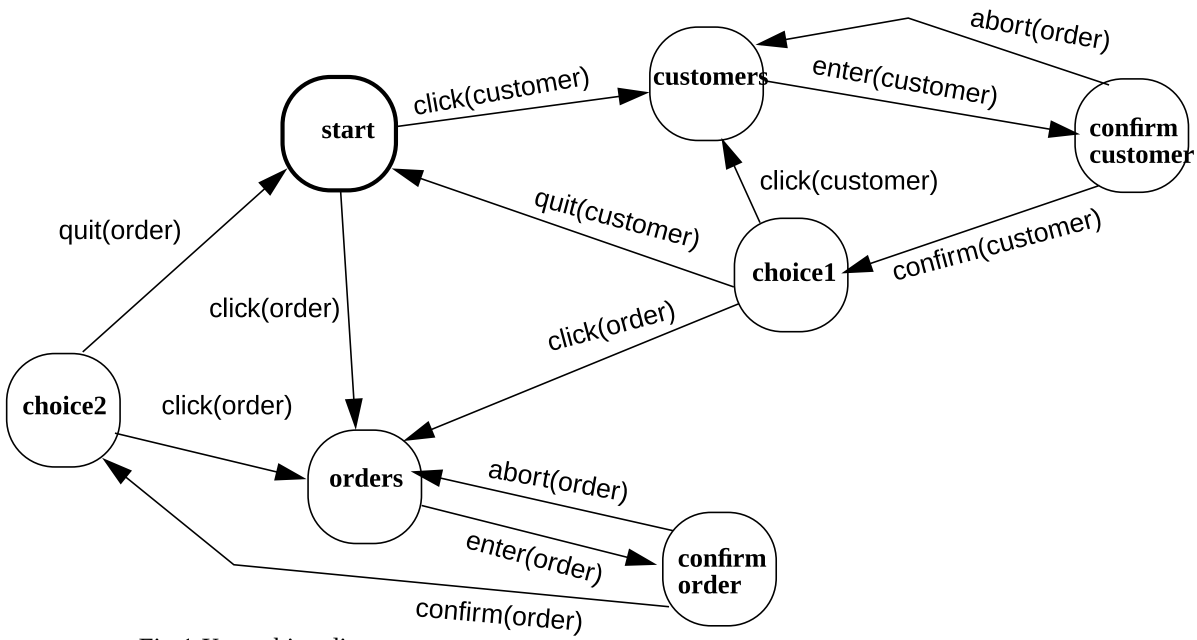

Fig.1 X-machine diagram

Now we try to identify from these stories, what is prompting change (inputs), what internal knowledge is needed (memory), what is the observable result (output) and how the memory changes after the event. We also try to identify the risk that the story will be changed during the course of the project as a means of trying to manage its evolution.

The table above describes some of the functions from the stories in this form.

From the diagram (Fig.1) one can see how the basic functions are organised. Each state has associated an appropriate screen with buttons, text fields etc. Of course, the model is simple and crude, there is no distinction between entering a new customer's details and editing an existing one but it is enough to explain the method. These refinements are, what they say they are, refinements that can be dealt with later and the work we do on test set generation here is built on them.

Test set generation.

Assuming that the basic functions in the table are correct and the design for test conditions are satisfied the test set is generated in the following way.

We start at the state **start** with the initial state of the internal memory, probably in some basic initialised state, and the aim is to visit every state in turn. When we have reached a state we need to confirm that it is the correct state and this is done by following more paths from that state until we get outputs that tell us, unambiguously, what the state was. Then we repeat the path to that state and check what happens if we try to apply every basic function from that state, some will succeeded but some should fail. Have the correct ones passed and failed? This is then repeated for every state.

Some example functions sequences are:

click(customer)::enter(customer);

click(customer)::enter(customer)::click(order)

(Here :: means concatenation or sequence connector.)

[Note that the first test has tried to access the state **confirm customer** correctly and should pass, the second has tried to apply an incorrect function from that state and should fail.]

The test generation, which is fully automated, will generate all the sequences needed to establish whether the implementation is correct, i.e. agrees with the model.

Now, this test set is not quite what we want since it is based on the set of *functions* which we cannot access directly, it needs to be converted to a sequence of *inputs*. So we choose suitable inputs that will trigger the correct functions as we trace through the diagram along the paths of functions generating sequences of inputs which are our actual tests. The design for test conditions allow this to happen, the mathematical details and proof of correctness are in [1] and [2].

Thus we have the following test sequences corresponding to the sequences above:

customer_button_click::customer_details_entered

customer_button_click::customer_details_entered:: orders_button_clicked

where customer_button_click is the event (or input)

corresponding to the clicking of the customer button on the start screen, this should trigger the first function in the sequence.

Of course, as this is a high level test set, the input customer_details_entered represents a more complex series of activities. If the customer screen was structured with a number of data slots representing different parameters, eg. customer_name, customer_address, etc., then this will be modelled with a lower level machine involving more lower level, basic functions which need to be tested first. What this amounts to is that the code associated with the screen for customer data entry needs to be written and tested first.

The memory structure now needs to be discussed. Essentially we need to think about this in terms of what basic types of memory structure is relevant at the different levels. At the top level, for example we could represent it as a small vector or array of compound types of the form:

customer_details × order_details

filling in the actual details later. It may be, for example, that these will represent part of a structured database with a set of special fields which relate to the design of the screens associated with these operations. So customer_details would involve name, address etc. which would be represented as some lower level compound data structure, perhaps and there would be basic functions which insert values into the database table after testing for validity etc.

Fault detection.

Since detecting faults is a major aspect of testing and a key ingredient in any process attempting to improve the quality of the final software product it is worth looking at the way in which typical faults are trapped using these test sets.

Suppose that click(order) did something unwanted when the orders button was clicked whilst in state **confirm customer**. This would be exposed in the testing if the output observed was not an error, signifying that the function is not implemented from that particular state.

Another type of fault might be a missing transition, which, again would be exposed since the response to the test would be an error instead of the expected output.

4 CONCLUSIONS AND FURTHER WORK

There is still many aspects of the relationship between XP and testing to explore. Ultimately we need to build smart test tools which interface naturally with the XP process. Traditionally testing has been left to the end of the coding, the V model tries to encourage designers to derive their unit tests from unit specifications, their system (or function) tests from system specifications and requirements but, unfortunately, this is rarely done since these specifications are rarely stable or suitable and the methodology doesn't force you to focus on the test sets in the way that XP does.

In XP we focus much more on the iterative progression *from requirements to test sets to code* and this presents many new challenges. There are incredible savings in time and gains in quality by using smart test strategies in XP. This paper is an attempt to explain how one of the most powerful test generation approaches could be put to use.

We are currently building some test tools to support our work on using XP in industrial contracts. Further descriptions of these developments and their consequences will follow in further papers.

ACKNOWLEDGEMENTS

We would like to thank our colleagues Francisco Macias, Tony Simons and Mike Stannett for many helpful suggestions and comments on this paper.

REFERENCES

1. M. Holcombe & F. Ipate, "Correct systems - building a business process solution". Springer, Applied Computing Series, 1998.
2. F. Ipate & M. Holcombe, "Specification and testing using generalised machines: a presentation and a case study." Software testing, Verification and Reliability, 8, 61-81, 1998.
3. F. Ipate & M. Holcombe, "A method for refining and testing generalised machine specifications." Int. Jour. Comp. Math. 68, 197-219, 1998.
4. K. Beck, *Extreme Programming Explained: Embrace Change*, Addison Wesley, 1999.
5. XProgramming Web site, On-line at <http://www.XProgramming.com/ >
6. K. Bogdanov & M. Holcombe, "Test generation for a statechart with a relatively large number of states", submitted.

Functional Test Generation for Extreme Programming

Mike Holcombe
University of Sheffield,
Regent Court, Portobello Street,
Sheffield, S1 4DP, UK,
+44 114 222 1802
m.holcombe@dcs.shef.ac.uk

Kirill Bogdanov
University of Sheffield,
Regent Court, Portobello Street,
Sheffield, S1 4DP, UK,
+44 114 222 1837
k.bogdanov@dcs.shef.ac.uk

Marian Gheorghe,
University of Sheffield,
Regent Court, Portobello Street,
Sheffield, S1 4DP, UK,
+44 114 222 1843
marian@dcs.shef.ac.uk

ABSTRACT
Test generation and the engineering, including maintenance, of the set of test cases are a key part of the Extreme Programming approach. Since so much depends on the viability of these test sets it is therefore important that methods for constructing them make use of the best available techniques. Total testing provides a mechanism whereby test sets are created which can detect ALL possible faults in an implementation, provided that a number of key conditions are satisfied. This paper describes how total testing can be used in Extreme Programming and illustrates the concepts with a simple case study. The methods proposed here are being used in a number of industrial projects and some interim conclusions from these are presented.

Keywords
Extreme programming, test set generation, general (X-) machines, total testing, test refinement.

1 INTRODUCTION

All software systems are subject to testing - for some of them testing is the major activity in the project. In Extreme Programming [4], [5] the production of test cases is now a vital part of the initial phases of a project. Testing, however, rarely gets the attention it deserves from researchers and developers, partly because its foundations are very weak and ill-understood. The principal purpose of testing is to detect (and then remove) faults in a software system. A number of techniques for carrying out testing, and in particular, for the generation of test sets exist. Many sophisticated (and expensive) tools are available on the market and many developers look to these to provide a solution to the problems of building fault-free systems. We consider the problem of fault detection and note that few, if any, of the existing methods really address the real issues. In particular no methods allow us to make any statement about the *type* or *number of faults* that remain undetected after testing is completed. Thus we cannot really measure the effectiveness of our testing activities in any rigorous way.

Emphasis is usually placed on coverage measures which really indicate effort rather than effectiveness. However, by considering testing from a straightforward, theoretical point of view we demonstrate that a new method for generating test cases can provide a more convincing approach to the problem of detecting ALL faults and allows us to make sensible claims about the level and type of faults remaining after the testing process is complete. We can then integrate this approach into XP in a simple and designer-friendly way. The approach is outlined in the following section, illustrated by an example in section 3 and conclusions made in section 4.

2 THE FUNDAMENTALS OF TOTAL TESTING

The basis of the testing method is that of computational modelling with X-machines (these date from the mid 70s) which are a simple and elegant way of visualising the dynamics of a software system.

The model identifies the set of events and *inputs* that produce observable change, these are mouse clicks, data entry, sensor inputs etc. and the observable *outputs* such as screen displays, commands to peripheral devices etc. Alongside these are a model of - essentially a global - *memory* which describes what the system knows and needs to know in order to permit effective operation of the key functions of the system, examples might be a database or various internal variables.

The concept of a key or *basic* function is one that takes a pair of parameters consisting of the current input and the current state of the memory and produces an output whilst updating the memory. We call these basic functions *business processes* in [1]. These business processes are then *integrated* into a state-based machine model, the *stream X-machine*. This then provides us with a mechanism for building extremely powerful test strategies, [1], [2], [3]. Note that the machine constructed above could itself be regarded as a basic function for a higher level system thus providing the hierarchical leap that makes the method work so well. Essentially the approach allows us to manage the test process, we start at the bottom and test the lowest level basic functions. The method then lets us test the integration of these into the lowest level machine in such a way that, under suitable assumptions (see below) we can detect ALL faults in an implementation. Now we can repeat the process at a

higher level until we are able to test the full integration of the system.

Design for test.

Not many software developers realise that the design can affect the effectiveness of testing and the issue of *design for test* is one that should be considered more. Organising the system in a particular way can make an enormous difference, some systems are almost impossible to test properly if this is not done.

The two key issues are *controllability* and *observability.* By controllability we mean putting in enough functionality so that we can drive the system to any state and apply any basic function from that state under all the necessary conditions needed. This is usually done by designing in special test inputs that can set up the system in a suitable way, these would not be used in normal operation. Observability simply means arranging for enough information to be output that we can distinguish between the various basic functions that might have fired. We achieve this by making suitable data available as outputs, for example printing out appropriate variable values at appropriate times, these need to be disabled at delivery. This is a process that can be a source of error but one that, if done in a controlled and careful way, can significantly increase the effectiveness of testing.

We need to assume that the system satisfies these design for test requirements. There are a couple of other conditions, firstly we need to be sure that the basic functions are correct, this can be done by a separate functional testing method, using category partition and boundary values is an effective way of doing this, or maybe you are using tried and trusted components, for example, functions that take keyboard input and echo it to a screen or put it in a register or perhaps a function that accesses a cell in a database table

The final condition is some sort of estimate of how many extra states there might be in the implementation, compared to the model, usually there are few extra states but one can be pessimistic at the cost of larger test sets.

The test generation algorithm, itself, is best described using an example.

3 A SIMPLE EXAMPLE

Suppose that we are building a simple customer and orders database. We might identify a number of stories such as the following:

1. Customer details are entered customer by customer.

2. Customer details can be edited.

3. Orders are entered by customer

4. Orders can be edited when necessary.

The details of the structure of the customer and orders details are left until later, we try to build an abstract model of the user interface and then refine it. The test approach permits us to generate an abstract high level test strategy and to refine the test cases *in parallel* with the design [1], [2], thus saving enormously in test case size for large examples - a recent case study, involving 3 million transitions, demonstrated this, [6].

story	function	input	current memory	output	updated memory	change risk
1	click(customer)	customer button click	-	new customer screen	-	low
1	enter(customer)	customer details entered	current customer database	confirmation details screen	-	medium (nature of details liable to change)
1	confirm(customer)	customer confirm button clicked	(current customer database)	OK message and start screen button	updated customer database	low
3	click(order)	orders button clicked	-	new orders screen	-	low
3	enter(order)	new order details entered	current orders database	confirmation orders screen	-	high (nature of details of orders liable to change)
3	confirm(order)	orders confirm button clicked	(current orders database)	Ok message and start screen button	updated orders database	low
3	quit()	click on return to start button	-	start screen	-	low

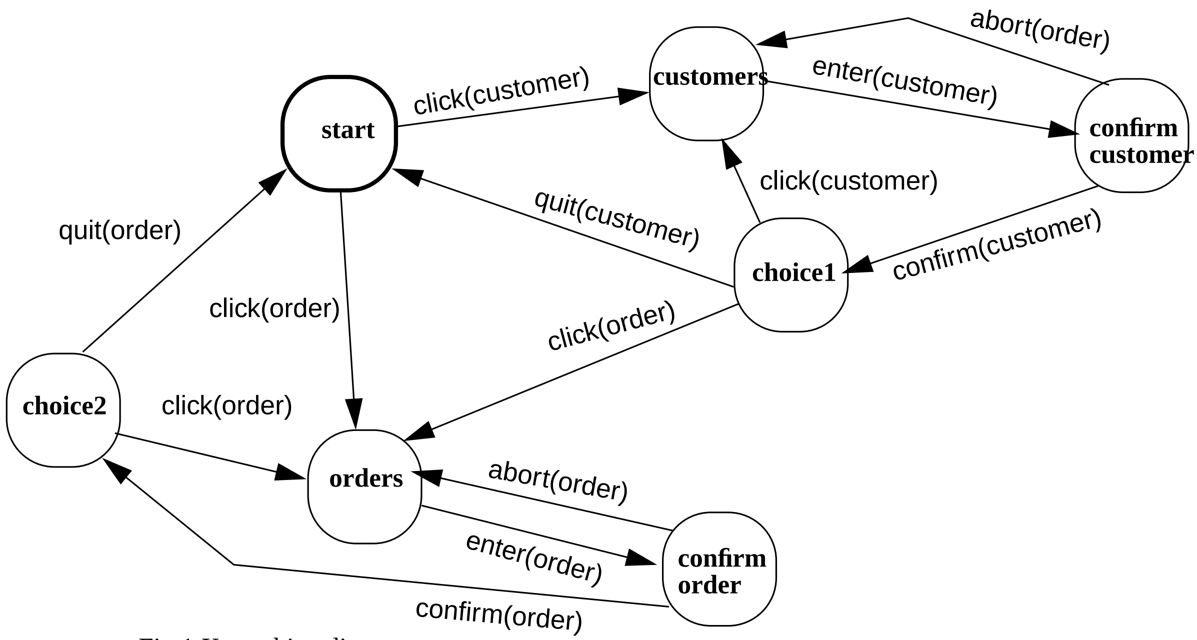

Fig.1 X-machine diagram

Now we try to identify from these stories, what is prompting change (inputs), what internal knowledge is needed (memory), what is the observable result (output) and how the memory changes after the event. We also try to identify the risk that the story will be changed during the course of the project as a means of trying to manage its evolution.

The table above describes some of the functions from the stories in this form.

From the diagram (Fig.1) one can see how the basic functions are organised. Each state has associated an appropriate screen with buttons, text fields etc. Of course, the model is simple and crude, there is no distinction between entering a new customer's details and editing an existing one but it is enough to explain the method. These refinements are, what they say they are, refinements that can be dealt with later and the work we do on test set generation here is built on them.

Test set generation.

Assuming that the basic functions in the table are correct and the design for test conditions are satisfied the test set is generated in the following way.

We start at the state **start** with the initial state of the internal memory, probably in some basic initialised state, and the aim is to visit every state in turn. When we have reached a state we need to confirm that it is the correct state and this is done by following more paths from that state until we get outputs that tell us, unambiguously, what the state was. Then we repeat the path to that state and check what happens if we try to apply every basic function from that state, some will succeeded but some should fail. Have the correct ones passed and failed? This is then repeated for every state.

Some example functions sequences are:

click(customer)::enter(customer);

click(customer)::enter(customer)::click(order)

(Here :: means concatenation or sequence connector.)

[Note that the first test has tried to access the state **confirm customer** correctly and should pass, the second has tried to apply an incorrect function from that state and should fail.]

The test generation, which is fully automated, will generate all the sequences needed to establish whether the implementation is correct, i.e. agrees with the model.

Now, this test set is not quite what we want since it is based on the set of *functions* which we cannot access directly, it needs to be converted to a sequence of *inputs*. So we choose suitable inputs that will trigger the correct functions as we trace through the diagram along the paths of functions generating sequences of inputs which are our actual tests. The design for test conditions allow this to happen, the mathematical details and proof of correctness are in [1] and [2].

Thus we have the following test sequences corresponding to the sequences above:

customer_button_click::customer_details_entered

customer_button_click::customer_details_entered::
orders_button_clicked

where customer_button_click is the event (or input)

corresponding to the clicking of the customer button on the start screen, this should trigger the first function in the sequence.

Of course, as this is a high level test set, the input customer_details_entered represents a more complex series of activities. If the customer screen was structured with a number of data slots representing different parameters, eg. customer_name, customer_address, etc., then this will be modelled with a lower level machine involving more lower level, basic functions which need to be tested first. What this amounts to is that the code associated with the screen for customer data entry needs to be written and tested first.

The memory structure now needs to be discussed. Essentially we need to think about this in terms of what basic types of memory structure is relevant at the different levels. At the top level, for example we could represent it as a small vector or array of compound types of the form:

customer_details × order_details

filling in the actual details later. It may be, for example, that these will represent part of a structured database with a set of special fields which relate to the design of the screens associated with these operations. So customer_details would involve name, address etc. which would be represented as some lower level compound data structure, perhaps and there would be basic functions which insert values into the database table after testing for validity etc.

Fault detection.

Since detecting faults is a major aspect of testing and a key ingredient in any process attempting to improve the quality of the final software product it is worth looking at the way in which typical faults are trapped using these test sets.

Suppose that click(order) did something unwanted when the orders button was clicked whilst in state **confirm customer**. This would be exposed in the testing if the output observed was not an error, signifying that the function is not implemented from that particular state.

Another type of fault might be a missing transition, which, again would be exposed since the response to the test would be an error instead of the expected output.

4 CONCLUSIONS AND FURTHER WORK

There is still many aspects of the relationship between XP and testing to explore. Ultimately we need to build smart test tools which interface naturally with the XP process. Traditionally testing has been left to the end of the coding, the V model tries to encourage designers to derive their unit tests from unit specifications, their system (or function) tests from system specifications and requirements but, unfortunately, this is rarely done since these specifications are rarely stable or suitable and the methodology doesn't force you to focus on the test sets in the way that XP does.

In XP we focus much more on the iterative progression *from requirements to test sets to code* and this presents many new challenges. There are incredible savings in time and gains in quality by using smart test strategies in XP. This paper is an attempt to explain how one of the most powerful test generation approaches could be put to use.

We are currently building some test tools to support our work on using XP in industrial contracts. Further descriptions of these developments and their consequences will follow in further papers.

ACKNOWLEDGEMENTS

We would like to thank our colleagues Francisco Macias, Tony Simons and Mike Stannett for many helpful suggestions and comments on this paper.

REFERENCES

1. M. Holcombe & F. Ipate, "Correct systems - building a business process solution". Springer, Applied Computing Series, 1998.
2. F. Ipate & M. Holcombe, "Specification and testing using generalised machines: a presentation and a case study." Software testing, Verification and Reliability, 8, 61-81, 1998.
3. F. Ipate & M. Holcombe, "A method for refining and testing generalised machine specifications." Int. Jour. Comp. Math. 68, 197-219, 1998.
4. K. Beck, *Extreme Programming Explained: Embrace Change*, Addison Wesley, 1999.
5. XProgramming Web site, On-line at <http://www.XProgramming.com/ >
6. K. Bogdanov & M. Holcombe, "Test generation for a statechart with a relatively large number of states", submitted.

Functional Test Generation for Extreme Programming

Mike Holcombe
University of Sheffield,
Regent Court, Portobello Street,
Sheffield, S1 4DP, UK,
+44 114 222 1802
m.holcombe@dcs.shef.ac.uk

Kirill Bogdanov
University of Sheffield,
Regent Court, Portobello Street,
Sheffield, S1 4DP, UK,
+44 114 222 1837
k.bogdanov@dcs.shef.ac.uk

Marian Gheorghe,
University of Sheffield,
Regent Court, Portobello Street,
Sheffield, S1 4DP, UK,
+44 114 222 1843
marian@dcs.shef.ac.uk

ABSTRACT

Test generation and the engineering, including maintenance, of the set of test cases are a key part of the Extreme Programming approach. Since so much depends on the viability of these test sets it is therefore important that methods for constructing them make use of the best available techniques. Total testing provides a mechanism whereby test sets are created which can detect ALL possible faults in an implementation, provided that a number of key conditions are satisfied. This paper describes how total testing can be used in Extreme Programming and illustrates the concepts with a simple case study. The methods proposed here are being used in a number of industrial projects and some interim conclusions from these are presented.

Keywords

Extreme programming, test set generation, general (X-) machines, total testing, test refinement.

1 INTRODUCTION

All software systems are subject to testing - for some of them testing is the major activity in the project. In Extreme Programming [4], [5] the production of test cases is now a vital part of the initial phases of a project. Testing, however, rarely gets the attention it deserves from researchers and developers, partly because its foundations are very weak and ill-understood. The principal purpose of testing is to detect (and then remove) faults in a software system. A number of techniques for carrying out testing, and in particular, for the generation of test sets exist. Many sophisticated (and expensive) tools are available on the market and many developers look to these to provide a solution to the problems of building fault-free systems. We consider the problem of fault detection and note that few, if any, of the existing methods really address the real issues. In particular no methods allow us to make any statement about the *type* or *number of faults* that remain undetected after testing is completed. Thus we cannot really measure the effectiveness of our testing activities in any rigorous way.

Emphasis is usually placed on coverage measures which really indicate effort rather than effectiveness. However, by considering testing from a straightforward, theoretical point of view we demonstrate that a new method for generating test cases can provide a more convincing approach to the problem of detecting ALL faults and allows us to make sensible claims about the level and type of faults remaining after the testing process is complete. We can then integrate this approach into XP in a simple and designer-friendly way. The approach is outlined in the following section, illustrated by an example in section 3 and conclusions made in section 4.

2 THE FUNDAMENTALS OF TOTAL TESTING

The basis of the testing method is that of computational modelling with X-machines (these date from the mid 70s) which are a simple and elegant way of visualising the dynamics of a software system.

The model identifies the set of events and *inputs* that produce observable change, these are mouse clicks, data entry, sensor inputs etc. and the observable *outputs* such as screen displays, commands to peripheral devices etc. Alongside these are a model of - essentially a global - *memory* which describes what the system knows and needs to know in order to permit effective operation of the key functions of the system, examples might be a database or various internal variables.

The concept of a key or *basic* function is one that takes a pair of parameters consisting of the current input and the current state of the memory and produces an output whilst updating the memory. We call these basic functions *business processes* in [1]. These business processes are then *integrated* into a state-based machine model, the *stream X-machine*. This then provides us with a mechanism for building extremely powerful test strategies, [1], [2], [3]. Note that the machine constructed above could itself be regarded as a basic function for a higher level system thus providing the hierarchical leap that makes the method work so well. Essentially the approach allows us to manage the test process, we start at the bottom and test the lowest level basic functions. The method then lets us test the integration of these into the lowest level machine in such a way that, under suitable assumptions (see below) we can detect ALL faults in an implementation. Now we can repeat the process at a

higher level until we are able to test the full integration of the system.

Design for test.

Not many software developers realise that the design can affect the effectiveness of testing and the issue of *design for test* is one that should be considered more. Organising the system in a particular way can make an enormous difference, some systems are almost impossible to test properly if this is not done.

The two key issues are *controllability* and *observability.* By controllability we mean putting in enough functionality so that we can drive the system to any state and apply any basic function from that state under all the necessary conditions needed. This is usually done by designing in special test inputs that can set up the system in a suitable way, these would not be used in normal operation. Observability simply means arranging for enough information to be output that we can distinguish between the various basic functions that might have fired. We achieve this by making suitable data available as outputs, for example printing out appropriate variable values at appropriate times, these need to be disabled at delivery. This is a process that can be a source of error but one that, if done in a controlled and careful way, can significantly increase the effectiveness of testing.

We need to assume that the system satisfies these design for test requirements. There are a couple of other conditions, firstly we need to be sure that the basic functions are correct, this can be done by a separate functional testing method, using category partition and boundary values is an effective way of doing this, or maybe you are using tried and trusted components, for example, functions that take keyboard input and echo it to a screen or put it in a register or perhaps a function that accesses a cell in a database table

The final condition is some sort of estimate of how many extra states there might be in the implementation, compared to the model, usually there are few extra states but one can be pessimistic at the cost of larger test sets.

The test generation algorithm, itself, is best described using an example.

3 A SIMPLE EXAMPLE

Suppose that we are building a simple customer and orders database. We might identify a number of stories such as the following:

1. Customer details are entered customer by customer.

2. Customer details can be edited.

3. Orders are entered by customer

4. Orders can be edited when necessary.

The details of the structure of the customer and orders details are left until later, we try to build an abstract model of the user interface and then refine it. The test approach permits us to generate an abstract high level test strategy and to refine the test cases *in parallel* with the design [1], [2], thus saving enormously in test case size for large examples - a recent case study, involving 3 million transitions, demonstrated this, [6].

story	function	input	current memory	output	updated memory	change risk
1	click(customer)	customer button click	-	new customer screen	-	low
1	enter(customer)	customer details entered	current customer database	confirmation details screen	-	medium (nature of details liable to change)
1	confirm(customer)	customer confirm button clicked	(current customer database)	OK message and start screen button	updated customer database	low
3	click(order)	orders button clicked	-	new orders screen	-	low
3	enter(order)	new order details entered	current orders database	confirmation orders screen	-	high (nature of details of orders liable to change)
3	confirm(order)	orders confirm button clicked	(current orders database)	Ok message and start screen button	updated orders database	low
3	quit()	click on return to start button	-	start screen	-	low

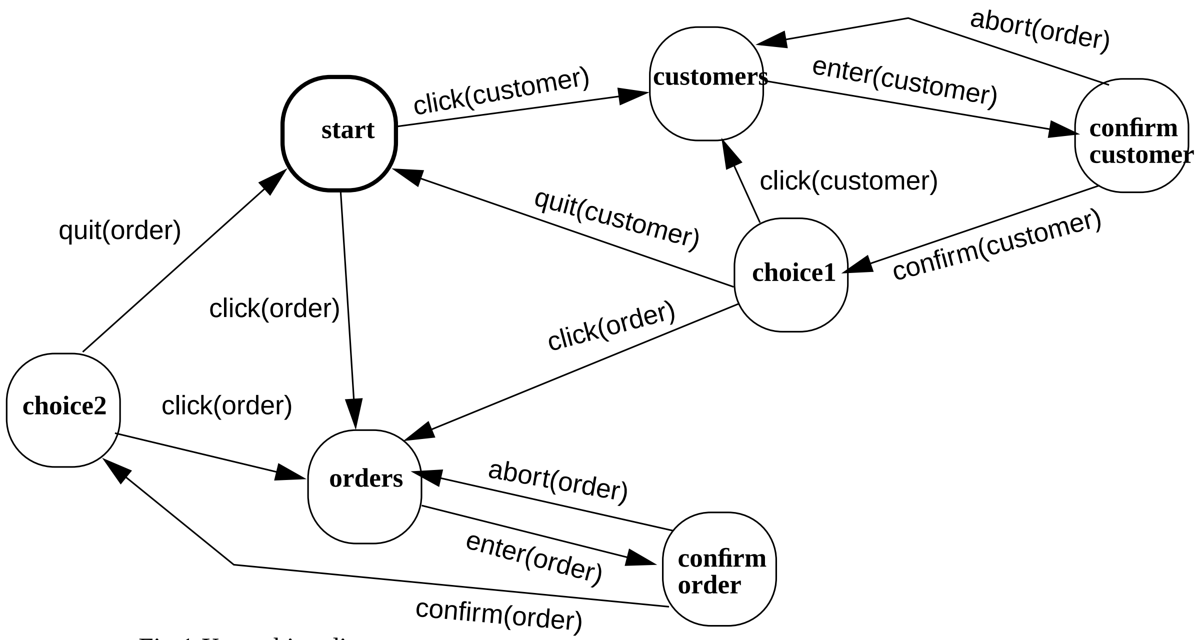

Fig.1 X-machine diagram

Now we try to identify from these stories, what is prompting change (inputs), what internal knowledge is needed (memory), what is the observable result (output) and how the memory changes after the event. We also try to identify the risk that the story will be changed during the course of the project as a means of trying to manage its evolution.

The table above describes some of the functions from the stories in this form.

From the diagram (Fig.1) one can see how the basic functions are organised. Each state has associated an appropriate screen with buttons, text fields etc. Of course, the model is simple and crude, there is no distinction between entering a new customer's details and editing an existing one but it is enough to explain the method. These refinements are, what they say they are, refinements that can be dealt with later and the work we do on test set generation here is built on them.

Test set generation.

Assuming that the basic functions in the table are correct and the design for test conditions are satisfied the test set is generated in the following way.

We start at the state **start** with the initial state of the internal memory, probably in some basic initialised state, and the aim is to visit every state in turn. When we have reached a state we need to confirm that it is the correct state and this is done by following more paths from that state until we get outputs that tell us, unambiguously, what the state was. Then we repeat the path to that state and check what happens if we try to apply every basic function from that state, some will succeeded but some should fail. Have the correct ones passed and failed? This is then repeated for every state.

Some example functions sequences are:

click(customer)::enter(customer);

click(customer)::enter(customer)::click(order)

(Here :: means concatenation or sequence connector.)

[Note that the first test has tried to access the state **confirm customer** correctly and should pass, the second has tried to apply an incorrect function from that state and should fail.]

The test generation, which is fully automated, will generate all the sequences needed to establish whether the implementation is correct, i.e. agrees with the model.

Now, this test set is not quite what we want since it is based on the set of *functions* which we cannot access directly, it needs to be converted to a sequence of *inputs*. So we choose suitable inputs that will trigger the correct functions as we trace through the diagram along the paths of functions generating sequences of inputs which are our actual tests. The design for test conditions allow this to happen, the mathematical details and proof of correctness are in [1] and [2].

Thus we have the following test sequences corresponding to the sequences above:

customer_button_click::customer_details_entered

customer_button_click::customer_details_entered::
orders_button_clicked

where customer_button_click is the event (or input)

corresponding to the clicking of the customer button on the start screen, this should trigger the first function in the sequence.

Of course, as this is a high level test set, the input customer_details_entered represents a more complex series of activities. If the customer screen was structured with a number of data slots representing different parameters, eg. customer_name, customer_address, etc., then this will be modelled with a lower level machine involving more lower level, basic functions which need to be tested first. What this amounts to is that the code associated with the screen for customer data entry needs to be written and tested first.

The memory structure now needs to be discussed. Essentially we need to think about this in terms of what basic types of memory structure is relevant at the different levels. At the top level, for example we could represent it as a small vector or array of compound types of the form:

customer_details × order_details

filling in the actual details later. It may be, for example, that these will represent part of a structured database with a set of special fields which relate to the design of the screens associated with these operations. So customer_details would involve name, address etc. which would be represented as some lower level compound data structure, perhaps and there would be basic functions which insert values into the database table after testing for validity etc.

Fault detection.

Since detecting faults is a major aspect of testing and a key ingredient in any process attempting to improve the quality of the final software product it is worth looking at the way in which typical faults are trapped using these test sets.

Suppose that click(order) did something unwanted when the orders button was clicked whilst in state **confirm customer**. This would be exposed in the testing if the output observed was not an error, signifying that the function is not implemented from that particular state.

Another type of fault might be a missing transition, which, again would be exposed since the response to the test would be an error instead of the expected output.

4 CONCLUSIONS AND FURTHER WORK

There is still many aspects of the relationship between XP and testing to explore. Ultimately we need to build smart test tools which interface naturally with the XP process. Traditionally testing has been left to the end of the coding, the V model tries to encourage designers to derive their unit tests from unit specifications, their system (or function) tests from system specifications and requirements but, unfortunately, this is rarely done since these specifications are rarely stable or suitable and the methodology doesn't force you to focus on the test sets in the way that XP does.

In XP we focus much more on the iterative progression *from requirements to test sets to code* and this presents many new challenges. There are incredible savings in time and gains in quality by using smart test strategies in XP. This paper is an attempt to explain how one of the most powerful test generation approaches could be put to use.

We are currently building some test tools to support our work on using XP in industrial contracts. Further descriptions of these developments and their consequences will follow in further papers.

ACKNOWLEDGEMENTS

We would like to thank our colleagues Francisco Macias, Tony Simons and Mike Stannett for many helpful suggestions and comments on this paper.

REFERENCES

1. M. Holcombe & F. Ipate, "Correct systems - building a business process solution". Springer, Applied Computing Series, 1998.
2. F. Ipate & M. Holcombe, "Specification and testing using generalised machines: a presentation and a case study." Software testing, Verification and Reliability, 8, 61-81, 1998.
3. F. Ipate & M. Holcombe, "A method for refining and testing generalised machine specifications." Int. Jour. Comp. Math. 68, 197-219, 1998.
4. K. Beck, *Extreme Programming Explained: Embrace Change*, Addison Wesley, 1999.
5. XProgramming Web site, On-line at <http://www.XProgramming.com/ >
6. K. Bogdanov & M. Holcombe, "Test generation for a statechart with a relatively large number of states", submitted.

Functional Test Generation for Extreme Programming

Mike Holcombe
University of Sheffield,
Regent Court, Portobello Street,
Sheffield, S1 4DP, UK,
+44 114 222 1802
m.holcombe@dcs.shef.ac.uk

Kirill Bogdanov
University of Sheffield,
Regent Court, Portobello Street,
Sheffield, S1 4DP, UK,
+44 114 222 1837
k.bogdanov@dcs.shef.ac.uk

Marian Gheorghe,
University of Sheffield,
Regent Court, Portobello Street,
Sheffield, S1 4DP, UK,
+44 114 222 1843
marian@dcs.shef.ac.uk

ABSTRACT

Test generation and the engineering, including maintenance, of the set of test cases are a key part of the Extreme Programming approach. Since so much depends on the viability of these test sets it is therefore important that methods for constructing them make use of the best available techniques. Total testing provides a mechanism whereby test sets are created which can detect ALL possible faults in an implementation, provided that a number of key conditions are satisfied. This paper describes how total testing can be used in Extreme Programming and illustrates the concepts with a simple case study. The methods proposed here are being used in a number of industrial projects and some interim conclusions from these are presented.

Keywords

Extreme programming, test set generation, general (X-) machines, total testing, test refinement.

1 INTRODUCTION

All software systems are subject to testing - for some of them testing is the major activity in the project. In Extreme Programming [4], [5] the production of test cases is now a vital part of the initial phases of a project. Testing, however, rarely gets the attention it deserves from researchers and developers, partly because its foundations are very weak and ill-understood. The principal purpose of testing is to detect (and then remove) faults in a software system. A number of techniques for carrying out testing, and in particular, for the generation of test sets exist. Many sophisticated (and expensive) tools are available on the market and many developers look to these to provide a solution to the problems of building fault-free systems. We consider the problem of fault detection and note that few, if any, of the existing methods really address the real issues. In particular no methods allow us to make any statement about the *type* or *number of faults* that remain undetected after testing is completed. Thus we cannot really measure the effectiveness of our testing activities in any rigorous way.

Emphasis is usually placed on coverage measures which really indicate effort rather than effectiveness. However, by considering testing from a straightforward, theoretical point of view we demonstrate that a new method for generating test cases can provide a more convincing approach to the problem of detecting ALL faults and allows us to make sensible claims about the level and type of faults remaining after the testing process is complete. We can then integrate this approach into XP in a simple and designer-friendly way. The approach is outlined in the following section, illustrated by an example in section 3 and conclusions made in section 4.

2 THE FUNDAMENTALS OF TOTAL TESTING

The basis of the testing method is that of computational modelling with X-machines (these date from the mid 70s) which are a simple and elegant way of visualising the dynamics of a software system.

The model identifies the set of events and *inputs* that produce observable change, these are mouse clicks, data entry, sensor inputs etc. and the observable *outputs* such as screen displays, commands to peripheral devices etc. Alongside these are a model of - essentially a global - *memory* which describes what the system knows and needs to know in order to permit effective operation of the key functions of the system, examples might be a database or various internal variables.

The concept of a key or *basic* function is one that takes a pair of parameters consisting of the current input and the current state of the memory and produces an output whilst updating the memory. We call these basic functions *business processes* in [1]. These business processes are then *integrated* into a state-based machine model, the *stream X-machine*. This then provides us with a mechanism for building extremely powerful test strategies, [1], [2], [3]. Note that the machine constructed above could itself be regarded as a basic function for a higher level system thus providing the hierarchical leap that makes the method work so well. Essentially the approach allows us to manage the test process, we start at the bottom and test the lowest level basic functions. The method then lets us test the integration of these into the lowest level machine in such a way that, under suitable assumptions (see below) we can detect ALL faults in an implementation. Now we can repeat the process at a

higher level until we are able to test the full integration of the system.

Design for test.

Not many software developers realise that the design can affect the effectiveness of testing and the issue of *design for test* is one that should be considered more. Organising the system in a particular way can make an enormous difference, some systems are almost impossible to test properly if this is not done.

The two key issues are *controllability* and *observability.* By controllability we mean putting in enough functionality so that we can drive the system to any state and apply any basic function from that state under all the necessary conditions needed. This is usually done by designing in special test inputs that can set up the system in a suitable way, these would not be used in normal operation. Observability simply means arranging for enough information to be output that we can distinguish between the various basic functions that might have fired. We achieve this by making suitable data available as outputs, for example printing out appropriate variable values at appropriate times, these need to be disabled at delivery. This is a process that can be a source of error but one that, if done in a controlled and careful way, can significantly increase the effectiveness of testing.

We need to assume that the system satisfies these design for test requirements. There are a couple of other conditions, firstly we need to be sure that the basic functions are correct, this can be done by a separate functional testing method, using category partition and boundary values is an effective way of doing this, or maybe you are using tried and trusted components, for example, functions that take keyboard input and echo it to a screen or put it in a register or perhaps a function that accesses a cell in a database table

The final condition is some sort of estimate of how many extra states there might be in the implementation, compared to the model, usually there are few extra states but one can be pessimistic at the cost of larger test sets.

The test generation algorithm, itself, is best described using an example.

3 A SIMPLE EXAMPLE

Suppose that we are building a simple customer and orders database. We might identify a number of stories such as the following:

1. Customer details are entered customer by customer.

2. Customer details can be edited.

3. Orders are entered by customer

4. Orders can be edited when necessary.

The details of the structure of the customer and orders details are left until later, we try to build an abstract model of the user interface and then refine it. The test approach permits us to generate an abstract high level test strategy and to refine the test cases *in parallel* with the design [1], [2], thus saving enormously in test case size for large examples - a recent case study, involving 3 million transitions, demonstrated this, [6].

story	function	input	current memory	output	updated memory	change risk
1	click(customer)	customer button click	-	new customer screen	-	low
1	enter(customer)	customer details entered	current customer database	confirmation details screen	-	medium (nature of details liable to change)
1	confirm(customer)	customer confirm button clicked	(current customer database)	OK message and start screen button	updated customer database	low
3	click(order)	orders button clicked	-	new orders screen	-	low
3	enter(order)	new order details entered	current orders database	confirmation orders screen	-	high (nature of details of orders liable to change)
3	confirm(order)	orders confirm button clicked	(current orders database)	Ok message and start screen button	updated orders database	low
3	quit()	click on return to start button	-	start screen	-	low

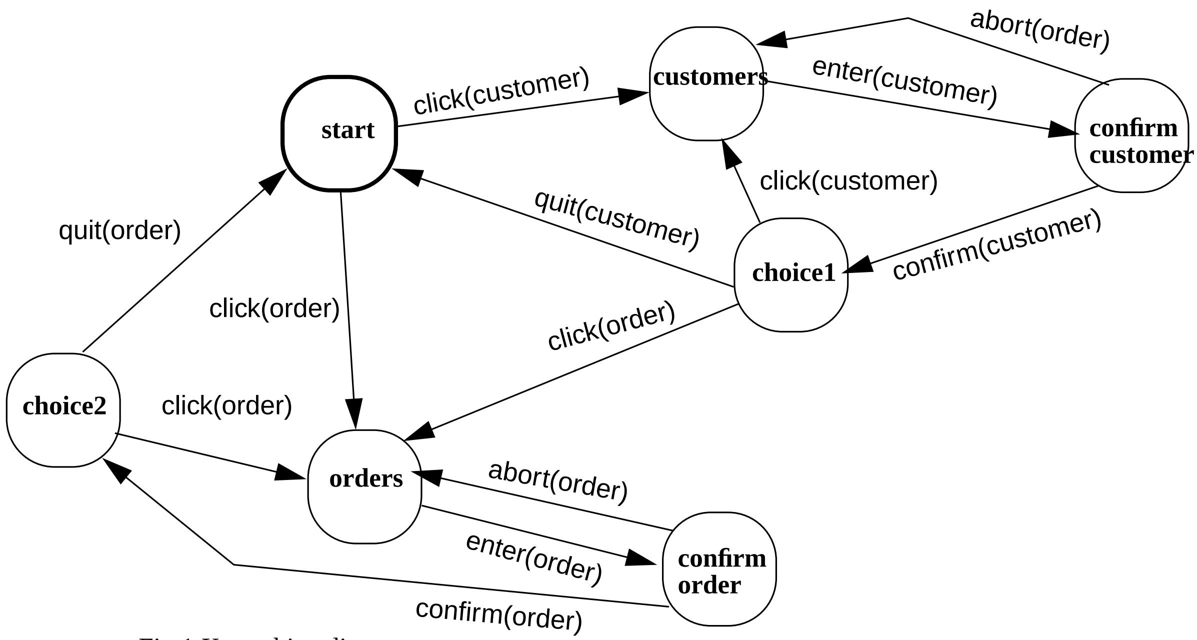

Fig.1 X-machine diagram

Now we try to identify from these stories, what is prompting change (inputs), what internal knowledge is needed (memory), what is the observable result (output) and how the memory changes after the event. We also try to identify the risk that the story will be changed during the course of the project as a means of trying to manage its evolution.

The table above describes some of the functions from the stories in this form.

From the diagram (Fig.1) one can see how the basic functions are organised. Each state has associated an appropriate screen with buttons, text fields etc. Of course, the model is simple and crude, there is no distinction between entering a new customer's details and editing an existing one but it is enough to explain the method. These refinements are, what they say they are, refinements that can be dealt with later and the work we do on test set generation here is built on them.

Test set generation.

Assuming that the basic functions in the table are correct and the design for test conditions are satisfied the test set is generated in the following way.

We start at the state **start** with the initial state of the internal memory, probably in some basic initialised state, and the aim is to visit every state in turn. When we have reached a state we need to confirm that it is the correct state and this is done by following more paths from that state until we get outputs that tell us, unambiguously, what the state was. Then we repeat the path to that state and check what happens if we try to apply every basic function from that state, some will succeeded but some should fail. Have the correct ones passed and failed? This is then repeated for every state.

Some example functions sequences are:

click(customer)::enter(customer);

click(customer)::enter(customer)::click(order)

(Here :: means concatenation or sequence connector.)

[Note that the first test has tried to access the state **confirm customer** correctly and should pass, the second has tried to apply an incorrect function from that state and should fail.]

The test generation, which is fully automated, will generate all the sequences needed to establish whether the implementation is correct, i.e. agrees with the model.

Now, this test set is not quite what we want since it is based on the set of *functions* which we cannot access directly, it needs to be converted to a sequence of *inputs*. So we choose suitable inputs that will trigger the correct functions as we trace through the diagram along the paths of functions generating sequences of inputs which are our actual tests. The design for test conditions allow this to happen, the mathematical details and proof of correctness are in [1] and [2].

Thus we have the following test sequences corresponding to the sequences above:

customer_button_click::customer_details_entered

customer_button_click::customer_details_entered::
orders_button_clicked

where customer_button_click is the event (or input)

corresponding to the clicking of the customer button on the start screen, this should trigger the first function in the sequence.

Of course, as this is a high level test set, the input customer_details_entered represents a more complex series of activities. If the customer screen was structured with a number of data slots representing different parameters, eg. customer_name, customer_address, etc., then this will be modelled with a lower level machine involving more lower level, basic functions which need to be tested first. What this amounts to is that the code associated with the screen for customer data entry needs to be written and tested first.

The memory structure now needs to be discussed. Essentially we need to think about this in terms of what basic types of memory structure is relevant at the different levels. At the top level, for example we could represent it as a small vector or array of compound types of the form:

customer_details × order_details

filling in the actual details later. It may be, for example, that these will represent part of a structured database with a set of special fields which relate to the design of the screens associated with these operations. So customer_details would involve name, address etc. which would be represented as some lower level compound data structure, perhaps and there would be basic functions which insert values into the database table after testing for validity etc.

Fault detection.

Since detecting faults is a major aspect of testing and a key ingredient in any process attempting to improve the quality of the final software product it is worth looking at the way in which typical faults are trapped using these test sets.

Suppose that click(order) did something unwanted when the orders button was clicked whilst in state **confirm customer**. This would be exposed in the testing if the output observed was not an error, signifying that the function is not implemented from that particular state.

Another type of fault might be a missing transition, which, again would be exposed since the response to the test would be an error instead of the expected output.

4 CONCLUSIONS AND FURTHER WORK

There is still many aspects of the relationship between XP and testing to explore. Ultimately we need to build smart test tools which interface naturally with the XP process. Traditionally testing has been left to the end of the coding, the V model tries to encourage designers to derive their unit tests from unit specifications, their system (or function) tests from system specifications and requirements but, unfortunately, this is rarely done since these specifications are rarely stable or suitable and the methodology doesn't force you to focus on the test sets in the way that XP does.

In XP we focus much more on the iterative progression *from requirements to test sets to code* and this presents many new challenges. There are incredible savings in time and gains in quality by using smart test strategies in XP. This paper is an attempt to explain how one of the most powerful test generation approaches could be put to use.

We are currently building some test tools to support our work on using XP in industrial contracts. Further descriptions of these developments and their consequences will follow in further papers.

ACKNOWLEDGEMENTS

We would like to thank our colleagues Francisco Macias, Tony Simons and Mike Stannett for many helpful suggestions and comments on this paper.

REFERENCES

1. M. Holcombe & F. Ipate, "Correct systems - building a business process solution". Springer, Applied Computing Series, 1998.
2. F. Ipate & M. Holcombe, "Specification and testing using generalised machines: a presentation and a case study." Software testing, Verification and Reliability, 8, 61-81, 1998.
3. F. Ipate & M. Holcombe, "A method for refining and testing generalised machine specifications." Int. Jour. Comp. Math. 68, 197-219, 1998.
4. K. Beck, *Extreme Programming Explained: Embrace Change*, Addison Wesley, 1999.
5. XProgramming Web site, On-line at <http://www.XProgramming.com/ >
6. K. Bogdanov & M. Holcombe, "Test generation for a statechart with a relatively large number of states", submitted.