Automation of Test Case Generation from Behavior Tree Requirements Models

Peter A. Lindsay, Sentot Kromodimoeljo, Paul A. Strooper
The University of Queensland
School of ITEE
St Lucia, Queensland, Australia
p.lindsay@uq.edu.au, p.strooper@uq.edu.au

Mohamed Almorsy
Swinburne University of Technology
School of Software and Electrical Engineering
Hawthorn, Victoria, Australia
malmorsy@swin.edu.au

Abstract —**Behavior Trees (BTs) are a graphical notation for requirements capture that is easier to read than other formal notations, with direct traceability between individual functional requirements and their representation in the BT model. This paper investigates whether this relationship can be extended to generation of test cases, using a symbolic model checker to ensure correctness and completeness of test cases with respect to the model. To do so it was necessary to provide mechanisms for test planner input and to control the combinatorial explosion of test cases that results from models containing parallel behaviour. The result is an automated process for generating a complete set of natural-language test cases, with tracing back to the original requirements and with correctness and completeness guaranteed by the model checker. The approach is demonstrated on an Automated Teller Machine example and then applied to an example from air traffic control in a model with multi-threaded behaviour.**

Keywords—Test case generation; Behavior Trees; model-based testing; symbolic model checking

I. INTRODUCTION

Software engineering standards for critical systems such as DO-178C [1] require traceability from system requirements to test cases. Typically test case development and tracing is done manually, which is difficult, labour-intensive, error-prone and costly. Model Based Testing has been proposed as a solution to this problem [2], but modelling is difficult, often several different notations are required, and stakeholders have difficulty understanding and reviewing the models.

The Behavior Tree (BT) notation and method were introduced by Geoff Dromey as a means for capturing functional requirements in a single model [3]. The notation has been found to be easier to understand and trace back to natural language requirements than modelling notations such as UML [4, 5]. The notation has a formal semantics, which has enabled development of a range of support tools such as animators and model checkers [6].

This paper proposes a means for generating functional test cases directly from BT models of system requirements. The BT model acts as the oracle for testing. The test planner adds annotations to the BT model to control the granularity and number of test cases generated, and thereafter the process is largely automated. A prototype tool has been developed to generate the test cases and calculate their preconditions.

The main difference between BT notation and other system modelling notations is that system functionality is localised in BTs, in the sense that individual functional requirements are captured in more-or-less contiguous subtrees of the model [7]. This results in several key advantages for Test Case Generation (TCG): there is a close relationship between individual requirements and their associated test cases; tracing from requirements to test cases is easy; and test cases can automatically be converted to natural language instructions to testers using terminology directly from the requirements specification. The approach supports different levels of testing including unit testing, integration testing, factory testing and acceptance testing. Other contributions of the paper include provision of a path generation facility for BT models based on symbolic model checking, and a mechanism to control the combinatorial explosion in test cases for models with parallel behaviour.

The paper is structured as follows: Section II introduces the BT notation and gives an overview of the approach. Section III describes the BT notation in more detail, illustrated on the Automated Teller Machine (ATM) example specified by Russell Bjork [8]. The proposed TCG approach is illustrated on the ATM example in Section IV; the results are compared with Bjork's informal approach and with Utting and Legeard's application of the traditional UML approach [2]. Section IV introduces a second example, to illustrate how the approach applies to BT constructs that were not covered in the ATM example; the example is based on a tool used by air-traffic controllers. Section V discusses limitations of the approach and future work. Section VI discusses related work.

II. OVERVIEW

A. Requirements modelling notation

A BT model is a directed tree whose nodes represent states and conditions of system components, and events and actions such as external input/output and inter-component messaging, similar to the information in nodes and transition labels in state machine notations such as UML State Diagrams [2], although the BT notation does not support hierarchical states. The edges in BT models capture control flow: i.e., the order in which, and conditions under which, components change state and actions occur. The BT notation is explained in detail in Section III.

The BT method's goal is improved understanding and specification of system requirements by stepwise, manual text-based translation of functional requirements into tree segments in the BT notation, and then integrating the segments into a single BT model [3]. BT models thus resemble Functional Flow Block Diagrams in that they capture behaviour as sequences of functions, represented as subtrees [7]. They are also like UML Sequence Diagrams (SDs), in that they capture scenarios as paths through the tree; unlike SDs, which require a new diagram for each scenario, BT branching constructs enable all behaviours to be captured in a single model. For more

details and comparison with other systems modelling approaches, see Lindsay [7].

In terms of coverage and concepts, the BT notation is closest in spirit to state machine (SM) notations, since both track how component states change in response to external events and inputs. The main structural difference is that SM models group together state transitions by component whereas BT models group them by behaviour. The BT structure facilitates tracing back to requirements since functionality is localised rather than being spread across different state machines. This in turn makes it much easier to verify BT models against requirements, which we believe is the main reason they are more popular with stakeholders who are not experts in formal methods [4].

A wide variety of tools have been developed to support BT modelling and analysis, including editors and model checkers. The BT Analyser tool developed by Sentot Kromodimoeljo [9] has been extended to support the approach defined here.

B. TCG preparation

The aim is to provide automated support to the test planner for the test case identification and specification process, where the end deliverables include textual descriptions of the test cases that can be used as instructions for testers.

In outline, test cases will be generated from paths through a BT model of the system requirements. The test planner first needs to decide on the granularity of testing and provide testing-related information that is not present in the requirements specification. Specifically, the test planner needs to perform four tasks:

1. Choose which component states are going to represent the system states from which the testing part of test cases start and end. These are called the *Check Points (CPs)* in what follows. A particular CP is nominated as the *initial system state*, from which test cases begin and end; typically this will be a node at the root of the tree, but it doesn't have to be. In practice it may be desirable to allow multiple possible initial system states, but here we restrict to a single CP to simplify the explanation of the approach.
2. Choose which specific component states and/or outputs will be observable to the tester as system responses, and describe the expected observations in words taken from the original requirements.
3. For events in the model and external inputs from components that will be under the tester's control, describe the corresponding tester action.
4. For external inputs from components not under the tester's control, describe conditions that will cause the desired input.

The choice of CPs dictates the granularity of test cases, such as whether there are a small number of cases consisting of many steps, or a larger number of cases consisting of only one or two steps. The choice of initial CP determines how far back the tester needs to go when resetting the system after each test case. For example, in the ATM example in Section IV below, Bjork has a number of test cases within a single customer session, whereas Utting shuts down the ATM completely after each step. Our approach leaves the choice to the test planner. In Section IV we use Bjork's choices for illustration.

Steps 2 and 3 are a simple reverse translation of the usual BT modelling step, from BT nodes to natural language. Step 4 can be more challenging: see Section IV for an example and more discussion. The nature of the testing determines which components the tester has control over: for unit testing the tester controls inputs from all components other than the one being tested; for factory testing the tester controls (or simulates) inputs only from external components; and for acceptance testing the tester controls no components at all, other than the operator interface.

In multi-threaded models there are typically many execution paths which differ only in the order of interleaving of behaviours. This leads to a combinatorial explosion of possibilities, often differing only in minor details. To reduce the number of test cases our approach allows the test planner to specify test-case equivalence relations by nominating as *Nodes Of Interest (NOI)* the key state changes and/or events whose occurrence and order of occurrence are of interest. For example, the test planner can opt for full black box test coverage by nominating all the external input events as the NOI: the BT Analyser will return examples of all external input event sequences that change the system state. Alternatively, the test planner may simply be interested in test cases that affect the state of key components: this can be achieved by nominating the relevant component states as the NOI. This mechanism is illustrated in Section V below.

C. Automated support for TCG

Once the CPs and NOI have been nominated, the BT Analyser tool generates test cases in an abstract form. A subsequent processing step converts them automatically into a textual description that can be given directly to the tester.

The BT Analyser extracts a representative of each equivalence class of feasible paths between successive CP nodes in the BT model. A *test case path (TCP)* is a sequence of BT nodes, starting and ending in a CP node but with no other CP nodes in-between. Note that a TCP might start and end with the same system state. TCPs will be used in two ways below: either to tell the tester what actions they need to take to move the system from one CP to another, or to generate a sequence of user actions and expected system responses that forms the basis of a test case.

Conceptually, the TCP-generation process simulates traversing the BT model top-down, forking paths at non-deterministic branch points (i.e., creating a new path for each branch, with a copy of the path to this point for each one), and following jumps indicated by flags at leaf nodes, until the next CP is reached. The process is fully automated.

Path generation is based on symbolic model checking: a path is generated only if it is semantically *feasible* in the underlying formal model. Infeasible paths are eliminated during path generation: if a selection or guard occurs with a condition which is inconsistent with the component's current state, the BT Analyser eliminates the path from consideration, since it can never be executed. The number of feasible paths is typically far less than would be generated by simple tree traversal with interleavings. Moreover, the tool finds the preconditions for each path, indicating the conditions that need to be in place for the path to be feasible. The BT Analyser

performs a reachability check to pre-process BT models, to check that all nodes can feasibly be reached

The final step is to create a test case corresponding to each TCP. Each test case consists of the following three parts:

1. *Pre-amble*: a precondition and sequence of steps from the initial system state to the test case starting state, SS say. This will generally involve stringing together several paths, if SS is not directly reachable from the root of the BT. Either the test planner could choose which particular paths are used, such as the ones with the least number of actions, or the BT Analyser can find one automatically.
2. *The test steps*: a sequence of user actions and observable system responses generated from each path that starts in state SS. The BT model acts as an oracle, describing the expected system response to each user action. Note that several user actions may be required before an effect is observable, and conversely several observable effects may occur before another user action is required.
3. *Post-amble*: a sequence of steps that returns the system to its initial state; i.e., "resets" the system. Again, this could be chosen by the test planner or found automatically.

Rather than returning the system to its initial state after each test case, the test planner might choose some other intermediate state as the reset point, from which to start the next test case. We will not go into the details but simply note that path generation between CPs provides the mechanism to support this; it is a matter for test planning, whereas this paper focuses on test case generation.

The path generation and path translation processes are fully automated and explained in more detail and illustrated on two examples below. Each test case is a step-by-step translation of the corresponding BT path. In some cases it is necessary to add conditions to the test-case pre-amble to ensure that the path will be executed correctly and completely during testing.

III. Background: Behavior Trees

The BT notation has evolved over time: this paper uses the BT syntax standardised in [10] and the semantics of Colvin and Hayes [11]. This section describes the core BT syntax for modelling functional behaviour and illustrates it on the well-known Automatic Teller Machine (ATM) example, based on the description supplied by Russell Bjork [8]. This is the same example that Utting and Legeard [2] use to illustrate the approach to model-based testing using UML.

A. The BT notation

In overview: a BT model is a directed tree made up of different types of nodes (Figure 1) and two types of branching (Figure 2). Non-deterministic branches are indicated by '[]' in the child nodes. Edges in the tree represent control flow; leaf nodes can also contain flags that indicate how control flows from that point (Figure 3). A BT model is intended to represent all possible behaviours of a multi-threaded system.

Each node has a system component or external agent (e.g., a user) associated with it. In more detail, the different types of nodes shown in Figure 1 are:

- *state realisation*: indicating that the named component is currently in the indicated state;
- *selection*: control passes this point only if the component is in the named state (or not in that state, if the state name has 'not' before it), otherwise the thread dies;
- *event*: control waits at this point until the named event occurs;
- *guard*: control waits until the named component enters the named state;
- *external input*: the named component receives the named message and data value; and
- *external output*: the named component sends out the named message and data value.

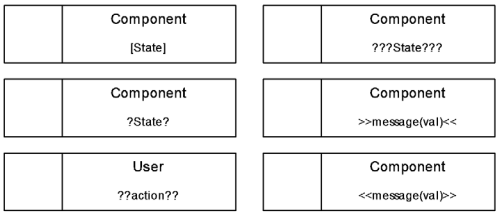

Figure 1. BT node types (left column first): state realisation; selection; event; guard; external input; external output

There is also an option to assign and check attribute values, as a refinement of component states. Finally, nodes can be combined atomically, depicted as being joined by an edge without an arrowhead, meaning they get executed without intervening steps of other threads.

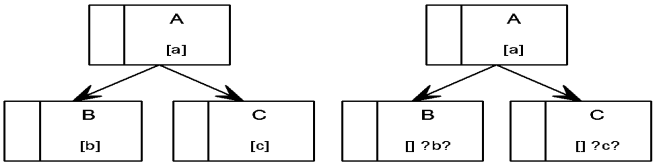

Figure 2. BT branching types: parallel; non-deterministic

System behaviour consists of components changing states in response to events and inputs from external systems. A BT model captures this as a set of all possible multi-threaded "execution paths" through the tree. In a particular execution, control flows from the root of the tree down edges. At non-deterministic branching nodes, exactly one of the child branches is taken. Typically non-deterministic branches start with a selection, event or external input node. Control forks at parallel branching nodes: i.e., all of the child branches get executed. The "prioritised" semantics of BT execution is used here [6], whereby transitions involving external inputs and events are delayed until all other enabled transitions have been executed.

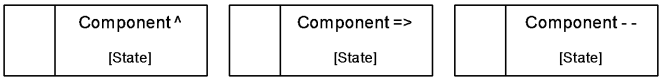

Figure 3. BT control flow flags: reversion; reference; kill

When execution reaches a leaf node the following rules apply, according to what flag, if any, occurs (Figure 3). The full syntax also includes a synchronisation construct and internal

I/O for message passing between components. See [3] for details of the BT notation and semantics.

B. The BT method

This section illustrates the BT notation on the ATM example [8]. For simplicity the ATM functionality has been restricted here to performing transactions. Details such as PIN checks, unreadable cards, logging and cash dispensing have been omitted for reasons of space. It would be straightforward to add such details to the BT model, but this approach highlights the differences better. The BT model for the ATM appears in full in Figure 4.

Note that one of the strong points of BT models is that system functionality is localised: i.e., the nodes corresponding to individual requirements are typically contiguous within the model. This makes it easy to trace between requirements and the model. Since test cases will be generated from individual parts of the model and tags are carried through, traceability between requirements and test cases is ensured. The numeric tags in the left part of each node are normally used for tracing back to the requirement identifiers but are instead numbered here for ease of reference in the full model. (The example in Section V illustrates requirements traceability.)

Customer selections of an account and withdrawal amount are shown as parameters in external input messages received through the console (nodes 7 and 9). The withdrawal request is sent via the communication link (*Coms*) to the bank for approval, with the relevant data as arguments. If withdrawal is approved, the updated balance is returned as part of the approval message from the bank; this fact was not made explicit in Bjork's requirements.

As the model develops, unstated requirements and assumptions often become apparent. For example, a BT completeness check is that the nodes below a non-deterministic branch point should exhaust all of the possible options. The branch below node 22 covers the case in which the customer selects an account that is not linked to the card. Bjork does not consider this case explicitly, so we assume the system simply shows an error, with the reason, and asks them to choose again. This is an example of the kind of missing requirement that the BT method is good at uncovering. Arguably an error should also be returned if the account balance is below the minimum possible withdrawal amount.

C. The BT Analyser tool

The BT Analyser was developed as an extension of Sentot Kromodimoeljo's symbolic model checker for finding multiple counterexamples [9]. To explain its use here some definitions are first required: Given a BT model M and a set E of nodes of M representing the NOI, a node sequence ns from E* is said to be *feasible* between CP nodes N1 and N2 in M if there exists an execution path through M which: starts at a node matching N1, ends at a node matching N2 and does not pass through any other CP nodes; and which matches ns when projected to E. (Note that tags and flags are ignored.)

Two execution paths are said to be *E-equivalent* if they project to the same node sequence from E*.

The BT Analyser can determine whether there is a feasible execution path between any two CP nodes in M and if so, will return a representative of each E-equivalence class of execution paths for the node pair. A shortest such path is returned. (There may be several different shortest paths.) These are the subpaths that are used for test case generation.

For example, if the ATM states are used as the CPs for the ATM example and the Coms unit input nodes as the NOI, then the execution paths 3-5-27-28(13) and 3-5-7-29(27)-28(13) are equivalent, since neither of them involves NOI. (Intermediate nodes in execution have been elided to save space.) On the other hand, the following paths are not equivalent, since they involve different sequences of NOI (shown in square brackets after the path): 3-5-7-9-11-13 [11]; 3-5-7-9-19-21(8)-9-11-13 [19,11]; 3-5-7-9-19-21(8)-9-19-21(8)-9-11-13 [19,19,11]. For this choice of NOI the generated tests check behaviour associated with the banking system interface and ignore system usability issues such as whether the user can cancel interactions.

IV. Usage Example: The ATM

The TCG process from Section II is explained in detail below, illustrated on Bjork's ATM example [8].

A. Test planning decisions

As described in Section II.B above, the test planner's first step is to nominate *Check Points*: i.e., the component states from which test cases start and end. Bjork has different sets of test cases for the different phases of ATM operation, such as for system start-up and shutdown, starting a customer "session", and dealing with each of the different transaction types. Of these, only a subset of the tests concerning session management and the withdrawal transaction are relevant here, due to the reduced system scope explained in Section III. To recreate Bjork's test cases using our approach, the following nodes would be chosen as the CPs, grouped by matching nodes: 1 (=18), 4, 6 (=24), 8 (=21) and 13 (=28). For step 2, Table 1 shows the system responses Bjork nominated as observables for the ATM system, with the corresponding nodes from the BT model; error node 22 has been added for completeness. The descriptions are a straightforward reverse-translation of BT component states and external output messages back to natural language.

Step 3 involves explaining in words what actions the tester will take during testing. For events and external input messages involving the customer, such as inserting the card into the Reader, selecting from menus, and entering amounts via the Console, this is a simple matter of reverse translation, similar to step 2 above. For example, the "cancel" event (node 27) translates back to the user action "hit the cancel key", and *Reader>>AcceptCard<<* (node 2) translates back to "insert the card into the card reader slot".

Step 4 involves explaining the conditions the tester will need to establish before executing a test case, in order to stimulate the desired inputs from components that are outside their control. For system testing of the ATM example, the inputs in question would be the three possible responses from the bank in response to a withdrawal request (nodes 11, 19 and 22). Table 2 describes the corresponding conditions. These conditions will be included as instructions to the tester in the precondition part of the pre-amble of the corresponding test cases. The instructions indicate what values should be assigned to the parameters *acc* and *amt* when the test execution reaches the corresponding customer inputs (nodes 7 and 9, respective-

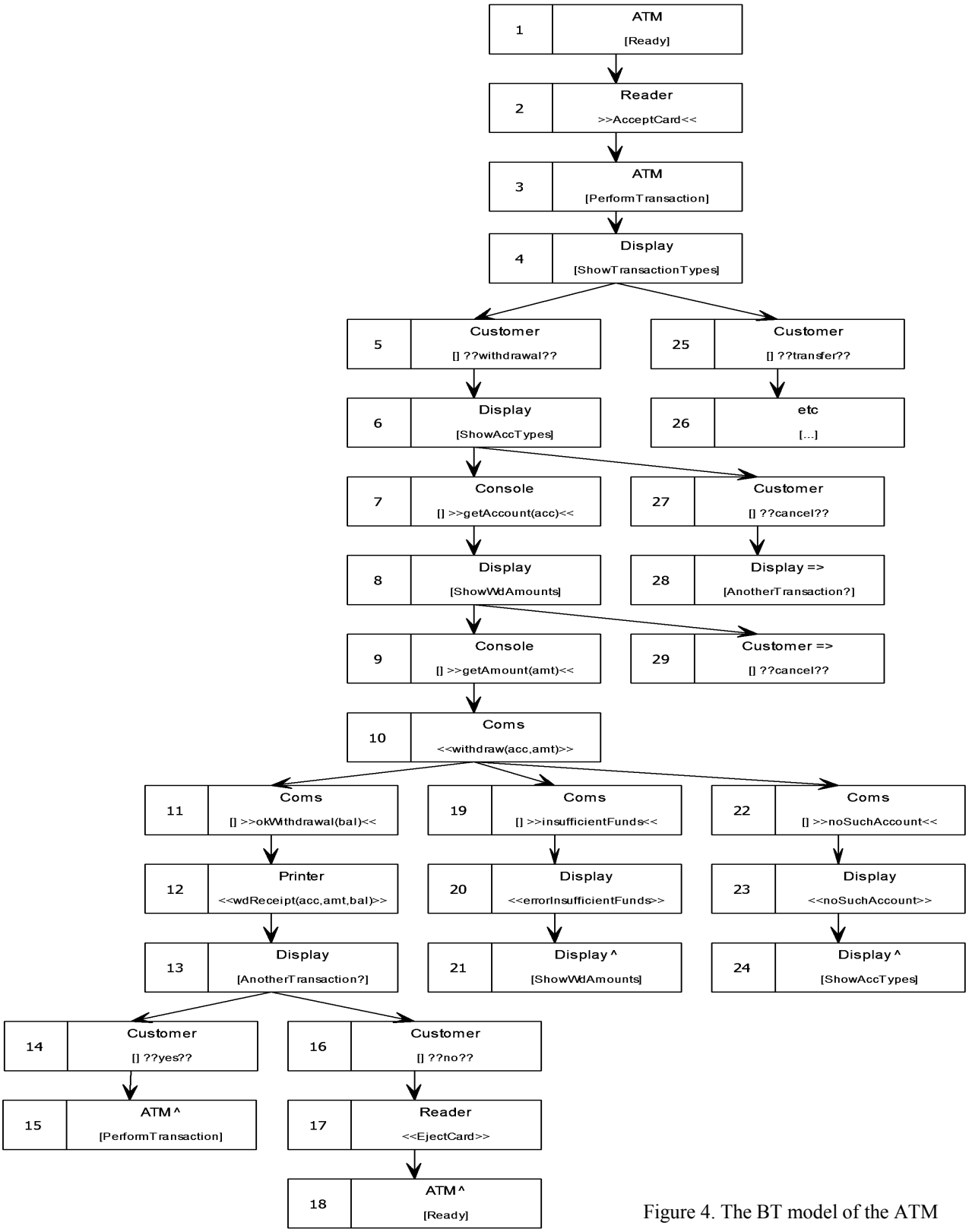

Figure 4. The BT model of the ATM

ly). More generally step 4 will require detailed understanding of how external systems such as the bank behave: unfortunately this is often not fully specified in the system requirements.

Table 1. Observable ATM system responses

Node	Observable effect
1	system waiting for card to be entered
4	display shows menu of transaction options
6	display shows menu of account types
8	display shows menu of withdrawal amounts
12	printed receipt shows withdrawal details, including the account, amount and new balance
13	display asks if another transaction wanted
17	card gets ejected
19	error shown (insufficient funds)
22	error shown (no such account)

Table 2. Instructions for stimulating external inputs

Input message	Instruction
okWithdrawal(bal)	Ensure that the card has an account with a balance exceeding the minimum withdrawal amount, and select that account and amount when prompted.
insufficientFunds	Ensure that the card has an account with a balance less than the maximum allowed withdrawal amount, and select that account and the maximum amount when prompted.
noSuchAccount	Use a card that does not have all account types and, when prompted, select an account type that is not linked to the card.

B. Path generation for the ATM example

As noted in Section II.C above, the TCP-generation process involves traversing the BT top-down, forking paths at non-deterministic branch points, and following reversions and jumps, until the next CP-matching node is reached. Table 3 shows in short-hand form the paths that the tool generated for the ATM example; intervening nodes have been elided.

Instructions need to be generated for the pre-amble of each test-case telling the tester how to initialise the system. For the sake of illustration let's take ATM in the *Ready* state as the initial CP. The first four CPs can easily be reached from the initial state, by taking the path 1-4-6-8; the fifth CP can be reached by taking the path 1-4-6-27-28: i.e., by inserting the card and selecting the withdrawal option, then hitting the cancel key.

Likewise, instructions need to be generated for the post-amble saying how to reset the system. From system states 6 and 8 the path 27-28(13)-16-18 takes the ATM back to the *Ready* state: i.e., the tester hits the cancel key then selects 'no'. For system state 4 the user first has to select the withdrawal option. For system state 13 it is simply a matter of selecting `no'. This treatment assumes the bank computer always responds. If not, some other way will need to be found for resetting the system in test cases that proceed past node 9.

Table 3. Paths generated for the ATM example

#	Path	#	Path
1	1-4	6	8-19-21(8)
2	4-6	7	8-22-24(6)
3	6-8	8	8-29(27)-28(13)
4	6-27-28(13)	9	13-14-15(3)-4
5	8-11-13	10	13-16-18(1)

C. Path translation

As noted in Section II.C above, the process of translating a BT path into a test case is a matter of working down the path, translating events and external input node into a user action, and translating nominated observable nodes into an expected observable system response, using the reverse translation approach explained in Section II.C. State changes that have not been declared to be observable, such as node 3, and other node types simply get ignored.

As a simple example, test case 1 from Table 3 becomes the following test steps: insert a card (node 2) and check that the display shows the transaction options menu (node 4).

As a more interesting example, consider test case 5 (path 8-11-13). The path contains an external input message *okWithdrawal(bal)* so the corresponding condition from Table 2 needs to be added to the pre-amble: i.e., the tester needs to ensure the card has an account *acc* with a balance exceeding the minimum withdrawal amount *amt*. The pre-amble will also tell the tester how to bring the system to the state from which testing begins: namely, insert the card, select the withdrawal transaction, and select account *acc* from the accounts menu. There is one test step in this case: select the minimum withdrawal amount *amt* from the withdrawal amounts menu. The system should respond by printing a receipt and asking if another transaction is wanted. The post-amble instructs the tester to reset the system by answering 'no'.

Now consider test case 6 (path 8-19-21). Node 19 causes the following instruction to be added to the pre-amble: use a card with an account (*acc*) whose balance is less than the maximum allowed withdrawal amount; insert the card, select withdrawal, and then select account *acc*. The test then consists of selecting the maximum withdrawal amount; the system should respond by showing the 'insufficient funds' error, then displaying the menu of withdrawal amounts. The other paths from Table 3 translate in a similarly straightforward manner. Sections V and VI discuss translation of the BT constructs not used in the ATM model.

D. Comparison with other approaches

Table 4 shows how Bjork's test cases can be derived from the paths in Table 3 by the above approach. The numbering corresponds to his 'session' and 'withdrawal' test cases [8]: e.g., s1 is his first test case for the 'session' use case, and w3 is his third test case for the 'withdrawal' use

case. Cases s2, s3 and w4 are ruled out by the scoping down of the example and have been omitted: e.g., because they relate to PIN checks or cash dispersal. Otherwise, our test cases cover all of Bjork's plus one for the extra "no such account" error message.

Table 5 shows how Utting & Legeard's test cases [2] compare to ours. Note that most of their 31 test cases are outside the scope of the current study. In test 31 they test that the Cancel key is ignored between nodes 9 and 12; such test cases are not captured in the current approach, but would be if the BT model were augmented with behaviours that say explicitly that the system does not change state when certain events occur. Note also that they do not explicitly record the expected test results: the latter have to be inferred from the models. And note that in their model, hitting Cancel from the transaction option menu results in the card being rejected, which seems different from Bjork's requirement.

Table 4. Comparison with Bjork's test cases

Bjork test #	s1	s4	s5	s6	w1
BT test #	1	2;3;5	9	10	2
Bjork test #	w2	w3	w5	w6	w7
BT test #	3	5	6	4	8

Table 5. Comparison with Utting & Legeard's test cases

Utting test #	T10	T16	T22	T26
BT test #	1;2;3;5;10	1;2;4	1;2;3;6	1;2;3;7

We contend that the BT model captures the information important for testing that is contained in the combination of state machines and sequence diagrams used by Bjork, and the state machines, object diagrams and OCL post-conditions used by Utting. Moreover, it does so in a single, easy to understand model. Admittedly Utting's models also contain important data information that is not captured in the BT model above, such as the relationship between the withdrawal amount and the account balance before and after the withdrawal; we would have to use the relational extension of BTs [12] to handle this, but that tool support for that part of the BT method is not yet well-developed.

V. The Bearing & Range Line example

This section illustrates the approach on a problem adapted from a real-life example provided by Thales. The example has been stripped back to illustrate the treatment of BT constructs that were not present in the ATM example above, such as parallel branching, selections, guards and kill nodes. Traceability from system requirements to test cases is a requirement of DO-278A [13], the DO-178C-related software engineering standard for Air Traffic Management software such as this.

A. System description

The example concerns an interactive tool used by air-traffic controllers, called the Bearing & Range Line (BRL). A *track* is a moving object on the controller's display indicating an aircraft's current position. The controller selects a track and initiates the BRL tool, which then displays information about the position of a second object relative to the selected track; the object can be a fixed point, such as a waypoint or airport, or another track. The BRL consists of a *vector* from the track to the object and a *label* displaying the relative-position information. The type of information displayed on the label depends on the type of the object: namely, whether it is a point or a track. For the current study the type of the information is the important thing rather than the value displayed, which is the responsibility of other tools.

The BRL system requirements are:

1. The user initiates the BRL by moving the pointer over a track and hitting the BRL key.
2. When the pointer is moved away from the selected track a vector is drawn dynamically from the track to the object currently under the pointer. As the user moves the pointer, the BRL label changes depending on the type of object currently under the pointer.
3. The user can create a fixed BRL by moving the pointer to the desired end object and left clicking. An error message is displayed if the original track is selected as the end (since zero-length vectors are not allowed), and the BRL remains in dynamic mode. But otherwise the vector end becomes anchored to the selected object, and further movement of the pointer does not change it.
4. Hitting the Escape button deletes the BRL.

Figure 5 shows the resulting BT model, with nodes numbered for ease of reference; where a node has two tags (e.g., "3,1") the second number is a reference back to the requirement from which the subtree was derived. In this model the BRL is composed of two components: a *Vector* and a *Label*; both are initially *null*. A *Dummy* component has been introduced for convenience, to allow the user's control of the Pointer (the subtree below node 23) to be separated out from behaviour of the BRL itself.

Parallel branching occurs below nodes 2 and 8. The looping thread on the left, consisting of nodes 9-13, models how the type of the label changes as the pointer is moved while the BRL is in dynamic mode (requirement 2 above): *Label* starts in *track-point* mode once the pointer moves off the selected track and then changes to *track-track* mode if the pointer moves over a track. The nature of the display means that the pointer must move over a point immediately upon moving off the selected track. Nodes 9 and 11 are guards, meaning control waits at the node until the condition becomes true. This looping behaviour continues until the user left clicks on an object other than the initially selected track, causing the vector to become fixed (node 20); node 19 kills thread 9-13, so that the label-type remains fixed at its last value thereafter. Note that reversion at node 17 also kills the thread, but it restarts again automatically from node 8.

Table 6. Observable BRL system responses

Node	**Observable effect**
1	No vector displayed
8	One end of the vector follows the pointer around
10	The label is in track-point format
12	The label is in track-track format
16	Error displayed
20	Vector is fixed and pointer can be moved freely

The observable system responses are listed by node

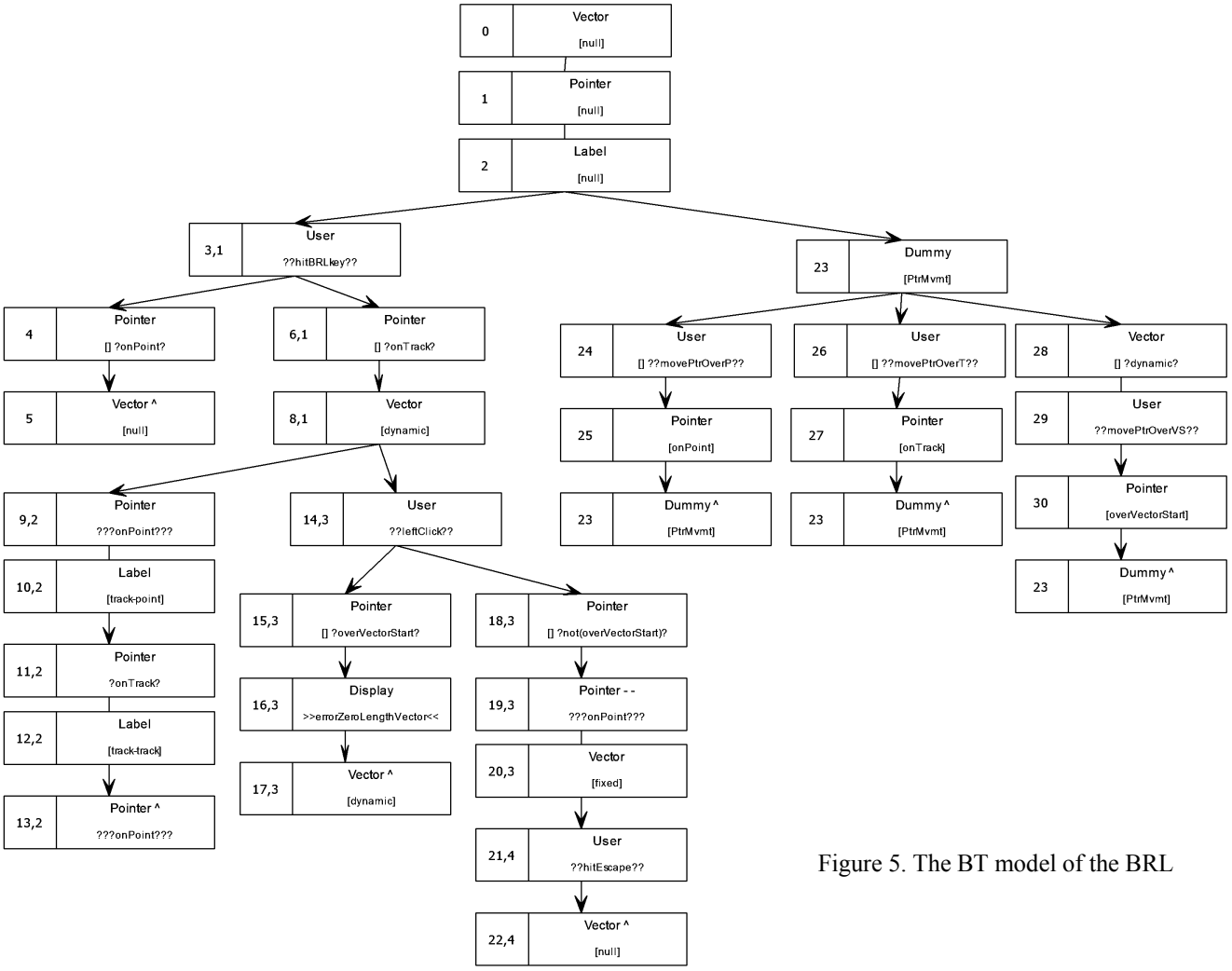

Figure 5. The BT model of the BRL

The observable system responses are listed by node number in Table 6 with their reverse translations. Step 3 is a straightforward reverse translation of user events: e.g. 'movePtrOverT' simply translates back to tester action "move the pointer over a track". All components are under the user's control in this example, so step 4 is not needed. The other two TCG preparation steps are discussed below.

B. Test case generation for BRL

The obvious CPs to choose for this example are the three *Vector* states (nodes 0, 8 and 20). The path for reaching node 8 from node 0 is 0-23-26-23-3-8 and from there node 20 is reached via 8-24-23-9-14-20. (As usual intermediate nodes are elided to save space.) From there the path 20-21-22 resets the system.

The process for path generation was explained in Section *II C*. Parallel branching increases the number of different interleavings that would need to be considered if simple tree traversal were used. Many user actions have no effect on the system state, such as repeatedly moving the pointer over different points in dynamic mode while the label is in the track-point format. But the model checker is able to eliminate many of the combinations as not being feasible, and to find the shortest path from each equivalence class of the feasible paths.

For testing purposes the key properties of the system are the format of the label and the production of error messages. For test case generation we thus nominate nodes 10, 12 and 16 as the NOI. The resulting paths are given in Table 7, with intermediate nodes and pointer movement events elided from paths, and equivalence classes indicated. Note that it is sufficient to traverse the looping thread 9-13 once, to test all of the functionality it captures; after that no further state combinations will be introduced no matter how many times the loop is iterated, as verified by the model checker. The last column in Table 7 traces test cases back to the requirements from Section V.A that gave rise to them.

The process for translating paths to test cases proceeds as before. As an example, the test case for path 7 in Table 7 is as follows: First bring the system into dynamic-vector mode by moving the pointer over a track and hitting the BRL key (path 1-3-6-8). Then move the pointer over a point (enabling node 9); the label should be in track-point mode. Then move the pointer away from the vector start position (the action enabling selection node 18) and left click; the vector should be in fixed mode. Finally, to reset the system, hit Escape (the user action for path 20-22).

Table 7. Test case paths for the BRL example

#	Path	NOI	Reqs
1	1-3-5(1)	None	1
2	1-3-8	None	1
3	8-14-15-17(8)	16	3
4	8-9-14-15-17(8)	10, 16	2,3
5	8-9-11-14-15-17(8)	10, 12, 16	2,3
6	8-14-18-20	None	3
7	8-9-14-18-20	10	2,3
8	8-9-11-14-18-20	10, 12	2,3
9	20-22(1)	None	4

VI. Discussion and Future Work

The examples above have not illustrated the synchronisation and internal messaging BT constructs, but the BT Analyser handles them in test path generation, and the test planner does not need to provide any extra information for them. The BT Analyser also handles parameterised components, sets and quantifiers [12], and we have added a construct for conjunction of conditions involving parametrised conditions and applied the approach successfully to systems using these constructs. Generally speaking though, the BT notation and symbolic model checking are good for describing and analysing system behaviour but poor for data aspects; that limitation extends to test case generation.

The ATM and BRL examples are relatively small examples. We have applied the BT Analyser on bigger examples to assess its scalability. The biggest example tried is called SSM (for Sensor System Monitor). The SSM with 9 sensors has 357 BT nodes with 53 of them being CP nodes. It took almost 6 hours for BT Analyser to find 1126 test paths for the 9-sensor SSM example with no NOI. A laptop with a 2.7GHz i7 CPU and 16GB of RAM was used and BT Analyser was run with a virtual memory limit of 8GB.

There are examples of BTs smaller than the 9-sensor SSM example that require more resources, but the SSM example shows that BT Analyser can be applied to industry-sized examples. In as much as the BT model captures all of the desirable behaviours of a system, and only the desirable behaviours, the structure of the BT model should not make a difference to the results of the TCG process; it can however have an effect on the efficiency of the model checker. The SSM example is interesting in that the way the BT model is structured appears to mitigate the state explosion problem in model checking; how exactly it does this needs further investigation.

Judicious selection of NOI can significantly reduce the number of test cases in systems with multi-threaded behaviour. The BT Analyser provides other useful checks, such as that all nodes in the BT model are reachable. One of our students has developed a simple user interface to support the test case generation process, including test planner selections and translation back to natural-language test case descriptions.

Future work includes providing further support for the test planner by providing more information about path preconditions, and helping them plan test campaigns if they provide a BT model of the test environment. We also plan to evaluate how well the approach supports requirements change management on a large industry case study. Developing any formal model for a large complex system can be costly and time-consuming; a promising alternative approach might be to piggyback on the Engineering Change Proposal process, and develop models incrementally as needed to support change analysis and testing.

VII. Related work

This section describes related work on test case generation from state-based modelling notations.

Wendland *et al.* [14] propose that BT models provide an ideal basis for test planning, but they generate test cases by hand. They point out that if the system does not change in response to an external event, the event is simply left out of a typical BT model. In system-level testing, however, it is necessary to check that the system really does not respond to such an event, so they explicitly augment BT models with null behaviours. Our approach applies equally well to this form of model, since it is simply another form of BT model.

Hakimipour and Strooper [15] propose a BT-based TCG approach that generates a test case for each functional requirement represented by a single branch behaviour tree. They generate further test scenarios that cover multiple functional requirements. A generated test case specifies a set of user actions to reach a system state. The approach does not consider loops. Moreover, it does not consider discriminating between observable behaviour and internal system behaviours. The format of the generated test cases does not reflect the expect outcome(s). Salem and Hassan [16] introduce a TCG and prioritization technique based on BTs. The test case generator produces a test case for every single transition between system states, which may result in a large number of test cases. Moreover, it does not trace the generated test cases back to requirements, unlike here.

The traditional approach of using model checking for test case generation (e.g., [17]) is to specify a property, often called "trap property", and a counterexample to the property becomes a test case. In contrast, our approach searches for test paths directly using lower level functionalities of the model checker. It is unlikely that we would use full LTL model checking.

Our approach is also different from the finite state automata (FSA) approach (e.g., [18]). The FSA approach requires that the automaton be deterministic. A non-deterministic automaton would need to be transformed into a deterministic automaton, which may cause an exponential blowout, since in general a non-deterministic automaton with n states may need to be transformed to a deterministic automaton with 2^n states. In contrast, non-determinism is handled directly by the underlying model checking framework in our approach. However, the state explosion problem remains.

Nebut *et al.* [19] propose a TCG approach based on use cases. They extend the use case notations with contracts that can capture use case pre- and post-conditions. The revised use case specification is used to build a state transition machine that reflects dependencies between use cases. This transition machine is used to identify all possible orderings of use cases. Each sequence of use cases (path) is considered

as a test objective. Given that use cases describe system behaviour at an abstract level, the use cases in the transition machine are replaced with sequence diagrams to reflect more refined behaviour. The approach can generate all possible scenarios based on the pre- and post-conditions. However, the approach suffers from scalability problems because it requires exponential runtime. Moreover, it is tightly coupled with the object-oriented analysis and design paradigm.

Lee and Friedman [20] propose a requirements-based TCG approach using a cause(input)-effect(output) model as a requirements model capturing input-output requirements. This model is manually transformed into two models: a reference model, which covers inputs and system behaviour (modelled using MathWorks StateFlow), and a test oracle, which covers expected outputs. The StateFlow Simulink Design Verifier generates test cases that provide full coverage of the system behaviour. These test cases are then linked to the test oracle.

VIII. Summary

We have introduced a new requirements-based testing technique based on BT models. We contend that the structure of BT models lends itself better to this process than other approaches, which typically use combinations of state machines and sequence diagrams. The BT model forms the basis for the test case generation process and is used as the test oracle. The test planner selects: which system states will act as test case start and end states; which system states and/or external outputs will be observable; and which events or states will define the equivalence relation on test cases. A tool is then used to generate system test cases by finding representatives of all possible execution paths between Check Points in the BT model. Note that the output is a set of natural language test cases with tracing back to the individual requirements that gave rise to them: the tester does not need to have any knowledge of formal methods, and does not even need to see the BT model. We have illustrated our approach on two case studies: an Automated Teller Machine and an example from the Air Traffic Management domain.

The process leads to a rigorous set of test cases, confirmed by model checking. The combination of BT modelling and automated test case generation yields full two-way traceability from requirements to test cases. This means that when individual requirements change, it is a simple matter to identify which tests need to change and how. This in turn is expected to accelerate and improve the quality of requirements change management.

Acknowledgements: This work was supported by grant LP130100201 from the Australia Research Council and Thales Australia. The BRL case study was provided by Dean Kuo.

References

[1] RTCA, "DO-178C: Software Considerations in Airborne Systems and Equipment Certification (aka EUROCAE ED-12C)," Radio Technical Commission for Aeronautics, 2012.

[2] M. Utting and B. Legeard, *Practical model-based testing: a tools approach*. San Francisco, CA: Morgan Kaufmann Publishers, 2007.

[3] R. G. Dromey, "From requirements to design: Formalizing the key steps," in *1st Int. Conf. on Software Engineering and Formal Methods (SEFM)*, 2003, pp. 2-11.

[4] D. Powell, "Behavior engineering-a scalable modeling and analysis method," in *8th IEEE Int. Conf. on Software Engineering and Formal Methods (SEFM)* 2010, pp. 31-40.

[5] R. Glass, "Is this a revolutionary idea, or not?," *Software,* vol. 47, pp. 23-25, 2004.

[6] L. Grunske, P. Lindsay, and K. Winter, "An automated failure mode and effect analysis based on high-level design specification with Behavior Trees," in *Integrated Formal Methods (IFM),* LNCS 3771: Springer, 2005, pp. 129-149.

[7] P. A. Lindsay, "Behavior Trees: From Systems Engineering to Software Engineering," in *8th IEEE Int. Conf. on Software Engineering and Formal Methods (SEFM)*, 2010, pp. 21-30.

[8] R. Bjork. (2004). Automated Teller Machine example. [Online]. http://www.math-cs.gordon.edu/courses/cs211/ATMExample/ accessed 21 Aug 2015.

[9] S. Kromodimoeljo, "Controlling the Generation of Multiple Counterexamples in LTL Model Checking," PhD thesis, University of Queensland, 2014.

[10] The Behavior Tree Group. (2007). Behavior Tree Notation v1.0. [Online] http://www.itee.uq.edu.au/sse/dccs. accessed 21 Aug 2015

[11] R. J. Colvin and I. J. Hayes, "A semantics for Behavior Trees using CSP with specification commands," *Science of Computer Programming,* vol. 76, pp. 891-914, 2011.

[12] K. Winter, R. Colvin, and R. G. Dromey, "Dynamic relational behaviour for large-scale systems," in *Australian Software Engineering Conference (ASWEC)*, 2009, pp. 173-182.

[13] RTCA, "DO-278A: Software Integrity Assurance Considerations for Communication, Navigation, Surveillance and Air Traffic Management (CNS/ATM) Systems (aka EUROCAE ED-109A)," Radio Technical Commission for Aeronautics, 2011.

[14] M.-F. Wendland, I. Schieferdecker, and A. Vouffo-Feudjio, "Requirements-driven testing with Behavior Trees," in *IEEE 4th Int. Conf. on Software Testing, Verification and Validation Workshops (ICSTW)*, 2011, pp. 501-510.

[15] N. Hakimipour and P. Strooper, "Exploring an Approach to Model-Based Testing from Behavior Trees," in *SATA Worskhop, Proc 19th Asia-Pacific Software Engineering Conference*, IEEE, 2012, pp. 80-86.

[16] Y. I. Salem and R. Hassan, "Requirement-based test case generation and prioritization," in *2010 International Computer Engineering Conference*, 2010, pp. 152-157.

[17] A. Gargantini and C. Heitmeyer, "Using model checking to generate tests from requirements specifications," in *Proc 7th European Software Engineering Conference*, LNCS 1687: Springer, 1999, pp. 146-162.

[18] S. Fujiwara, G. v. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi, "Test selection based on finite state models," *IEEE Transactions on Software Engineering,* vol. 17, pp. 591-603, 1991.

[19] C. Nebut, F. Fleurey, Y. Traon, and J. Je´ze´quel, "Automatic Test Generation : A Use Case Driven Approach," *IEEE Transactions on Software Engineering,* vol. 32, pp. 140–155, 2006.

[20] C. C. Lee and J. Friedman, " Requirements modeling and automated requirements-based test generation," *SAE International Journal of Aerospace,* vol. 6, pp. 607-615, 2013.