BT-TCG User Guide

Peter A. Lindsay & Sentot Kromodimoeljo
School of IT&EE, The University of Queensland,
Brisbane, Qld 4072, Australia
(p.lindsay,s.kromodimoeljo)@uq.edu.au

July 28, 2016

Abstract

This is a guide to BT-TCG: a tool supporting test case generation (TCG) from models written in the Behavior Tree (BT) modelling notation.

Overview

The Behavior Tree (BT) notation and method were introduced by Geoff Dromey as a means for capturing functional requirements of a multi-threaded system in a single graphical model [3]. [1] The notation has been found to be easier to understand and trace back to natural language requirements than modelling notations such as UML [4, 12]. The notation has a formal semantics [2], which has enabled development of a range of support tools such as animators and model checkers [5].

A new type of symbolic model checker, with the capability to generate multiple counterexamples [7], has recently been adapted to generate test cases from BT models. The result is called *BT-TCG*. Because the model checker understands the semantics of the BT notation, the test cases it generates are guaranteed to be feasible and complete with respect to the model (in a sense defined below), as well as the shortest possible.

In overview, BT-TCG offers the following new capabilities:

1. generation of test cases for manual testing, as described in [10]
2. generation of test cases for automated testing of web sites and Android applications

[1] See `en.wikipedia.org/wiki/Behavior_tree` for a short history. The AI community has more recently introduced a semi-formal notation also called Behavior Trees, but their approach is very different to Dromey's.

3. model checking support, including verification of properties stated in LTL

4. support for test planning, whereby test cases are strung together for efficiency, instead of resetting the system between tests

5. generation of documentation, such as user guides

6. input of BT models developed in BESE

This guide compiles information from a number of different sources, for people interested in using BT-TCG. The reader is assumed to have some familiarity with BTs, for example by having studied [3]. §1 has a summary of the BT notation. §2 has three examples: the corresponding BT models and results are included in the tool release. §3 has a brief summary of the model checker and its use for verifying properties of BT models. §4 describes different approaches to test case generation supported by BT-TCG.

Acronyms

BT	Behavior Tree
CP	Check Point
FTP	Full Test Plan (§4.2)
LTL	Linear Temporal Logic
LTP	Long Test Plan (§4.5)
NOI	Node(s) of Interest
SUT	System Under Test
TCG	test case generation
TCP	test case path (§4.2)

Contents

1 The BT notation and method

1.1 Overview and tools

The BT notation has evolved over time: BT-TCG uses the BT syntax standardised in [13] and the semantics of Colvin and Hayes [2].

In overview: A BT model (or Behavior Tree for short) is a directed tree consisting of different types of nodes and two types of branching. Edges in the tree represent control flow, starting from the root. Leaf nodes can contain operators called *flags* that indicate how control flows from that point; if there is no flag, the thread simply stops at the leaf. See §2 below for examples. Extensions of the notation have been developed to support parameterisation, in the form of multiple components with the same behaviour: see §1.7 below.

A number of different tools are available for editing BT models, including:

ComBE and TextBE: [2] Eclipse plug-ins supporting a concise textual syntax, with conversion to graphical format. This is the main input format (`.btc`) supported by BT-TCG. ComBE is preferred because it supports commenting, and links nodes in the graphical format back to the textual format, although its support for tags is limited. Currently ComBE only works for the L (Lunar) version of Eclipse. BT-TCG has only been tested on the M (Mars) version of Eclipse. The automatic tree layout is good for small models, but for large models it sometimes rearranges the order of branches or even messes up display of the tree altogether.

BESE: [3] A commercial tool maintained by Dendronix. Free licences are available for research purposes (only). BT-TCG accepts BESE models as input, but ignores relationships ("what, where, why, etc") and several other BESE elements.

Integrare: [4] A stand-alone graphical editing tool developed by Larry Wen. Tree layout is under user control, with generally better looking results than ComBE, which makes it good for preparing models for publication. [5]

1.2 Node graphical syntax

Fig. 1 shows an example BT node as displayed in graphical form. The different elements of the node are explained in Table 1.

[2] `http://bt-tools.github.io/ComBE/`

[3] Contact Dan Powell (dpowell72@me.com) for details

[4] `www.beworld.org/BE/home/be-resources/#tools`

[5] Here's one way of generating B&W trees from BT editing tools: copy and paste the tree into Photoshop or similar, set the lightness to 50%, then apply greyscale, and save as png.

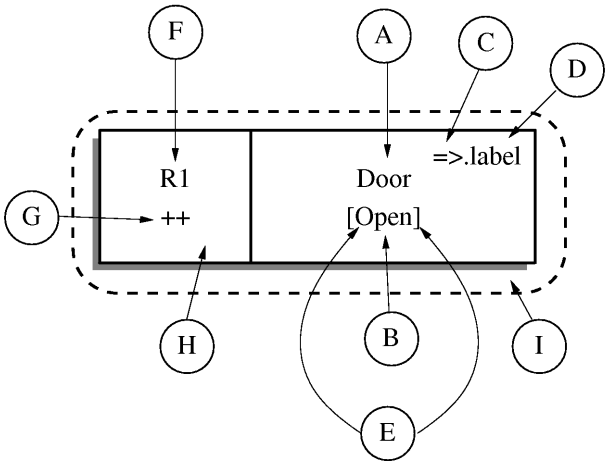

Figure 1: Example BT node

Nodes can be combined atomically – depicted as being joined by an edge without an arrowhead – meaning they effectively get executed at the same time, without intervening steps of other threads.[6]

A *node profile* consists of a component name (A) and behaviour (B and E). Two nodes are said to *match* if they share the same node profile.

1.3 Node behaviour types

The different behaviour types are as follows:[7]

State realisation (#S) '`C[S]`': component `C` is currently in, or transitions into, state `S`. State realisations can include assignment of values to component attributes (written '_ := _').

Selection (#L) '`C?P?`': control passes this point only if the condition `P` is satisfied; otherwise the thread dies. `P` can be a state (in which case the condition is that `C` is in the named state), or that an attribute has a particular value, or the negation of one of these. '`C?else?`' stands for the disjunction of the negations of the sibling nodes.

Event (#E) '`C??e??`': control waits at this point until event `e` occurs.

Guard (#G) '`C???P???`': control waits until condition `P` is satisfied.

Internal input (#II) '`C>m<`': `C` receives message `m` from another component.

Internal output (#IO) '`C<m>`': `C` broadcasts message `m` to other components.

[6]ComBE uses ';' for sequential composition (steps) and ';;' for atomic composition.

[7]The letters in parentheses are the ComBE identifier for the behaviour type.

Label	Name	Description
A	Component Name	Specifies the component: can be a system component or an external system or agent, such as a user
B	Behavior Name	In combination with the behaviour type, specifies the behaviour associated with the component, such as what state it transitions to or what condition it must satisfy
C	Flag (aka Operator)	(optional) Flags modify the control flow: see §1.4 below for details.
D	Label	An optional label for disambiguating nodes, in case the tree contains multiple matching nodes
E	Behavior Type	Delimiters on the behaviour indicate the type of behaviour involved
F	Traceability Link	In the BT method: a reference to the requirements that give rise to this node. Often however it is used simply for referencing nodes.
G	Traceability Status	In the BT method, indicates the status of the node with respect to the requirements document, such as whether it is explicitly mentioned, or implied, or missing: see [13] for details. BT-TCG has a different use for this: see §4.6 below for details.
H	Tag	The box on the left-hand side of the node (by default, contains traceability information, but may be used differently, or omitted, in different contexts): see §1.5 below for details.
I	Behavior Tree Node	The main types of BT nodes are explained in this section, but quantifier nodes are also possible: see §1.7 below.

Table 1: Elements of a BT node

External input (#EI) '`C>>m<<`': `C` receives message `m` from an external system.

External output (#EO) '`C<<m>>`': `C` sends message `m` to an external system.

1.4 Control flow and flags

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 ("*runs*") through the tree. Control can branch in two ways in BT models: parallel branching (#P in ComBE), corresponding to forking of different threads of behaviour; and non-deterministic branching (#N in ComBE), when different outcomes are possible. Graph-

ically, non-deterministic branching is indicated by '[]' in the child nodes in BESE and Integrare, and in the parent node in ComBE and TextBE; if there is no indication, parallel branching is assumed.

In a particular execution, control flows from the root of the tree down edges in steps from parent node to child. 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, as separate, interleaved threads.

When execution reaches a leaf node the following rules apply, according to what flag, if any, occurs:

reversion '^': control passes to the closest matching ancestor of the leaf node, and any parallel threads that were forked below that ancestor node get killed.

reference '=>': control passes to a matching node without a flag; such a node must be unique in the tree

There are two other kinds of flag:

kill '- -': the thread starting from the closest matching sibling of an ancestor node gets killed.

synchronisation '=': execution waits at the node until all threads with a matching node and synchronisation flag reach that node; thereafter execution continues as normal.

1.5 Tags and traceability

A BT model is typically developed from a set of natural-language functional requirements for a system by translating each requirement into a BT segment then integrating the segments together [3]. The integration process has been shown to be particularly effective at identifying inconsistencies and incompleteness in requirements specifications [11]. Conversely, the BT model can be reverse translated (by hand only at this stage) into a clear set of natural-language functional requirements.

In the BT method, *tags* are used for tracing back to the natural-language requirement(s) that gave rise to the segment(s) containing the node. In BT-TCG they are used to in a more general way, to refer to particular instances of nodes in the BT model. To avoid confusion (e.g., when choosing NOI[8]), the modeller should try to avoid using the same tag for matching nodes.

The traceability status indicator changes the colour of the node in the graphical view. Table 2 has a short summary.

[8]Nodes of Interest (NOI) are explained in §4.3 below.

Symbol	colour	Description
none	green	node explicitly mentioned in corresponding requirement
+	yellow	node implicitly described in corresponding requirement
-	red	node missing from corresponding requirement
++	blue	node added during development
+-	none	node of indeterminate status

Table 2: Traceability status types

1.6 Paths

A *path* is a sequence of BT nodes capturing a particular partial run of the BT model. In very simple models, paths are contiguous, but in models involving parallelism, a path is an interleaving of steps from separate threads in the model.

A BT model is intended to represent all possible executions – and only possible executions – of the system being modelled. In the standard "prioritised" semantics of BT execution, steps involving external inputs and events get delayed until all other enabled steps have been executed [5]; this ensures that all internal behaviours are fully executed between occurrences of external events.[9] States of the underlying Kripke model in which none of the internal transitions are enabled are called *stalled states*.

1.7 Quantifiers and parameters

Zafar et al extended the BT notation with "quantifiers" for parameterised parallel and alternative branching [15]. FOR_ALL (parallel branching) is depicted as '$\|x : X$' and FOR_ONE (alternative branching) as '$[]x : X$'.[10] The scope of x is through the subtree below the node: see Fig. 2.

In the course of industry applications, we discovered that a parameterised AND construct was also needed, to capture conditions that apply across all of the parameterised components simultaneously. (Disjunctions can already be handled by the FOR_ONE quantifier.) The new construct is written '$\|\& x : X$', with a list of nodes connected to it by atomic composition (see Fig. 2). The scope of x is through the nodes joined atomically to the AND node.

The operational semantics of these constructs can be explained as follows: In a pre-processing step, BT Analyser converts the parameterised model into an ordinary BT model, starting from the deepest occurrences of quantifiers and working upwards to the BT root. It creates a separate copy of each subtree below the quantifier, instantiated with each value from X, and joined

[9] The non-prioritised semantics is typically used only when race conditions are being investigated.

[10] BESE and Integrare have constructs for quantifiers, but in COMBE and TextBE we use the assertion (#A) operator.

below a dummy node using the appropriate branching construct. BT Analyser processes the AND construct by creating an atomic combination of copies of the nodes, with x replaced by each specific value from X.

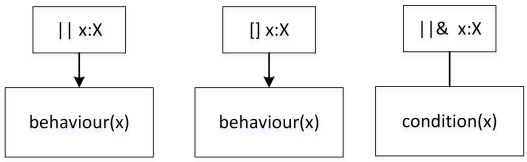

Figure 2: FOR_ALL tree, FOR_ONE tree, AND node

Typically, the AND construct is applied to nodes which are all state realisations, or all selections, or all guards, but mixtures of these nodes types are allowed where permitted by the BT semantics [2]. An AND node can be joined to other nodes by atomic or sequential composition.

1.8 Comparison with other modelling notations

The BT notation is a rich system modelling notation that can capture the logic of how components change state in response to events and interaction between themselves. One of the main strengths of the BT notation is that functionality is localised: typically, individual natural-language functional requirements map to contiguous segments of the model, unlike notations such as State Diagrams, Activity Diagrams and Petri Nets, where functions get spread across different components [9]. The closest UML modelling approach is Sequence Diagrams, but unlike the latter, all behaviour is captured in a single BT model. Functional Flow Block Diagrams, which are widely used in systems engineering, are similar in shape to BT models (in that they capture flow of functions), but do not explicitly capture state.

[6] describes a translation from BT models into UML Statecharts.

2 BT examples

This section gives 3 example BT models, illustrating different aspects of the BT notation. For legibility, black and white versions of the Integrare models are given here.

2.1 ATM example

The first example is a simple Automatic Teller Machine (ATM), based on the description supplied by Russell Bjork [1]. This is the same example that Utting and Legeard [14] use to illustrate the approach to model-based testing using UML.

A BT model of the ATM is given in Fig. 3. See [10] for more explanation of how the model relates to Bjork's and Utting's versions. It involves simple sequential execution, without parallelism apart from the dummy branch for the 'transfer' transaction. The `Coms` component links the ATM with the bank-headquarters accounts database, and is treated here as an external system. The ATM's (Card) Reader, Console, Display and Printer are also treated as external systems for the purposes of illustration.

2.2 BRL example

The second example (Fig. 4) is the Bearing and Range Line (BRL) system from [10]. It illustrates the use of parallel branching, selections, guards and kill nodes.

2.3 SSM example

The third example (Fig. 5) is a Sensor System Monitor (SSM), based on a safety-watchdog subsystem of a large distributed system. It illustrates the use of quantifiers and parameterisation. We go into it in more detail here, since the example hasn't been published yet.

FOR_ALL quantifiers are used at nodes 11 and 40, to model the sensor-related behaviour: receiving a valid signal from one of the sensors, and a sensor's timer timing out, respectively. The AND construct is used in three places: to stop all sensor times (node 1), restart all sensor times (node 4), and check if all sensor timers have stopped (node 44).

SSM interacts with the following subsystems: an indeterminate number of external sensors that send signals to the SSM; a filter that identifies the sensor that has sent the signal and determines if it is valid for that sensor; a timer for each sensor s, that keeps track of when the last valid signal was received from s, and times out after a pre-determined period; a self-test that sends an "I'm alive" signal from the platform on which the SSM is running; and a self-test timer TimerST that keeps track of when the last valid self-test signal or valid sensor signal was received, and times out after a pre-determined period ST-timeout. (Note the lack of symmetry in the above: receiving a valid sensor signal results in TimerST also getting reset, but not the reverse.) The operator can stop SSM or restart it any time after it has been started.

The SSM has four main states as follows: *Stopped:* the SSM is not operational; *Unknown:* the SSM has recently (re)started and is waiting to receive a signal from a sensor or from the self-test; *Ok:* the SSM has received a valid signal from at least one sensor since its timer was last reset, and a self-test signal within ST-timeout; *Failed:* TimerST and/or all of the sensor timers have timed out.

Formally, the following temporal logic formulae characterise the two opera-

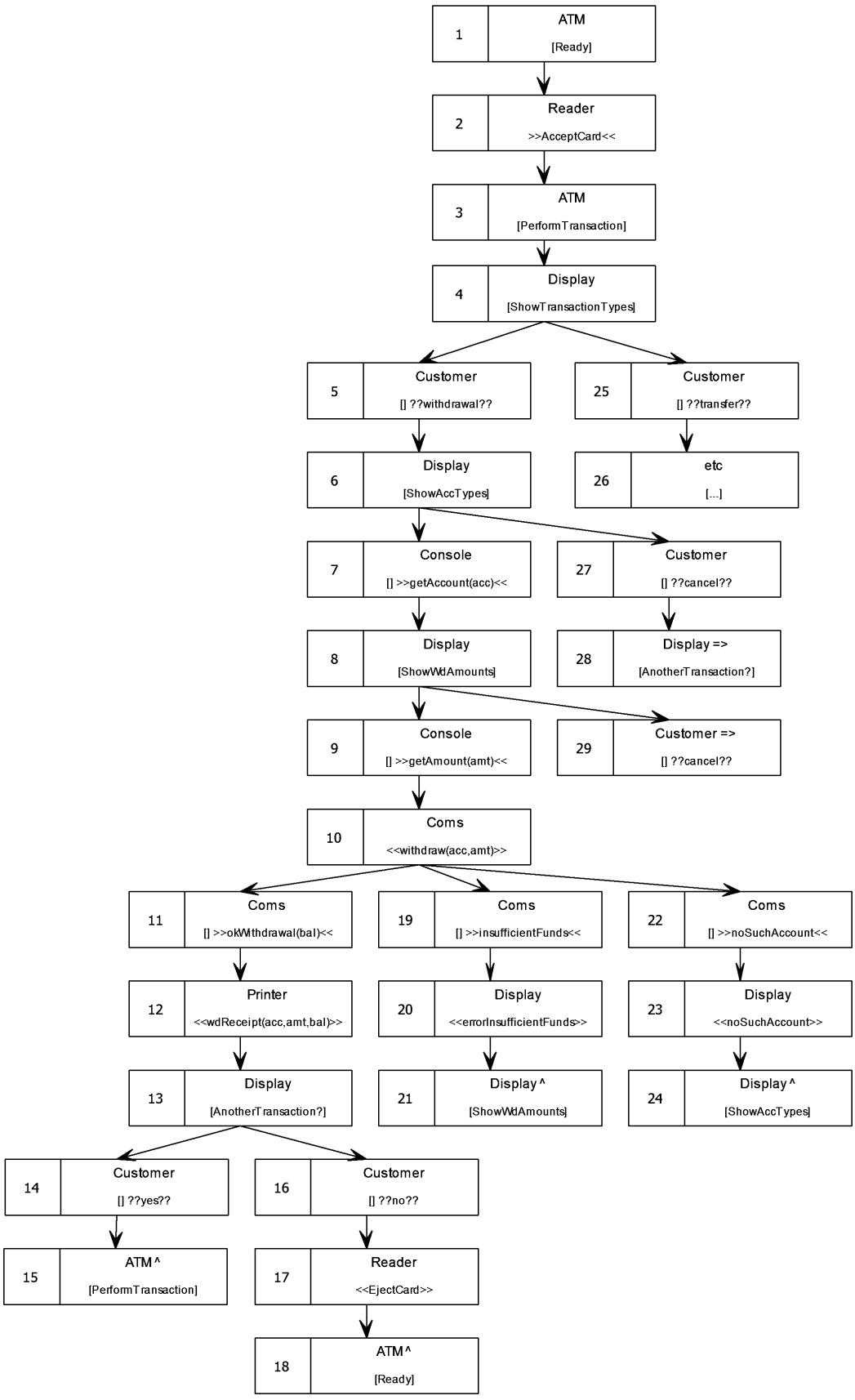

Figure 3: BT model of the ATM – version 150508

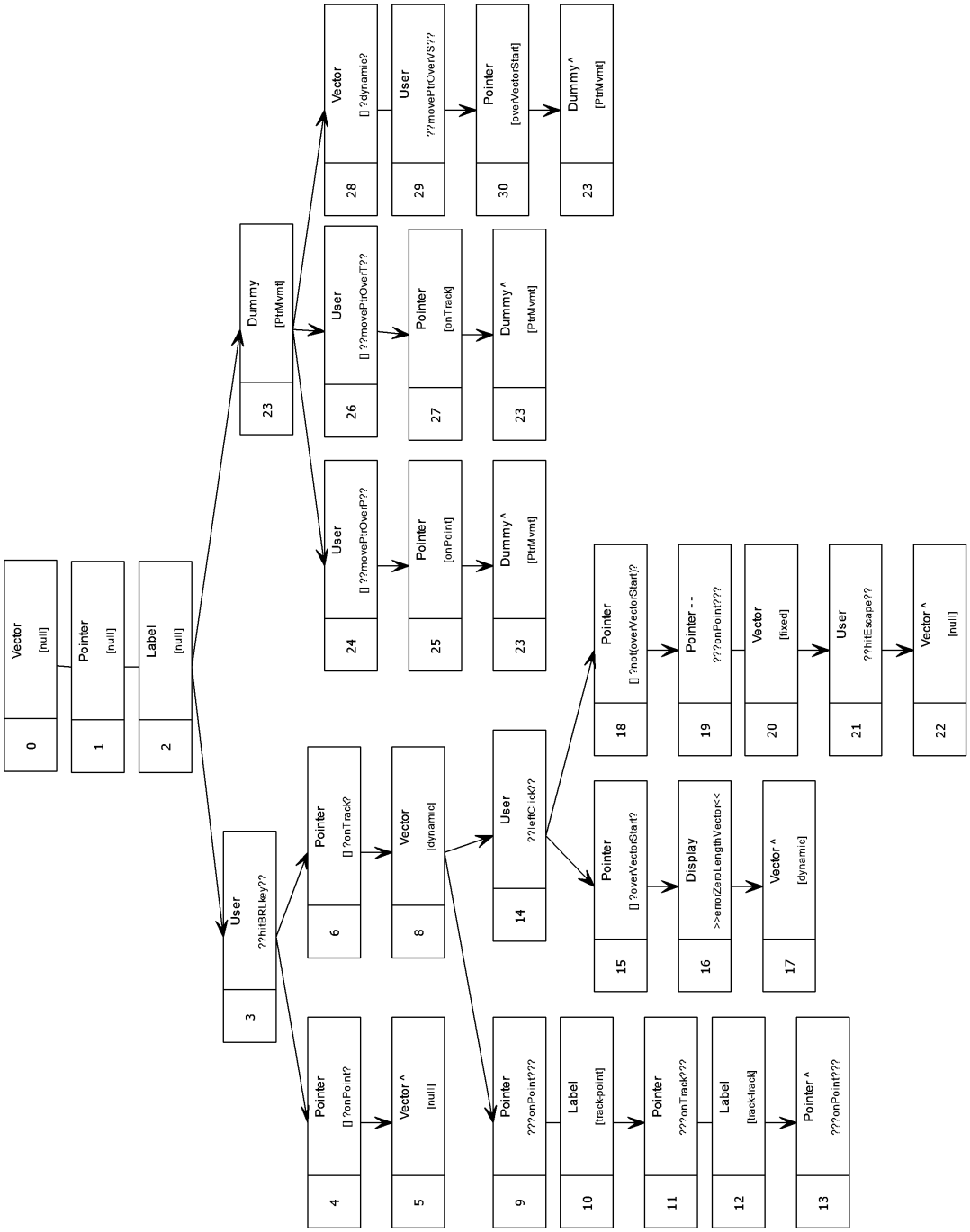

Figure 4: BT model of the BRL – version 150515

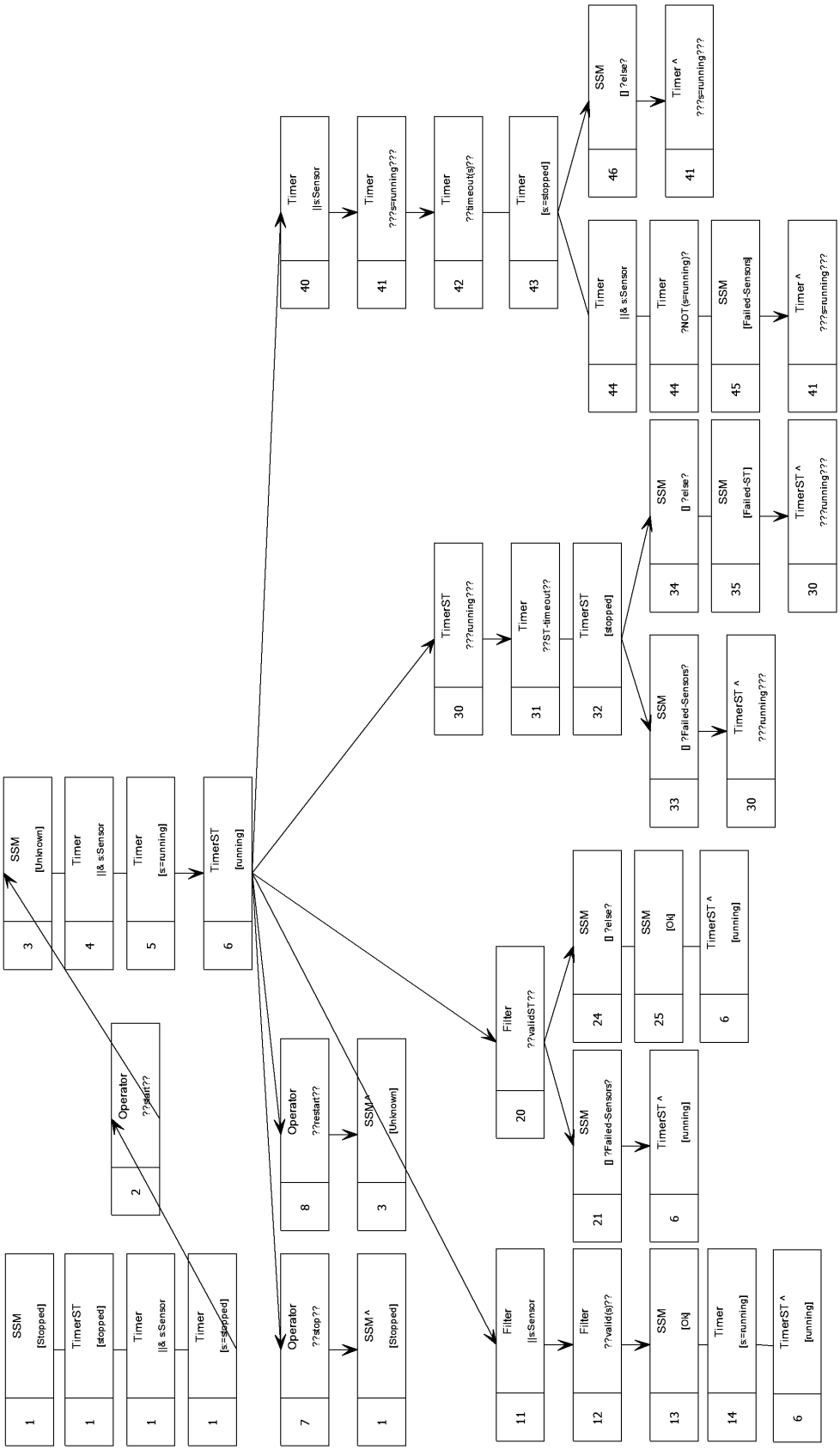

Figure 5: Parameterised BT model of the SSM – version 160429p

tional states of the system:

$$(SSM = Stopped \vee SSM = Failed) \Leftrightarrow TimedOut$$
$$Restart \vdash \mathbf{X}(SSM = Unknown \mathbf{W}(ValidSignal \vee TimedOut))$$
$$ValidSensorSignal \vdash \mathbf{X}(SSM = Ok \mathbf{W}(Restart \vee TimedOut))$$
$$ValidSTSignal \wedge \neg SensorsTimedOut \vdash \mathbf{X}(SSM = Ok \mathbf{W}(Restart \vee TimedOut))$$

where *Restart* represents the operator start or restart of the system, *ValidSignal* the reception of any valid signal, and *TimedOut* the timeout of TimerST and/or all of the sensor timers, etc. ($\mathbf{X}$ is the next temporal operator and $\mathbf{W}$ is weak until. $A \vdash B$ stands for $\mathbf{G}(A \Rightarrow \mathbf{X}B)$, where $\mathbf{G}$ is the always operator.) $\mathbf{X}$ is needed to allow state realisation to take place after an event has occurred.

3 BT Analyser

BT Analyser [7] is a symbolic Linear Temporal Logic (LTL) model checker in which models can be specified using the BT notation. It was designed to support incremental analysis of BT models for the purpose of generating multiple counterexamples; by default it uses the prioritised BT semantics. As well as the traditional fix-point approach to symbolic LTL model checking, on-the-fly symbolic LTL model checking can be performed using BT Analyser. BT Analyser has been used in safety analysis for automatic generation of minimal cut sets [8].

The `readme` file in the BTAnalyser folder has details of how to install and compile BT Analyser, and the encoding of BT nodes into LTL. The `api` file describes the functions provided by BT Analyser. BT Analyser runs in a variety of different Lisp environments, but BT-TCG requires SBCL.

3.1 Background

An example of using the incremental analysis capability of BT Analyser for the purpose of generating multiple counterexamples is as follows:

1. Tell BT Analyser to compute (using fixpoint computation) the set F_φ of states that are fair with respect to an LTL formula φ. The states in F_φ can potentially be involved in counterexamples[11] for $\neg \varphi$ (see [8] for the fixpoint formulation of F_φ). The intersection of F_φ with the set I of initial states and the set S_φ of states committed to satisfying φ, determines whether or not there are counterexamples.

2. If the intersection is not empty, tell BT Analyser to produce a counterexample path.

[11] a counterexample for $\neg \varphi$ is a path that satisfies φ.

3. Based on an analysis of the counterexamples produced so far, formulate a global constraint gc that rules out the counterexamples already produced, and tell BT Analyser to produce a counterexample path in which each of the states satisfies gc.

Step 3 can be repeated until there are no more counterexamples. Often it is advantageous to tell BT Analyser to first compute general reachability and have subsequent analysis be performed on reachable states (especially if there are not too many interleavings in the BT model).

The generation of multiple counterexamples as described above can sometimes be automated. An automation for generating *minimal cut sets* for safety analysis is described in [8]. The automation uses a feature of the BT Analyser that allows a cycle constraint cc to be specified independent of gc in generating counterexample paths. The cycle constraint cc must be satisfied by every state in the cycle part of a counterexample path (where a counterexample path consists of a finite prefix followed by a finite cycle that is repeated forever). Both global constraints and cycle constraints are used in [8].

The generation of multiple counterexamples as described above is often preferable than traditional approaches of generating multiple counterexamples. For example, traditional model checkers that generate multiple counterexamples tend to generate too many counterexamples that are slight variations of each other, while with the above approach, one can control the classes of counterexamples and only generate one counterexample path from each class. For model checkers without multiple counterexample generation capability, modifying the temporal formula or the model and rerunning the model checker to find additional counterexamples is both time consuming and error-prone.

The design of BT Analyser enables it to be extended to provide very efficient special-purpose tools. In some cases, disciplined access to the mid-level functionalities of BT Analyser is required. The main mid-level functions are the *image* and *pre-image* (under the transition relation) functions for elementary blocks, where an elementary block corresponds to a BT node or an atomic composition of BT nodes. In a symbolic framework, these functions are predicate transformers.

3.2 Checking LTL properties

The temporal logic properties from §2.3 have been verified using the BT Analyser model checker, using the following definitions:

$$
\begin{aligned}
&SSM = Failed \ ::= \ SSM = Failed\text{-}ST \vee SSM = Failed\text{-}Sensors \\
&SensorsTimedOut \ ::= \ \textstyle\bigwedge_{s:Sensors} Timer.s = stopped \\
&TimedOut \ ::= \ TimerST = stopped \vee SensorsTimedOut \\
&Restart \ ::= \ Operator??start?? \vee Operator??restart?? \\
&ValidSensorSignal \ ::= \ \textstyle\bigvee_{s:Sensors} Filter??valid(s)?? \\
&ValidSTSignal \ ::= \ Filter??validST??
\end{aligned}
$$

where $C??e??$ is the assertion that the event `C??e??` is enabled (i.e, all enabled internal steps have already been taken). The first of the properties was only shown to hold in stalled states of the underlying Kripke model, since it may fail for intermediate states such as before component state changes have been realised.

4 Test case generation

In overview, BT-TCG provides mechanisms for generating test cases as paths through the BT model. Events and external inputs from the BT model become test actions, and the BT model acts as the oracle.

BT-TCG supports many different approaches to test case generation (TCG). §4.1–4.3 describe TCG for manual testing, as introduced in [10]. §4.4 discusses the extension to automated testing and §4.5 discusses stringing together test cases (instead of resetting the system between each test case). §4.6 discusses generation of user documentation from BT models – something that might not normally be regarded as testing, but can be very useful none-the-less.

The `quick-start` file in the BT-TCG release root folder has instructions for how to install and run BT-TCG. The `add-on-guide` file is a guide for researchers and students planning to extend BT-TCG.

4.1 Configuring BT-TCG for manual testing

In order to generate test cases for manual (human-in-the-loop) testing, 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 or BT model. This step is called *test case configuration*.

To configure BT-TCG, after reading in the BT model the test planner needs to perform five tasks:

1. Choose which component states are going to represent the system states from which the testing part of test cases start and end. The corresponding node profiles are called the *Check Points (CPs)* in what follows.

2. Nominate a particular CP node as the *initial system state*, from which test cases begin and end: the node cannot be a reversion or reference leaf node. BT-TCG selects the earliest occurrence of a CP-matching node as the default initial system state, but the user can override this.

3. Choose which specific component states and/or outputs will be observable to the tester as system responses, and describe the expected observations in words.[12] The description may need to include instructions such as how long to wait before the expected response will occur.

[12]Not all such nodes are observable. For example, when performing acceptance testing of

4. For events in the model and external inputs from components that will be under the testers control, describe the corresponding tester action in words.

5. For external inputs from components not under the testers control, describe conditions that will stimulate the desired input.[13] BT-TCG provides a tick-box for the test planner to indicate which external inputs will not be under the testers control.

If the BT model was developed from a requirements document, the natural-language descriptions are simply reverse-translations of the nodes in question: the words can usually be taken directly from the original requirements. See [10] for example configurations for ATM and BRL.

See the `quick-start` file for details of the BT-TCG user interface. Test case configurations can be saved from BT-TCG as an XML file (with suffix tcc.xml), for sharing or editing offline.

In what follows, the difference between a CP (/action/observation/NOI) node profile and the nodes in the tree that match them will be blurred. Where the distinction is important, it will be noted.

4.2 Test cases and test case generation

BT Analyser generates test cases in the form of paths between CPs, with no intervening CPs. For each pair of nodes matching CPs, BT Analyser generates three paths: a path from the initial state node to the starting CP, called the *pre-amble*; a path from the starting CP to the ending CP, called the *test case path (TCP)*; and a path from the ending CP to the initial state node, called the *post-amble*. It also generates a path from the root of the tree to the initial state node. In each case shortest path is generated, in terms of the number of non-atomic nodes in the path. When joined together, the 3 paths are called a *full test path (FTP)*.

For each such path, BT-TCG then generates a natural-language *test case* with four parts as follows:[14]

Precondition: The conditions needed to stimulate any external inputs that occur on the FTP.

---

the ATM, the messages sent between the ATM and the central bank database will not be observable.

[13] Whether or not an input is under tester control depends on the test environment and the kind of testing being undertaken. For module testing, the tester is assumed to have access to all variables in the module's interface, so all external inputs are within their control. For acceptance testing, where the SUT is connected to the target external systems, the latter typically will not be under tester control.

[14] Test cases starting from reversion and reference nodes are dropped, since they are already covered by test cases starting from their target node.

Pre-amble: A sequence of instructions to the tester corresponding to the actions in the pre-amble path.

Test steps: A sequence of tester actions and observable system responses from the TCP. Note that several tester actions may be required before an effect is observable, and conversely several observable effects may occur before another tester action is required.

Post-amble: A sequence of instructions to the tester for how to return the system to its initial state.

To generate test cases the user hits the 'Generate test cases' button: the output is displayed on the 'Test planner' tab. The results can also be output as an excel spread sheet with a list of individual test cases in one of three forms: as a test plan, with just tester actions and observations included;[15] as a full path; or in a simplified form for system documentation, as explained in §4.6 below.

The individual test cases can be done in any order the tester chooses.

The generated test cases are guaranteed to be feasible, assuming the BT model is correct with respect to the System Under Test (SUT). Likewise, the set is complete in the sense that, if it is possible for the SUT to transition from (the state corresponding to) one CP to another without passing through any other CP, then a test case will be generated between the pair. In particular, if all nodes in the BT model are reachable (which is not always the case), then all branches and nodes in the BT model get covered. Moreover, the generated test case will be the shortest, in terms of the number of steps in the path.[16]

4.3 Nodes of Interest

Although a short path is generally preferred for test efficiency, sometimes the tester wants to test particular scenarios, such as paths in which particular events or states do, or do not, occur. To support such a capability, [10] introduced the notion of *Nodes of Interest (NOI)*.

For each pair of CPs and each NOI, BT Analyser generates a shortest path between the CPs that does not involve the NOI (if any) and one that does involve the NOI (again, if any).[17]

This approach supports a wide range of test-coverage types, such as black-box testing and white-box (BT-oriented) testing, and enables development

[15] The tags of the CPs involved are also noted, for reference back to the BT model. This is particularly handy when debugging models, or to recreate the context of an error, if a test fails.

[16] From a test planner perspective, the number of actions and observations might be a more useful metric, but number of steps is more general and easier to compute with.

[17] This is slightly different to the proposal in [10], where different permutations of NOI were considered: this notion proved to be computationally expensive and difficult to guarantee completeness. The new notion proved to be adequate for most of our purposes.

of test cases for a wide range of interesting scenarios, without an explosion in the number of test cases. See [10] for illustration of use of NOI for the BRL example. (Note that NOI are not relevant to models that do not involve parallelism, such as ATM, since there can only ever be one path between each pair of CPs in such cases.)

The user selects NOI node profiles in the right-hand pane of the Test Configuration tab in BT-TCG, where nodes from the BT model are listed in depth-first order. NOI are stored as part of the test case configuration file. NOI are highlighted where they appear in the list of nodes under the tooltip in the Test Planner pane in BT-TCG.

4.4 Automated testing

As part of their theses projects, several students have investigated use of BT-TCG to generate test cases as test scripts for automated testing. The approach is the same as for manual testing above, except that the test planner provides code snippets in place of manual instructions for actions and observations. Support for patterns that commonly occur when testing web and Android apps has been included in BT-TCG as WebDriver snippets in Python syntax, selected by right-clicking in the right-hand pane of the Actions and Observables tabs.

For example, an event `User??select C??` might translate to a Python command to click on the element corresponding to `C` in the app. An observation `C[S]` might translate to a Python command to check that `C` has value `S`.

See the `codegen-notes` file for details.

4.5 Long test paths

As an alternative to resetting the SUT between each test case, BT-TCG supports development of a *long test path (LTP)* , formed by stringing test case paths together. To illustrate, consider the ATM example in §2.1. When configured as in [10], BT-TCG returns the ten TCPs shown in Table 3. The TCPs could be strung together into a single LTP as follows: `1, 2, 3, 7, 6, 4, 9, 2, 3, 8, 10, 1, 2, 3, 5.` [18]

BT Analyser keeps track of the system state as the LTP develops, and checks which test cases can feasibly be concatenated onto the end of the LTP. It can also check if any given test case is reachable from the current state, and find which intermediate steps to insert in order to bring the system into

[18]Note that where individual test cases involve preconditions (such as test cases 5, 6 and 7), the preconditions need to be worded carefully so it is clear which parameter (such as account type or withdrawal amount) the tester should choose at which step. This part of BT-TCG could usefully be improved to support parameterisation better, in order to support automated testing.

ID	Path	ID	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)

Table 3: Test case paths generated for the ATM example

a state where the selected test case can be applied; moreover, it checks which test cases are covered by intermediate steps.

The Test Planner tab in BT-TCG provides a simple user interface for developing LTPs. All of the generated test cases are listed in the LH panel, colour-coded according to whether they have been covered (grey) or not (white). The colour bar immediately to the right of a test case indicates whether it can be reached from the current state: green indicates it is immediately reachable; yellow indicates intermediate steps are required; and red indicates it is not reachable. If the user selects a yellow test case, the test case and its intermediate steps are appended to the end of the LTP and BT Analyser checks whether the intermediate steps cover any other test cases. For more details, see the `BT-TCG-design` document in the release.

4.6 Generating simplified user instructions

It is sometimes useful to be able to reverse-translate BT models into natural language descriptions, for example to generate excerpts for user manuals or for on-screen help.[19] Simple overview user guides often only include instructions for how to use a particular system function, whereas the tester needs more information, such as how to create the preconditions to invoke the function, and how to close the function in order to reset the system to its initial state. BT-TCG provides a facility for using a single BT model for both purposes, by using the traceability status indicator (§1.5) to enable the modeller to indicate which action and observable nodes are needed for simple user guides, and which ones for testers.

To use this facility, the modeller needs to change the traceability status indicator in action and observable nodes as shown in Table 4. To generate the simplified documentation, the user configures BT-TCG and generates test cases as usual for testing, but selects 'export to excel in simplified format' from the BT-TCG File function.[20] Actions should be phrased in passive mode, such as 'the icon is displayed', since nodes corresponding to preconditions in the simplified format get the words 'This function only applies if' prepended to

[19]The motivation for this problem came from another part of the ARC LP project with Thales which was concerned with automating HMI generation from configuration files.

[20]A late change to the way reversions are handled in TCG means that some redundant documentation cases are generated.

them.

Symbol	Use in simplified output format
+	node will appear verbatim as an instruction
++	node will appear as 'This function only applies if <instruction>'
+-	node will not appear

Table 4: Use of traceability status indicator for generating user instructions

A simple example WIMP interface is included in the release (`WIMP160728`). The BT model is composed of two parts: generic mouse behaviour, consisting of button clicks, holds and releases; and pointer behaviour, as the user moves the pointers over various icons in the window.[21]

5 Final words

5.1 Known limitations

The following limitations currently apply:

1. BT models need to be prepared in a different tool. A useful extension would be to integrate a BT model editor-simulator into the tool and link BT-TCG output back to the model, for example to visualise test path execution in the model.

2. Apart from `else` and `not`, propositional connectives (such as 'and' and 'or') are not supported in BT guards and selections.

3. It would be handy to have support for writing BT model properties in LTL.

4. Some BESE constructs (such as relations and 'what', 'why', etc) are not supported.

5. Currently only state realisation nodes can be selected as check points, but sometimes it would be useful to be able to select other types of node.

6. Currently test case configuration is done using node profiles, but sometimes it would be useful to be able to configure individual nodes separately. In particular, it would be handy to be able to select particular nodes as NOI – for example, to generate test cases relevant to a particular segment of a BT model, without leading to a combinatorial explosion of cases.

[21] The model is a stripped down version of a much larger case study – only showing actions associated with the left button here – but the overall structure has been preserved.

7. As noted in §4.5, the particular parameters that get chosen for user actions sometimes depends on the context (/test case). While this can be handled in the precondition part of test cases for manual testing in the basic approach, this is not adequate for automated testing and long test paths. A better solution is needed.

5.2 Acknowledgment

This work was supported by grant LP130100201 from the Australia Research Council and Thales Australia.

References

[1] R. Bjork. Automated teller machine example, 2004.

[2] R. J. Colvin and I. J. Hayes. A semantics for Behavior Trees using CSP with specification commands. *Science of Computer Programming*, 76(10):891–914, 2011.

[3] R. G. Dromey. From requirements to design: Formalizing the key steps. In *Proc. 1st Int. Conf. on SwEng. and Formal Methods (SEFM)*, pages 2–13. IEEE, 2003.

[4] R. L. Glass. Is this a revolutionary idea, or not? *Comm. ACM*, 47(11):23–25, 2004.

[5] L. Grunske, K. Winter, N. Yatapanage, S. Zafar, and P. A. Lindsay. Experience with fault injection experiments for FMEA. *Software: Practice and Experience*, 41(11):1233–1258, 2011.

[6] S.-K. Kim, T. Myers, M.-F. Wendland, and P. A. Lindsay. Execution of natural language requirements using state machines synthesised from Behavior Trees. *Journal of Systems and Software*, 85(11):2652 – 2664, 2012.

[7] S. Kromodimoeljo. *Controlling the Generation of Multiple Counterexamples in LTL Model Checking*. PhD thesis, The University of Queensland, Australia, 2014.

[8] S. Kromodimoeljo and P. A. Lindsay. Automatic generation of minimal cut sets. In *Proc. 4th Int. Wkshp on Eng. Safety & Security Systems (ESSS)*, pages 33–47, 2015.

[9] P. A. Lindsay. Behavior Trees: from systems engineering to software engineering. In *Proc. Sw. Eng. and Formal Methods*, pages 21–30. IEEE, 2010.

[10] P. A. Lindsay, S. Kromodimoeljo, P. A. Strooper, and M. Almorsy. Automation of test case generation from Behavior Tree requirements models. In *Proc. 24th Australasian Software Eng. Conference (ASWEC)*, pages 118–127. IEEE, 2015.

[11] D. Powell. Requirements evaluation using Behavior Trees – findings from industry. In *Industry track of Aust. Sw. Eng. Conf. (ASWEC)*, 2007. www.beworld.org.

[12] D. Powell. Behavior engineering: a scalable modeling and analysis method. In *Proc. 8th Int. Conf. on Sw. Eng. & Formal Methods*, pages 31–40. IEEE, 2010.

[13] The Behavior Tree Group. Behavior Tree notation v1.0, 2007.

[14] M. Utting and B. Legeard. *Practical model-based testing: a tools approach*. Morgan Kaufmann, 2010.

[15] S. Zafar, R. Colvin, K. Winter, N. Yatapanage, and R. G. Dromey. Early validation and verification of a distributed role-based access control model. In *Proc. 14th Asia-Pacific Software Eng. Conference (APSEC)*, pages 430–437. IEEE, 2007.