BT-TCG Design and Implementation

Sentot Kromodimoeljo

July 27, 2016

Abstract

This document describes the design and implementation of BT-TCG: a tool for generating test cases from Behavior Tree (BT) models. It is intended to help the reader who wants to extend or revise BT-TCG for research and/or commercial purposes. This document assumes that the reader is familiar with the BT notation.

1 Introduction/Overview

The BT model shown in Figure 1 is used as a running example in this document. Section 2 describes the model-based test case generation methodology that BT-TCG supports. Section 3 describes the actual test case generation in BT-TCG. Section 4 describes how BT-TCG supports the translation of test cases into instructions (for human or machine) consisting of actions and observations. Section 5 describes how BT-TCG supports the construction of long test plans from test cases. Section 6 describes the main calls from BT-TCG to its inference engine: BT Analyser. Section 7 discusses support for test code generation in BT-TCG. Section 9 provides a guide on BT-TCG and BT Analyser documentation that may help the reader in extending and/or commercialising BT-TCG. Finally, Section 10 provides a conclusion.

2 Test Case Generation Methodology

BT-TCG is based on the methodology described in [7], which uses a notion of *check points* (CPs): state realisation nodes that correspond to (potential) changes[1] in *system states*. A test case in the methodology is a *feasible* path (sequence of transitions) in the BT model that starts from a CP and ends at a CP with no CPs in between, where a transition corresponds to the "execution" of a BT node or atomic composition of BT nodes[2]. The methodology has since undergone minor revisions, but the main idea of paths between CPs remains.

Associated with test case generation is a notion of *coverage*. In general, many test cases are required to obtain some kind of coverage. Topics on various kinds of coverage are beyond the scope of this document. Instead, the notion of coverage in the methodology is in terms of CPs and *nodes of interest* (NOIs): additional BT nodes whose presence or

[1]The fact that CPs are (potential) state changes rather than states introduces a subtlety of which the reader needs to be aware.

[2]Since an atomic composition of BT nodes consists of a sequence of BT nodes, a path also corresponds to a sequence of BT nodes.

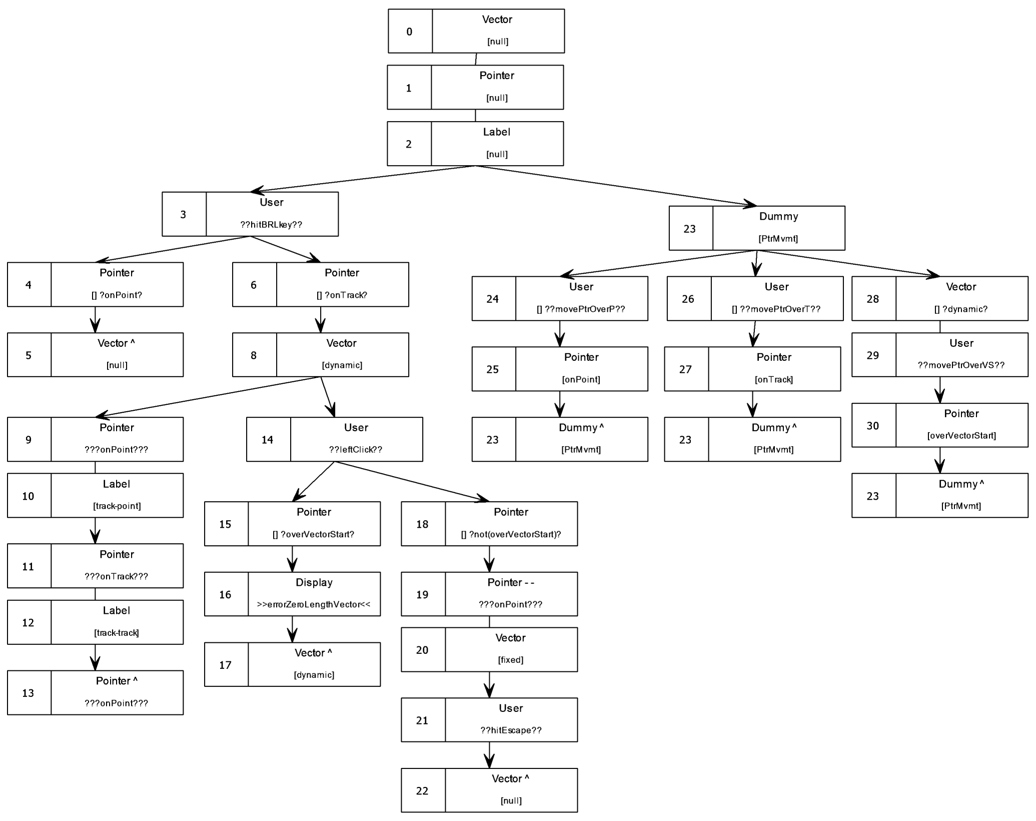


Figure 1: BT Model for Bearing and Range Line (BRL) version 150515

absence in a test case are of interest. The hope is a desired coverage can be mapped into coverage in terms of CPs and NOIs.

In the methodology, between each ordered pair of CPs, at the very least a test case must be generated, provided, of course, that a path between the pair is feasible. In general, a shortest path that represents a class of test cases is desired. Thus, a shortest path between an ordered pair of CPs is a good candidate for a test case and should be generated.

Sometimes we want more coverage between a pair of CPs. We might be interested in the existence of a path between the CPs where a BT node — NOI — is present, and in the existence of a path between the CPs where NOI is absent, and have test cases generated accordingly. More elaborate coverage can be envisioned where the occurrence and order of occurrence of multiple NOIs in a path are important. However, this can quickly lead to a combinatorial explosion. Thus combinations of NOIs must be handled very carefully. The actual implementation of test case generation will be described in Section 3.

In addition to the coverage aspect of test case generation, there is the *target* aspect of test case generation. What is important about the target is *how* testing is to be performed rather than *what* is to be tested. The *how* of performing tests involves the following issues:

- determining whether tests are to be performed "manually" by humans, or if code needs to be generated to perform the tests,
- identifying *actions* required in the tests, and
- identifying effects that need to be *observed* during the tests.

The focus of the original methodology was in human-performed testing. However, the methodology seems to be equally suited for testing with various degrees of automation.

Actions can be associated with external events (including external inputs) in the BT model. For human-performed testing, the events can be translated into human actions. However, each action event can also be translated into code or a combination of code and human actions.

Observations can be associated with state realisations and external outputs. If the target is human-performed testing, the relevant BT nodes can be translated into direct human observations. As with actions, however, each relevant BT node can be translated into code or a combination of code and human observations.

The support for actions and observations in BT-TCG will be described in Section 4.

3 Actual Test Case Generation in BT-TCG

CPs were described in Section 2 as being BT state realisation nodes. In practice and reality, we have the following complications:

- Multiple state realisation nodes can occur in a BT that have the same *component* and *state*. In practice, if we choose a state realisation node as a CP, we would like all state realisation nodes with the same component and state to be CPs, and in fact this was the assumption in [7]. Thus for the purpose of specifying CPs, it is preferable to treat a set of state realisation nodes with the same component and state as a single entity rather than separate nodes.

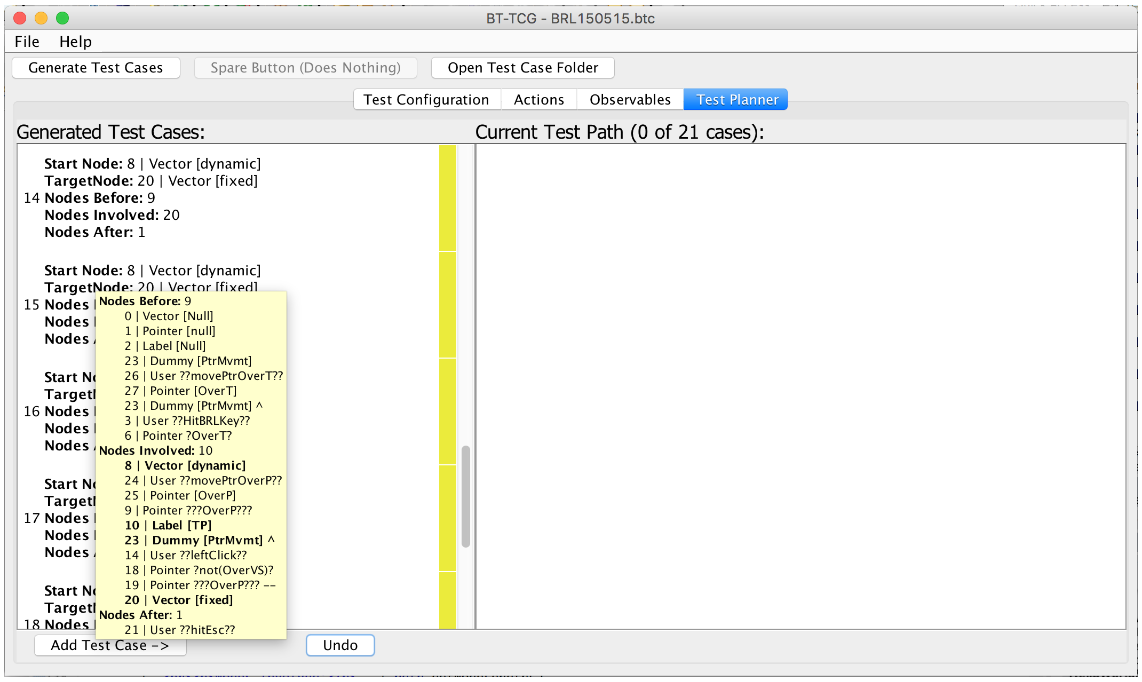
- A CP state realisation node can be in an atomic composition of BT nodes. If such a CP is not the first or the last node in the atomic composition, the node cannot be the first or the last node in a path respectively.

In BT-TCG, a CP is specified by a BT node *profile* rather than a specific BT node. The profile of a BT node is determined by the BT node's component and *behaviour* (for a state realisation node, the behaviour is simply the state part of the node), and whether the BT node is a *kill* node. When a BT node profile is designated as a CP, all BT nodes with that profile become CPs. This results in a simpler user interface, at the expense of some flexibility. The benefit of using node profiles for CPs far outweigh the small loss of flexibility.

Because an atomic composition of BT nodes is treated as a single transition in a path, if a CP is in an atomic composition, the entire atomic composition in effect becomes a CP. Thus a test case might not start or end with a CP node but in such a case, it must start or end with an atomic composition involving a CP respectively.

In BT-TCG test case generation, a particular BT node that is a CP, is designated as the *initial state*. The initial state is not necessarily the root of the BT model (the root need not be a CP). Although the initial state serves as a starting point, for testing, some steps may need to be performed to reach the starting point (represented by a path from the root of the BT model to the initial state).

More generally, for each individual test case, if the test path does not start at the root, a *pre-amble* is generated by BT-TCG: a sequence of transitions preceding the test path, starting at the root. The pre-amble followed by the test path produces a path that is feasible in the BT model. A *post-amble* is also produced, independent of the need for a pre-amble, that leads back to the initial state, if it is possible to go to the initial state after



```
Generated Test Cases:                          Current Test Path (0 of 21 cases):

     Start Node: 8 | Vector [dynamic]
     TargetNode: 20 | Vector [fixed]
  14 Nodes Before: 9
     Nodes Involved: 20
     Nodes After: 1

     Start Node: 8 | Vector [dynamic]
     TargetNode: 20 | Vector [fixed]
  15 Nodes Before: 9
           0 | Vector [Null]
           1 | Pointer [null]
           2 | Label [Null]
           23 | Dummy [PtrMvmt]
           26 | User ??movePtrOverT??
           27 | Pointer [OverT]
           23 | Dummy [PtrMvmt] ^
           3 | User ??HitBRLKey??
           6 | Pointer ?OverT?
     Nodes Involved: 10
           8 | Vector [dynamic]
           24 | User ??movePtrOverP??
           25 | Pointer [OverP]
           9 | Pointer ???OverP???
           10 | Label [TP]
           23 | Dummy [PtrMvmt] ^
           14 | User ??leftClick??
           18 | Pointer ?not(OverVS)?
           19 | Pointer ???OverP??? --
           20 | Vector [fixed]
     Nodes After: 1
           21 | User ??hitEsc??
```

Figure 2: Example Test Case

the test case. Note that whereas a pre-amble is always possible (albeit sometimes empty), there might not be any feasible post-amble. BT-TCG does not distinguish between an impossible post-amble and an empty post-amble (where the test path already leads to the initial state). The optional pre-amble, followed by the test path, followed by the post-amble, produces a path that is feasible in the BT model.

Figure 2 shows an example test case (the part in light yellow). The pre-amble is represented by `Nodes Before`, the actual test case by `Nodes Involved`, and the post-amble by `Nodes After` (recall from Section 2 that a path can be specified in terms of nodes as well as transitions). For `Nodes Involved`, the CPs and NOIs appear in bold font (the first and last BT nodes that appear in bold font are CPs and the rest are NOIs).

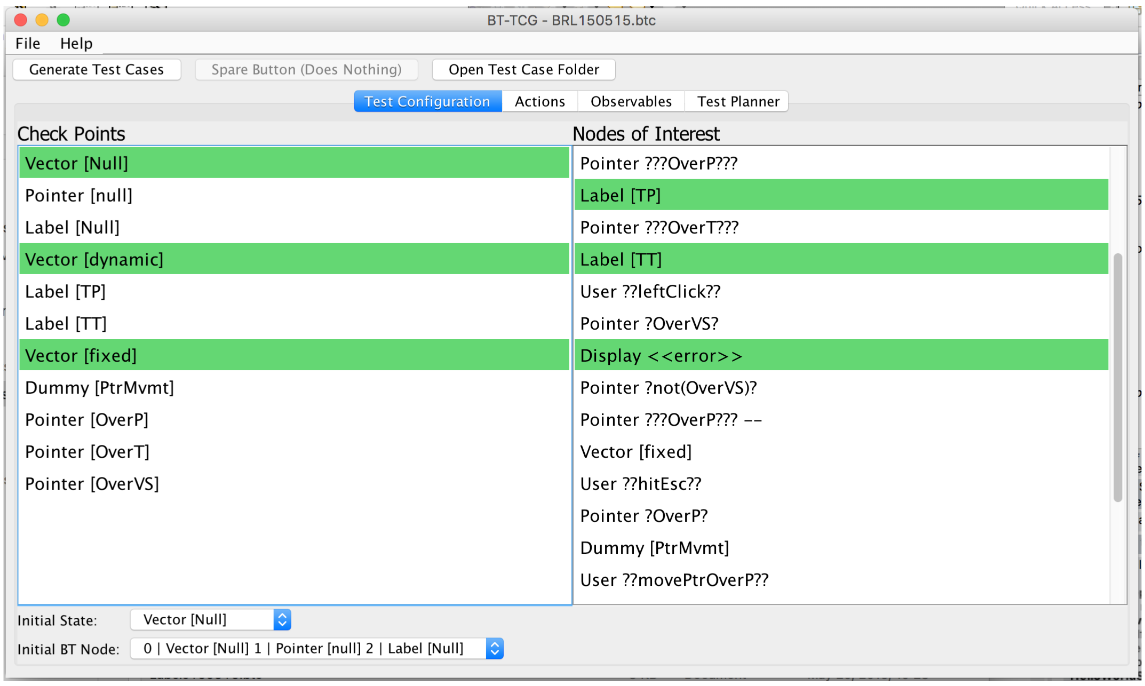
NOIs are also specified using node profiles in BT-TCG[3]. However, unlike CPs, which are restricted to state realisations, there are no restrictions for NOIs. Reference nodes and reversion nodes that are not CPs are automatically NOIs. For each ordered pair of CPs, as well as generating a shortest path between the pair of CPs, for each NOI:

- if there is a feasible path between the pair that does not contain the NOI, then there is a generated test path between the pair that does not contain the NOI, and
- if there is a feasible path between the pair that does contain the NOI, then there is a generated test path between the pair that contains the NOI.

Sometimes a path that starts with a *reversion* or *reference* differs from another path only in the starting transition, with the other path starting at the target of the reversion or reference. In such a case[4], BT-TCG considers the path that starts with a reversion or reference to be *subsumed* by the other path, and the first path is discarded. A path

[3]Future versions of BT-TCG may include graphic representation of BTs where specific BT nodes may be chosen as NOIs.

[4]Atomic compositions can also be present in such cases, and the complications they introduce are taken into account by BT-TCG.



BT-TCG - BRL150515.btc

File Help

Generate Test Cases | Spare Button (Does Nothing) | Open Test Case Folder

Test Configuration | Actions | Observables | Test Planner

Check Points

- Vector [Null]
- Pointer [null]
- Label [Null]
- Vector [dynamic]
- Label [TP]
- Label [TT]
- Vector [fixed]
- Dummy [PtrMvmt]
- Pointer [OverP]
- Pointer [OverT]
- Pointer [OverVS]

Nodes of Interest

- Pointer ???OverP???
- Label [TP]
- Pointer ???OverT???
- Label [TT]
- User ??leftClick??
- Pointer ?OverVS?
- Display <<error>>
- Pointer ?not(OverVS)?
- Pointer ???OverP??? --
- Vector [fixed]
- User ??hitEsc??
- Pointer ?OverP?
- Dummy [PtrMvmt]
- User ??movePtrOverP??

Initial State: Vector [Null]

Initial BT Node: 0 | Vector [Null] 1 | Pointer [null] 2 | Label [Null]

Figure 3: Test Configuration Tab

is, of course, subsumed by an identical path. Other kinds of subsumption are being investigated but have not been incorporated into BT-TCG.

CPs and NOIs can be entered and revised in the `Test Configuration` tab of BT-TCG (see Figure 3). A test configuration can be saved into a file and can be read from a file.

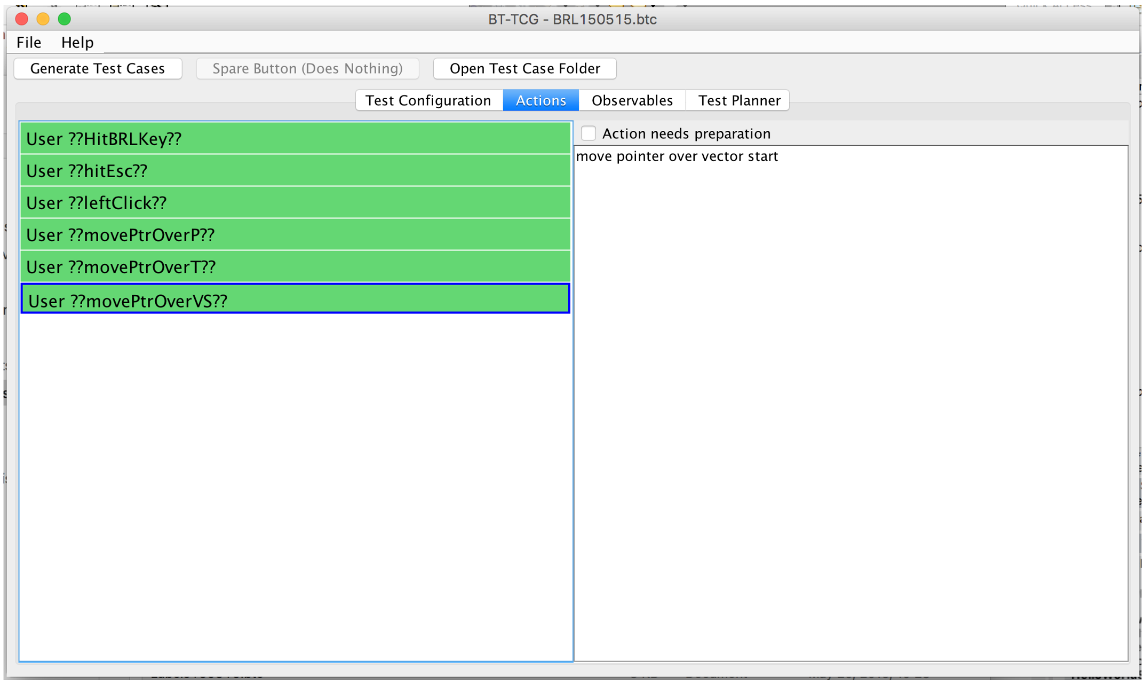
4 Actions and Observations

In addition to CPs and NOIs, a *test configuration* in BT-TCG includes mappings for *actions* and *observations*. A node profile that corresponds to an action or observation is mapped to text. The idea is a test path generated by BT-TCG can be translated into a sequence of instructions for actions and observations in the actual test. The sequencing of the instructions follows the sequencing of the corresponding nodes in the test path. In BT-TCG, the mapping for actions can be entered and revised in the `Actions` tab (see Figure 4).

The checkbox — `Action needs preparation` — when ticked means extra preparation may need to be performed before the test in order for the action to have the desired effect, for example, the conditions for stimulating an external input may need to be established prior to testing.

The mapping for observations can be entered and revised in the `Observables` tab (see Figure 5). The tab is similar to the `Actions` tab, except there is no checkbox.

The text for an action or observation can be instructions for humans, code, or a combination of code and instructions for humans. The form for entering and revising the text is a free-text area, which provides much flexibility. To help the user of BT-TCG produce the mappings, particularly for code generation, templates for common situations need to be developed. There are example templates in BT-TCG that can insert code into the text area based on the node profile, however, much work needs to be done to

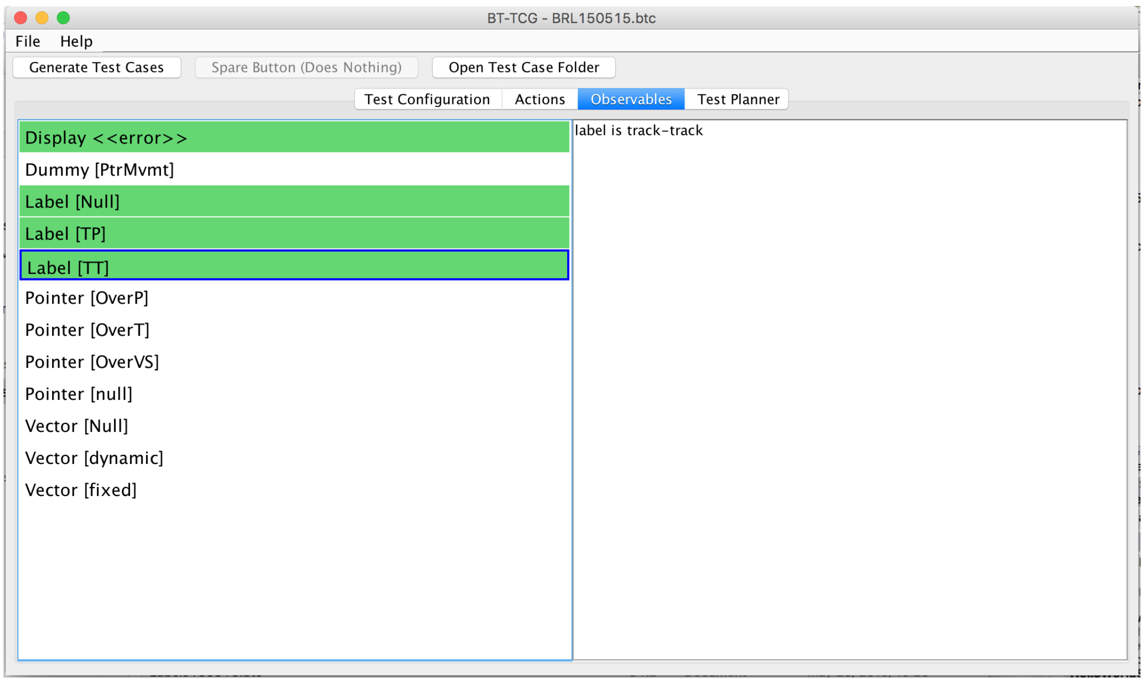


User ??HitBRLKey??
User ??hitEsc??
User ??leftClick??
User ??movePtrOverP??
User ??movePtrOverT??
User ??movePtrOverVS??

Action needs preparation

move pointer over vector start

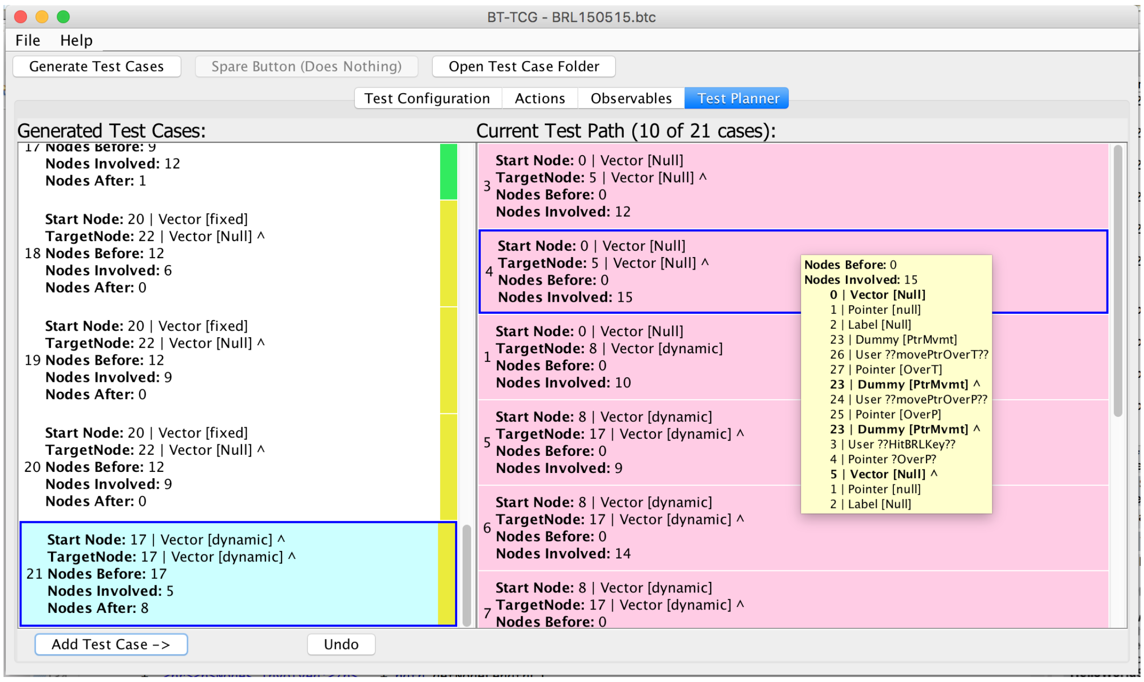
Figure 4: Actions Tab



Display <<error>>
Dummy [PtrMvmt]
Label [Null]
Label [TP]
Label [TT]
Pointer [OverP]
Pointer [OverT]
Pointer [OverVS]
Pointer [null]
Vector [Null]
Vector [dynamic]
Vector [fixed]

label is track–track

Figure 5: Observables Tab



```
Generated Test Cases:
17 Nodes Before: 9
   Nodes Involved: 12
   Nodes After: 1

   Start Node: 20 | Vector [fixed]
   TargetNode: 22 | Vector [Null] ^
18 Nodes Before: 12
   Nodes Involved: 6
   Nodes After: 0

   Start Node: 20 | Vector [fixed]
   TargetNode: 22 | Vector [Null] ^
19 Nodes Before: 12
   Nodes Involved: 9
   Nodes After: 0

   Start Node: 20 | Vector [fixed]
   TargetNode: 22 | Vector [Null] ^
20 Nodes Before: 12
   Nodes Involved: 9
   Nodes After: 0

   Start Node: 17 | Vector [dynamic] ^
   TargetNode: 17 | Vector [dynamic] ^
21 Nodes Before: 17
   Nodes Involved: 5
   Nodes After: 8
```

```
Current Test Path (10 of 21 cases):
  Start Node: 0 | Vector [Null]
  TargetNode: 5 | Vector [Null] ^
3 Nodes Before: 0
  Nodes Involved: 12

  Start Node: 0 | Vector [Null]
  TargetNode: 5 | Vector [Null] ^
4 Nodes Before: 0
  Nodes Involved: 15

  Start Node: 0 | Vector [Null]
  TargetNode: 8 | Vector [dynamic]
1 Nodes Before: 0
  Nodes Involved: 10

  Start Node: 8 | Vector [dynamic]
  TargetNode: 17 | Vector [dynamic] ^
5 Nodes Before: 0
  Nodes Involved: 9

  Start Node: 8 | Vector [dynamic]
  TargetNode: 17 | Vector [dynamic] ^
6 Nodes Before: 0
  Nodes Involved: 14

  Start Node: 8 | Vector [dynamic]
  TargetNode: 17 | Vector [dynamic] ^
7 Nodes Before: 0
```

```
Nodes Before: 0
Nodes Involved: 15
    0 | Vector [Null]
    1 | Pointer [null]
    2 | Label [Null]
    23 | Dummy [PtrMvmt]
    26 | User ??movePtrOverT??
    27 | Pointer [OverT]
    23 | Dummy [PtrMvmt] ^
    24 | User ??movePtrOverP??
    25 | Pointer [OverP]
    23 | Dummy [PtrMvmt] ^
    3 | User ??HitBRLKey??
    4 | Pointer ?OverP?
    5 | Vector [Null] ^
    1 | Pointer [null]
    2 | Label [Null]
```

Figure 6: Test Planner Tab

improve this capability. Technical details on how to add templates to BT-TCG and how to extend BT-TCG in general can be found in the add-on guide for BT-TCG [1].

5 Long Test Plans

The test cases discussed so far are individual test cases between CPs in isolation, although a test case is guaranteed to be feasible in the BT model in the sense that there is a path from the root of the BT model to the test case. The idea of stringing the individual test cases into a *long test plan* was originally developed as part of a student project for Mitchell Savell. The long test plan support that has been incorporated into BT-TCG includes a high degree of automation. Long test plan development is done in the `Test Planner` tab of BT-TCG as shown in Figure 6. The long test plan is represented by the *current test path.*

Implicitly, there is a notion of *current state* when developing a long test plan in BT-TCG. Initially the current state is the state at the root of the BT model[5]. Thereafter, it is the state reached after the last transition of the current test path. In the `Generated Test Cases` part of the tab, test cases that are *immediately available* from the current state are marked green in the side bar, where a test case is immediately available if no transitions need to be inserted before the (main part of the) test case can be added to the current test path. Test cases that are not immediately available but is reachable from the current state are marked yellow in the side bar. Test cases that are not reachable from the current state are marked red in the side bar. Test cases that have been added to the current test path are shaded in light blue colour.

The user selects a test case from the generated test cases and adds it to the current test path (if the test case selected has already been added to the current test path, then

[5]Strictly speaking it is the set of possible states at the root of the BT model.

BT-TCG does nothing — a test case is added to the current test path at most once). The example in Figure 6 shows the effect of adding test case 21 to the current test path where the test path was previously empty. BT-TCG inserted 9 other test cases before case 21. The manner in which BT-TCG inserts additional test cases is roughly as follows:

- If the test case being added is immediately available, then each additional test case is a *loop* case (ends by reverting to its starting node).
- If the test case being added is not immediately available, then it must be reachable, and a shortest *gap* between the current state and the test case is computed. BT-TCG then tries to fill this gap with test cases that are not yet covered, with as little *overhead* as possible.

An overhead is a sequence of transitions in the current test path that is not part of a test case in the path. An overhead is associated with each test case in the current test path, and its size is indicated by the `Nodes Before` count. Normally, the test cases automatically inserted by BT-TCG have no overhead. However, if the current test path is empty and no test case is immediately available (which can happen if the root of the BT model is not a CP), then the first test case automatically inserted may have an overhead.

The gap is filled by matching yet-to-be-covered test cases with an initial subsequence of the gap (which can be the entire gap). If BT-TCG finds a match, the matching test case is inserted into the current test path and the gap reduced by the test case. At any stage in the gap-filling process, yet-to-be-covered looping cases can be inserted. BT-TCG is smart enough to transfer the overhead of a case to the first looping case when required. Once no more yet-to-be-covered test cases (other than the test case to be explicitly added) can be inserted, the remaining gap, if any, becomes an overhead for the explicitly added test case.

The user can remove test cases from the current test path using the undo facility. The user selects a test case in the current test path and clicks on the `undo` button: all test cases from the selected test case on are removed from the current test path.

At any stage in the long test plan development, BT-TCG ensures that the current test path is feasible in the BT model.

6 Interface with BT Analyser

BT-TCG is written in Java using the Eclipse development environment. It has only been tested on the Mars version of Eclipse and has never been made into a self-contained *jar* file for Java.

BT-TCG uses BT Analyser as its inference engine, which it automatically spawns as a process. It communicates with BT Analyser through a socket interface. BT-TCG issues commands in *s-expression* syntax to BT Analyser and receives the replies in XML format. The s-expression syntax and the XML formats are described in the API document for BT Analyser [2]. More documentation for BT Analyser is provided by [5].

BT-TCG and BT Analyser can read BT models from files generated by TextBE, ComBE and BESE tools, although some features of BESE cannot be handled[6]. The parsing of these files are done by BT Analyser. For a source file from TextBE or ComBE, the `process-bt-file` command is issued to BT Analyser:

[6]BT-TCG supports a subset of BESE roughly equivalent to the BT notation supported by TextBE.

```
(process-bt-file "BRL150515.btc")
```

and for a source file from BESE, the `process-bese-file` command is issued:

```
(process-bese-file "BRL150515.xml").
```

If the parsing is successful, the following response would be obtained from BT Analyser:

```
<result>SUCCESS</result>.
```

After parsing, BT-TCG would ask BT Analyser to compute reachability information on the model. Then BT-TCG obtains the parsed representation of the BT model by issuing:

```
(print-bt)
```

with a returned result of the following form:

```
<result><block> ... </block> ... ... <block> ... </block></result>
```

where a *block* represents a BT node or an atomic composition of BT nodes.

The main command for obtaining test cases is `find-test-paths`:

```
(find-test-paths 0 (24) 3 (0 3 12 14 16 19))
```

where 0 is the index of the block representing the BT node or atomic composition of BT nodes for the starting CP, (24) is a list of block indices for NOIs, 3 is the block index for the ending CP, and (0 3 12 14 16 19) is a list of block indices for all CPs. BT Analyser will produce a (possibly empty) list of paths:

```
<result>
<path>
<block-index>0</block-index><block-index>21</block-index>
<block-index>25</block-index><block-index>26</block-index>
<block-index>27</block-index><block-index>1</block-index>
<block-index>2</block-index><block-index>3</block-index>
</path>
<path>
<block-index>0</block-index><block-index>21</block-index>
<block-index>22</block-index><block-index>23</block-index>
<block-index>24</block-index><block-index>25</block-index>
<block-index>26</block-index><block-index>27</block-index>
<block-index>1</block-index><block-index>2</block-index>
<block-index>3</block-index>
</path>
</result>.
```

Other BT Analyser commands used in test case generation and test planning include `test-path-preamble`, `test-path-postamble`, `test-path-gap` and `check-test-path` for computing pre-ambles, computing post-ambles, computing gaps and checking the feasibility of paths respectively.

7 Support for Test Code Generation

Section 4 describes how BT-TCG supports the translation of BT-nodes into arbitrary text fragments. A text fragment can be code in some test scripting notation such as Groovy, Python or Java. However, it would be tedious for the user to type or copy-paste code into the `Actions` and `Observables` tabs. To better support test code generation, templates for code need to be developed for the tabs.

In addition, an add-on tab for code generation may be needed where the user can specify setup and tear-down code and other parameters for generating test code. There is a guide for writing add-ons for BT-TCG [1] including extra tabs and templates for the `Actions` and `Observables` tabs. Issues for test code generation in BT-TCG are discussed in [4]

8 File Loading and Saving

In addition to being able to load BT model files, BT-TCG can save configurations (set in the `Test Configuration`, `Actions` and `Observables` tabs) that can later be loaded, not necessarily in the same session. When a configuration file is loaded, the associated BT model is also loaded.

BT-TCG also allows test cases and long test plans to be exported in various formats (currently as Excel files). With the development of more support for test code generation, it is expected that more formats will be supported.

9 Guide to Other Documentation

BT-TCG and BT Analyser documentation that may be useful for the reader who plans to extend and/or commercialise BT-TCG includes:

- the `readme` for BT Analyser [5], is the main documentation for installing and using BT Analyser;
- the `API` documentation for BT Analyser [2], describes the API for the socket interface of BT Analyser;
- the `quick start guide` for BT-TCG [3], a guide to easy installation and use of BT-TCG;
- the `user guide` for BT-TCG [6], a user's guide for BT-TCG that contains more information than the quick start guide;
- the `add-on guide` for BT-TCG [1], a guide for writing add-ons for BT-TCG explaining, among other things, the main data structures and methods of BT-TCG, as well as how add-ons are to be hooked into BT-TCG; and
- the `notes on test code generation` [4] for BT-TCG.

10 Conclusion

The design and implementation of BT-TCG has been briefly described in this document. It provides an overview of BT-TCG, with many of the missing details provided by other documents included in a BT-TCG distribution as cited in the references.

11 Acknowledgement

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

References

[1] Sentot Kromodimoeljo. A guide for writing BT-TCG add-ons. Included in BT-TCG distribution as file: add-on-guide.pdf, 2016.

[2] Sentot Kromodimoeljo. API for BT Analyser. Included in BT Analyser distribution as file: api.txt, 2016.

[3] Sentot Kromodimoeljo. BT-TCG Quick Start. Included in BT-TCG distribution as file: quick-start.pdf, 2016.

[4] Sentot Kromodimoeljo. Notes on Test Code Generation in BT-TCG. Included in BT-TCG distribution as file: codegen-notes.pdf, 2016.

[5] Sentot Kromodimoeljo. Readme File for BT Analyser. Included in BT Analyser distribution as file: readme.txt, 2016.

[6] Peter A. Lindsay and Sentot Kromodimoeljo. BT-TCG User Guide. Included in BT-TCG distribution as file: BT-TCG_UserGuide.pdf, 2016.

[7] Peter A. Lindsay, Sentot Kromodimoeljo, Paul A. Strooper, and Mohamed Almorsy. Automation of test case generation from behavior tree requirements models. In *24th Australasian Software Engineering Conference, ASWEC 2015, Adelaide, SA, Australia, September 28 - October 1, 2015*, pages 118–127. IEEE Computer Society, 2015.