A guide for writing BT-TCG add-ons

Sentot Kromodimoeljo

July 11, 2016

1 Introduction

BT-TCG is a Java program for producing test cases from BT models that uses BT Analyser as its engine. Currently BT-TCG runs as an Eclipse project, but the goal is to eventually have it runnable as a jar file. The core of BT-TCG is a major rewrite, by Mitchell Savell, of the core of an existing tool, TCGen-UI, written by Wei Yu Soh.

The BT-TCG core has a somewhat limited list of features. You can load a BT model, configure the test parameters (check points, observables, actions and nodes of interest), save and load configurations, generate all test paths, and little else. To be a more effective tool, BT-TCG would need more capabilities to be added, thus the idea of *add-ons*.

The purpose of this document is to show how add-ons are to be incorporated into BT-TCG. This document assumes that you have a computer with Eclipse (preferably Mars) and the SBCL Lisp system both installed, and you possess some basic skills in system administration. On a unix system, you must have administrator privileges to run BT-TCG (this is because BT Analyser is run as a socket server).

2 Installing BT-TCG

To obtain high integration, BT-TCG has its own copy of BT Analyser in the `BTAnalyser` subfolder of the BT-TCG project folder. The copy of BT Analyser in the subfolder needs to be compiled for the specific OS platform of the computer (instructions for compiling are in `readme.txt` in the subfolder). Currently BT-TCG assumes that SBCL is used as the Lisp system.

If the computer is running a unix system (e.g., Mac OS or Linux), then the file `start.sh` needs to be made executable. From the project folder, using a command line interface, type:

```
chmod +x start.sh
```

BT-TCG must then be imported into Eclipse as an existing project. From the `file` menu in Eclipse, select import and then select `Existing Projects into Workspace` (see Figure 1). Once BT-TCG has been successfully imported as an existing project, you should be able to run BT-TCG. Note that the project folder need not be inside the workspace folder of Eclipse.

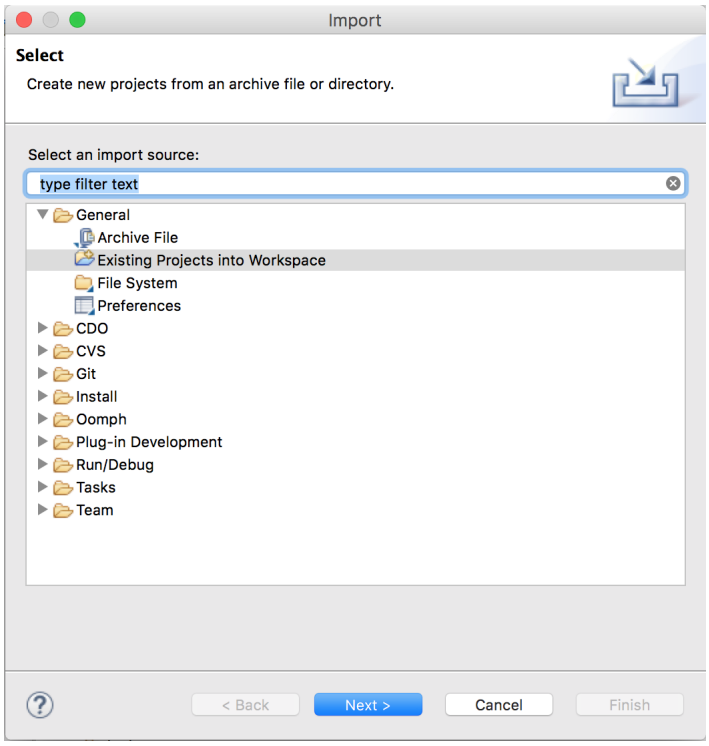

Figure 1: Import BT-TCG as an existing project

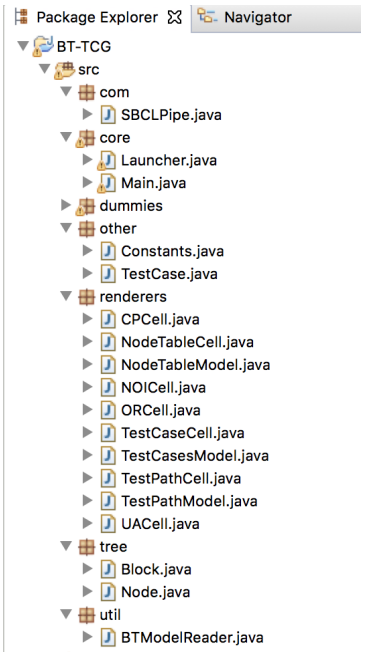

Figure 2: BT-TCG Java packages

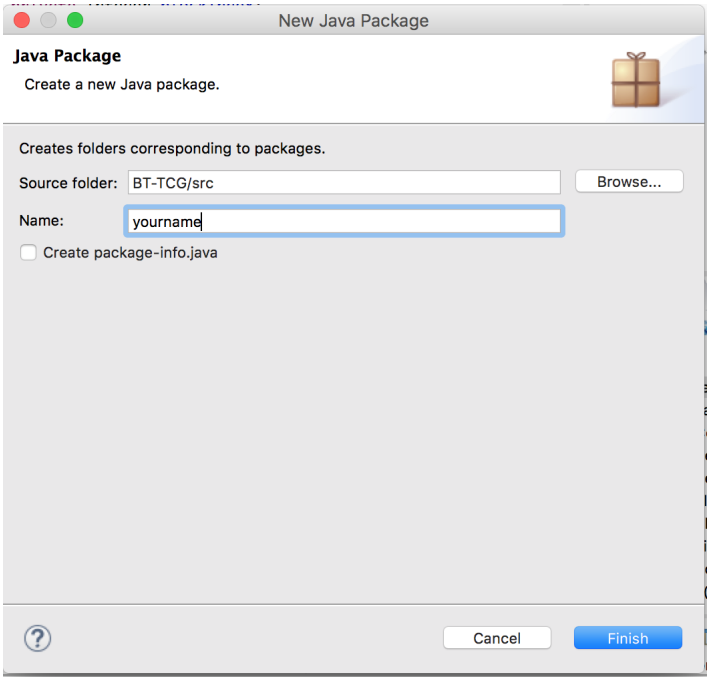

Figure 3: Creating a new package for an add-on

3 BT-TCG Java packages

Once BT-TCG has been successfully imported as a project into Eclipse, you should be able to see the Java packages that implement BT-TCG via the package explorer of Eclipse, such as shown in Figure 2.

The Java classes for an add-on would reside in its own package. You would not be changing any existing Java files except `Launcher.java` in the `core` package. The `Launcher` class is where the BT-TCG core and the add-ons are glued together.

4 Creating an add-on

Three kinds of add-ons are proposed for BT-TCG:

- *tab* add-ons, represented by tabs in the main BT-TCG window,
- *file menu* add-ons, for additional entries in the main file menu, and
- *popup* add-ons, for adding popup menus for existing *text areas*.

A tab add-on would be implemented by a *JComponent* class residing in its own package. A file menu add-on would be implemented by a *JMenu* class residing in its own package. A popup add-on can reside entirely in `Launcher.java`, or it can have some of its code in a separate class in an add-on package.

To create a tab add-on, first you need to create a new package in the project. In Eclipse, click on `File->New->Package`. Type in the name of your new package, e.g., `yourname`, in the dialog window (see Figure 3) and make sure that `BT-TCG` is selected as the project if you have multiple projects in your Eclipse workspace.

Now you are ready to create a new class for your tab add-on that will reside in your own package (`yourname` in the example). Click on `File->New->Package` in the name of

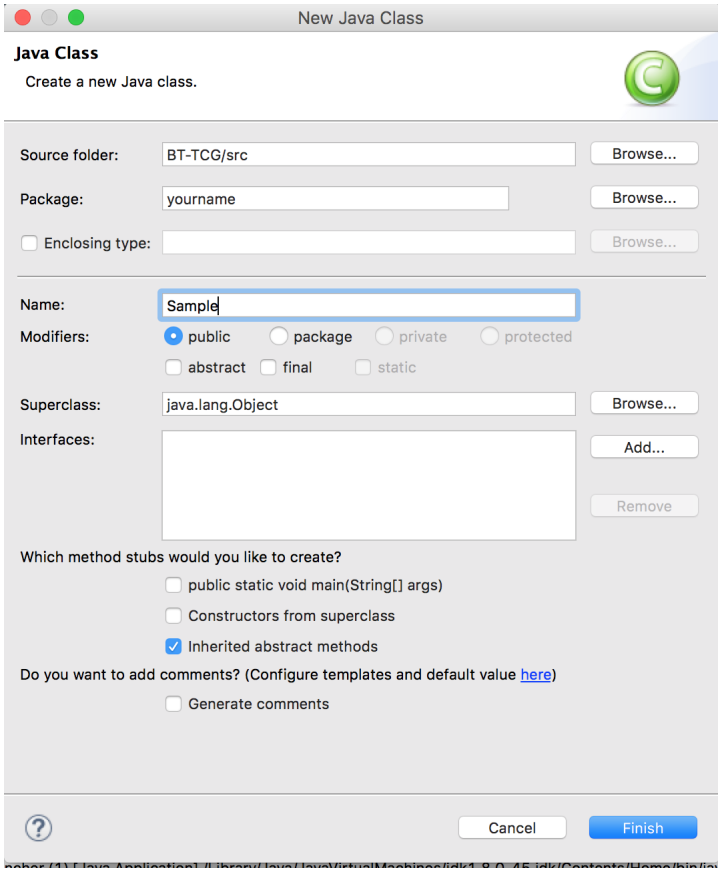

Figure 4: Creating a new class for an add-on

your new class in the dialog window (see Figure 4) and make sure that your package is selected as the package. Figure 5 shows the skeleton that Eclipse created for the new class. You can now proceed to add details to your new class.

The class that implements a tab add-on can be any subclass of *JComponent*, e.g., a subclass of *JPanel*. In fact, if we had specified `javax.swing.JPanel` as the superclass in the dialog, instead of the default java.lang.Object, Eclipse would have created a different skeleton code, as in Figure 6.

After you have created your add-on, you would need to "glue" your add-on to the BT-TCG core. There are three steps that you can follow:

1. Import your class in `Launcher.java` (see line 20 in Figure 7).

2. Add to `Launcher.java` the code to add your add-on as a tab (see lines 88-89 in Figure 8).

3. Add a public static void method to clear the add-on's data and add a call to the method from the `clear` method in the `Launcher` class.

Even without performing step 3, you can run BT-TCG and the add-on has been added as a tab as shown in Figure 9.

Obviously you need to add code to your add-on to interact with the BT-TCG core and do something interesting. For that, you would need

- some Java Swing programming (Eclipse can make it easy for you),

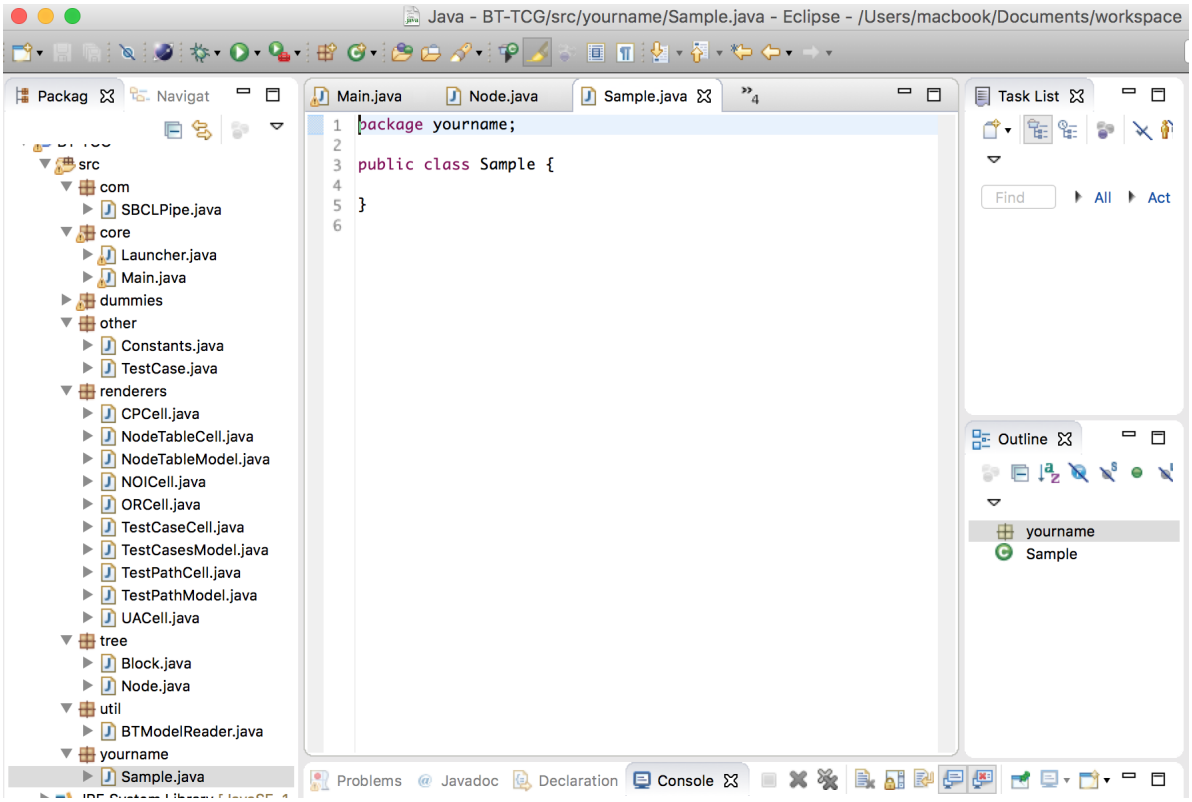

Figure 5: Newly created class

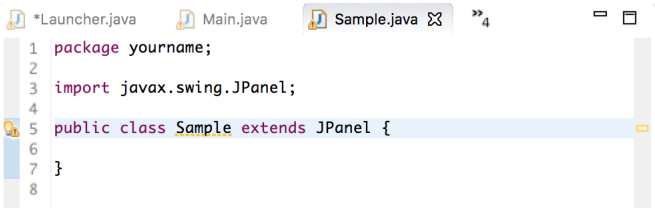

Figure 6: Skeleton created for JPanel subclass

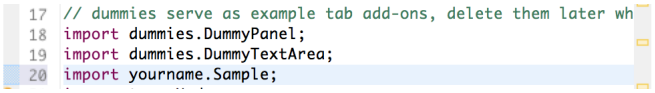

Figure 7: Import the class for the add-on in Launcher.java

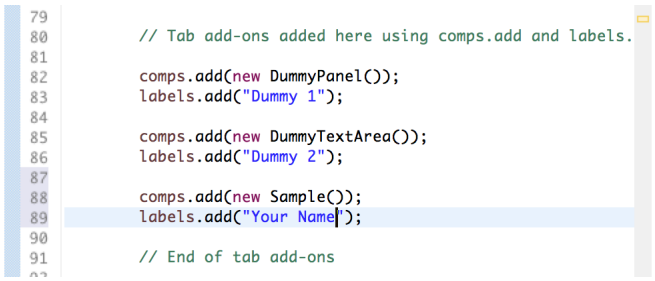

Figure 8: "Glue" the add-on in Launcher.java

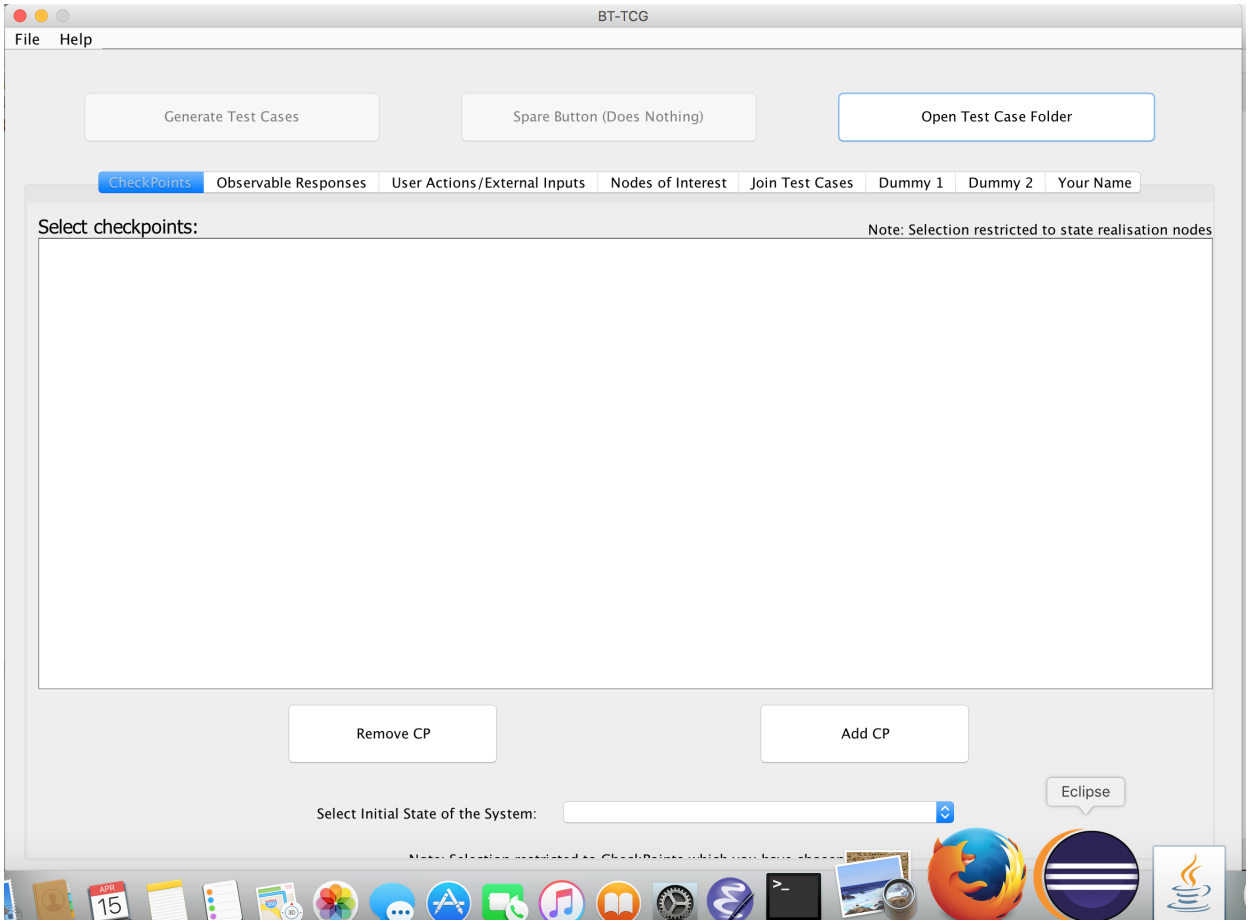

Figure 9: BT-TCG with the add-on

- knowledge of the important data structures of BT-TCG (e.g., the `Node` class), and
- knowledge of the important public methods of the `Main` class.

This document will not cover Java Swing programming (there are a lot of tutorials on Java Swing programming on the internet). The rest of the document will cover the important data structures of BT-TCG, the important public methods of the `Main` class, and various tips on dealing with the data structures and organising your own data structure, including resetting.

The process of adding a file menu add-on is similar to that of adding a tab add-on. An example of a file menu add-on is the *Save Test Cases as Excel* operation implemented by the `SaveExcel` class in the *excel* package. See how it is glued to the BT-TCG core in `Launcher.java` by the addition of an "entry" in the method

```
public static ArrayList<JComponents> fileMenuItems ()
```

(with `excel.SaveExcel` imported).

To add popup menu add-on, see the examples in the method

```
public static void main(String[] args)
```

in `Launcher.java`.

5 BT-TCG data structures

5.1 Node

An object of the `Node` class is intended to represent a BT node. You SHOULD NOT create or modify `Node` objects directly, leaving it to the BT-TCG core to do so. The class resides in the `tree` package.

There are 6 fields of a `Node` object that may be of interest, with their accessors:

```
String getTag(),
String getComponent(),
String getBehaviourType(),
String getBehaviour(),
String getFlag(),
Integer getBlockIndex().
```

A `Node` object normally belongs to a `block` object (see Section 5.3). The method `getBlockIndex()` produces the index for the block to which the node belongs.

In addition to the accessors, you may be interested in the methods for equality testing and converting to string:

```
Boolean equals(Object otherNode),
String toString().
```

5.2 NodeProfile

An object of the `NodeProfile` class is intended to be used as a pattern for matching BT nodes. As with `Node` objects, you SHOULD NOT create or modify `NodeProfile` objects directly. The class resides in the `tree` package.

There are 4 fields of a `NodeProfile` object that may be of interest, with their accessors:

```
String getComponent(),
String getBehaviourType(),
String getBehaviour(),
Boolean getKillFlag().
```

A node profile ignores the *flag* of a BT node (as well as the *tag*), except if it is the *kill* flag. The last accessor produces an indicator whether the profile belongs to a BT node with a *kill* flag. A `NodeProfile` object does not belong to any specific `Node` object.

In addition to the accessors, you may be interested in the methods for equality testing and converting to string:

```
Boolean equals(Object otherNode),
String toString().
```

5.3 Block

An object of the `Block` class is intended to represent a BT block. As with `Node` and `NodeProfile` objects, you SHOULD NOT create or modify `Block` objects directly. The class resides in the `tree` package.

There are 5 fields of a `block` object that may be of interest, with their accessors:

```
Integer getParent(),
List<Integer> getChildren(),
Integer getIndex(),
String getBranchType(),
List<Node> getNodes().
```

A block is uniquely identified by its index. Section 6.2 describes how you can access the basic data structures of the BT-TCG core, including blocks.

There is a method to convert a `Block` object to a `String` object:

```
String toString()
```

which shows multiple BT nodes in the case of an atomic composition.

5.4 TestCase

An object of the `TestCase` class is intended to represent a test case (a test path and possibly its prologue and epilogue paths). Currently, test cases are produced by the BT-TCG core (using the `Generate Test Cases` button). There is a facility for producing tailored test cases (see Section 8). You SHOULD NOT create or modify `TestCase` objects directly, except for the `index` field of a tailored test case. The class resides in the `other` package.

There are 6 fields of a `TestCase` object that may be of interest, with their accessors:

```
Integer getIndex(),
Integer getStart(),
Integer getEnd(),
ArrayList<Integer> getSteps(),
ArrayList<Integer> getBlocksBefore(),
ArrayList<Integer> getBlocksAfter().
```

The integer that is the index of a `TestCase` object is used simply to identify a test case from a collection of test cases produced by the BT-TCG core. All of the other integers are block indices. Thus you have access to the starting block, the ending block, and the paths (`getSteps` gives you the main steps, `getBlocksBefore` gives you steps from the root of BT to the beginning of the main steps, and `getBlocksAfter` gives you steps from the end of the main steps to the designated initial state).

The test case represented by a `TestCase` object is a "raw" test case. In contrast, a test case in an Excel file produced by BT-TCG is more polished in that it presents the test case in a human readable format, and with the default format, the focus is on the main test path (some information in the prologue and epilogue are suppressed). The class `SaveExcel` in the `excel` package shows how ready-to-use test cases can be produced from raw test cases.

5.5 TestSegment

Individual test cases (each in isolation) are represented using the `TestCase` class, described in Section 5.4. BT-TCG has a facility to help the test planner produce an overall test plan: a sequence of steps that cover all or a subset of the test cases. The sequence of steps is broken into segments, with each segment representing a test case. In the context

of a test plan, steps before and steps after in a test case are irrelevant; instead, we are interested in the in between steps (steps that fill a possible gap between consecutive test cases in a test plan). The `TestSegment` class is intended to represent test segments.

There are 3 fields of a `TestSegment` object that may be of interest, with their accessors:

```
getPreamble(),
getTestCase(),
getRealStart().
```

The method `getPreamble` gives you the steps that fill the gap (if any) between the previous test case and the current test case. The method `getTestCase` gives you the test case associated with the segment. The method `getRealStart` gives you the actual starting step of the test case (this is needed because sometimes the (test case part of a) segment starts from a reversion step.

6 Important public methods of the Main class

6.1 Method for accessing BT Analyser interface

The method for accessing the BT Analyser interface is

```
SBCLPipe getSBCL().
```

You need this if you want to send commands to the BT Analyser. The method in SBCLPipe to send a command is

```
String sendCommand(String command).
```

Thus to send the command `(print-bt)` you simply invoke

```
Main.getSBCL().sendCommand("(print-bt)");
```

which gives you a string containing the XML result. You would need an XML parser to process the result.

6.2 Methods for accessing basic data structures of the BT model

The methods for accessing the basic data structures of the BT model are:

```
Boolean modelIsLoaded(),
String getBTFilePath(),
Integer numberOfBlocks(),
Block getBlock(Integer i),
ArrayList<NodeProfile> getNodeProfiles().
```

The method `modelIsLoaded()` tells you whether a BT model is currently loaded in BT-TCG. The method `getBTFilePath()` gives you the string representation of the path for the BT file loaded. The method `numberOfBlocks()` gives you the number of blocks in the model, and the method `getBlock(i)` gives you the block with index `i`. Finally, the method `getNodeProfiles()` gives you the node profiles for the BT model.

6.3 Methods for accessing test configuration parameters

The methods for accessing the test configuration parameters are static methods of the `Main` class in the `core` package. For check points and nodes of interest, the methods operate on profiles:

```
Boolean isChosenCP(NodeProfile profile),
Boolean isChosenNOI(NodeProfile profile).
```

The methods return `true` if the node profile is chosen as a check point (for `isChosenCP`) or a node of interest (for `isChosenNOI`). Otherwise the methods return `false`.

The method for accessing the initial block (the block that contains the initial BT node) is:

```
Block getInitialBlock().
```

For accessing observables and actions (and whether an action needs preparation), there are methods that operate on node profiles:

```
String getObservable(NodeProfile profile),
String getAction(NodeProfile profile),
String getPrepAsString(NodeProfile profile),
Boolean getPrepAsBoolean(),
```

as well as nodes:

```
String observableResponse(Node node),
String userAction(Node node),
Boolean needsPreparation(Node node).
```

6.4 Methods for accessing generated test cases and the test plan

The methods for accessing generated test cases and the test plan are static methods of the `Main` class in the `core` package. The methods are:

```
ArrayList<TestCase> getTestCases(),
ArrayList<TestSegment> getTestPath().
```

The method `getTestCases()` gives you the test cases generated, while the method `getTestPath()` gives you the test plan constructed so far.

Note that the 2 methods never produce `null`, but can produce an empty `ArrayList`. Thus, to check if test cases have been generated, for example, we can use the test `(Main.getTestCases().size() == 0)`. You can process individual test cases using methods described in Section 5.4, while individual test segments can be processed using methods in Section 5.5.

7 Access to text areas in BT-TCG core

The *Actions* tab and the *Observables* tab of BT-TCG each contains a text area (implemented using the `JTextArea` class) to enter descriptions of user actions / external inputs and observable responses respectively. In some cases, you might want access to these areas, directly from your add-on, or using popup menus, to modify their contents.

An example of using a popup menu to modify a text area would be to insert text into the text area, based on a template applied to the selected BT node, when you select an entry in the popup menu for the text area. The template can be specific to your add-on so, for example, if your add-on is to generate Java code, then the template can be made to produce Java code snippet.

An example of modifying a text area directly from your add-on would be to automatically fill in the text area based on a dictionary in your add-on.

7.1 Methods for Actions tab

The following methods are static methods of the `Main` class in the `core` package:

```
void insertUATextInput(String text),
void setUATextInput(String text),
void appendUATextInput(String text),
NodeProfile getSelectedUAProfile().
```

For popup menu operations, you would want to insert text into the text area of the Actions tab at the cursor position, using `insertUATextInput(text)`. However, for more general use from your add-on, you can also set the entire text area or append text at the end of the text area.

You can get the profile that is selected in the Actions tab using `getSelectedUAProfile()`.

7.2 Methods for Observables tab

The following methods are static methods of the `Main` class in the `core` package:

```
void insertORTextInput(String text),
void setORTextInput(String text),
void appendORTextInput(String text),
NodeProfile getSelectedORProfile().
```

As with the text area of the Actions tab, for popup menu operations, you would insert text at the cursor position. As with the Actions tab, you can also set the entire text area or append text at the end of the text area.

You can get the profile that is selected in the Observables tab using `getSelectedORProfile()`.

7.3 Popup menus

The popup menus for text areas of the Actions tab and the Observables tab are managed from the `Launcher` class using the following fields:

- `JPopupMenu popupForUA`,
- `JPopupMenu popupForOR`.

You can modify the popup menus by adding appropriate code that modify the above fields in the `Launcher` class. You would need some knowledge about the `JPopupMenu` class.

8 More advanced functionalities

Many of the information you need are already provided by the BT-TCG core. However, if you need more advanced analysis from the BT Analyser, you can call:

```
Main.getSBCL().sendCommand(command);
```

You will need to pass the BT Analyser command as a string and the method will return the resulting XML response as a string. If an error occurs, the method returns an error string instead of an XML string.

To have your add-on perform the more advanced analysis, you would need a good knowledge of the functionalities of BT Analyser and use an XML parser to parse results from BT Analyser.

A simpler interface is provided if all you want the BT Analyser to do is generate tailored test paths. The `Main` class provides the following static method:

```
ArrayList<TestCase>
   sendTestCaseCommand
     (Integer start, ArrayList<Integer> nois, Integer end).
```

The parameter `start` specifies the index of the starting block, `nois` specifies the block indices of the nodes of interest, and `end` specifies the index of the ending block. This method allows you finer control of the test case generation, especially with respect to the nodes of interest. This can be useful because the test case generation may be incomplete if you specify more than 1 node of interest (the more nodes of interest, the less complete the test case generation). If you generate test cases using this method, you can set the index field of each test case generated using the `setIndex(Integer value)` method of the `TestCase` class.