

UI Clicker

- VCC -

First release: September 2022. Updated: Oct 2024

Table of contents:

1. Introduction
2. Action options and action editor
3. The debugger
4. The HTTP API
5. Testing
6. Clicker client
7. Application security
8. Other

1. Introduction

UIClicker is a development tool, used for automating UI interactions with different desktop applications. It features searching for controls (buttons, listboxes etc.), image recognition, executing apps, clicking on controls, setting control text etc. UI interactions are organized into actions, which can be further organized into callable action templates (lists of actions). UIClicker can interact with local applications (on the same machine), or it can execute actions on another instance from a remote machine. Actions can be executed automatically, or can be run using a built-in debugger.

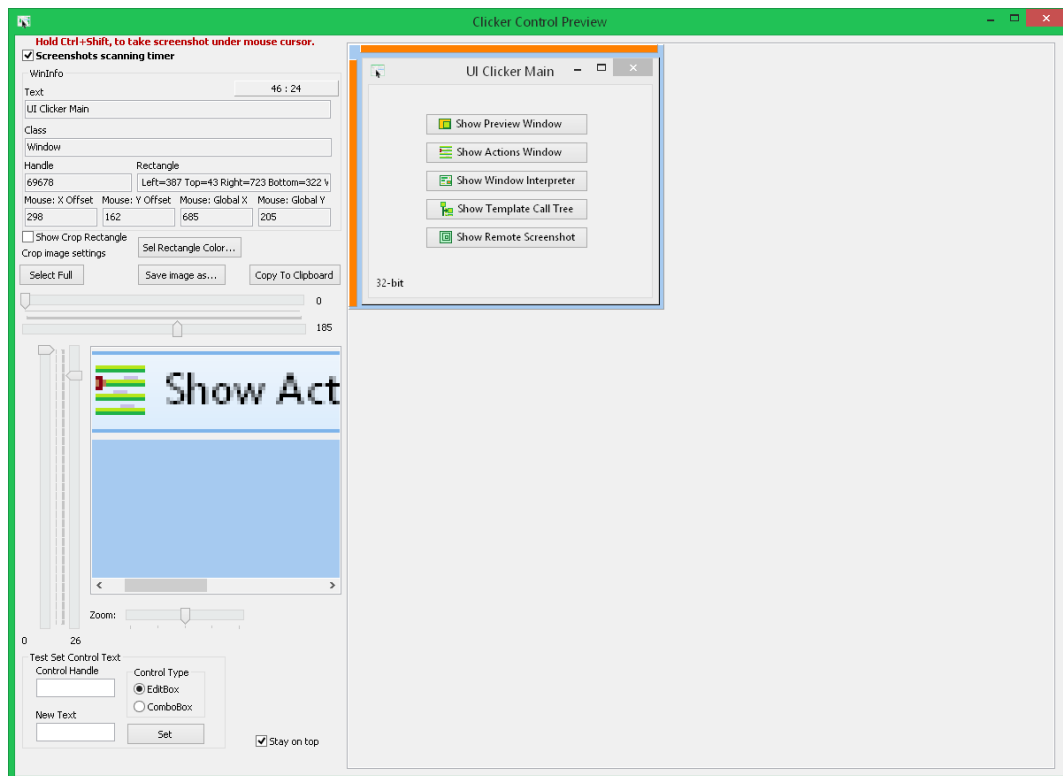
The tool comes as a single executable (can be 32-bit or 64-bit), with multiple built-in subtools:

- "Preview window", used for taking screenshots and displaying window/control details.
- "Actions window", the action editor, which includes action execution engine and debugger.
- "Window interpreter", can scan a window (or a part of it) and create a map of controls.
- "Template call tree", can take a list of action templates and display them as a call tree.
- "Remote screenshot" window, displays a screenshot of the remote desktop, where a second UIClicker is running.

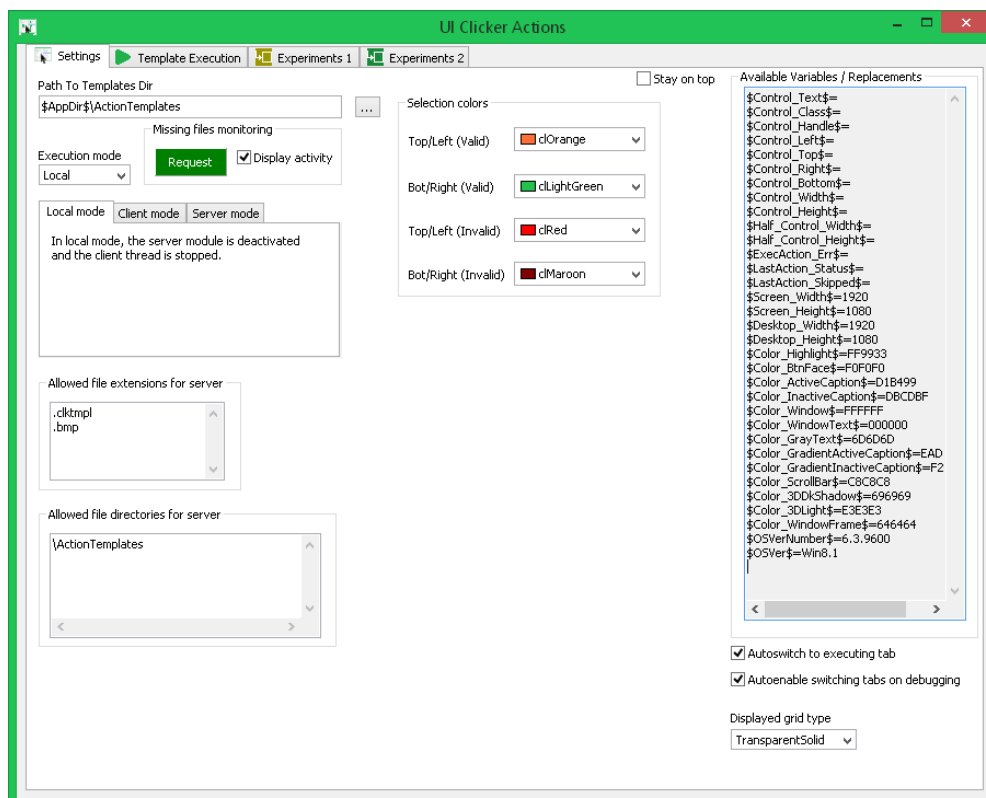


The application can be compiled for 32-bit or 64-bit and it comes as a native Windows application, so it won't need extra frameworks, runtimes or redistributables. It uses the basic win32 API, so it can run even on older versions of Windows (probably down to Windows XP).

To automate the interaction with an application, most of the operations will be done on the "Actions" window. A common interaction pattern is to search for a window, then search for one or more controls on that window and then click or set text on those controls. The Actions window has a small "tool", which can be used to get basic control information from another window, but when users want more information about a control (like current handle, position and size), they can use the Preview window. This is also required when the target window (the one which is automated) is designed to autohide when it loses focus. The Preview window can get window/control information without being focused. It can also be used to display a window/control with zoom.



UIClicker can also search for subcontrols (also buttons, checkboxes, listboxes etc), which are interactable areas without a handle number, allocated by the operating system. These subcontrols can't be found on a window, using the same win32 API used to find controls. They can be found using image recognition, i.e. searching for a predefined text or screenshot of a subcontrol.



The screenshot of a subcontrol, can be taken using the Preview window or the Window Interpreter. All the screenshots, used for subcontrol searching, have to be available as bitmaps, at action execution time.

The Actions window is organized into multiple pages, the first of which, being "Settings". Most of the application settings, displayed on this page, involve the "remote execution" feature. From the "Execution mode" combobox, users can set where the actions will be executed (on the local machine, targeting local apps, on the local machine as a client, and on a remote machine as a server). There are also two editboxes, which allow users to select which file types and file locations can be used when transferring files from client to server, during template execution.

Action templates can be located anywhere on disk, and be specified as full paths when called, but they can also reside in a "Templates Dir", configured from "Path To Templates Dir" editbox, from where they can be called, without a path.

On the same settings page, the application displays a list of available variables, which can be used during action execution. Some of them are automatically initialized on application startup.

The second page, "Template Execution", shows a list of actions, for the currently loaded template, and an action editor. During action execution, multiple templates can be loaded automatically and displayed as subpages, depending on the call stack (see "Main Player" page).



There are 13 types of actions: Click, ExecApp, FindControl, FindSubControl, SetControlText, CallTemplate, Sleep, SetVar, WindowOperations, LoadSetVarFromFile, SaveSetVarToFile, Plugin and EditTemplate. Each of these, has various settings, which can be edited from the action editor (the lower part of the Actions window). For example, the "Click" action can be configured to emulate a click event at absolute screen coordinates, or on a specific control on a window. It can include x/y offsets, can use internal variables, and can emulate a multi-click or a mouse drag event. Every action can be set to be executed or skipped, by a condition (see "Condition" page). Also, after executing a Find(Sub)Control action, in case the search criteria didn't result in a match, i.e. the control is not found, the "Debugging" page may display some useful information. The lower part of the "Template Execution" page is a logger/console, which displays debugging information. It can also be used to quickly evaluate variables or built-in functions. There are also two built-in template editors, under the "Experiments" pages, which can be used to edit a template, while working on another one, in the main editor.

The Window Interpreter is a tool, used for creating a "map of controls", as they exist on the target application. It can do this on a local window, or on the remote machine, where UIClicker is running in server mode.



The yellow panel, in the upper left area of the window, is a "drag panel", which is used to get the basic information from a target window/control, that is to be "interpreted". The process is also called "recording". The recording of a window/control consists of getting the OS allocated handles on every point on that window/control, to determine where the controls are. They are displayed, both as a map of controls and as a tree. Similar to the Preview window, the screenshot of the selected control can be copied to clipboard or saved to bitmap file (right-click on screenshot for pop-up menu). The selected control on screenshot, is "highlighted" by selection lines, which can be manually adjusted. In addition to that, the tree can be saved as a binary file, which can be loaded later. The tree can also be exported as a yml file. When saving a tree, the currently loaded screenshot and control maps are also saved as png files. If they are available as files, when loading a tree, they are loaded as well. From the arrow button, next to the "Start Recording" button, users can record a window from a remote machine, or record a window with subcontrols on the local machine. Searching for subcontrols works by automatically moving the mouse over a window and looking for changes in appearance. The process is imperfect, i.e. it will miss some subcontrols or find non-existent subcontrols. It is also time consuming and CPU demanding.

The Template call tree window is a small tool, which attempts to display a call tree out of multiple loaded templates. It may be useful when working on a project with many templates and it becomes hard to figure out how they are called. In case of recursive calling, the templates may end up being removed from the tree, because they may be counted multiple times.



The last tool, which can be displayed from UIClicker's main window, is the "Remote Screenshot" window. This is required by Window Interpreter, when recording a window on a remote machine. The selected control, from the Remote Screenshot window is going to be the "root" control in the tree, generated by Window Interpreter. Clicking the "Refresh screenshot" button, will take a screenshot of the remote machine and display it on "Remote Screenshot" window. Depending on the "Mouse tool" combobox selection, clicking on the screenshot can select a component (window or other control), or can send the "MouseDown" / "MouseUp" events to the remote machine. The arrow button, next to "Refresh screenshot", opens a pop-up menu, with an item, which when set, allows automatically refreshing the screenshot after a "MouseUp" event.



2. Action options and action editor

As mentioned in the previous sections, actions are organized into action templates. These templates can be saved to files and called from other templates. The flow control is simple. It features conditions for every action, to decide if an action is going to be executed or skipped. Conditions are defined as boolean values, evaluated from existing variables. The execution is mostly linear, without explicit loops. To execute one or more actions multiple times, can be done either by copying those actions multiple times in a template, or calling the template recursively. The CallTemplate action has support for calling a template in a loop, both as a for and as a while.

There is only one variable namespace in a template call chain, i.e. the same list of variables is passed to a template when called, and updated back on return. The experiment pages from the Action window, have their own variable instances. When executing remote actions, the list of variables from server, is overlayed on top of the existing local (client) list. This happens after every explicitly executed action. Variables are automatically evaluated when used in action execution.

Currently, the whole automation process is done with the following actions types:

Click, ExecApp, FindControl, FindSubControl, SetControlText, CallTemplate, Sleep, SetVar, WindowOperations, LoadSetVarFromFile and SaveSetVarToFile. Each of these actions has multiple options, which can be set from the action editor.

Only one action can be edited at a time, using the ObjectInspector (the left part of the action editor) and the extra editors (the left part of the action editor).

As can be seen in ObjectInspector, all actions have the same common set of properties and an action-specific set of properties.

An action can be added to the list, in multiple ways. The action palette can be displayed from the "Action Palette" button (the one with the yellow arrow), from where, an action can be dragged to the list of actions. Once in the list, it can be edited. The second way of adding an action, is from the "Add" button and its arrow button. This will add the current content from the action

editor as a new action. The third way is to paste actions from clipboard. Depending on the current focusing state, the list of actions may not always recognize Ctrl-V as a keypress. While editing an action, the "Update" button displays a red text, indicating that the content in editor is modified, but not the action. When clicking "Update", the content from the editor is updated in the list of actions.

2.1. The "Click" action

This action would usually be found as the final action in a chain of actions, involving the interaction with a control. A click event can be emulated at absolute screen coordinates, at a control left/top or right/bottom coordinates, or some x/y coordinates given by user defined variables. It can be done with the left, right or middle mouse buttons. The event can be a click, a drag, a simple mouse move, a single mouse down or mouse up, or a mouse wheel event. Usually, the mouse cursor is returned to the initial position (the one before executing the action). Optionally, the mouse cursor can be left at the target coordinates. It may be required as the proper behavior of different controls, or simply as a debugging option. A double-click may be emulated as two clicks. A MouseWheel event can be used to scroll a page. Its "MouseWheelAmount" property has to be set to the number of scroll points (a positive or a negative value, depending on scroll direction).

The duration of "mouse-down" and the duration of "mouse-move" (including dragging), can be configured through properties. They apply to Click and Drag operations only. As limitations, the horizontal mouse wheel (tilting) might not be supported by all target applications.

| Property | Value | DataType |
|-------------------------------|------------------|----------|
| Action specific | | |
| XClickPointReference | xrefLeft | Enum |
| YClickPointReference | yrefTop | Enum |
| XClickPointVar | \$Control_Left\$ | String |
| YClickPointVar | \$Control_Top\$ | String |
| XOffset | 4 | String |
| YOffset | 4 | String |
| MouseButton | mbLeft | Enum |
| ClickWithCtrl | False | Boolean |
| ClickWithAlt | False | Boolean |
| ClickWithShift | False | Boolean |
| Count | 1 | Integer |
| LeaveMouse | False | Boolean |
| MoveWithoutClick | False | Boolean |
| ClickType | Click | Enum |
| XClickPointReferenceDest | xrefLeft | Enum |
| YClickPointReferenceDest | yrefTop | Enum |
| XClickPointVarDest | \$Control_Left\$ | String |
| YClickPointVarDest | \$Control_Top\$ | String |
| XOffsetDest | 7 | String |
| YOffsetDest | 7 | String |
| MouseWheelType | mwvVert | Enum |
| MouseWheelAmount | 1 | String |
| DelayAfterMovingToDestination | 50 | String |
| DelayAfterMouseDown | 200 | String |
| MoveDuration | -1 | String |










There are also options to emulate key events for Ctrl, Alt and Shift, while executing the click events. Most of the properties accept variables as values. Users can hover the property values with the mouse cursor, to display their tooltips. They mention whether they accept variables or not.

2.2 The "ExecApp" action

Executing other applications may be required as part of the automation, or simply to do some operation, which UIClicker can't do. The executed application can receive a string through StdIn and will store the StdOut in "\$ExecAction_StdOut\$" variable. Because the list of variables uses CRLF as delimiter for its internal structure, the content of "\$ExecAction_StdOut\$" variable is automatically modified, to replace CRLF occurrences with ASCII 4 and ASCII 5. A similar format is expected from the string passed through StdIn. A future version of UIClicker may have a different structure for the list of variables, where no delimiter is needed. Users can call the following two built-in functions, to do the CRLF <-> #4#5 conversions:

```
"$FastReplace_45ToReturn(<some_string>)$"
```

```
"$FastReplace_ReturnTo45(<some_string>)$"
```

| Property | Value | DataType |
|---|----------------------------------|----------|
|  Common | | |
|  Action specific | | |
|  PathToApp | C:\Windows\System32\ipconfig.exe | String |
|  ListOfParams | | String |
|  WaitForApp | True | Boolean |
|  AppStdIn | | String |
|  CurrentDir | | String |
|  UseInheritHandles | uihOnlyWithStdInOut | Enum |
|  NoConsole | False | Boolean |

The "UseInheritHandles" option is one of the parameters of CreateProcess function, from the win32 API. It is required when passing data through StdIn and StdOut. It can also create a problem when executing an application, which will have to stay open, if the UIClicker is running in server mode. That is, the executed application will keep open the socket, created by UIClicker as server. This will prevent new instances of UIClicker to listen on its own port number. At the expense of a bit of CPU overhead, the socket reuse is switched off, so the above behavior should not be present. When the "NoConsole" property is set to True, executed console applications will not be displayed.

2.3 The "FindControl" action

This is one of the most common actions, users will have to work with. UI interactions usually start by searching for a window (which is also a control), then following a chain of controls on that window, end up to the desired control, which has to be clicked, or having its text box set to a specified value. This action can find controls only, i.e. components which have a handle number, allocated by the operating system. It is a fast operation, because it can rely on available win32 API. A control can be searched on the whole screen (this is usually the case when searching for a window), or it can be searched in a specific area, usually, the rectangle of the parent control/window. A control is most of the times, identified by a class and/or text and is found in a specific area. When multiple controls can be found by the search criteria, users have to constrain the criteria by limiting the search area.

UIClicker has limited support for returning multiple controls. By default, the first found control will be returned as the search result. This happens when the "GetAllControls" property is set to False. The action automatically sets multiple variables to the first found control (like '\$Control_Handle\$', '\$Control_Text\$', '\$Control_Class\$', '\$Control_Left\$', '\$Control_Top\$', '\$Control_Right\$', '\$Control_Bottom\$', '\$Control_Width\$', '\$Control_Height\$' etc.).

When the "GetAllControls" property is set to True, FindControl does not stop on the first found control. It continues searching for matching controls until it times out. Since multiple controls are expected to be found, the result will be a list of control handles. They are placed as #4#5-separated list of numbers, in the '\$AllControl_Handles\$' variable. For example, it can have the following value: 594196 | 332530 | 266902 | . The above mentioned variables are still set to the first found control. To work with the other found controls, the Left/Top/Width/Height variables would have to be set to their suitable values. This can be done, from the "SetVar" action, by calling the '\$UpdateControlInfo(<Handle>)\$' function. Its argument has to be one of the desired handles. The list can also be iterated, using the "CallTemplate" action. The number of items from the returned list, can be obtained by calling '\$ItemCount(\$AllControl_Handles\$)\$', by "SetVar" action. Individual items are indexed in a similar way, using '\$GetTextItem(\$AllControl_Handles\$, <Idx>)\$'. When FindControl was set to return multiple controls, the best results were obtained in "sfcmGenGrid" mode (see "SearchForControlMode" property, from "MatchCriteria" property).

Warning: the EnumerateWindows option may not work, because the callback specified by EnumWindows (see win32 API) function, receives a lot of invalid window handles from the operating system, and sometimes, no valid handles at all.

Currently, the "sfcmEnumWindows" has no implementation for multiple results, since it doesn't work properly, anyway. The last option, "sfcmFindWindow" will return multiple results, only if it has multiple text or classname values to search for (e.g. multiple strings are comma separated). If there are multiple controls with the exact text and classname, only one result is returned.

Some parts of the editor are shared with the FindSubControl action, thus all of the options will be displayed on both action types. Eventually, they may be separated in a future version of the application. For example, under the "MatchCriteria" section, the "WillMatchText" and "WillMatchClassName" are FindControl specific options, while "WillMatchBitmapText", "WillMatchBitmapFiles" and "WillMatchPrimitiveFiles" are FindSubControl specific. Similarly, the same "WillMatchText" property, is used both for FindControl and FindSubControl actions.

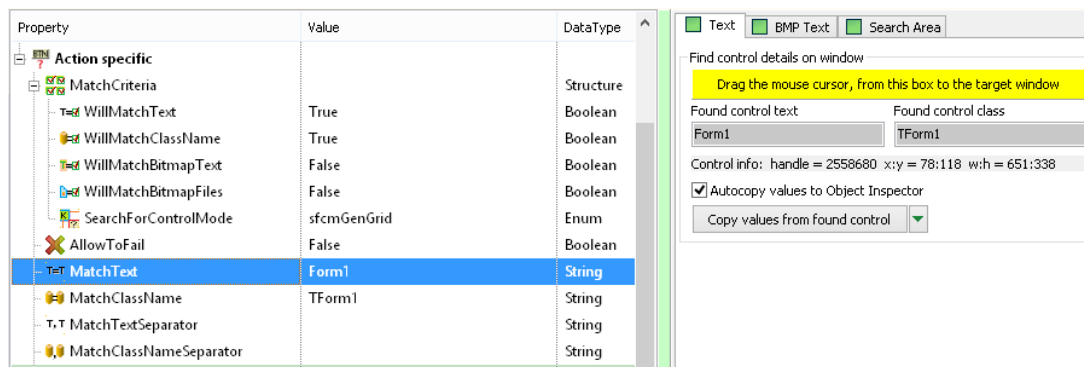
These actions are allowed to fail, because sometimes, a control is expected not to exist. When a Find(Sub)Control is set to be allowed to fail, and it fails, it sets its execution result as "Allowed Failed". This response can be used for flow control. There is also an option to wait for a control to be hidden or to be destroyed. It can be used to wait for a dialog box to stop indicating that some operation is in progress.

Since the Find(Sub)Control action executes in the UI thread, UIClicker can be blocked if the target application is not responding. The default search mode, set by "SearchForControlMode" property, is set to generate a grid, as it works both for windows and other types of controls. The other two use window specific APIs. All of the three options are affected by non-responding windows.

| Property | Value | DataType |
|------------------------------|-----------------------------|-----------|
| Action specific | | |
| MatchCriteria | | Structure |
| WillMatchText | True | Boolean |
| WillMatchClassName | True | Boolean |
| WillMatchBitmapText | False | Boolean |
| WillMatchBitmapFiles | False | Boolean |
| SearchForControlMode | sfcmGenGrid | Enum |
| AllowToFail | False | Boolean |
| MatchText | | String |
| MatchClassName | | String |
| MatchTextSeparator | | String |
| MatchClassNameSeparator | | String |
| MatchBitmapText [0..0] | | Array |
| MatchBitmapFiles | \$TemplateDir\$\SomeBmp.bmp | Array |
| MatchBitmapAlgorithm | mbaBruteForce | Enum |
| MatchBitmapAlgorithmSettings | | Structure |
| InitialRectangle | | Structure |
| UseWholeScreen | True | Boolean |
| ColorError | 0 | String |
| AllowedColorErrorCount | 0 | String |
| WaitForControlToGoAway | False | Boolean |

When using the Find(Sub)Control action, always set a timeout, greater than 0. Otherwise, the action may keep searching for a control, which will never be found. Working with FindControl and FindSubControl, can quickly become frustrating when the desired control is not found. That is because the search criteria may not be properly set, or the search area does not include the searched control. For example, there are applications, which display a variable window title, based on their loaded project(s). For example, a window title may contain the project name, or a "*", or the word "modified". There are cases when the window class may contain a different random number, every time the application is started. UIClicker has support for searching for these types of windows/controls, by allowing wildcards or multiple predefined text/class values to be searched in a single action. The "MatchTextSeparator" and "MatchClassNameSeparator" properties, accept

user-defined strings, which can be used to separate multiple values in the "MatchText" and "MatchClassName" properties. Currently, there is no support for regex.

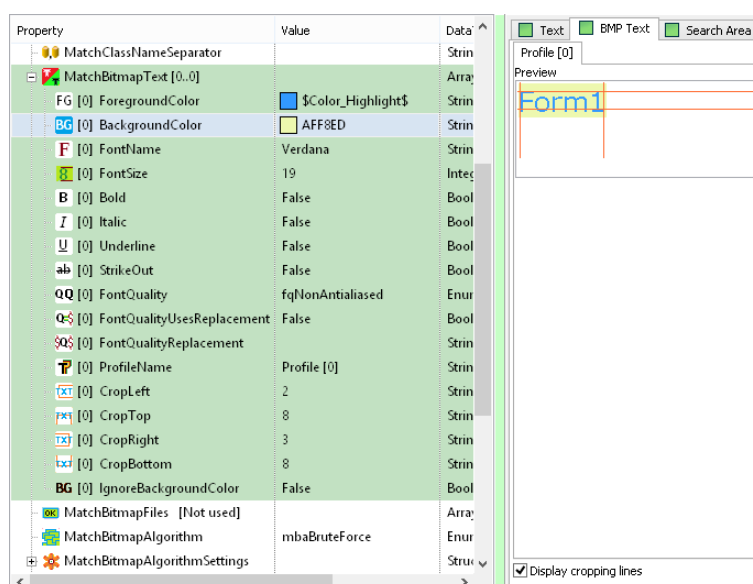


The yellow panel can be used to get a control's text and class, by starting a mouse drag operation from anywhere on that panel, to the target control. When releasing the (left) mouse button, the control's text and class are displayed in the "Found control text" and "Found control class" editboxes. From there, they can be copied to the "MatchText" and "MatchClassName" properties, using the "Copy values from found control" button. Next to this button, there is an arrow button, which opens a pop-up menu, from where similar values can be copied from Preview window or Window Interpreter. It has to be mentioned, that not all controls will have a text. However, all controls should have a class. When searching for a control, which can have any text value, matching text should be disabled (set "WillMatchText" property to False).

2.4 The "FindSubControl" action

The "WillMatchBitmapText", "WillMatchBMPFiles" and "WillMatchPrimitiveFiles" properties are FindSubControl specific options, which configure how to search for a screenshot of a subcontrol. If the subcontrol is "defined" as a text, most of the time, the text can be generated by UIClicker, using the options under "MatchBitmapText" section.

The "image" of the searched text can have multiple font profiles per action, so that it can match a subcontrol in multiple states. For example, focused, unfocused, selected, unselected. Most of the time, text color and background color are standard system colors. UIClicker implements built-in variables, initialized with these color values, for the most common system colors. Two fonts are also commonly found in various system menus and standard controls, which are Tahoma 8 and Segoe UI 9. Depending on application implementation or OS, these fonts may be configured with different "font qualities" (i.e. anti-aliasing), which can also be set, from the "FontQuality" property.

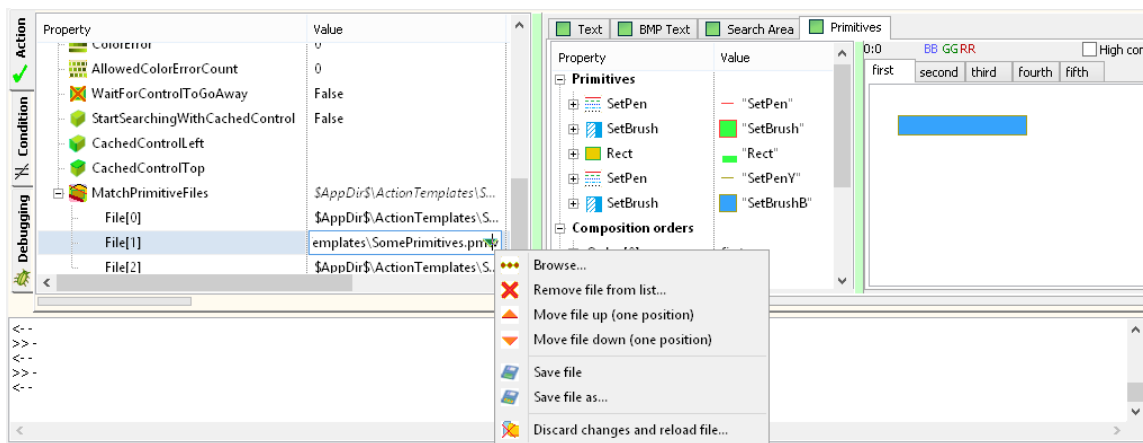


The generated text, as displayed in the "Preview" box, is searched on the window/control, defined by the search area. It can be matched pixel to pixel with 100% or less accuracy. When searching for antialiased text, usually, there won't be 100% match, so users will have to configure an expected search error. This can be done from the "ColorError" and "AllowedColorErrorCount" properties. Unfortunately, UIClicker does not provide an automated way of finding the minimum error level. The searched text can be cropped, so it can be matched even when its edges are being overlayed by a focus rectangle. The searched pixels, of background color (usually text background), can be ignored, if the IgnoreBackgroundColor property is True. It is useful for background with gradients.

The "MatchBitmapFiles" property is a list of bitmap filenames, which contain screenshots of the searched subcontrol. Similarly to the generated text, this allows searching for multiple screenshots of the subcontrol, in its different states.

The advantage of searching for a predefined text, is that it can be easily configured to include multiple font profiles, whenever the operating system settings might influence the text appearance. The disadvantage of using bmp files, is that they have to be present on disk, at the configured location, making the whole project less portable.

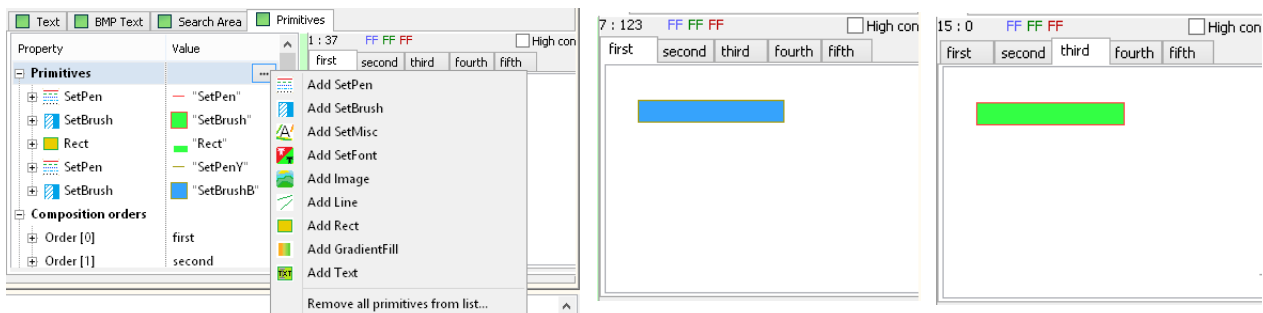
A new feature (still in work) is about searching for bitmaps, "described" as a vectorial content, by various primitives (lines, rectangles, text, image, gradient fill etc.). The option is set as "WillMatchPrimitiveFiles" under the "MatchCriteria" property.



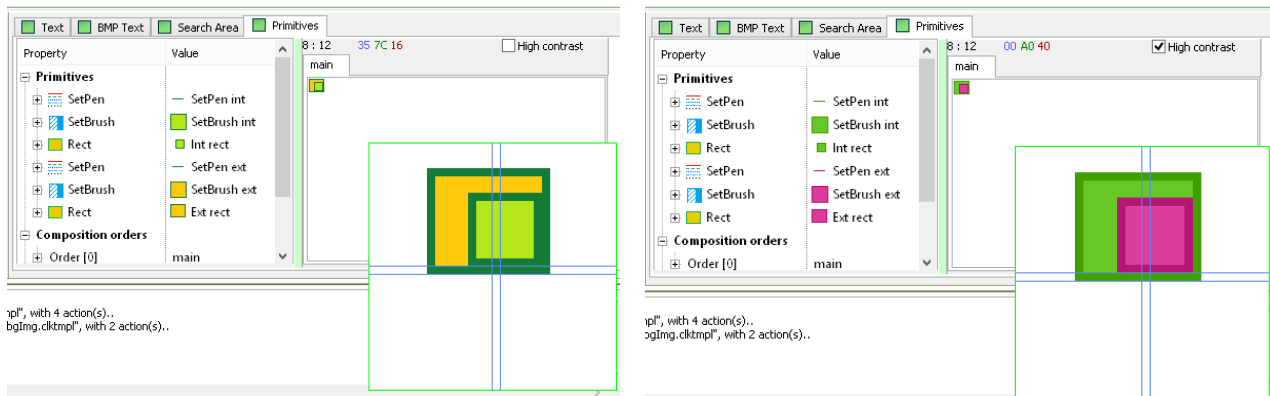
The "MatchPrimitiveFiles" property is a list of primitives files (*.pmtv), which is searched from top to bottom. Adding a new file or an existing file can be done from the "..." button, next to the "MatchPrimitiveFiles" property value.

To load a pmtv file into the editor (on the right hand side of the window), the item has to be clicked in the ObjectInspector ("File[0]", "File[1]" etc items), under "MatchPrimitiveFiles".

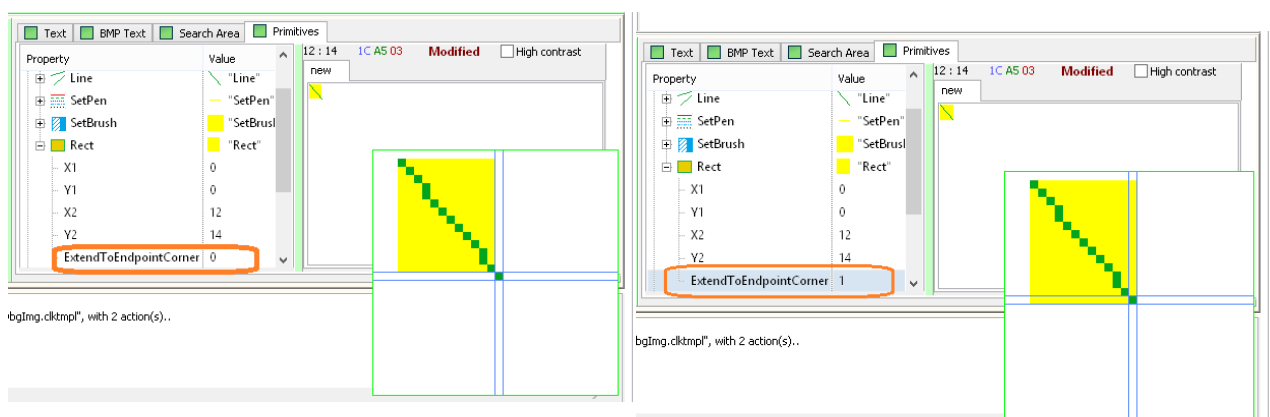
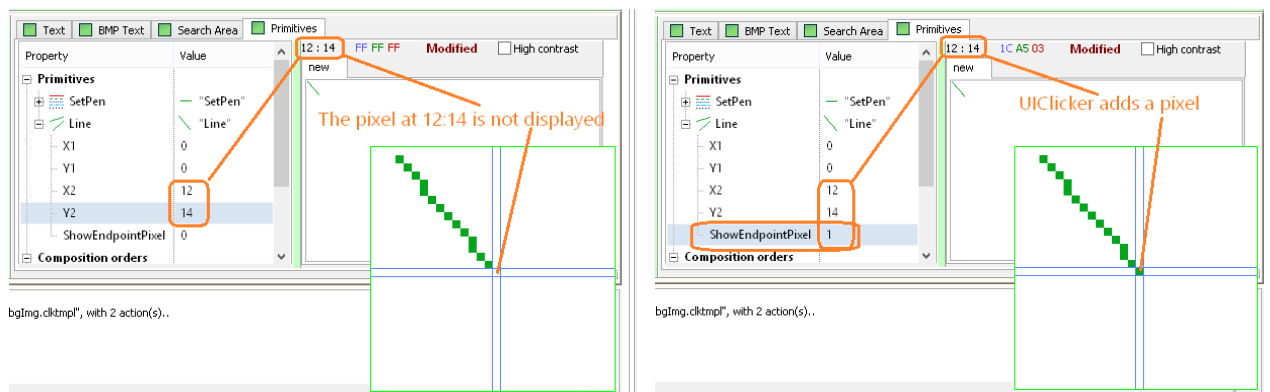
Each primitives file contains a list of drawing instructions, for both primitive properties and primitives themselves. Each primitives file supports multiple composition orders. All the items, under the "Primitives" section, appear under every composition order. This is where they can be arranged in any order they have to be "executed" by the compositor. More than that, the compositor direction can be set top->bottom or bottom->top, from the "Settings" section (the one after "Composition orders"). After all, the last displayed primitive, from a composition, would be the most visible, since it covers all the others. Every order is displayed in its own tab.



Sometimes, it may happen that the colors for primitives, are either identical or very similar. Because of that, it is difficult or impossible to make sense of primitives within a composition. A simple solution would be to preview the composition with totally different colors, without changing their properties. This can be done, by checking the "High contrast" checkbox.

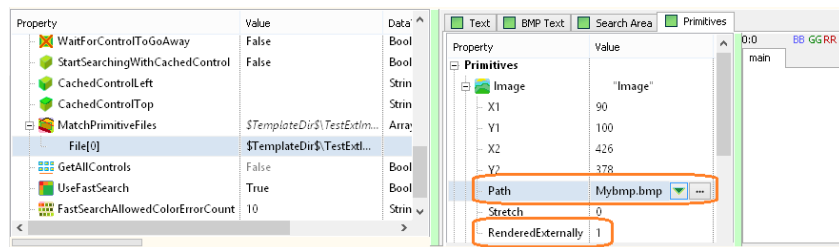


One limitation, which comes from how lines are displayed by the OS, is that the last pixel of a line is not displayed. That is, when a line has its start pixel at $X1:Y1$ and its end pixel at $X2:Y2$, the end pixel does not appear. To overcome this, every line primitive has a property, called "ShowEndpointPixel", which has to be set to 1. A similar problem is found on drawing rectangles. In this case, the property is called "ExtendToEndpointCorner". It often happens that these properties have to be set to 1, if the background color, around the most exterior rectangle, matters when searching for a subcontrol. This is because the last pixel would be invisible, but the final image size is computed from the right-most and bottom-most pixels from all primitives. As a current limitation, some font settings are not applied when computing the final image size.



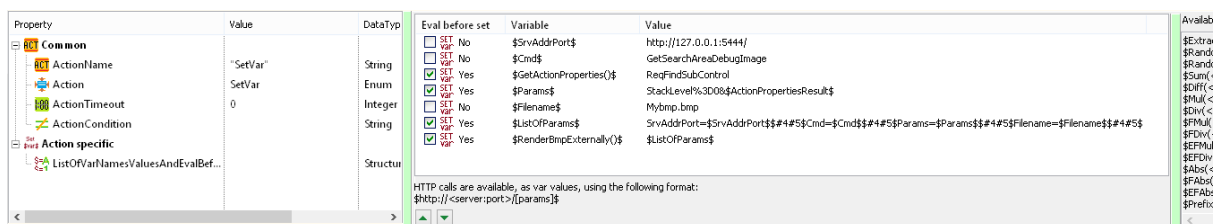
A feature, useful to primitives, is "external bitmap rendering". This allows sending some "rendering" information to a server, through an http request and receive the rendered bitmap, to be used as searched image. It is also available, to be directly used by FindSubControl as the bitmap to be searched on (i.e. the background bitmap, instead of a screenshot). External rendering is useful, mostly for image recognition of a subcontrol which looks differently, based on its state. For example, a trackbar has a slider button, which can be in one of a finite set of positions. By using external rendering, UIClicker can find the subcontrol in one of the possible states (with approximation). It can compose the "image" of the subcontrol, from available screenshots of its different parts (e.g. trackbar rail and slider). Also, external rendering allows creating a searched bitmap, with a complex content, by a server which doesn't have the limitations of UIClicker.

An http request for external bitmap rendering, requires the filename of the bitmap, which will be stored in an in-mem file system for externally rendered bitmaps, when the server responds. For now, only a single bitmap per request is accepted. From multiple such requests, UIClicker can store different bitmaps in memory and use them in FindSubControl actions, via primitives. To use a received bitmap, the "RenderedExternally" property, of an image primitive, should be set to 1. If left to its default (0), the bitmap is loaded from disk.



The actual http request can be done from the SetVar action (described later), using the "\$RenderBmpExternally()" function. It requires both http specific parameters and rendering specific parameters. It doesn't use any special protocol. It is the same type of request as used by the debugger module (see for example, chapters 4.9 or 4.10). The request is synchronous and expects either the rendered bitmap or an error message. If successful, the bitmap is kept in memory. Otherwise, the error message is found in "\$ExternallyRenderedBmpResult\$" variable.

Calling the "\$RenderBmpExternally()" function, requires creating the request string in a variable and encode it (along with rendering parameters) using the \$#4#5\$ separator. Each item is a key-value pair, of the "<key>=<value>" format. Because of this encoding, the "=" character has to be escaped, using its ASCII hexa value of 0x3D (see "%3D" from standard http encoding).

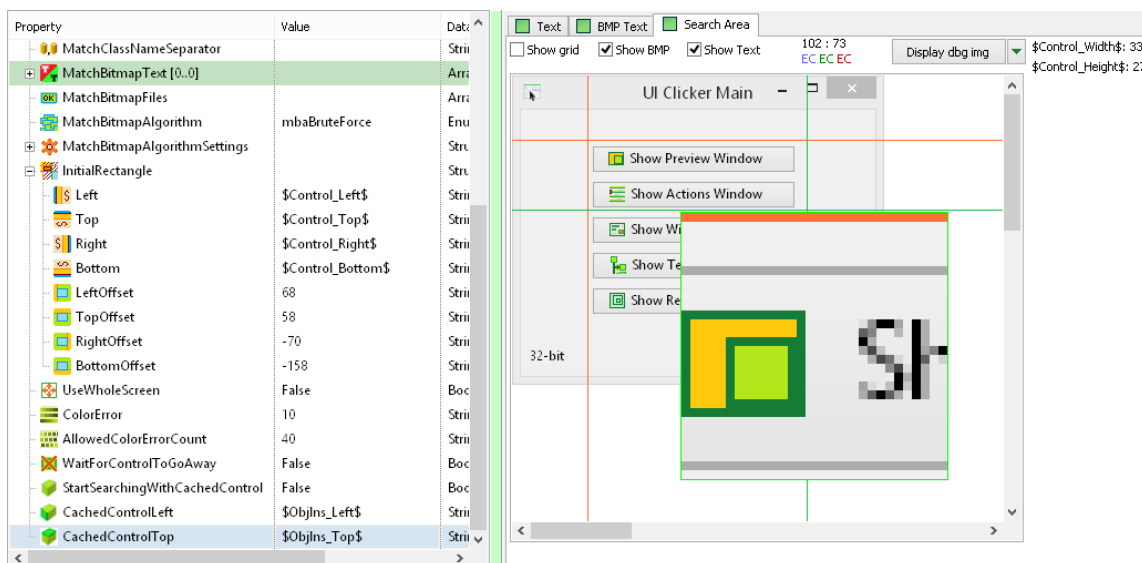


As a requirement, the "\$RenderBmpExternally" function name should be placed in the "Variable" column and its argument, in the "Value" column of a SetVar action. Similarly, the "\$GetActionProperties()" function, is used to encode all the properties of a FindSubControl action, to later become part of the "Params" paramter. This way, the rendering server receives more details.

The "Filename", "SrvAddrPort" and "Cmd" parameters are mandatory. The "Params" parameter is what comes after the "Addres:Port/Cmd" section of a request. Here, the rendering parameters can be included. This parameter is optional (if the server returns a single bitmap only). Another optional parameter, is "IncludeFilenameInRequest", which will cause the filename to be a (sub)parameter of "Params", if set to 1. The values of all these parameters are server specific.

Asynchronous requests are possible, by using the already existing \$http\$ function for starting the rendering, then using "\$RenderBmpExternally", for getting the result (bmp).

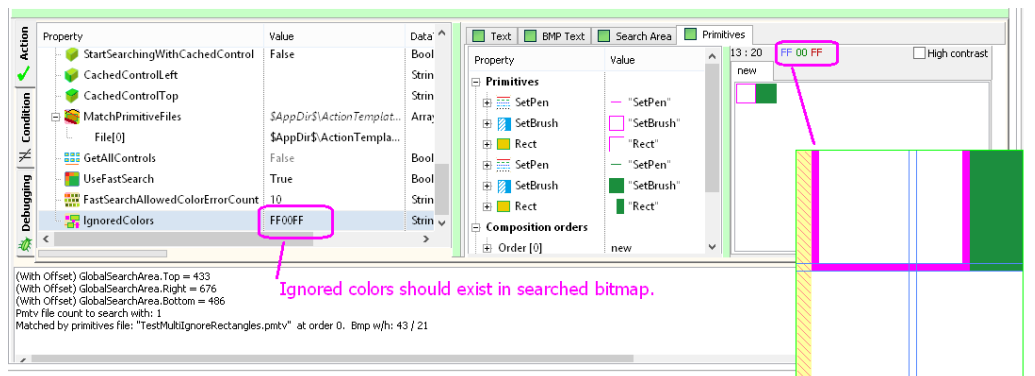
Back to the main FindControl properties, as the name suggests, the "InitialRectangle" section is a collection of properties, defining where to search for a control or a subcontrol. Users will have to take care of the "UseWholeScreen" property, when searching for a window (the whole screen) or when searching for any other control. Sometimes, a control can be found if the property is True, i.e., searched all over the screen, but it is better to limit the search to a specific window or other parent control. The search area is defined by a left/top/right/bottom rectangle. These values are usually found in the "\$Control_Left\$", "\$Control_Top\$", "\$Control_Right\$", "\$Control_Top\$" variables, automatically set by the previous action(s), which searched for a parent control. The search area can be further limited by offsets. Depending on how the searched control or subcontrol is aligned to its parent control/window, the search area can be defined by different variables. This makes the search more versatile and immune to window resizing, since a control is usually aligned to one of the window's edges. The smaller the search area, the faster the (sub)control will be found.



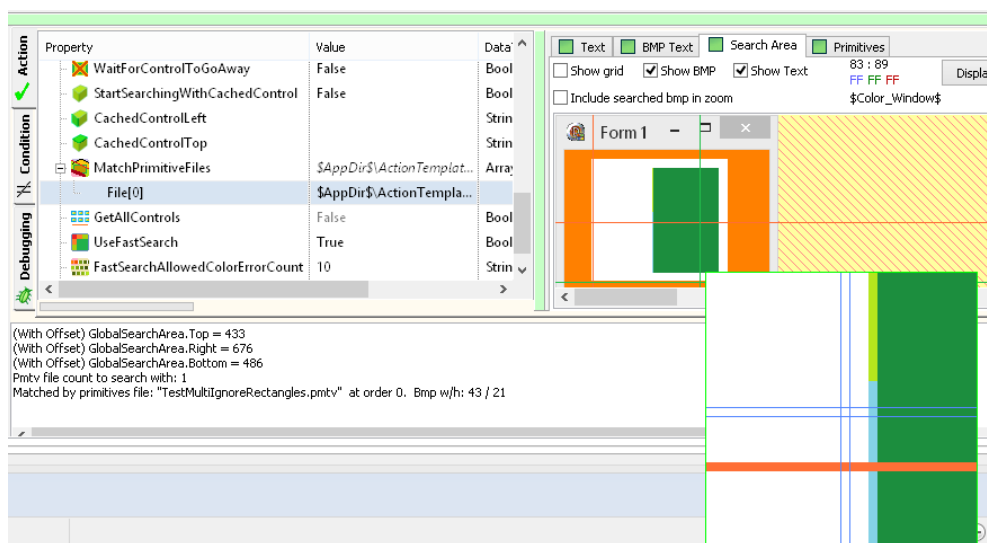
The "Display dbg img" button, takes a screenshot of the search area, using the current configured edges, by the last executed action. That is, when pressing the button, a successful action has to already be executed, which will define the desired search area. Do not execute the currently worked on action, when pressing this button, because that will result in modifying the "\$Control_Left\$", "\$Control_Top\$", "\$Control_Right\$", "\$Control_Top\$" variables, leading to an invalid search area. The "Show BMP" and "Show Text" checkboxes can be used to display either one of the searched texts or bitmap files, overlayed on top of the search area. This helps to visually compare the searched subcontrol, to the searched content. Generating the preview text or preview bmp can be done from the arrow button, next to the "Display dbg img" button. The "Offset" properties from this "InitialRectangle" section, can automatically update the position of the orange-green limit lines, from the preview image. This can be done by holding Ctrl or Ctrl-Shift, while "dragging" one of these values, up or down with the mouse left button. This feature works only for the case when "UseWholeScreen" property is set to False. The "Offset" values are also updated when dragging the lines.

The "ColorError" and "AllowedColorErrorCount" properties, can set the color difference and how many error points are allowed for a successful match. When "UseFastSearch" property is True, a 5px x 5px square (top-left part of the searched bitmap) is searched first, then if that matches, the full bitmap is verified. This approach can speed up the search by 25-100 times the existing one. If using it, it is important to make sure the content of the top-left corner of the search bitmap, to have some unique features, which won't match on every verified coordinates. Since the 25 verified pixels will get a lower number of color matching errors, a different property is required to set the error level. This is "FastSearchAllowedColorErrorCount", and it should be set to a low value.

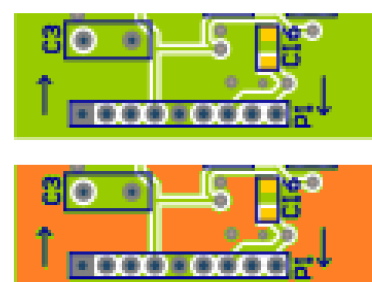
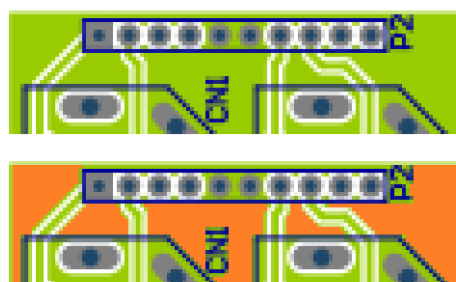
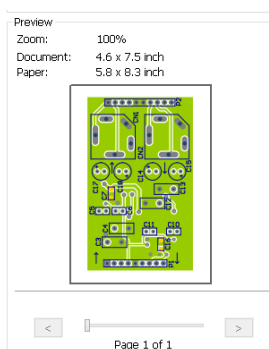
A similar property to "IgnoreBackgroundColor" (from "MatchBitmapText" property), is "IgnoredColors". This can be set to a comma-separated list of colors (6-digit hexa values, or var/replacements), which if found on the searched bitmap, are no longer verified on the search area. For most cases, a single ignored color is enough to be set.



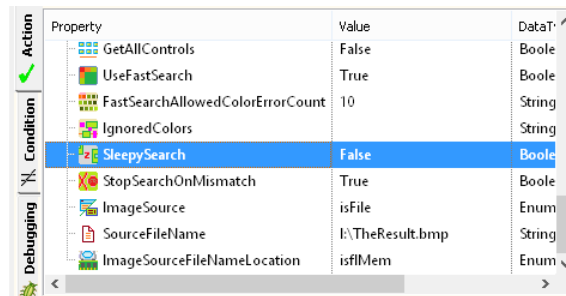
If the searched bitmap already exists as a bmp file, it is possible that multiple ignored colors have to be set to the "IgnoredColors" property. The ignored colors do not have to exist on the searched area. The matching of ignored colors, is affected by "ColorError". This property can be set to 0, for an exact match, or can be greater than 0. When greater than 0, the configured ignored colors do not have to match exactly the ones from the search bitmap. This way, a range on ignored colors can be set around the configured colors.



This feature is useful on finding the edges of an area from a screenshot, when that area is enclosed by antialiasing pixels. Commonly, the colors of antialiasing pixels take a wide range, so they are better to be ignored, rather than be covered by color errors. Example of antialiasing pixels on drawing edges (the background is colored in orange, to show the remaining green edges):



To prevent CPU overhead from the computationally intensive bitmap matching algorithm, the "SleepySearch" property can be set to True. When that's the case, the algorithm randomly calls the OS's "Sleep" function with a random number of milliseconds (which may take up to 40ms). This has been the default behavior prior to implementing this property. Since the resolution of Sleep function is pretty poor at small values, the overall duration of the algorithm is severely affected, although not necessary to keep the CPU overhead that low. This behavior can still be useful when there is a high probability that searching for a subcontrol would fail in most of the times. However, if it's likely to be found, then the property can be safely set to False (the current default behavior). The search duration is many times lower, but it keeps a CPU core to its maximum (no GPU implementation so far).



| Property | Value | DataT |
|----------------------------------|------------------|--------------|
| GetAllControls | False | Boole |
| UseFastSearch | True | Boole |
| FastSearchAllowedColorErrorCount | 10 | String |
| IgnoredColors | | String |
| SleepySearch | False | Boole |
| StopSearchOnMismatch | True | Boole |
| ImageSource | isFile | Enum |
| SourceFileName | E:\TheResult.bmp | String |
| ImageSourceFileNameLocation | isflMem | Enum |

The "StopSearchOnMismatch" property allows finding the minimum pixel error count as the result of a FindSubControl search. However, the value is meaningful only when the searched bitmap is of the same size as the bitmap it is searched on (i.e. the background bitmap (can be a screenshot)). This is useful to compare two bitmaps, to have a numerical difference (the number of mismatching pixels) between them. The default value of this property is False, which causes the search algorithm to stop and set the action result to "Failed". When set to True, the algorithm continues past the number of mismatching pixels, set by the "AllowedColorErrorCount" property, so it can count all of them. In that case, the action result would be set to "Successful".

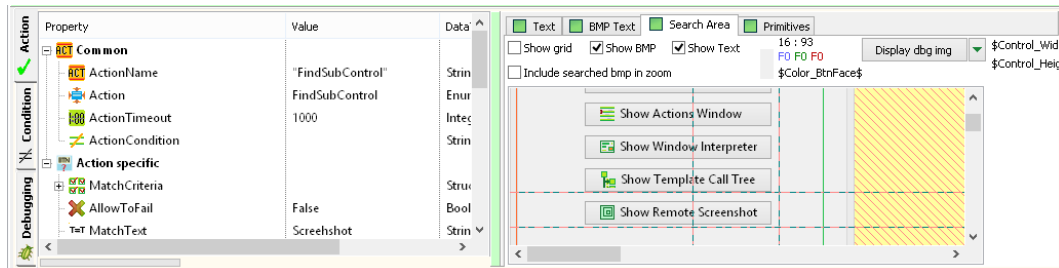
By default, FindSubControl takes a screenshot at the control, identified by the "\$Control_Handle\$" variable, using the cropping settings from the "InitialRectangle" property. This happens if the "ImageSource" property is set to "isScreenshot". In order to use a custom background, instead of a screenshot, the "ImageSource" property can be set to "isFile". It allows a bitmap file to be used as the bitmap to be search on (i.e. the background). This file can be loaded from disk or from the in-mem file system, used for externally rendered bitmaps. When used, the "SourceFileName" property points to the filename of the file to be loaded. If the file is loaded from disk, the filename has to follow the required filename conventions. However, when loaded from memory, it can have almost any possible name.

To control whether the file is loaded from disk or from the in-mem file system, the "ImageSourceFileNameLocation" property can be set to "isflDisk" or "isflMem". It defaults to "isflMem", as would be used by the "\$RenderBmpExternally()" function.

In addition to the above mentioned function, the bitmaps from the in-mem file system, can be saved by action plugins, as part of their API.

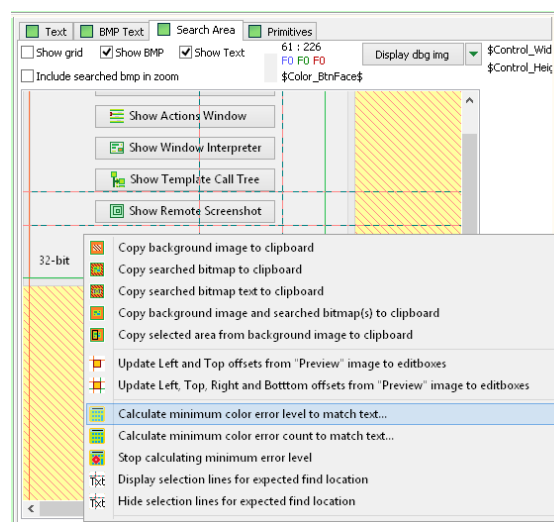
It has to be mentioned that when using the "isFile" settings for "ImageSource", the values from the "InitialRectangle" property are only partially ignored. They are not used, but at the same time they are validated, so that a valid rectangle has to "formed" from those settings. For example, the "Left" and "Top" subproperties can be set to 0, while the "Right" and "Bottom" would be set to 1. The offsets can also be set to 0. This would result in a 1px by 1px wide rectangle to be validated. Since such a rectangle is valid, the algorithm moves further to the actual searching. As opposed to using the "isScreenshot" setting, the bitmap to be searched on would be used as it is, without any extra cropping. It is expected that it is already cropped. This is a limitation of the current implementation.

From the pop-up menu of the preview image, it is possible to search for a minimum color error level and a minimum color error count, which can be used as "ColorError" and "AllowedColorErrorCount" properties of a FindSubControl action. The algorithm for finding these values, is far from perfect, but it is able to get close to some usable results. Although they would work, probably a margin would still be required, so they don't get too tight.



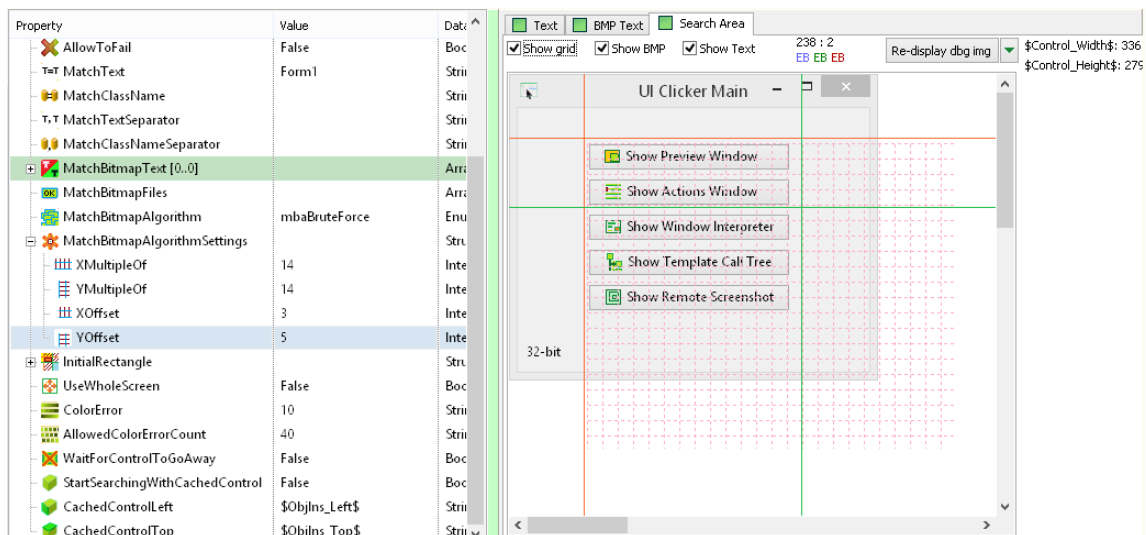
Searching for these two values is done separately, from two different menu items. As input, the search algorithm uses the currently existing (selected) FindSubControl action, but it verifies a range of different values for the searched parameter. It does this by executing the FindSubControl action. For "ColorError", the initial interval is [0..100], while for "AllowedColorErrorCount", it is [0..1000]. The tested value is half of the current interval. If the tested value results in a match, the upper endpoint is updated to the currently tested value. If it fails to match, the lower endpoint is updated to the currently tested value. Thus, the search interval is reduced on each iteration, until either its endpoints are equal, or a match is found.

Because it is possible to get false positives on high color error values or color error count values, the search algorithm has an additional input. This is called the "expected search area", which can be defined by selecting it from the preview image, by holding the Alt key and using the classicMouseDown-MouseMove-MouseUp selection. This area is displayed using additional selection lines (displayed as dashed lines). If no expected area is defined, the default values are set to the entire preview image.



Users can stop the search, either by holding Ctrl-Shift-F2, which also stops the FindSubControl action, or from the same pop-up menu, using "Stop calculating minimum error level" item. As mentioned before, when using the preview image, the previous action has to be executed successfully, to properly set the control variables (\$Control_Left\$, \$Control_Top\$ etc).

Currently, there is no algorithm, which can search for both mentioned properties at once. Usually, the color error level is a small value, somewhere between 0 and 20. The "AllowedColorErrorCount" highly depend on the text length and font size. It also depends on the antialiasing algorithm, which in many cases will generate a different content, on every new text location.



At the moment of writing this, there are only two search algorithms, a "brute force" and a "grid" search. The brute force algorithm verifies the searched bmp (text or file) pixel-by-pixel, starting with the top-left corner of the search area, and it continues, towards the bottom-right pixel. The grid algorithm does a similar search, but only for the points matching a multiple of "X" and "Y" values, defined by the "XMultipleOf" and "YMultipleOf" properties, under the "MatchBitmapAlgorithmSettings" section. The grid can be shifted by different amounts, from the "XOffset" and "YOffset" properties. The grid algorithm is useful when the search area is known to have a structure (like a listbox, tree, menu etc.), displaying items at specific coordinates.

When the "Show grid" checkbox is checked, the preview image shows also the generated grid. There is also an option to cache the search coordinates, so that the search algorithm would verify them first (see "StartSearchingWithCachedControl" property). It has to be mentioned that caching is not efficient when using multiple font profiles, because a cache miss will fall back to searching for text on that particular font profile, before moving to the next one.

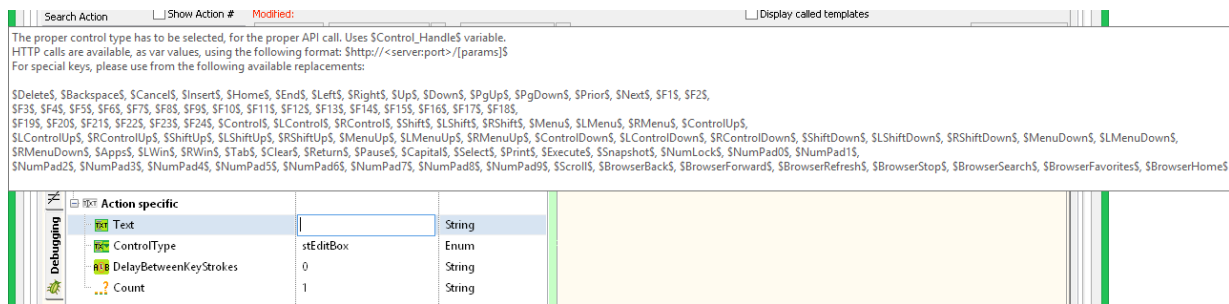
When a control or subcontrol is not found, although "it should be", users will have to double check the match criteria, the expected control text and class, font settings, search area and color error. Since this is a search algorithm, it is very difficult to implement a mechanism to track down what is the cause of missing a searched (sub)control.

The action timeout on a FindSubControl action, will stop the action, only after one full search (the entire search area), which can take longer than the timeout itself. It should not be set to 0.

2.5 The "SetControlText" action

Similar to clicking, the (sub)control can be interacted with, by emulating the keyboard. The most common control this action will target, is the editbox. It comes either as a standard system control, or can be part of a UI framework, where it is more of a subcontrol (does not have a handle number). When targeting a control, the "SetControlText" action may be configured to use the "EditBox" option, which is the fastest. This implies setting the editbox using native win32 API. However, it is possible, that depending on the implementation of target application, simply setting the text, won't trigger any "OnChange" event, needed to properly detect that the editbox content has changed. If that's the case, the action has to be configured, to emulate keystrokes, by selecting the "stKeyStrokes" option. This will trigger any "OnChange" event, configured for that editbox. This option is a bit slower than the "EditBox" one and it requires the editbox to have focus. The "ComboBox" option may be used, when the target editbox is part of a combobox component. It is similar to the "EditBox" one.

Emulating special keys, like Ctrl, Shift, Alt, Esc, Return, F1, F2..., Del, PgUp, PgDn etc, is available when setting the "stKeyStrokes" option to the "ControlType" property. The tooltip of the "Text" property shows what var/replacements can be used for these special keys. The Ctrl, Shift, Alt keys, have replacements like "\$ShiftUp\$", "\$ShiftDown\$", which allow further key combinations.

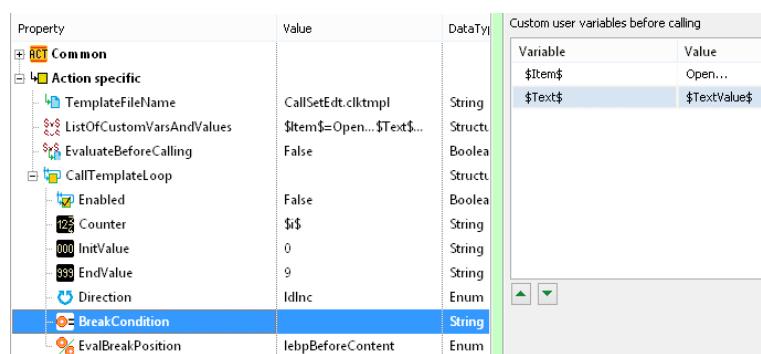


The content for the "Text" property, can be automatically obtained from an HTTP server, which should respond with the expected text, when being requested with a string of the following format: "\$http://<server:port>/[params]\$" (without double quotes). A similar request can be made by the "http" function, on most action properties.

When "ControlType" is set to "stKeyStrokes", the delay between the simulated keystrokes is set by the "DelayBetweenKeyStrokes" property. "Count" sets how many times the text is sent.

2.6 The "CallTemplate" action

As mentioned before, templates can call each other, passing a set of variables and setting new or existing variables when returning. Because the implementation is simple, there is only one namespace, i.e., there is only one set of variables. The ones, mentioned under the "Custom user variables before calling" list, will simply be added to the list of existing variables, right before calling the template. They will remain there, appended to the main list. This is one way of creating new variables. The other way would be from the "SetVar" action.



This action has one option of evaluating the variables, before calling the template. The path to the called template can be an absolute path, or can be omitted. If only the template filename is mentioned, UIClicker assumes the template should be found in the configured template directory (see "Settings" page).

A CallTemplate action can be set to call the "called" template in a loop. It has to be mentioned from the beginning that, at the moment of writing this documentation section, this feature is in work, so not all corner cases are covered yet. Specifically, when remote debugging a looped CallTemplate action, the Stop button should not be pressed, because at server side, the loop will continue, leading to a bad state. However, the debugging should instead continue using the StepOver button as many times as the loop goes. The other uses of a looped CallTemplate action should be working.

The loop can be enabled using the "Enabled" property. It can use a variable as a counter, as in for loops, which should be filled in to the "Counter" property (e.g. \$i\$). This "counter" variable will be automatically set to the value (can be an evaluated variable) from the "InitValue" property and goes up to and including the value (can be an evaluated variable) of the "EndValue" property. The "Direction" property specifies if the counter variable should be incremented or decremented. As a limitation, it can't step over multiple values, i.e. being increased or decreased by 2, 3, 4 etc. When the "EndValue" is greater than the "InitValue", the loop should be configured to go up, by setting the "Direction" property to Inc. Otherwise, it should be set to Dec. If these do not match, the loop stops.

If the user wants to allow UIClicker to evaluate the direction, based on Init and End values, the "Direction" property should be set to Auto.

A loop can be stopped, before having its counter variable reaching the "End value", by a break condition. Using the "..." button, of the "BreakCondition" property, users can open the condition editor, to create or edit a break condition. Using the editor, ensures that the expression syntax is met. It uses the same mechanism as action conditions. The condition can be evaluated before or after making the call to the "called" template. It can be configured from the "EvalBreakPosition" property.

Every variable, modified by the "called" template is added/updated back to the caller's list of variables, allowing them to even be part of a break condition. Unfortunately, the namespace of a "called" template is the same as the caller's, so a "called" template can modify the iteration counter variable. If users modify this on purpose, it can lead to infinite loops.

The "CallTemplateLoop" properties allow various function calls to be evaluated. For example, if the iteration counter "\$i\$", has to iterate from 0 to "\$n\$" -1, the "InitValue" should be set to 0, while the "EndValue" should be set to "\$Diff(\$n\$,1)\$" (no quotes). The "\$Diff()\$" function returns the n-1 value in this case. As with any other property, the ones from a looped CallTemplate, are evaluated at every loop iteration. If, for example, one or both loop endpoints are modified during execution, this has to be taken into account, to both avoid stopping the loop unexpectedly or entering an infinite loop.

When using a looped CallTemplate, the user variables are set on each loop iteration. If some or all of these variables have to be set only once, before the loop, they should be moved to a SetVar action, before the looped CallTemplate action. This is a small inconvenience when converting a CallTemplate into a looped CallTemplate.

By having the loop feature, implemented in CallTemplate action only, it makes the templates a little more clean, as it requires a set of actions to be refactored into a separate template, to actually be called repeatedly. Unfortunately, this limits the readability of a set of actions or keeping track of how variables are modified by a called template.

As an example, the following is the setup of a looped CallTemplate, to iterate through the items of a list with CRLF separated strings, read from a text file:

| Action Name | Action | Timeout [ms] | Condition / Misc | Text |
|--------------|--------------------|--------------|------------------|------------------|
| SetVar | Set \$var\$ SetVar | 0 | \$FileContent\$ | \$FileContent\$ |
| CallTemplate | CallTemplate | 0 | ExpCalee1.c... | ExpCalee1.dktmpl |

| Eval before set | Variable | Value |
|-----------------|-----------------|-------------------------------------|
| SET Yes | \$FileContent\$ | \$LoadTextFile(C:\abc\myfile.txt)\$ |

| Property | Value |
|-------------------|--|
| Enabled | True |
| Counter | \$i\$ |
| InitValue | 0 |
| EndValue | \$Diff(\$ItemCount(\$FileContent\$),1)\$ |
| Direction | IdInc |
| BreakCondition | \$GetTextItem(\$FileContent\$, \$i\$) \$==3... |
| EvalBreakPosition | lebpBeforeContent |

2.7 The "Sleep" action

There are cases when a dialogbox is blinking or automatically resizing, while being displayed, causing UIClicker to miss a searched control.

| Property | Value | DataType |
|-----------------|---------|----------|
| Common | | |
| ActionName | "Sleep" | String |
| Action | Sleep | Enum |
| ActionTimeout | 0 | Integer |
| ActionCondition | | String |
| Action specific | | |
| Value | 1000 | String |

Use this action only as a last resort (e.g. blinking)

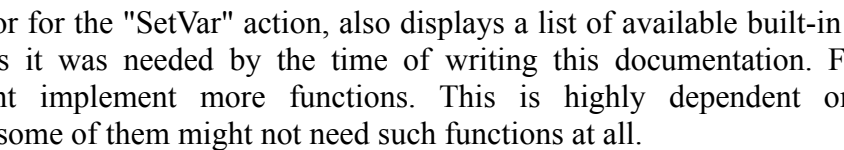
Elapsed Time [ms]:

Remaining Time [ms]:

This is a type of race condition, because the searched control is there, but it is still inaccessible. Most of the time, there is no proper way of waiting for a stable state of that dialogbox, so only a

While executing, the remaining time is displayed on the action editor.

There are cases, when one or more variables will have to be set, before executing some other actions. The "SetVar" action allows, both setting existing variables to new values, or creating new variables, by simply adding them to the list. By how they are used, the UIClicker variables are also called replacements, as they are automatically evaluated and replaced with their values when used. Variables are defined as their names, enclosed by two "\$" characters. Their values, can be simple strings, other variables, or built-in functions. Setting a variable to another variable can be evaluated "in-place" or later, depending on how it is configured.



and are used when converting from ASCII #4#5 to/from CRLF. Storing the ASCII #4#5 characters instead of CRLF, allows proper use of the list of variables, which can't store strings containing CRLF. The "\$Current_Mouse_X\$" and "\$Current_Mouse_Y\$" functions return the live position of the mouse cursor, at the time of calling. The "\$CRLF\$" function returns the ASCII #13#10 characters, i.e. the "return" string, as used on MS Windows. The "\$#4#5\$" function returns the ASCII #4#5 string. The "\$Now\$" function returns a timestamp, at the call time. To stop the execution of the current template, users can add the "\$Exit(<ExitCode>)\$" to the list. Using "\$CreateDir(<PathToNewDir>)\$", a new directory is created. It also creates parent directories.

If a control handle is already known, the "\$UpdateControlInfo(<Handle>)" function can update all the "\$Control_<property>\$" variables (e.g. \$Control_Text\$, \$Control_Class\$ etc.) with the information about that control (text, class name, position, size etc). For example, if calling

"\$GetSelfHandles()\$" (as part of a SetVar action), "UpdateControlInfo" can be used to set the "\$Control_<property>\$" variables to one of the UIClicker's windows. This is done by passing one of the generated/updated variables (e.g. \$frmUIClickerMainForm_Handle\$) to "UpdateControlInfo". Using self handles, is a requirement when testing UIClicker.

There are OS desktop themes or other mechanisms, by which applications display text, using slightly altered system colors. Because of that, UIClicker provides \$IncBrightness\$ and \$DecBrightness\$ functions (and their R, G, B subsets). Users can pass system colors to these functions, then use the result as the expected colors when executing a FindSubControl action.

Although some of the above functions look like ordinary variables, they should not be used to store custom user values. If for example, "\$Current_Mouse_X\$" is assigned a value from the console, it is added as a variable to the list of variables, and its value would be returned before being evaluated as a function.

The "SetVar" action also has a "FailOnException" property, which can be used to cause the action to fail, when one of the above function raises an exception. This is usually the case with http requests (either the "\$HTTP\$" function, or the "\$RenderBmpExternally()\$" function or some other similar functions). When set to True, the exception message is placed in "\$ExecAction_Err\$" variable (in addition to, optionally, a function-specific result variable). For example, if the "\$RenderBmpExternally()\$" function cannot connect to the configured server, it sets the "\$ExternallyRenderedBmpResult\$" variable to "Client exception: Connect timed out." error message. The same error message is copied to "\$ExecAction_Err\$".

2.8 The "WindowOperations" action

UIClicker provides three simple window operations, to directly interact with one of the windows from the target application. These are "Bring to front", "Move / Resize" and "Close". The "Bring to front" operation, focuses the target window and brings it to the top-level position on desktop. This is usually required, to have full access to a window when taking a screenshot or when interacting with one of its (sub)controls.

| Property | Value | DataType |
|------------------------|----------------|----------|
| Common | | |
| Action specific | | |
| Operation | woBringToFront | Enum |
| NewX | | String |
| NewY | | String |
| NewWidth | | String |
| NewHeight | | String |
| NewPositionEnabled | False | Boolean |
| NewSizeEnabled | False | Boolean |

The next available operation is "Move / Resize", which can move and resize a window to the desired position and size. What win32 API allows, is that any standard Windows control can be moved or resized, not only windows. Moving a button or resizing it on a 3rd party application, might not be of any practical use, but it should be mentioned, as the API allows it. Positioning and resizing a window may be required, when the actions are configured to expect certain controls in specific areas of the target windows. For example, there may be few actions at the beginning of a template, to arrange the target windows on desktop, then interact with them.

The last available operation is "Close", which can close a window, or the entire application. All of the above operations, rely on the value of '\$Control_Handle\$' variable, as it is automatically set by a FindControl action.

2.9 The "LoadSetVarFromFile" action

For having a way of saving the list of variables to disk, there is a pair of actions, LoadSetVarFromFile and SaveSetVarToFile. They use a SetVar action, to specify the exact variables, which are going to be loaded/saved from/to a file. Currently, both actions have the same list of properties, which are made of: "FileName" and "SetVarActionName". "FileName" must be set to the path of a file, from which the action will load the values for the mentioned variables.

| Property | Value |
|------------------------|-----------------------------|
| Common | |
| ActionName | "LoadSetVarFromFileA" |
| Action | LoadSetVarFromFile |
| ActionTimeout | 0 |
| ActionCondition | |
| Action specific | |
| FileName | \$TemplateDir\$\SetVarA.ini |
| SetVarActionName | "SetVarA" |

The file format of that file is set by SaveSetVarToFile action. That means that a SaveSetVarToFile action has to be called, prior to LoadSetVarFromFile, to create the file. The action fails if the file does not exist or its path is not set to "FileName" property. Similarly, if the "SetVarActionName" property is not set, or the SetVar action of the specified name, is not found in the current action template, the action fails. If the file contains a superset of the variables list, with regard to the SetVar action, those variables, which are not found in SetVar, are going to be ignored.

2.10 The "SaveSetVarToFile" action

This is the pair action type of LoadSetVarFromFile. These two actions do not have to exist in the same action template at the same time. Similarly to LoadSetVarFromFile, if the "FileName" property is not set, the action fails. However, if the file does not exist, it is created. The current behavior is to rewrite the entire file content. The SetVar, mentioned by the "SetVarActionName" property, has to exist in the current action template.

In the current implementation, there is no way of specifying to load/save the entire list of variables.

Usually, the SetVar action, used by LoadSetVarFromFile and SaveSetVarToFile, has to be disabled, so that it won't be executed. Only the list of var names is going to be used from that action. The var values or the evaluation options will be ignored. If a variable is mentioned by the SetVar action and it doesn't exist in the list of variables, the SaveSetVarToFile action saves it to file with a value of empty string. When LoadSetVarFromFile opens the file, it adds the non-existent variable to list.

2.11 The "Plugin" action

When none of the above actions can implement the desired functionalities, users can write their own action plugins. They come as dll files, and are loaded and executed in place, i.e. when executing the action. The list of properties of this action is composed of a fixed part (the hardcoded properties) and a dynamic part, which is a plugin-specific list of properties. For now, there is only one fixed property, i.e. the plugin filename. This has to point to a valid dll file, which implements the UIClicker's plugin API. The dynamic list of properties is user implemented and is exposed by every plugin, using a function from its API. UIClicker calls that function whenever it has to load the content of the ObjectInspector. That also happens after setting the "FileName" property.

| Property | Value | DataType |
|------------------------------|--------------------------------|----------|
| Common | | |
| ActionName | "Plugin" | String |
| Action | Plugin | Enum |
| ActionTimeout | 0 | Integer |
| ActionCondition | | String |
| Action specific | | |
| FileName | \$AppDir\$\...\UIClickerFin... | String |
| FindSubControlTopLeftCorner | FindSubControlTopLeftC... | String |
| FindSubControlBotLeftCorner | FindSubControlBotLeftC... | String |
| FindSubControlTopRightCorner | FindSubControlTopRight... | String |
| FindSubControlBotRightCorner | FindSubControlBotRight... | String |
| FindSubControlLeftEdge | FindSubControlLeftEdge | String |
| FindSubControlTopEdge | FindSubControlTopEdge | String |
| FindSubControlRightEdge | FindSubControlRightEdge | String |
| FindSubControlBottomEdge | FindSubControlBottomE... | String |
| ParentFindControl | FindWindowWithScreens... | String |
| BorderThickness | 6 | String |
| MatchWindowEdges | True | String |

Plugin debugging

Stop

Continue All

Step Over

Scroll to current line

```

1229 SetTemplateVar(CActionPlugin_ExecutionResultVar_Control_Bottoms
1230
1231 if MatchWindowEdges = 'True' then
1232 begin
1233     DbgPoint('Searching for window edges', 'start');
1234
1235     SetLength(AllFindSubControlExecutionResults_Edges, 0);
1236
1237 //Executing FindSubControl actions
1238 s := 'unset action';
1239 for i := 0 to ActionCount - 1 do
1240 begin
1241     GetActionInfoByIndex(i, s, ActionTypeDWord);
1242
1243 if ActionTypeDWord <> $FFFFFFFF then
1244 begin
1245     ActionType := TCikAction(ActionTypeDWord);
1246
1247 if ActionType in [acFindSubControl, acPlugin] then
1248 for j := CFindSubControlLeftEdge_PropIndex to CFindSubC
1249 if s = AllRequiredFindSubControlNames[j] then

```

A plugin has access to the list of UIClicker's variables (get and set), the list of actions from the current template (names and contents), it can execute actions by name, it can log to UIClicker's log, it can update the result bitmap (the one used by FindSubControl, from the "Debugging" tab), it can load bitmaps from disk, using UIClicker's allowed directories list, it can save bitmaps to its in-mem file system and it can implement debugging breakpoints. Using these breakpoints, UIClicker's debugger is extended, so it can step-in and jump through plugin's code. It does not implement actual dll debugging, but it is still useful to pause execution at various breakpoints. When stepped-in, the plugin uses a different set of debugging buttons (see previous screenshot). As a limitation, none of the plugin variables are exposed to the debugging interface. They will have to be logged at debugging breakpoint calls.

Currently as part of the plugin API, the plugin has to export only three functions: "GetAPIVersion", "GetListOfProperties" and "ExecutePlugin". The "GetAPIVersion" function is called by UIClicker to verify if its API matches the plugin API. Whenever an API change happens, the API version has to be incremented (see the "CActionPlugin_APIVersion" constant from the ClickerActionPlugins.pas source file). Both UIClicker and plugins will have to use this file, to implement the same API.

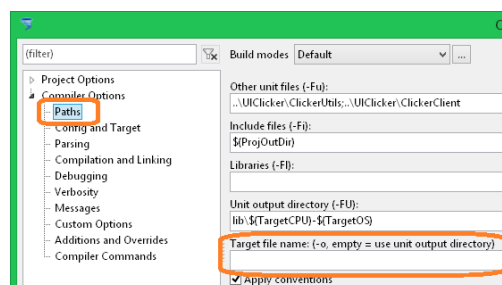
As mentioned above, when UIClicker has to get the list of plugin-specific properties, it calls its "GetListOfProperties" function.

UIClicker actually loads the plugin multiple times, one type of loading being to call "GetAPIVersion" and "GetListOfProperties", while the other, to call "ExecutePlugin" on action execution. The entire plugin execution happens inside the exposed "ExecutePlugin" function. This function takes as arguments the addresses of multiple callback functions, which are used to interact with UIClicker. All the callback headers are defined and described in ClickerActionPlugins.pas file.

Both the above three functions and the mentioned callbacks are defined to use the "cdecl" calling convention, because it is widely available. As for the datatypes, only simple types are used. When passing strings, memory has to be allocated in advance, which is overwritten during the call.

For debugging, UIClicker requires some sort of debugging symbols. UIClicker comes with a small tool, called "ClkDbgSym", found in the "\Tools\ClkDbgSym\" directory. This tool has to be configured to be called by the compiler, before any compilation of a plugin. It should receive, through command line, the name of the ".DbgSym" file (destination file) and then all the source filenames of the plugin, which are expected to be debugged. The tool searches for "DbgPoint(" string, which will use as an indicator for debugging points. It then generates the ".DbgSym" file, out of all these breakpoints. When debugging the plugin, UIClicker loads the ".DbgSym" file and monitors the "DbgPoint(" calls, so it can display where the actual plugin execution is currently at. It does that by matching the string, passed to the "DbgPoint(" function. That means the arguments have to be unique.

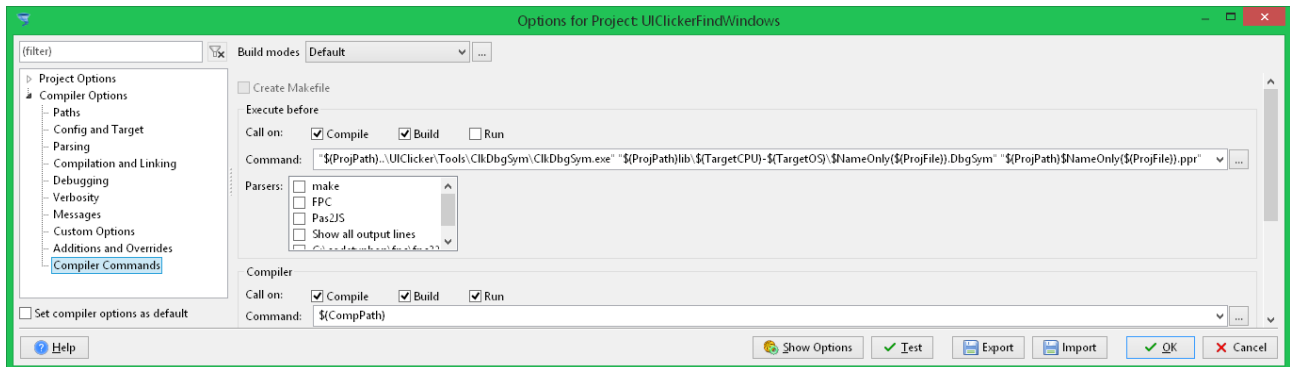
As an example, the plugin project is configured to instruct the compiler to generate the dll file in the \lib\<TargetCPU>\<TargetOS> directory.



This allows a predictable file path, both for 32-bit and 64-bit dlls. It will be required when building the command line for calling the "ClkDbgSym" tool. An example of this command line would be:

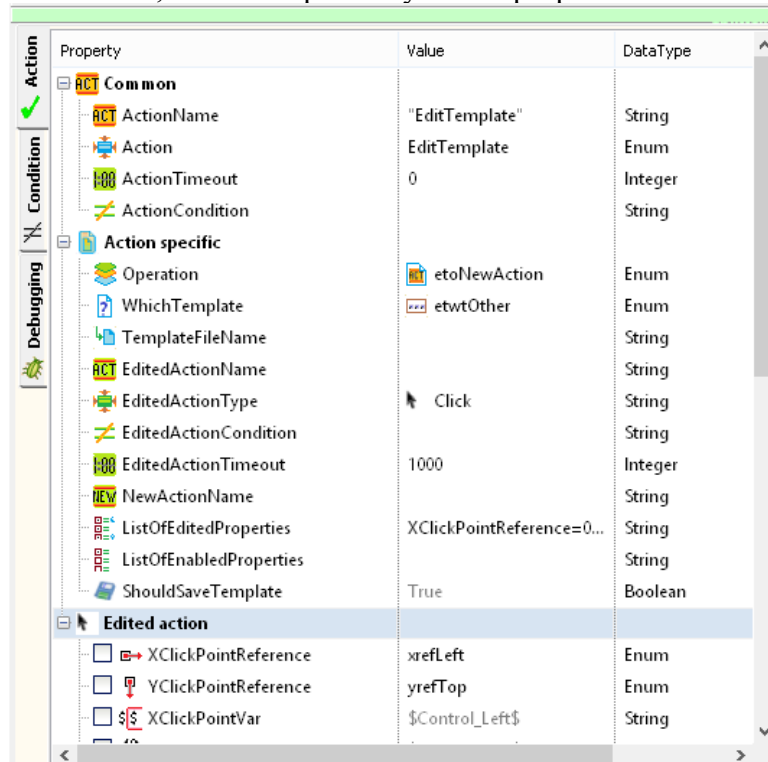
```
"$(ProjPath)..\UIClicker\Tools\ClkDbgSym\ClkDbgSym.exe" "$(ProjPath)lib\$(TargetCPU)-$(TargetOS)\$NameOnly$(ProjFile).DbgSym" "$(ProjPath)$NameOnly$(ProjFile).ppr"
```

Using the above command line, the plugin project has to exist in a directory, at the same level as UIClicker's project directory. It has to be filled in to the "Command" editbox as in the following screenshot of "Project Options" dialog. Also, it has to be executed on every Compile and Build actions.



2.12 The "EditTemplate" action

This is an action, capable of editing templates, by creating and editing actions and also saving those templates. It can target a template on disk or the currently loaded template (also called self template). As a special type of action, it has a third section of properties in ObjectInspector, which displays the same list of properties as displayed by the type of the edited property. For example, if the edited property is Click, the properties under "Edited action" section displays Click properties. When set to ExecApp, it displays ExecApp properties and so on. As edited properties, they can be checked/unchecked, to control precisely which properties to set and which to leave.



Currently, there are 14 operations, set by the Operation property: etoNewAction, etoUpdateAction, etoMoveAction, etoDeleteAction, etoDuplicateAction, etoRenameAction, etoEnableAction, etoDisableAction, etoGetProperty, etoSetProperty, etoSetCondition, etoSetTimeout, etoExecuteAction and etoSaveTemplate.

- When set to etoNewAction, EditTemplate creates a new action and uses the properties, from

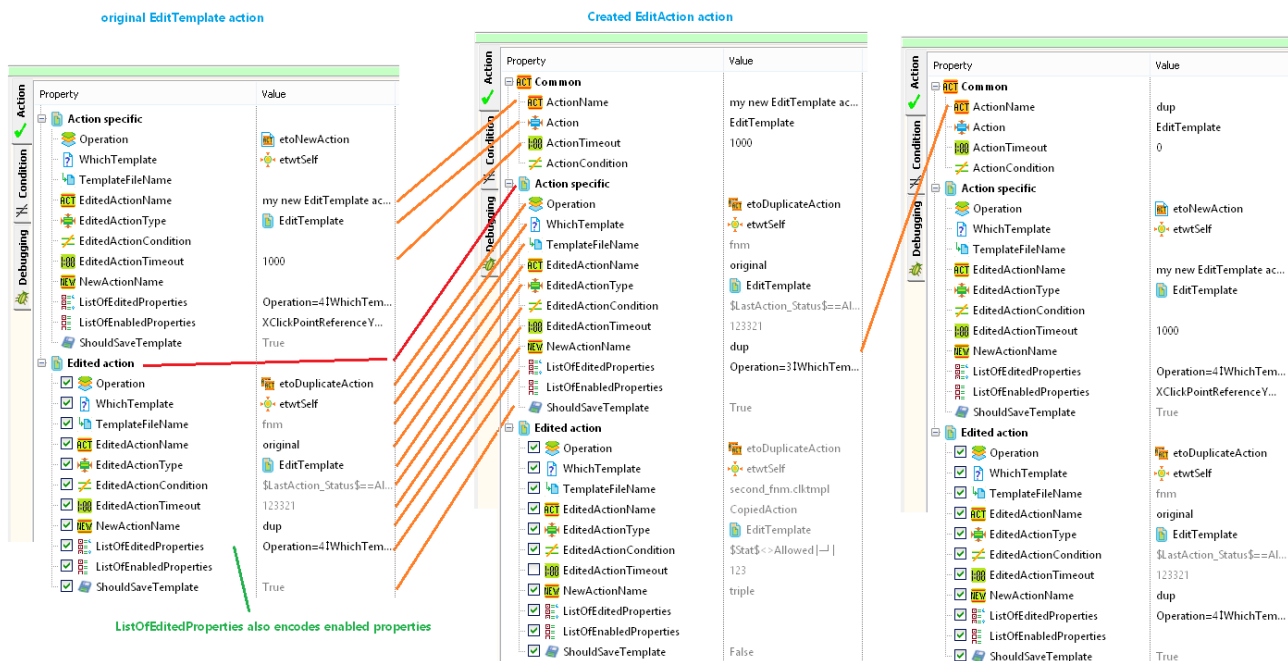
under the "Edited action" section, to set the properties of the newly created action. Only the checked properties are going to be set. All the others are left alone. As a requirement, the new action name, which has to be set to `EditedActionName` property, has to be unique to the template. Although templates are not required to have unique action names, `EditTemplate` requires unique names to identify actions.

- When set to `etoUpdateAction`, `EditTemplate` updates an existing action. It works in a same way as `etoNewAction`. It also uses the `EditedActionName` property to identify the action. If there are two actions with the same name, it updates the first one.
- When set to `etoMoveAction`, `EditTemplate` moves an existing action, from the position mentioned by the `EditedActionName` property, to the position of another existing action, mentioned by the `NewActionName` property. Both actions have to exist.
- When set to `etoDeleteAction`, `EditTemplate` deletes the mentioned action from the template.
- When set to `etoDuplicateAction`, `EditTemplate` duplicates an existing action, mentioned by the `EditedActionName` property as a new action, with a name set to the `NewActionName` property. The new action is placed at the end of the template. In the current implementation, `EditTemplate` doesn't verify if there is an action with the same name as the new action.
- When set to `etoRenameAction`, `EditTemplate` renames an existing action, mentioned by the `EditedActionName` property, to a name, set to the `NewActionName` property. The new name must not exist in the edited template.
- When set to `etoEnableAction`, `EditTemplate` enables an action in the list o actions.
- When set to `etoDisableAction`, `EditTemplate` disables an action in the list o actions.
- When set to `etoGetProperty`, `EditTemplate` reads the checked properties of the mentioned action and returns them as variables with the following format: `$Property_<PropertyName>_Value$`.
- When set to `etoSetProperty`, `EditTemplate` sets all the checked properties of the mentioned action.
- When set to `etoSetCondition`, `EditTemplate` sets the action condition, specified by the `EditedActionCondition` property.
- When set to `etoExecuteAction`, `EditTemplate` executes an existing action, specified by the `EditedActionName` property, in the context of the current template.
- When set to `etoSaveTemplate`, `EditTemplate` saves the currently loaded template. This operation affects only the loaded templates, i.e. when the `WhichTemplate` property is set to `etwtSelf`. When set to `etwtOther`, the template is automatically saved after each editing operation.

The `WhichTemplate` property controls which template contains or will contain the edited or new action. If set to `etwtSelf`, the currently loaded template is targeted. When set to `etwtOther`, the `TemplateFileName` property mentions which template to work with. In that case, the template has to be a valid file. The `EditedActionType` controls which type of action to work with. It can be one of the implemented action types.

The `ListOfEditedProperties` property is a list of "key=value" items (separated by ASCII-18 characters), which contains all the values, specified by the checked actions from under the "Edited action" section (see `ObjectInspector`). This is a read-only property and is updated automatically. When `EditTemplate` is set to create/update another `EditTemplate` action, the `ListOfEditedProperties` property, from under the "Edited action" section can be manually edited (is not read-only, like the one from under the "Action specific" section). The `ListOfEnabledProperties` property is a list of property names and reflects, which properties are checked under the "Edited action" section. The read-only state is similar to the one from `ListOfEditedProperties` property.

There is a special corner case, where an `EditTemplate` action has to create or update another `EditTemplate` action. In the current implementation, there is no extra set of properties, used to control the content of the new/updated action, from under the "Edited action" section, so the `ListOfEditedProperties` property and `ListOfEnabledProperties` property can be manually edited (also under the "Edited action" section), so that content will end up in the target action. Also `ListOfEditedProperties` encodes the checked properties.



The `ShouldSaveTemplate` property is used to control whether the current template is saved or not when executing with a `etoSaveTemplate` operation.

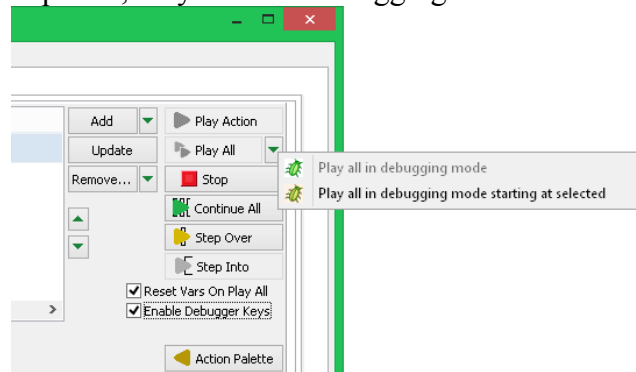
2.13 Action conditions

Every action is conditioned by the successful execution of the previous action, i.e., the value of the `"$LastAction_Status$"` variable has to be "Successful" or "Allowed Failed". It can also be empty string, for the first action of a template. If this variable is "Failed", the execution stops. To further control the execution, every action has its own condition, in the form of an "evaluable" expression, composed of AND / OR operators. This can be edited on the "Condition" page (see the vertical tabs). When comparing two variables, or a variable against a constant, they can be compared using the following operators: `"=="`, `"<"`, `"<"`, `">"`, `"<="`, `">="`. When an action condition is false, the execution is not stopped, the action is simply skipped and the `"$LastAction_Status$"` is set to "Successful". In addition to that, the `"$ExecAction_Err$"` variable is set to "Action condition is false.", to indicate why the action wasn't executed. The `"$LastAction_Skipped$"` variable is also set to "Yes".

3. The debugger

As in writing code, creating UIClicker templates, requires a lot of trial and error, i.e. setting options and testing them. Most of the time, users will have to add actions, then run them in place, to see if they are properly configured. Since many actions depend on previous actions, executing a Find(Sub)Control, twice in a row, won't lead to consistent results. This is where executing actions, using the built-in debugger, is required.

An action can be executed (played) on its own, regardless of the execution status vars, from the "Play Action" button. The "Play All" button will start the execution of the entire template, from the first action. Executing all actions, depends on the action status of the previous actions. The execution will stop at breakpoints, only when in debugging mode.



To execute using the debugger, there is an arrow button, next to the "Play All" button, which has two options, to execute the entire template in debugging mode, or start with the first selected action. Once entered the debugging mode, the execution waits for the user to click one of the "Step" buttons, to move to the next action, or "Continue All", to continue normal execution (as without debugger). The "Step Into" button, is used to open a template in a new page (tab), when the execution gets to a CallTemplate action. It allows the debugger to step through the called template.

There are cases when clicking the debugger buttons will interfere with the automated application, because there may be menus, still open, which expect a click. That is, clicking on a debugging button, will close the menus of the target application. The solution was to implement shortcut keys. They can be enabled from the "Enable Debugger Keys" checkbox. In the first version(s) of UIClicker, these keys are not configurable. They are mapped as follows:

- F6 to take a screenshot of the search area (the same as what the "Display dbg img" button does)
- F7 for Step Into
- F8 for Step Over
- F9 for Start execution and Continue All
- Ctrl-Shift-F2 to stop execution (this may require holding the keys for a few seconds, to stop).

It has to be mentioned that, while debugging, the execution engine does not expect the list of actions to change. There is no mechanism to prevent adding/moving/deleting actions, while debugging. If that happens, executing an unexpected action will cause an access violation. It is also possible that adding/moving/deleting actions somewhere at the bottom of a template won't cause access violations, if the execution is currently somewhere at the top.

In addition to the debugging buttons, there is also a dedicated page, called "Debugging" (see the vertical tabs), where users can access all available variables, and see the execution result of FindSubControl action (see "Bitmaps" section), on "Debugging" page. There is also a console-like "logging" box, which displays various debugging information. It also contains an editbox, where users can type any variable to be evaluated, or assigned. It has to be mentioned that complex expressions can't be evaluated properly, as the parser has a simple implementation.

4. The HTTP API

UIClicker implements a simple HTTP API, to communicate between two instances, one in client mode and the other in server mode. The server module is set to use keep-alive connections,

because the Internet library (Indy: <https://github.com/IndySockets/Indy>) works by creating new threads for any new request. At server side, there is a thread pool (configured to 30 threads), which allows serving existing threads on new requests, without too much CPU overhead. However, if the client disconnects after receiving a response, the behavior is similar to the case where no keep-alive connection is available. UIClicker uses both types of requests as the implementation allows. At client side, there is a separate thread, which is responsible for sending templates and bitmaps to the server, as the server won't have them available during execution. This connection can benefit from the keep-alive option and will stay alive (connected) as long as possible, sending data continuously. Other requests, are made by a different client module, which is created / destroyed on every request. This can't benefit from the keep-alive option, so new connections are created for every new request. It is usually an additional overhead to the operating system.

The current implementation requires creating/destroying the client modules, because for every new started template (while calling new templates), a new client module is created. Every such module will keep the connections open, while executing its "subtemplates". This way, the client instance will have multiple connections to the server, where only one is actively transferring something (request/response). This can easily be seen while debugging, when multiple templates are open, multiple connections should also exist.

There is however a limitation, as a safety measure, which involves the read timeout, that will close the connection if no data will be transferred in a given amount of time. This timeout is currently hardcoded to 1h. A future implementation might be based on a totally different approach, where connections should be asynchronous and there won't be any waitings of such high timeout values. The current implementation is supposed to be simple, not necessarily efficient.

Some requests will get (request) various screenshots (either a debug image, or a full desktop screenshot) from server as uncompressed bitmap files. In the first version(s), there is no option to get the screenshots in a different file format or using other forms of compression.

Following, there are short descriptions of HTTP commands, implemented in UIClicker.

4.1 SendFileToServer

Type: PUT

Request:

`http://<server:port>/SendFileToServer?FileName=<PathToFileForServer>`

Payload:

The file content (.clktmpl or .bmp)

Expected response(s):

Expecting a valid filename.

Received file: "<Fnm>" of <Size> bytes in size.

Description: SendFileToServer is used to send templates and bitmaps from client to server, as the template execution happens at server side. The current implementation makes use of an in-memory file system, at server side, so all executed templates and the bitmap files they require, are eventually sent to server. They are not saved to disk, they are simply used from memory. The in-mem file system allows its files to have full paths, so the "PathToFileForServer" value will usually have the full path as it exists on disk, at client side. Usually, templates are created at client's in-mem file system and sent from there. Bitmaps are sent from client's disk and can have relative paths.

4.2 GetFileExistenceOnServer

Type: PUT

Request:

`http://<server:port>/GetFileExistenceOnServer?VerifyHashes=<0..1>`

Payload:

CRLF separated list of file paths to be verified

Expected response(s):

CRLF separated "0"s and "1"s, matching file existence on server, in its in-mem file system
Description: GetFileExistenceOnServer is used to find out which files already exist at server side, and which files will have to be sent. In addition to checking the file existence, the "VerifyHashes" parameter, if present and set to "1", causes an additional check, i.e. the file content. This is required, because the file on server side, may be out of date, so the current version would have to be sent. An out of date file is reported as non-existent. When the client wants to also verify file hashes, it sets the "VerifyHashes" to "1" (no quotes), and the list of filenames, to be sent to server, will also contain current file hashes, as CRLF separated lines, like: <filename>#1#4<filehash> where, #1#4 are ASCII characters 1 and 4.

4.3 GetListOfWaitingFiles

Type: GET

Request:

`http://<server:port>/GetListOfWaitingFiles`

Expected response(s):

CRLF separated list of file paths

Description: Executing a template at server side is a blocking call, until the template itself and its used files are present. When one or more files are missing, they are placed in a list of missing files, so the client can access this list, using the "GetListOfWaitingFiles" request. This is a continuous polling for missing files and is currently the only request which truly benefits from the keep-alive connections. The polling interval is currently hardcoded at 200ms. With such a low interval, it is clear that creating new connections would quickly become an overhead. Once the client receives the list of files, it sends the files to server, one by one, using the "SendFileToServer" request.

4.4 GetCompInfoAtPoint

Type: Get

Request:

`http://<server:port>/GetCompInfoAtPoint?X=<XPointOnComp>&Y=<YPointOnComp>`

Expected response(s):

#8#7 separated list of the following "key=value" pairs:

Err=OK or X or Y are invalid

Handle=<component handle>

Text=<component text> If this value contains CRLF string, only the first line is read.

Class=<component class>

ScreenWidth=<screen width at server side>

ScreenHeight=<screen height at server side>

CompLeft=<component rectangle, left edge>

CompTop=<component rectangle, top edge>

CompWidth=<component width>

CompHeight=<component height>

Description: This request is used by the RemoteScreenshot window, when clicking somewhere on the screenshot. It is expected, that any point in a screenshot, belongs to one control, either the desktop, or a window, or any other control on a window. Some of the received parameters from the clicked component, are displayed on RemoteScreenshot window, others are used internally.

The following commands currently use the same parameter validation, although some parameters won't be needed for various commands. Future (better) implementations might have these validations removed when not needed. Some of the commands address a specific loaded template, during template execution, so the required parameter would be template index, which can directly address, one of those templates. This parameter is StackLevel=<value> . It will have to be present and be assigned to a valid value, starting at 0. For most commands, setting StackLevel to 0, will be fine, as it will address the main loaded template. For the others, it will have to address an

existing loaded template, including the main one.

If this parameter is missing, the response would be:

```
[Server error] Stack level not specified.
```

When its value is out of range, the response is:

```
[Server error] Stack level out of bounds: <param value>
```

If, for some reason, the above validation fails, and the requested index is still out of range, the default response is:

```
[Server error] Can't get frame at index: <param value>
```

The frame, mentioned above, is the internal component, which stores the action template (+ editor).

4.5 ExecuteCommandAtIndex

Type: Get

Request:

```
http://<server:port>/ExecuteCommandAtIndex?  
StackLevel=<value>&ActionIdx=<ActionIndex>&IsDebugging=<0..1>
```

Expected response(s):

ok For a successful execution

If the ActionIndex parameter is invalid, the internal variable, "\$DbgCurrentAction\$" is set to Cannot select action, to load values at index <index>.,

and the variable "\$Exception\$", is set to

```
ActionIndex out of bounds: <index> in ExecuteActionAtIndex(<index>)
```

#8#7 separated list of all the internal variables

Description: ExecuteCommandAtIndex can execute an action at ActionIndex, from the template at StackLevel. This is similar to clicking the "Play Action" button on a selected action. This action, automatically calls the internal handler, assigned to the "GetExecuteCommandAtIndexResult" request. It contains an additional variable, "\$RemoteExecResponse\$", which can be 0 or 1, depending on error condition and action execution result. The value of this variable, controls whether the execution should continue or not at client side. This list will replace the current list of variables at client side.

4.6 GetExecuteCommandAtIndexResult

Type: Get

Request:

```
http://<server:port>/GetExecuteCommandAtIndexResult?StackLevel=<value>
```

Expected response(s):

#8#7 separated list of all the internal variables. It includes "\$RemoteExecResponse\$", which can be 0 or 1.

Description: Returns the list of all the internal variables, from any loaded template.

4.7 ExitTemplate

Type: Get

Request:

```
http://<server:port>/ExitTemplate?StackLevel=<value>
```

Expected response(s):

ok

Description: Closes a template, which has automatically been loaded as a result of calling it from another template. This is usually called by client on the last action of a template, in debugging mode.

4.8 GetAllReplacementVars

Type: Get

Request:

`http://<server:port>/GetAllReplacementVars?StackLevel=<value>`

Expected response(s):

#8#7 separated list of all the internal variables.

Description: This request is similar to `GetExecuteCommandAtIndexResult`, however, the returned list of variables will not include "\$RemoteExecResponse\$".

4.9 GetResultedDebugImage

Type: Get

Request:

`http://<server:port>/GetResultedDebugImage?StackLevel=<value>&Grid=<0..1>`

Expected response(s):

A bitmap file, containing the debug image from "Debugging" page (see action editor)

Description: The client sends this request, to get the debug image from "Debugging" page. It will display it as the current debug image. This request is made after executing a `FindSubControl` action. If the `Grid` parameter is set to "1" (no quotes), the returned image contains the configured search grid.

4.10 GetSearchAreaDebugImage

Type: Get

Request:

`http://<server:port>/GetSearchAreaDebugImage?StackLevel=<stack level>`

Expected response(s):

A bitmap file, containing the screenshot of the latest found (sub)control.

Description: After finding a control or a subcontrol, its position is kept in '\$Control_Left\$', '\$Control_Top\$', '\$Control_Right\$', '\$Control_Bottom\$' variables. These may be used on a subsequent `Find(Sub)Control` action, to narrow down another (sub)control. The operation of getting the current (sub)control screenshot is required to manually define a search area for the new action. On `UIClicker`'s action editor, this is done from the "Display dbg img" button, found on "Search Area" page under the "FindControl" or "Find SubControl" pages. In server mode, the `GetSearchAreaDebugImage` action is used to "click" the "Display dbg img" button remotely. When clicking this button on client side, a screenshot is taken on server machine and displayed on both server and client's editors. Users can define a new search area for the currently worked on action, from this screenshot. As mentioned before, the defined offsets are valid only after executing the action which finds the component from screenshot. If the user executes the currently worked on action (that which the search area is now being defined), the '\$Control_Left\$', '\$Control_Top\$', '\$Control_Right\$', '\$Control_Bottom\$' variables are modified, resulting in an outdated search area. Often, the user will have to test if the search area is valid, by executing the action. To avoid working with an invalid set of offsets (search area), all the `Find(Sub)Control` actions, above the currently worked on action, which are part of the component hierarchy, will have to be executed again, to properly update the variables. The screenshot may not have to be updated, as it usually won't change.

The `StackLevel` parameter should be specified as a valid value for the current execution. `UIClicker` sets this parameter, depending on the current editor, which sends the request.

4.11 GetScreenShotImage

Type: Get

Request:

`http://<server:port>/GetScreenShotImage?StackLevel=0`

Expected response(s):

A bitmap file, containing the desktop screenshot, at the screen size.

Description: This request is made by clicking the "Refresh screenshot" button from the "Remote

Screenshot" window. It gets a new screenshot on every request. The screen resolution is expected to be known in advance. In case it is not set yet, it is obtained by requesting `GetCompInfoAtPoint`.

4.12 `GetCurrentlyRecordedScreenShotImage`

Type: Get

Request:

```
http://<server:port>/GetCurrentlyRecordedScreenShotImage?StackLevel=0
```

Expected response(s):

A bitmap file, containing the current component screenshot.

Description: This is similar to `GetScreenShotImage`, but instead, it returns the current screenshot, as it is displayed on the Window Interpreter from server side.

4.13 `LoadTemplateInExecList`

Type: Get

Request:

```
http://<server:port>/LoadTemplateInExecList?
```

```
StackLevel=<value>&FileName=<PathToTemplateFile>
```

Expected response(s):

Loaded

Description: `LoadTemplateInExecList` is used to load an existing template into the Actions window. It is expected that the template already exists in server's in-mem file system, sent via the `SendFileToServer` request. If the file does not exist, the server puts its name into the list of missing files, then enters a waiting state. While waiting, it does not respond to the `LoadTemplateInExecList` request. The client has to use a separate connection, to monitor for missing files, via the `GetListOfWaitingFiles` request. It will get the name of the missing template, then it will have to send it to server. When available on server, the file is loaded, and the `LoadTemplateInExecList` request is served.

4.14 `GetFileExpectancy`

Type: Get

Request:

```
http://<server:port>/GetFileExpectancy?StackLevel=0
```

Expected response(s):

OnDisk or FromClient or Unkown, depending on the "Files existence" combobox, from the "Settings page", ("Server mode" tab).

Description: This request is used by the client, when calling a new template, to get the server setting of "Files existence". If the the response is `FromClient`, then the client sends missing files to server.

4.15 `RecordComponent`

Type: Get

Request:

```
http://<server:port>/RecordComponent?StackLevel=0
```

Expected response(s):

Binary content of the tree structure, as recorded by Window Interpreter. This is similar to the file, saved using the "Save Tree..." button.

Description: Records the selected component from Window Interpreter and returns the tree structure. This structure is then loaded by the Window Interpreter tool from client side, and displayed there. The request is made from the arrow button, next to the "Start Recording" button, from Window Interpreter.

4.16 TestConnection

Type: Get

Request:

`http://<server:port>/TestConnection`

Expected response(s):

`Connection ok`

Description: This is a simple request, which can be used to verify if the server is available.

4.17 ClearInMemFileSystem

Type: Get

Request:

`http://<server:port>/ClearInMemFileSystem?StackLevel=0`

Expected response(s):

`Done`

Description: This request is used to delete all files from the in-mem filesystem. Usually, it shouldn't be required. It is mostly used for debugging. All the existing file transfer mechanisms should already be capable of updating the required files, regardless of their existence.

4.18 SetVariable

Type: Get

Request:

`http://<server:port>/SetVariable?`

`StackLevel=0&Var=<$SomeVarName$>&Value=<SomeValue>`

Expected response(s):

`Done`

Description: This request is useful mostly for debugging, as setting a variable during action execution, may influence the execution outcome. It can change only one variable at a time. Sending this request with a variable name which does not exist, will "create" that variable, i.e. will add it to the list. There is no restriction about which variable can be changed, so updating a predefined variable may lead to an unusable application (the server instance), which can be "fixed" either by restarting it or setting the variable back to its initial value. This can also be useful if a new version of the operating system implements system colors which are not predefined in UIClicker. There are also cases when the automated application works with multiple projects and the user wants to automatically create new variables, which store project paths or other project related settings.

4.19 TerminateWaitingForFileAvailability

Type: Get

Request:

`http://<server:port>/TerminateWaitingForFileAvailability?`

`StackLevel=0&Loop=<All, Single, Multi>`

Expected response(s):

`Done`

Description: When UIClicker runs in server mode and has to execute a template in a CallTemplate action, or to execute a FindSubControl action, it requires one or more files (the template to be executed or some bmp files). When configured to look for these files in its in-mem file system, the server will wait for them either until they are received from client or the waiting loop times out. There are two types of waiting loops, one which waits for a single file only, usually a template, and the other, which is able to wait for multiple files, usually a list of bitmaps. This command can terminate one or both of the waiting loops, before they time out. When this command is sent to server, the action waiting for the missing files, will report that the files do not exist and then fails.

The timeout to wait for a file is hardcoded (for now) to 60s. When waiting for multiple files, the timeout is 60s multiplied by file count. That is, for 3 files, the timeout would be 3min. This

command would be useful in testing and when closing the server, in case it is waiting for some files.

A disadvantage of the current implementation, is that a usual command, for executing actions (see commands below) is a blocking call. Because of this, stopping the server's waiting loop, requires sending "TerminateWaitingForFileAvailability" from a separate thread, at client side.

4.20 StopTemplateExecution

Type: Get

Request:

```
http://<server:port>/StopTemplateExecution?StackLevel=<value>
```

Expected response(s):

Done

Description: This is the equivalent of pressing the stop button on a template. To actually close the template (list of actions and template editor) in UIClicker, an additional call to ExitTemplate, is required.

The following, are low level commands, mapped directly to internal action execution commands, as described in chapter 2. Some of them still depend on the action editor and the list of variables. Commands, like FindControl, FindSubControl, SetControlText and WindowOperations, rely on the current value of '\$Control_Handle\$' variable, to interact with a control, so they will need this variable to be set to the proper control handle. This can be done by calling the "SetVariable" command (from this API) in advance, or using the value automatically set by executing the previous action. The "FindControl" action, sets the '\$Control_Handle\$' variable to the searched control handle when it is found. Executing the "SetVar" action can also set the '\$Control_Handle\$' variable.

A significant difference between previous API commands and these "low-level" ones, is that the previous ones rely on creating a template in memory and sending it to the server, while the "low-level" ones, send each action setting as an HTTP URL parameter.

The actions, which depend on the editor (like FindSubControl or Sleep), will access the editor from the level 0 of the execution stack (the "Main Player"). This tab of the editor is set to allow creating new tabs for new stack levels. When an action reads or writes to a variable, it uses the list of variables from level 0. This is why, in the "low-level" action execution commands, the "StackLevel" parameter must be 0.

The advantage of using "low-level" action execution commands, over creating templates and executing them, is that the user has more control of the execution flow, by using, for example an if statement over multiple action calls, instead of setting the same condition to all those actions in a template, or by using programming features, which are not available in UIClicker, like while or for loops. This also allows combining custom user code with calls to these "low-level" commands.

Some of the action parameters are integers or enums used as integers, which come as string values in an HTTP call. If these values cannot be converted to an integer, they default to 0. Enum values are sent as integers in the range 0..MaxEnum, and currently they do not accept strings. The list of parameters, for every command, matches the list of fields in the structure for that particular command. When parameters are missing from an URL, their value is set to 0, for integers and enums, and to empty string for strings. One particular case is the "FindSubControl" action, which allows searching by multiple font profiles. These are represented as items of an array and should be indexed. Also, there is an additional parameter, which does not have an explicit field in the structure, for setting the array length.

All of the "low-level" action execution commands respond with a list of variables, similar to the "ExecuteCommandAtIndex" command. When an integer or enum parameter is out of range, the "\$ExecAction_Err\$" variable is set to an error message, like "MouseButton is out of range.". Similar to "ExecuteCommandAtIndex" command, the "\$RemoteExecResponse\$" variable is set to 0 when an action fails, and to 1, when it succeeds.

In the following examples, the URL parameters will be displayed, one per line, for easy readability, but they should not be separated by LF or CRLF when creating the URL.

4.21 ExecuteClickAction

Type: Get

Request:

```
http://<server:port>/ExecuteClickAction?
StackLevel=0&
XClickPointReference=<0..4>&
YClickPointReference=<0..4>&
XClickPointVar=<value>&
YClickPointVar=<value>&
XOffset=<value>&
YOffset=<value>&
MouseButton=<0..4>&
ClickWithCtrl=<0..1>&
ClickWithAlt=<0..1>&
ClickWithShift=<0..1>&
ClickWithDoubleClick=<0..1>&
Count=<1..2147483647>&
LeaveMouse=<0..1>&
MoveWithoutClick=<0..1>&
ClickType=<0..4>&
XClickPointReferenceDest=<0..4>&
YClickPointReferenceDest=<0..4>&
XClickPointVarDest=<value>&
YClickPointVarDest=<value>&
XOffsetDest=<value>&
YOffsetDest=<value>&
MouseWheelType=<0..1>&
MouseWheelAmount=<value>&
DelayAfterMovingToDestination=<value>&
DelayAfterMouseDown=<value>&
MoveDuration=<value>&
UseClipCursor=<0..1>
```

Expected response(s):

#8#7 separated list of all the internal variables.

Description: Depending on the `ClickType` parameter, this action can simulate one or more click/mouse-down/mouse-up operations, one or more drag operations, or a mouse-wheel operation. When configured to click/mouse-down/mouse-up (`ClickType` is 0, 2, 3 or 4), the parameters ending in "Dest" are ignored. Otherwise, they perform the same role as their non-Dest parameters, but for the destination point. The `ClickWithDoubleClick` parameter is present for backwards compatibility, but it is deprecated and may be omitted. It is superseded by the `Count` parameter, which allows multiple clicks.

The `XClickPointVar` and the `YClickPointVar` parameters can be set to the name of existing variables, which should be evaluated to valid integers, or can be set to some integer values. Negative values are allowed as long as they, together with the `XOffset` and the `YOffset` parameters, will result in values `0..ScreenWidth-1` and `0..ScreenHeight-1`, respectively. The `XClickPointVar` and the `YClickPointVar` parameters are evaluated, only when the `XClickPointReference` and/or the `YClickPointReference` parameters are set to 3, which correspond to "xrefVar" and "yrefVar" enum values. The same goes for their Dest equivalent parameters. For the `XClickPointReference` and the `YClickPointReference` parameters, as well as their Dest equivalent, the 0..4 possible values are defined as enum like:

```
xrefLeft, xrefRight, xrefWidth, xrefVar, xrefAbsolute
yrefTop, yrefBottom, yrefHeight, yrefVar, yrefAbsolute
```

Unfortunately, the above "xref" and "yref" values cannot be used as values. Only integer values, 0

to 4 are accepted. As mentioned before, invalid values or unset parameters are converted to 0.

The `MouseButton` parameter is also an enum, which has its values defined as:

`mbLeft`, `mbRight`, `mbMiddle`, `mbExtra1`, `mbExtra2`. The value has to also be set as 0..4.

When the `LeaveMouse` parameter is set to 1, the mouse cursor is moved and left at the specified coordinates, otherwise it is brought back to its initial position (the one before executing the action). The `MoveWithoutClick` parameter, when set to 1, causes the action to execute only a mouse move operation to the specified coordinates, without actually clicking.

To simulate a mouse-wheel event, the `ClickType` parameter has to be set to 4. The default wheel type is vertical, i.e. when the `MouseWheelType` parameter is 0. The amount is set by the `MouseWheelAmount` parameter, which can have a positive or negative value. This value is automatically multiplied internally by the "WHEEL_DELTA" constant (see win32 API docs), when using the vertical scrolling. The "Delay" and "Duration" properties apply to "click" and "drag" only.

`DelayAfterMovingToDestination` controls the duration between moving the cursor and executing `Click`. The duration of the `MouseDown` in a `Click` operation is controlled by `DelayAfterMouseDown`. The duration of moving the mouse from its current position to its destination is set by `MoveDuration`.

Setting the mouse cursor position is susceptible to errors when moving the physical mouse. To make sure the cursor stays on its desired location, the `UseClipCursor` property controls whether to use `ClipCursor` function (see MSDN for extra info). After executing the click operation, `ClipCursor` is called again to release the cursor boundaries.

4.22 ExecuteExecAppAction

Type: Get

Request:

```
http://<server:port>/ExecuteExecAppAction?
StackLevel=0&
PathToApp=<full path to executable>&
ListOfParams=<#4#5 separated list of command line arguments>&
WaitForApp=<0..1>&
AppStdIn=<#4#5 separated list of strings or a single line string>&
CurrentDir=<either executable directory or some "work" directory>&
UseInheritHandles=<0..2>&
NoConsole=<0..1>&
ActionName=<meaningful action name for error messages>&
ActionTimeout=<0..2147483647>
```

Expected response(s):

#8#7 separated list of all the internal variables.

Description: The path to the executed application is expected to be a full path, or can be relative to the current directory, as set when executing `UIClicker`. Unfortunately, this action cannot execute applications using the `PATH` environment variable.

The `ListOfParams` parameter should be set to a #4#5 separated list of command line arguments for the target application. None of the paths should include double quotes as used in consoles when specifying paths with blanks. That is why the `ListOfParams` parameter should not be a blank separated list, because that would interfere with the folder names, if they contain blanks.

When the `WaitForApp` parameter is set to 1, `UIClicker` does not return from the action, until the executed application terminates or the action times out. In case of a timeout, the action fails. It is required to be set to 1, when the target application outputs on `StdOut` and that output is required. The `ActionTimeout` parameter is used when `WaitForApp` is set to 1. It represents milliseconds.

Similar to the `ListOfParams` parameter, `AppStdIn` is a #4#5 separated list of strings, as should be passed through the `StdIn` stream (pipe). When set to empty string, or not set at all, this parameter is ignored, i.e. no `StdIn` stream is used. To properly send a string through `StdIn`, the `UseInheritHandles` parameter should be set to 1 or 2. The same goes when reading from `StdOut`.

The `CurrentDir` parameter is usually set to the directory where the executable can be found. It can also be set to other "work" directory, depending on target application.

For the `UseInheritHandles` parameter, there can only be three values, 0, 1, and 2. They correspond to the following enum values:

`uihNo`, `uihYes`, `uihOnlyWithStdInOut`

When executing console applications, they are usually displayed in a console window (similar to `cmd`). To avoid displaying these consoles, the "No console" checkbox has to be checked. There are UI applications, which can also create and display a console window. If they are run by `UIClicker`, then the console might not properly show the output, although it is there. As a limitation, if the executed console application, requires user interaction (like displaying text, then expecting the user to input something), then only a simple interaction might actually work. For example, the console application might expect the user to input three values and press Enter (return) after each. Before reading the values, the console might display a small text, saying what input it expects, one for each value. From the `StdIn/StdOut` perspective, this looks like a ping-pong interaction. If the number of expected values, is constant (or predictable), then a string can be constructed, to be passed through `StdIn`. Usually, multiple values are input to a console, by pressing return (i.e. CRLF). Thus, the string should contain `#4#5` (ASCII 4 and 5), which will automatically be converted to CRLF when sending the string to the executed application. When users need the `"#4#5"` string, they can type `"$#4#5$"` variable in `UIClicker`'s console, to be evaluated.

4.23 ExecuteFindControlAction and ExecuteFindSubControlAction

Type: Get

Request:

```
http://<server:port>/ExecuteFindControlAction?
StackLevel=0&
MatchCriteria.SearchForControlMode=<0..2>&
MatchCriteria.WillMatchText=<0..1>&
MatchCriteria.WillMatchClassName=<0..1>&
MatchCriteria.WillMatchBitmapText=<0..1>&
MatchCriteria.WillMatchBitmapFiles=<0..1>&
AllowToFail=<0..1>&
MatchText=<value>&
MatchClassName=<value>&
MatchTextSeparator=<value>&
MatchClassNameSeparator=<value>&
MatchBitmapText.Count=<0..100>&
MatchBitmapText[<0..MatchBitmapText.Count-1>].ForegroundColor=<6-digit hex BGR>&
MatchBitmapText[<0..MatchBitmapText.Count-1>].BackgroundColor=<6-digit hex BGR>&
MatchBitmapText[<0..MatchBitmapText.Count-1>].FontName=<value>&
MatchBitmapText[<0..MatchBitmapText.Count-1>].FontSize=<2..200>&
MatchBitmapText[<0..MatchBitmapText.Count-1>].Bold=<0..1>&
MatchBitmapText[<0..MatchBitmapText.Count-1>].Italic=<0..1>&
MatchBitmapText[<0..MatchBitmapText.Count-1>].Underline=<0..1>&
MatchBitmapText[<0..MatchBitmapText.Count-1>].StrikeOut=<0..1>&
MatchBitmapText[<0..MatchBitmapText.Count-1>].FontQuality=<0..6>&
MatchBitmapText[<0..MatchBitmapText.Count-1>].FontQualityUsesReplacement=<0..1>&
MatchBitmapText[<0..MatchBitmapText.Count-1>].FontQualityReplacement=<value>&
MatchBitmapText[<0..MatchBitmapText.Count-1>].ProfileName=<unique value>&
MatchBitmapText[<0..MatchBitmapText.Count-1>].CropLeft=<value>&
MatchBitmapText[<0..MatchBitmapText.Count-1>].CropTop=<value>&
MatchBitmapText[<0..MatchBitmapText.Count-1>].CropRight=<value>&
MatchBitmapText[<0..MatchBitmapText.Count-1>].CropBottom=<value>&
MatchBitmapText[<0..MatchBitmapText.Count-1>].IgnoreBackgroundColor=<0..1>&
MatchBitmapFiles=<#4#5 separated list of bmp file paths>&
MatchBitmapAlgorithm=<0..1>&
MatchBitmapAlgorithmSettings.XMultipleOf=<32-bit integer value>&
MatchBitmapAlgorithmSettings.YMultipleOf=<32-bit integer value>&
MatchBitmapAlgorithmSettings.XOffset=<32-bit integer value>&
MatchBitmapAlgorithmSettings.YOffset=<32-bit integer value>&
```

```

InitialRectangle.Left=<var name or 32-bit integer value>&
InitialRectangle.Top=<var name or 32-bit integer value>&
InitialRectangle.Right=<var name or 32-bit integer value>&
InitialRectangle.Bottom=<var name or 32-bit integer value>&
InitialRectangle.LeftOffset=<var name or 32-bit integer value>&
InitialRectangle.TopOffset=<var name or 32-bit integer value>&
InitialRectangle.RightOffset=<var name or 32-bit integer value>&
InitialRectangle.BottomOffset=<var name or 32-bit integer value>&
UseWholeScreen=<0..1>&
ColorError=<value>&
AllowedColorErrorCount=<value>&
WaitForControlToGoAway=<0..1>&
StartSearchingWithCachedControl=<0..1>&
CachedControlLeft=<value>&
CachedControlTop=<value>&
MatchPrimitiveFiles=<#4#5 separated list of pmtv file paths>&
GetAllControls=<0..1>&
UseFastSearch=<0..1>&
FastSearchAllowedColorErrorCount=<value>&
IgnoredColors=<value>&
SleepySearch=<0..1>&
StopSearchOnMismatch=<0..1>&
ImageSource=<0..1>&
SourceFileName=<value>&
ImageSourceFileNameLocation=<0..1>&
PrecisionTimeout=<0..1>&
FullBackgroundImageInResult=<0..1>&
MatchByHistogramSettings.MinPercentColorMatch=<value>&
MatchByHistogramSettings.MostSignificantColorCountInSubBmp=<value>&
MatchByHistogramSettings.MostSignificantColorCountInBackgroundBmp=<value>&
EvaluateTextCount=<value>&
CropFromScreenshot=<value>&
ActionName=<meaningful action name for error messages>&
ActionTimeout=<0..2147483647>
FileLocation=<Mem, Disk>

```

Expected response(s):

#8#7 separated list of all the internal variables.

Description: The underlying mechanism of executing FindControl and FindSubControl actions, is the same. Some of the settings are shared, the others are particular to one action type or the other.

For the FindSubControl action, the request starts with:

<http://<server:port>/ExecuteFindSubControlAction>

The parameters are identical. For example, the bitmap text and bitmap files – related parameters will be ignored on "FindControl", while they are required on "FindSubControl". The ActionTimeout parameter has to have a positive value (usually between 1000 and 30000 for this action), in case the action has to be stopped by timeout, if it doesn't find the searched control/subcontrol.

The settings for the MatchCriteria parameters can be found on the "Criteria" tab on the "Find (Sub) Control" section of the action editor. The MatchCriteria.SearchForControlMode can have one of the three values, 0, 1, 2, which correspond to the following enum values: sfcmGenGrid, sfcmEnumWindows, sfcmFindWindow

The MatchCriteria.WillMatchText parameter is used by both "FindControl" and "FindSubControl" actions as the searched text. The control class name can be specified on the MatchCriteria.WillMatchClassName parameter. The other three fields, WillMatchBitmapText, WillMatchBitmapFiles, and WillMatchPrimitiveFiles, are "FindSubControl" only.

This action is also expected to find nothing, so it can be allowed to fail, by setting the AllowToFail parameter to 1. Then, it sets the "\$LastAction_Status\$" variable to "Allowed Failed".

When multiple matches are expected for the Text or Class fields of a control, the MatchTextSeparator and MatchClassNameSeparator parameters should be set to the user-

defined strings, used as list separators. Then, `MatchText` and `MatchClassName` are expected to be lists of search strings.

If searching for a subcontrol, using bitmap text, the `MatchBitmapText.Count` parameter specifies the number of font profiles that action will have. It can be 0, when no font profiles are used (when searching for a control or when searching by bitmap files) and is limited to 100 profiles, to limit memory usage. All parameters under the `MatchBitmapText` prefix, will be indexed like array parameters. For example, the `FontName` parameter will appear as:

```
MatchBitmapText[0].FontName=Tahoma
MatchBitmapText[1].FontName=Verdana
MatchBitmapText[2].FontName=Segoe UI
```

The index starts at 0 and can go up to `MatchBitmapText.Count - 1`. Text color and background color are set by the `ForegroundColor` and `BackgroundColor` parameters, under the same indexed `MatchBitmapText`. They can be set to an existing variable name or to an RGB value, written as 6-digit hex BGR value (see win32 API, about BGR). For example, yellow would be 00FFFF, while aqua would be FFFF00. As mentioned before, `UIClicker` provides already existing variables for usual system colors. To set the expected font anti-aliasing algorithm, the `FontQuality` parameter can be one of the 0 to 6 values, corresponding to the following enum values:

```
fqDefault, fqDraft, fqProof, fqNonAntialiased, fqAntialiased, fqCleartype,
fqCleartypeNatural
```

There are cases when a template expects multiple font qualities (anti-aliasing algorithms), so the `FontQualityUsesReplacement` parameter should be set to 1, while the `FontQualityReplacement` parameter is expected to have one of the above seven string values, without the "fq" prefix. The `CropLeft`, `CropTop`, `CropRight`, `CropBottom`, parameters, are used to crop a text, before being sent to the matching algorithm. All four parameters expect 0 or positive values. Variables can also be used. If these parameters are not specified, they default to 0. Positive values will cause the text to be cropped towards the center, while 0 causes no cropping. Every font profile can have its own set of cropping values. This is useful when the searched text can be overlapped by a focus rectangle.

When searching for bitmaps as `SubControl`, the list of bmp files can be set by the `MatchBitmapFiles` parameter, as a #4#5 separated list of paths. This parameter is used only when `MatchCriteria.WillMatchBitmapFiles` is set to 1. If using a search grid, all the parameters under the `MatchBitmapAlgorithmSettings` prefix should be a valid 32-bit integer, defining the grid points and grid offset.

The `InitialRectangle` parameters correspond to the editboxes from the "Search Area" page, under `Find(Sub)Control` page of the action editor. They define the edges/offsets of the search area. Can be set to 32-bit integers or to existing var names. Combining these values, the search area has to be a "positive" rectangle, i.e. Left "edge" should be less than the Right "edge", while Top "edge" has to be less than Bottom "edge". The `Left`, `Top`, `Right`, `Bottom`, parameters, from `InitialRectangle`, are ignored when `UseWholeScreen` is set to 1, as they are internally set to the edges of the screen.

For `ColorError` and `AllowedColorErrorCount` parameters, the values can be 32-bit integers or var names. Searching can start with a cached position, defined by `CachedControlLeft` and `CachedControlTop`, when `StartSearchingWithCachedControl` is 1.

Primitives filenames should be passed to the `MatchPrimitiveFiles` parameter, as a #4#5 separated list.

To make `FindControl` search for multiple controls, set the `GetAllControls` parameter to 1.

The `UseFastSearch` parameter should be set to 1, to use the "FastSearch" feature. It also requires the `FastSearchAllowedColorErrorCount` parameter to have a valid value.

For `ImageSource` parameter, the values 0 and 1 correspond to `isScreenshot` and `isFile`.

The same goes, for `ImageSourceFileNameLocation` parameter, where the values 0 and 1 correspond to `isflDisk` and `isflMem`. If `PrecisionTimeout` is set to `True`, the action stops faster.

When `FullBackgroundImageInResult` is set to 0, only the relevant search area and debugging information will be displayed in the debugging image. This is useful on remote

execution setups, to transfer smaller images.

There is a third bitmap search algorithm, currently in work, based on bitmap content, evaluated using histograms. Its main settings are controlled by the subproperties under the `MatchByHistogramSettings` property. The `MinPercentColorMatch` subproperty sets how much of the searched bitmap histogram data should be found in background bitmap histogram data. Only identical colors are compared. `MostSignificantColorCountInSubBmp` sets how many of the most significant colors of the searched bitmap histogram, are compared.

`MostSignificantColorCountInBackgroundBmp` sets how many of the most significant colors of the background bitmap histograms, are available to be compared. For more details, see each property's editor tooltip in ObjectInspector.

To allow a text to be searched, which contains the "\$" character, without being replaced as a variable, the `EvaluateTextCount` property can be set to 0. When set to -1 (the default value), the searched text is evaluated (replaced), until no more variables are found to be encoded within it. To prevent infinite loops, the evaluation algorithm stops after 1000 iterations. When set to a value, greater than 0, this property control exactly how many evaluations to run. As an example, if the input text is "\$abc\$" (without quotes) and the user wants to search this exact text, then the property should be set to 0. If "\$abc\$" is present as a variable and its value is "\$def\$" (without quotes), then a single evaluation (the property is set to 1) results in searching for "\$def\$" (without quotes). If "\$def\$" is also defined as a variable, and the property is set to 2, it is evaluated further.

In the last few OS releases, there are various types of windows, which use a different device context, to render their content (as examples, those which classes "SearchPane" and "Windows.UI.Core.CoreWindow"). Similar windows may be those, which render OpenGL content. The `CropFromScreenshot` property, when set to True, sets the screenshot operation to function differently. By default (when False), it uses the control handle to get the screenshot. This doesn't work on these overlay windows, resulting in blank content. When set to True, the screenshot is taken from the whole screen, then cropped at window coordinates. There is however, a difference, when the target control/window is overlapped by another window. As a limitation, a full screenshot contains the overlapping window, affecting the target content, and also it is a bit slower, as it requires an extra operation.

Similar to "ExecuteExecAppAction" command, the `ExecuteFindControlAction` and `ExecuteFindSubControlAction` commands, require setting the `ActionName` parameter, because in case of not finding the searched (sub)control, they set an error message, containing the action name. Usually, the error message is either about a negative search area, or a timeout.

For debugging purposes, there is a parameter, which controls where `UIClicker` (in server mode) should look for files. It is `FileLocation` and can be `Mem` or `Disk`. If not specified, it defaults to `Disk`. The `FindSubControl` action may require bitmap files, which `UIClicker` will wait for, if not available. When `UIClicker` waits (in server mode) for missing files, and the user works with low-level API functions for executing actions, the waiting loop can be closed by starting a second instance of `UIClicker` and setting it to be a client, connected to the existing server. It automatically sends missing files to server if they are available at client side. However, users will eventually have to implement a client, capable of sending missing files to server, using the available API functions.

4.24 ExecuteSetControlTextAction

Type: Get

Request:

```
http://<server:port>/ExecuteSetControlTextAction?
StackLevel=0&
Text=<value>&
ControlType=<0..2>&
DelayBetweenKeyStrokes=<value>&
Count=<0..65535>
```

Expected response(s):

#8#7 separated list of all the internal variables.

Description: Based on the value of `ControlType` parameter, this command can set a control text (an editbox or a window title), or an editbox, as part of a system combobox, or to simulate keystrokes.

The 0..2 possible values correspond to the following enum values:

`stEditBox`, `stComboBox`, `stKeystrokes`

When using `stKeystrokes`, the delay after each keystroke is set with `DelayBetweenKeyStrokes`.

4.25 ExecuteCallTemplateAction

Type: Get

Request:

```
http://<server:port>/ExecuteCallTemplateAction?
StackLevel=0&
TemplateFileName=<path to template>&
ListOfCustomVarsAndValues=<#4#5 separated list of $var$=value pairs>&
EvaluateBeforeCalling=<0..1>&
Loop.Enabled=<0..1>&
Loop.Counter=<var name>&
Loop.InitValue=<var name, function call or value>&
Loop.EndValue=<var name, function call or value>&
Loop.Direction=<0..2>&
Loop.BreakCondition=<#4#5 separated list of condition items, created by editor>&
Loop.EvalBreakPosition=<0..1>&
IsDebugging=<0..1>&
FileLocation=<Mem, Disk>&
UseLocalDebugger=<0..1>
```

Expected response(s):

#8#7 separated list of all the internal variables.

Description: This request loads a template, specified by `TemplateFileName` parameter and calls it. The value of `TemplateFileName` can be a full path or the filename only, if the file is in the "Templates Dir" directory, as configured on the "Settings" page of UIClicker. The path should not be enclosed by double quotes, even if it contains blanks.

To add or set custom vars, before calling the template, they can be passed, through the `ListOfCustomVarsAndValues` parameter as a #4#5 separated list of `$VarName$=VarValue` pairs. When `EvaluateBeforeCalling` is 1, all the var values are evaluated if they have the var format (e.g. `$VarName$=$OtherVarName$`). Those, which are not already in the main list, will be added. When `IsDebugging` is set to 1, the template is executed in debugging mode. The debugger can be at client side or at server side, based on the `UseLocalDebugger` parameter. When set to 1, the debugger is local to the server. This will require manual interaction at server side, i.e., clicking the "Step Over", "Step Into" and "Continue All" buttons. When not specified, or set to 0, the debugger is controlled by client. This is the default mode when using two instances of UIClicker, one in client mode and the other in server mode. When in server mode, a 3rd party application can control the action execution of a debugger, either at action level, by setting `UseLocalDebugger` parameter to 0 and calling "ExecuteCommandAtIndex" for every action, or letting the debugger run on its own on server, requiring manual user interaction there.

The called template can set the "\$LastAction_Status\$" variable to "Allowed Failed", if its last action is a Find(Sub)Control and is allowed to fail. This status is preserved on template exiting. Similar to FindSubControl, there is `FileLocation` parameter, which can be set to `Mem` or `Disk`. If not specified, it defaults to `disk`. It controls where the server should look for the called template. When set to `Mem`, and the template is not found in server's in-mem file system, the server enters a waiting loop for missing files. The client should implement a mechanism of verifying the missing files and sending them to server (see "GetListOfWaitingFiles" and "SendFileToServer" commands).

All the looped `CallTemplate` parameters are prefixed with "Loop". The first one is `Loop.Enabled`, which can be 0 or 1, and controls whether to use a simple `CallTemplate` or a looped one. `Loop.Counter` is usually a variable name, which will be used as iteration counter. For loop endpoints, there are `Loop.InitValue` and `Loop.EndValue`, which can be simple variable names, function calls or constants. The `Loop.Direction` parameter can be 0, 1, or 2, which correspond to the following enum values: `ldInc`, `ldDec`, `ldAuto`. The `Loop.EvalBreakPosition` parameter can be 0 or 1, which correspond to the following enum values: `lebpAfterContent` or `lebpBeforeContent`.

4.26 ExecuteSleepAction

Type: Get

Request:

```
http://<server:port>/ExecuteSleepAction?
StackLevel=0&
Value=<value>&
ActionName=<meaningful action name for error messages>
```

Expected response(s):

#8#7 separated list of all the internal variables.

Description: This command requires the amount of milliseconds to sleep, set by the `Value` parameter, which can be a 32-bit integer, or a var name. It also requires an action name, in `ActionName`, for error messages.

4.27 ExecuteSetVarAction

Type: Get

Request:

```
http://<server:port>/ExecuteSetVarAction?
StackLevel=0&
ListOfVarNames=<#4#5 separated list of var names>&
ListOfVarValues=<#4#5 separated list of var values>&
ListOfVarEvalBefore=<#4#5 separated list of 0s or 1s>&
FailOnException=<0..1>
```

Expected response(s):

#8#7 separated list of all the internal variables.

Description: All three lists, from the `ListOf<..>` parameters, should have the same number of items. The items from `ListOfVarValues`, can contain names of existing variables. These values are evaluated if the corresponding items (of the same index) from `ListOfVarEvalBefore`, are set to 1.

4.28 ExecuteWindowOperationsAction

Type: Get

Request:

```
http://<server:port>/ExecuteWindowOperationsAction?
StackLevel=0&
Operation=<0..2>&
NewX=<value>&
NewY=<value>&
NewWidth=<value>&
```

NewHeight=<value>&
NewPositionEnabled=<0..1>&
NewSizeEnabled=<0..1>

Expected response(s):

#8#7 separated list of all the internal variables.

Description: There are three available operations, set by the `Operation` parameter as 0, 1, 2, corresponding to the following enum values: `woBringToFront`, `woMoveResize`, `woClose`

When set to 0 or 2, the other parameters are not required, so they are ignored. They are required when `Operation` is 1, i.e. for moving or resizing a window (or another control). Since the same operation is used for moving and resizing, these two can be individually enabled, from the `NewPositionEnabled` and `NewSizeEnabled` parameters, when set to 1. The `NewX` and `NewY` parameters, are used for position, while `NewWidth` and `NewHeight` are used for size.

Since this command requires a valid control handle, this would usually be set by a `FindControl` action. It is also possible that the desired control is a window, which can be partially or totally overlapped by another window. If that is the case, the `FindControl` action will have to be configured to use the "sfcmFindWindow" option, instead of the usual "sfcmGenGrid" (see "SearchForControlMode" subproperty, under "MatchGrid" property). This is because, when using the "sfcmGenGrid" option, `FindControl` requires the window to be visible, and not covered by other windows, or moved outside the screen. The "Find Window" option does not have this limitation.

4.29 ExecuteLoadSetVarFromFileAction

Type: Get

Request:

http://<server:port>/ExecuteLoadSetVarFromFileAction?
StackLevel=0&
FileName=<value>&
SetVarActionName=<value>

Expected response(s):

#8#7 separated list of all the internal variables.

Description: The `FileName` parameter has to point to a valid filename, from where the action will load the list of variables. There has to be a `SetVar` action in the currently loaded template, so that the `SetVarActionName` parameter would point to that action.

4.30 ExecuteSaveSetVarToFileAction

Type: Get

Request:

http://<server:port>/ExecuteSaveSetVarToFileAction?
StackLevel=0&
FileName=<value>&
SetVarActionName=<value>

Expected response(s):

#8#7 separated list of all the internal variables.

Description: The `FileName` parameter has to point to a valid filename, to where the action will save the list of variables. There has to be a `SetVar` action in the currently loaded template, so that the `SetVarActionName` parameter would point to that action.

4.31 ExecutePluginAction

Type: Get

Request:

http://<server:port>/ExecutePluginAction?
StackLevel=0&
FileName=<value>&
ListOfPropertiesAndValues=<value>&
IsDebugging=<0..1>

Expected response(s):

#8#7 separated list of all the internal variables.

Description: The `FileName` parameter has to point to a valid plugin file (a dll). The `ListOfPropertiesAndValues` parameter is a #4#5 separated list of key=value pairs. The keys are the names of the plugin-specific properties. The `IsDebugging` parameter enables plugin debugging.

4.31 ExecuteEditTemplate

Type: Get

Request:

```
http://<server:port>/ExecuteEditTemplate?
StackLevel=0&
Operation=<0..13>&
WhichTemplate=<0..1>&
TemplateFileName=<value>&
EditedActionName=<value>&
EditedActionType=<0..12>&
EditedActionCondition=<value>&
EditedActionTimeout=<value>&
NewActionName=<value>&
ListOfEditedProperties=<value>&
ListOfEnabledProperties=<value>&
ShouldSaveTemplate=<0..1>
```

Expected response(s):

#8#7 separated list of all the internal variables.

Description: See the descriptions in the "EditTemplate action" chapter.

As an example of what to fill-in to `EditedActionCondition`, please copy-paste the value from ObjectInspector, after editing the condition, using the Condition editor.

5. Testing

Most of the current testing implementation relies on the HTTP API. The UI part of the application (e.g. template editor) has a limited set of automated tests. The bitmap processing features require their own tests, which are yet to be written. Indeed, there are still some known interaction bugs or inconsistent behaviors across similar UI features of the action editor. Also, in the first version(s) there are incomplete implementations of some features, which appear to be bugs. For example, if UIClicker, in server mode, expects some files, it stays in a waiting loop and can't be closed, either until timeout or the files are received. The missing part of the implementation is to safely terminate those loops. Similarly, the application can't be closed while debugging, because of another waiting loop.

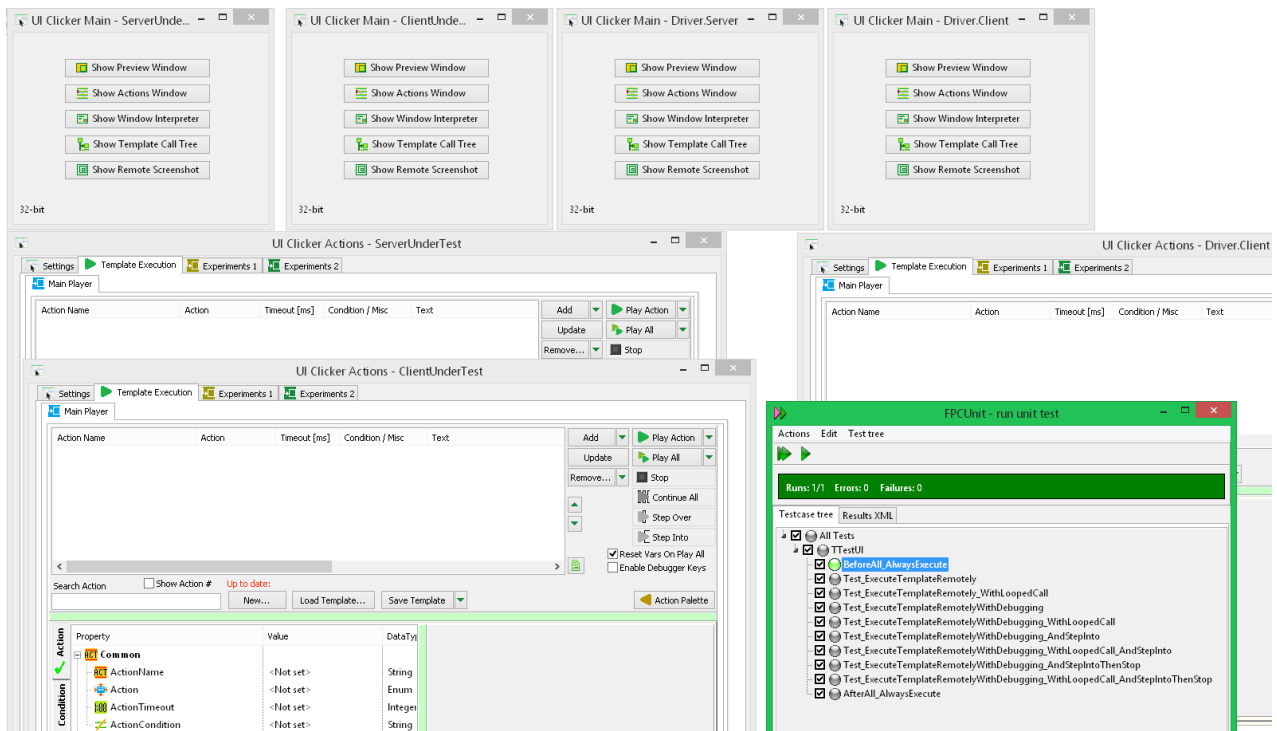
For automated testing, there is a "\Tests" directory, with a standard FPCUnit project. It is called "UIClickerHTTPTests" and it targets the HTTP API only. Since the HTTP API is split into low-level and template-level functions (from an action's perspective), the tests are also organized into multiple files. There is also a third file, with miscellaneous tests.

The "UIClickerHTTPTests" project should be compilable, both for 32 and 64-bit and be compatible (across the HTTP API) to both 32 and 64-bit versions of UIClicker. In the first version(s), the test coverage is pretty low, but it gives a good start on how to organize tests. Similarly to UIClicker, the test project depends on some units, from "UIClicker\..\Misc" directory.

Some of the tests require either built-in tools, or 3rd party applications. Among the built-in tools, there is the "\Tests\TestFiles\GradientText\GradientText.exe" application, which should be manually compiled, to generate the executable, to be available when running the tests. Also, it should be allowed, from the firewall, to listen to connections on its configured port (see its "Active" checkbox). Also, one of the tests uses the available web browser, installed on the test machine. There are tests, which verify the plugin interface, and those expect the "FindWindows" plugin to be built (by the user), prior to running those tests. See the "UIClickerFindWindowsPlugin" repository.

5.1 UI testing

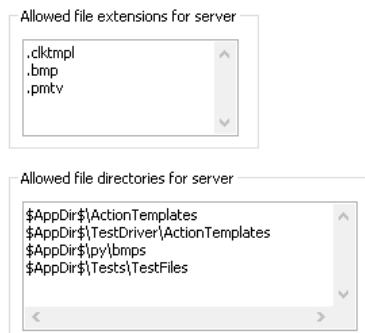
Another test suite, under Tests\UI directory, targets UI interaction and is mostly focused on template execution. This one has a more complicated setup, compared to the HTTP test suite. It uses four instances of UIClicker, out of which, three are set to work in server mode. The fourth one is switched back and forth between client mode and server mode. As more tests are being written, the setup may become more dynamic and complex.



In the current setup, the test application (built on FPCUnit), runs in client mode, and interacts with one of the UIClicker instances (in server mode), called "TestDriver". This TestDriver is instructed to load and execute predefined action templates, which interact with the test instance of UIClicker. The test instance is the one being switched between server mode and client mode. The TestDriver instance is another (up to date) copy of UIClicker.exe, placed in the TestDriver directory, near the main UIClicker.exe. The TestDriver copy, has its own set of action templates (used for testing), placed under TestDriver\ActionTemplates directory. By using the ActionTemplates directory, the test application won't have to specify a separate directory for testing templates.

UIClicker allows setting an additional string to the already existing window captions, by specifying it from command line, using the "--ExtraCaption" option. This helps the TestDriver identify all instances and even place them in a predefined layout on desktop, ready for interaction. The TestDriver instance has the extra caption, set to "Driver.Client". By default, its Actions window is not displayed as it's not needed in the current test suite. The "ClientUnderTest" instance is the one, which is configured to run in client mode, when running tests and is being switched by the TestDriver, to server mode when the test application reads its variables. At the moment of writing this chapter, this setup is limited to Windows only, as the Wine implementation can't properly set editboxes to a particular string (see SetControlText action). However, these actions can be configured, to use keystrokes.

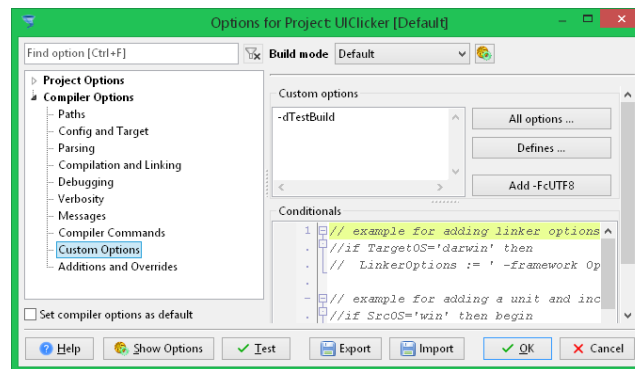
To allow proper file transfer between instances, the client under test has to be configured, from the application's "Settings" page, as in the next screenshot (UIClicker has to be run manually):



5.2 Command line options for testing

To further automate UI testing, a new feature of UIClicker is enabled, when the application is built in test mode, using the project-level definition, "TestBuild". It is found as "-dTestBuild" in the "Compiler Options" section from Project Options, in CodeTyphon. Since adding the command line options to UIClicker, a console had to be enabled, to allow it to output to StdOut pipe/stream. This was done by disabling "Win32 gui application", found under "Config and Target" section from Project Options, in CodeTyphon. Because of this, the console is displayed for a short amount of time and closed quickly, at every application startup, which does not involve command line, i.e. UI only. All the available commands are displayed in console, when UIClicker is run with the "--help" command line option.

```
Commandline options are available in TestBuild only
Usage:
  UIClicker.exe [Option]
Options:
  --ExtraCaption <Additional caption>
  --SetExecMode <Local, Client, Server>
  --ServerPort <Port used in server mode>
  --ConnectsTo <Server address and port, used in client mode>
  --ConnectionTimeout <Timeout, in milliseconds, used in client mode>
  --SkipSavingSettings <Yes>
  --AutoSwitchToExecTab <No, Yes>
  --AutoEnableSwitchTabsOnDebugging <No, Yes>
  --UseWideStringsOnGetControlText <No, Yes>
  --EnableSavingExceptionToFile <No, Yes>
  --AddAppArgsToLog <No, Yes>
Examples:
  UIClicker.exe --ExtraCaption MyServer --SetExecMode Server --ServerPort 15444
  UIClicker.exe --ExtraCaption MyClient --SetExecMode Client --ConnectsTo http://127.0.0.1:15444
```



As mentioned before, the "--ExtraCaption" option adds an extra string to all windows of that particular UIClicker instance. See the screenshot of the UI testing setup. When running on Wine, the application is automatically run as administrator, and in addition to the extra string, the caption on the Actions window will contain "[Is Admin]". For UI testing, a simple Wine detection is made, so a variable is set to " [Is Admin]", in TestDriver, to match the new captions.

To place a UIClicker instance in one of the available execution modes, the "--SetExecMode" option can be set to "Local", "Client" or "Server". This is the equivalent of setting the mode from the "Execution mode" combobox. This mode can be further changed during execution (from UI), as it is the case of the "ClientUnderTest" instance (see screenshot).

When "--SetExecMode" is set to "Server", an additional option "--ServerPort" can change the default port UIClicker is configured to listen to. Valid values are from 1 to 65535. However, it should not clash with already existing processes, listening on the same port.

When "--SetExecMode" is set to "Client", the "--ConnectsTo" option, can specify the connection string, using the format <http://address:port> .

A second option, used in client mode, is the "--ConnectionTimeout". This expects a valid positive integer, and represents the amount in milliseconds, the application will wait on establishing a connection to server, before timing out.

Most of the command line options affect one or more UI settings, which will eventually be saved to the applications config file (ini file). This can be prevented, by setting the "--SkipSavingSettings" option to "Yes".

The "--AutoSwitchToExecTab" and "--AutoEnableSwitchTabsOnDebugging" options can be set to "Yes" or "No" and control the two checkboxes of the same name, from the Settings page. These further control whether or not to display the action list while executing actions and whether or not to display it in debugging mode.

To overcome a limitation of Wine, to read ANSI strings from various controls of another application, the "--UseWideStringsOnGetControlText" option enables reading in Unicode (UTF-16) mode. This will switch to using the SendMessageW function, as opposed to the default SendMessageA. On Windows, SendMessageA works as expected, so this option is not needed.

When "--EnableSavingExceptionToFile" is set to "No", unhandled exceptions are not saved to text file anymore. This is useful, if running UIClicker in a directory without write permissions.

Application arguments are displayed in log when setting "--AddAppArgsToLog" to "Yes".

6. Clicker client

Because the implementation is not intended to become a full featured language, the HTTP API can be used by a 3rd party client, which should be able to implement a more complex logic than what UIClicker provides for action execution. There is a project, in "UIClicker\ClickerClient" directory, which compiles a native dll. It thus can be used by other applications/scripts, written in different languages. In the first version(s), it features mostly the template-level part of the HTTP API, as this one requires more than sending simple requests to execute actions, i.e. it can send templates and bitmaps to server, as "missing files", in a separate thread. The project should be compiled as 32 or 64-bit dll, depending on the loader bitness.

There is also a Python wrapper, which implements some of the exported functions from ClickerClient.dll. It can be found under "UIClicker\py" and it currently comes as two files, UIClickerTypes.py and wrapper.py. The primary limitation is the compatibility between the UTF8 implementation of Python strings and the UTF16 implementation in Windows API. There have been some implementation difficulties in UIClicker (both the executable and the client dll) for UTF-8, so the compatibility might be limited to the lower 128 ASCII characters.

~~At the moment of writing this section, there are some issues with certain exported functions, like "TestConnectionToServer", "GetServerAddress" and "SetServerAddress", which cause Python executable to crash. It is possible that there are unhandled exceptions, in extra threads, uncaught by the dll itself. Newer Python versions seem to work without the mentioned crashes.~~

7. Application security

As this is intended to be a small project, UIClicker relies on external security tools when being used as an HTTP server. The only security-like features of UIClicker, are the file extensions and allowed directory limitations, used in client mode (see Settings page from Actions window), and the command line options, available in testing mode only. Other than that, there is no other security feature. The application can execute pretty much anything it is asked to, including other applications, both as command line or using UI interaction. Users are strongly advised to use UIClicker in a closed environment, where no unwanted access is possible. As the protocol is HTTP only, without HTTPS, this application should not be exposed to the Internet. More than that, UIClicker should not be run as administrator. When running under Wine (in Linux), UIClicker detects that it is run as administrator and will display this on the Settings page.

8. Other

The source code of UIClicker, including the clicker client, can be found at:

<https://github.com/VCC02/UIClicker>

The UIClicker binaries (both 32-bit and 64-bit) are in a different repository (including this pdf):

<https://github.com/VCC02/UIClickerRel>

There are source code dependencies in a common repository:

<https://github.com/VCC02/MiscUtils>

When building, the UIClicker and MiscUtils repositories must be at the same level on disk. Other projects, like UIClicker action plugins, can be found at:

<https://github.com/VCC02/UIClickerFindWindowsPlugin>

<https://github.com/VCC02/UIClickerDistFindSubControlPlugin>

<https://github.com/VCC02/UIClickerTypewriterPlugin>

By October 2024, all the above three mentioned plugins are in work, with the "Typewriter" plugin being the most usable. The "FindWindows" plugin is also used by some HTTP API tests.