

User Interface - Feature #3704

enable automated testing for GUI (Swing and web clients)

08/28/2018 06:05 PM - Greg Shah

Status:	New	Start date:	
Priority:	Normal	Due date:	
Assignee:		% Done:	0%
Category:		Estimated time:	0.00 hour
Target version:			
billable:	No	vendor_id:	GCD
Description			
Related issues:			
Related to User Interface - Bug #6920: Ins and outs of Sikuli for GUI testing			New

History

#1 - 08/28/2018 06:34 PM - Greg Shah

----- Forwarded Message -----

Subject: automated GUI testing
Date: Mon, 19 Jan 2015 18:05:09 -0500
From: Greg Shah
To: Asofiei, Constantin, Hynek Cihlar
CC: Faulhaber, Eric

...customer discussion redacted...

The FWD project started in 2004. We deliberately made extreme compatibility with the 4GL runtime an important part of the offering. When we did that, we knew that automated testing would be important for several reasons:

1. For any non-trivial amount of code, it is impossible to get good test coverage without a large number of tests.
2. Running the tests frequently is necessary to ensure that even small changes can be tested. This allows changes to be isolated from one another, making it easier to see the cause of regressions.
3. We couldn't dedicate a large test team (or even a small test team) to the testing. The customer wasn't going to have a dedicated test team either.
4. Manual testing is error prone because it relies upon humans to detect problems and/or to duplicate the test the same way each time.

The initial approach was that pilot customer tried to use existing tools for automating ChUI screens. At the time, there was no good solution for TTY testing automation. One of the customer staff hacked something together using Jacareto/Jsat/cleverphl/picorder. With some of our help, they encoded just over 1000 tests using that. We started testing.

It turns out that that hacked together technology was quite unreliable. It was impossible to really use the test environment in any kind of productive way. Running the automated tests multiple times against the same FWD version would yield massively different results. A great deal of effort was spent trying to pass testing using that first approach.

At this point, I decided to write a proper test automation toolkit to replace the Jacareto approach that didn't work. This was the genesis of the [Automated Test Harness](#). Our team re-encoded the customer application's automated tests in the new "harness" and spent considerable time getting them working. Using that automated environment, we were able to greatly accelerate the development pace of the project.

I realize that our current set of ChUI automated tests are not 100% perfect. There is still some level of false positives (tests that sometimes fail due to some dependency which we did not adequately encode in the automated test). We do plan to resolve these issues and I believe there is no limitation in achieving that objective. Even so, they are reliable enough that testing will usually pass given two runs of the harness. When there are real regressions, they are very often found via this testing environment.

We do plan to expand this to implement unit tests and also to implement our own functional 4GL tests to allow comprehensive testing of specific functional areas of our compatibility. You can see some small movement in that direction in several places. For example, we have long had some useful tests in the date.java class that can be executed via a main() method. These could and should be shifted into being separated from the date code and executable as a set of unit tests, using something like JUnit. We also have a larger number of tests for specific functionality like native API calls (see testcases/uast/library_calls/). These are pretty extensive and more completely explore 4GL compatibility. The non-UI test suites would be executed in a batch without too much trouble. But still, we have not taken a properly structured approach to these tests. So they are neither automated nor are they comprehensive enough to really give us substantial coverage. On the positive side, since they are written in the 4GL, they do allow us to test the end-to-end 4GL support from conversion to runtime. I believe that a mixture of both traditional unit tests and these 4GL functional tests are required to really do the job right.

When we started to implement the 4GL GUI, we knew that GUI testing comes with a more complex set of problems. Our experience with testing tools like QTP demonstrated that they were fragile at best, and even then it was very hard coded to the low level details of the operating system platform or browser. This would break when running against FWD since the GUI design is not based on the WIN32 UI at all.

We have to put our own automated testing environment in place for GUI. With that in mind, I want to open a discussion with you all regarding how we should move forward with automated GUI testing. I will note some additional thoughts:

- Each test can generally be fairly well contained. Although we will need some tests that have multiple windows, complex focus management, complex control flow, nested triggers/wait-for... I would hope that most of our tests will be much simpler.
- I will consider that part of our tests can be pure Java and more designed as unit tests, but I think a substantial amount of our tests need to be real 4GL programs which we convert and run.
- It makes sense to be able to run the same tests against both the Swing GUI driver and the Javascript driver (actually running in the browser). There may be some things that can't work in both, but most things should work exactly the same in either place. I think this requirement means that most of the existing tooling won't work. Most projects are hard coded to Swing or Javascript/browser and I haven't yet found a real "cross-platform" approach that would work for GUI.
- I am OK with exposing an API in our client to allow the test suite to feed input and read output/state. I'm OK with using the widget hierarchy and widget state to confirm as much as possible rather than using only bitmap comparisons with the 4GL. On the other hand, I think it is inevitable that we do some non-trivial amount of bitmap comparison to ensure that we are exactly matching the look/feel of the 4GL.
- I can see one approach is to have the 4GL code itself do quite a bit of checking on things and the results of that code being fed back into the test results in some clean/architected manner. Our current testcase/uast/ suites that can be run non-interactively have some early approaches to this that need enhancement to record results and pinpoint failures more cleanly. It seems to me that this approach should handle both UI testing and non-UI testing, so that we have a single approach that can be extended to be more complete over time.
- This will be reliable from the start. No multiple runs to get results.
- Although TTY testing is easy to implement multiple user simulations on a single client, it is usually not possible to do that for GUI. In particular, if one uses Java's Robot functionality, that code actually integrates with the OS-level system event queue which makes multiple simultaneous clients difficult. Other issues like focus management also preclude that.
- There may be some use in extending our current harness environment to this purpose. Read through the [Harness Documentation](#). Most of the harness is not that specific to TTY.
- Capture/recording of tests should be considered, but I suspect that any such recorded test will need hand modifications to make it work reliably.

The following are links to the most obvious 3rd party (open source) tooling/frameworks:

<http://docs.seleniumhq.org/> (web browser only)
<https://code.google.com/p/fest/> (swing)
<http://www.uispec4j.org/> (swing)
<http://abbot.sourceforge.net/doc/overview.shtml> (swing/awt)
<https://jemmy.java.net/>
<http://sourceforge.net/projects/jacareto/>
<http://marathontesting.com/marathon/> (swing)
<https://developers.google.com/java-dev-tools/wintester/html/>

Some good docs that at least have thought provoking ideas (I may not agree 100% with all of this):

<http://abbot.sourceforge.net/doc/api/abbot/doc-files/about.html>
<https://jemmy.java.net/RecordingVSCoding.html>
<https://jemmy.java.net/AutomationEffectiveness.html>

#2 - 08/28/2018 06:38 PM - Greg Shah

Subject: Re: automated GUI testing
Date: Tue, 20 Jan 2015 15:34:35 +0200
From: Constantin Asofiei
To: Greg Shah, Hynek Cihlar
CC: Faulhaber, Eric

You may already be aware of this, but I'll note it either way: after looking at some of the tools you listed, I think all open source tools which work with either a browser or Swing rely on access to the existing DOM tree (for Browser) or to Swing's widget hierarchy (to be able to find a widget by name/ID).

As all the GUI drawing code draws directly to canvas, for these tools to work, they will need access to our FWD widgets. I don't know yet if we can create some kind of bridge between the FWD components and Swing, so that these tools can see them as Swing (or maybe some other way to expose the FWD widget hierarchy to the tool, or change the tool to access FWD instead of Swing/DOM).

About the non-interactive GUI tests; what we can do is manually encode in the test certain widget attributes (like position, size, screen-value, etc) and check the FWD values against the hard-coded ones. But I think we should consider a way to automate this; how I see it: we would have some 4GL libraries which, depending if the test is ran in 4GL or FWD, will either record the values for widget attributes/error-status/etc (when in 4GL) or will compare the current attributes/error-status/etc with the recorded ones (when ran in FWD). Ideally the libraries will automatically determine all the live widgets (maybe list all windows using the SESSION:FIRST-CHILD and go down the widget tree from there), and possibly filter recording only for certain widget types/IDs. We can use PRIVATE-DATA to uniquely identify each widget (as the HANDLE ID can't be used). We will need to circumvent some differences in the widget hierarchy (frame top-labels come to mind, and maybe some FIELD-GROUP issues). The point for this is that if the widget's dimension/location is correct, we have some assurance that the drawing should be (almost) correct.

For checking if the widgets are drawing properly, we (or the tool) will check a screen region given by the widget's pixel coordinates and compare with the screen region from the 4GL capture... but when texts are involved, almost sure these will never match perfectly, due to differences between the Java's built-in font rendering and Window's font rendering.

#3 - 08/28/2018 06:42 PM - Greg Shah

----- Forwarded Message -----

Subject: Re: automated GUI testing
Date: Tue, 20 Jan 2015 08:56:59 -0500
From: Greg Shah
To: Constantin Asofiei, Hynek Cihlar
CC: Faulhaber, Eric

Good thoughts.

You may already be aware of this, but I'll note it either way: after looking at some of the tools you listed, I think all open source tools which work with either a browser or Swing rely on access to the existing DOM tree (for Browser) or to Swing's widget hierarchy (to be able to find a widget by name/ID).

Yes. This is why I suggested that we consider enhancing our own harness instead. You're right that we could build a bridge that makes our hierarchy look like Swing but that seems complicated. Personally, I'm leaning toward enhancing our own harness, but I'm open to alternatives if the benefits are there.

Either way, it will be important for the tests to be able to read both the formatted and unformatted data from each widget, along with state like whether something is initialized, visible, has focus... so that we can really track editing/interactive operations.

I like the ideas about how we can use the 4GL to inspect/capture/compare and report.

but when texts are involved, almost sure these will never match perfectly, due to differences between the the Java's built-in font rendering and Window's font rendering.

This is true, we can expect things to be visually close but not identical. We can potentially read the labels and data values straight out of the widget hierarchy for comparison with captured baseline text. Another idea is to use some form of OCR to compare text from corresponding bitmaps.

#4 - 08/28/2018 06:44 PM - Greg Shah

----- Forwarded Message -----

Subject: Re: automated GUI testing
Date: Tue, 20 Jan 2015 20:35:53 +0100
From: Hynek Cihlar
To: Greg Shah
CC: Asofiei, Constantin, Faulhaber, Eric

I did the reading and here is set of my thoughts on the subject.

Due to the time pressure I think it will be important to gain for the 80/20 ratio, 80% coverage with 20% effort (whatever 100% effort means :-)). I am a fan of automation so I agree that this is the way to go. Also I would not put much work into unit testing of individual components as this introduces great overhead (with exceptions - components that have well-defined inputs/outputs).

Regarding the question whether to extend and use Test Harness or use a 3rd party tool I would incline towards the Test Harness. (1) TH has proven helpful testing ChUI, (2) it can be adapted to the requirements, no need to modify requirements to adapt to a specific 3rd party tool, (3) I think it is possible with TH to have same test sets for Swing as well as Javascript and for multiple OS platforms.

With the TH GUI changes I would consider some usability improvements. (1) Visualize the output differences (reports, screens), (2) output HTML results as each test finishes (this allows for a simple smoke check, it has happened to me several times that I was waiting for results when most of the tests failed due to issues in the environment or error in the code).

And now some more practical thoughts.

Account for GUI environment differences - fonts, DPI, UI scaling, UI themes, etc. The reference GUI test environment should be somehow formalized.

Text rendering will be different on 4GL and FWD/Swing (or even FWD/Javascript), so checking for absolute values (widget positions, sizes) or comparing bitmaps pixel-by-pixel will likely not work. I think the comparison of widget positional attributes will have to be a little more sophisticated (based on calculated text dimensions) or work on higher level of abstraction. By higher level I mean (1) work in character units and/or (2) check relative widget positions ("widget B is right of widget A with two characters spacing").

Proximity bitmap comparison. If feasible, this could help to achieve the desired automation for passing/failing outputs. Again I think that direct comparison of bitmaps will not work due to the rendering differences. So the idea is to employ an algorithm that would give a measure of equality. I am not an expert on image processing but there seem to be methods that could yield the desired results. For example by calculating image histograms (from color channels, texture directions, texture scales) values are generated which can be directly used to compute degree of similarity of two bitmaps. Surely this would require some experimentation but potential benefits are tempting.

UI component identification. In order to cleverly describe or query what needs to be on the screen we will need some kind of stable UI component identification. I think the ideal ID comes from the visuals - how the humans perceive the identity. So the ID should be calculated from the (1) widget type (2) label and in some cases (3) widget contents (when widget is missing for example). This I think would also help when manually writing tests or inspecting test results.

On what level will test instructions work? Would it be the elementary input actions, mouse cursor (providing absolute positional values) and button clicks, keyboard inputs? Or specialized instructions like `table_click_column(stableID, column_index)`? Here we are again hitting the issue of potential layout differences due to text rendering differences between 4GL and FWD.

Detection of GUI drawing finished. For web gui this will be easy, hook up with the `document.onReady` event, and provide the information to the test framework. For Swing GUI this will probably require some cooperation of the `OutputManager` and/or other components responsible for drawing

transactions.

Cross-browser compatibility issues - probably not a big issues, but I can imagine differences in control behavior (event handling), visual differences, differences in the DOM structure, etc.

#5 - 08/29/2018 09:49 AM - Greg Shah

Common testing tools (like Selenium) provide an automation model where they leverage the internal model of the controls/widgets to identify the specific widget and interact with it (read its text, send it input...).

FWD renders the UI in common Java code using a driver that implements a set of primitives. These primitives draw the UI and handle input events directly instead of delegating the task to higher level widgets in the browser or Swing.

In the browser this is handled via the HTML5 canvas. There is no 1-to-1 mapping of 4GL widgets to a DOM element. Instead, for each 4GL window there is an HTML5 canvas. All 4GL widgets, frames and window decorations are rendered in that single canvas by common Java code in the FWD client. Input (keys and mouse) provided to the canvas is forwarded to the FWD client (Java code) and interpreted as if it was processed on the focused 4GL widget. This allows total control over the appearance and behavior of the resulting UI, enabling absolute compatibility with the 4GL and also providing the ability to enhance/extend the UI with new capabilities. Using web forms and DOM elements directly for each widget would not allow compatibility nor would it allow the extensibility either.

The Swing approach is similar, using a single JPanel per 4GL window instead of Swing's built-in widgets. Walking the Swing widget tree does not match the 4GL widget tree.

This implementation is the reason that the standard Selenium approach (built-in to the Selenium web driver) will not work.

I agree that we want to leverage the Harness for our internal automated GUI testing. This has been envisioned in [#2863](#). To enable this in FWD, we will need to provide an API into our client UI implementation which exposes the ability to inspect and interact with the 4GL widgets and the resulting screens.

A common customer requirement is to use Selenium for web application testing. We will extend their "browser driver" model and map that into the same API that is going to be exposed to the Harness. This will enable testing from Selenium with the same enabling code that we would use internally.

This API should allow an external test toolset to identify specific widgets (find components by id, name or path), read state from a widget, write state into a widget and to interact with them by sending mouse and keyboard events. This kind of API can support the Selenium WebDriver4 approach which could be considered the industry standard for such tooling.

By supporting Selenium, customers will also have the ability to integrate (via the API) with a large number of available tools for test registration and replay.

A key point here (implicit in the above idea) is that we are not going to support execution against a WIN32 GUI. We are going to focus on providing first class automated GUI testing enablement for FWD without worrying about trying to execute the same tests against OpenEdge.

OpenEdge implements its GUI via direct usage of WIN32 controls and APIs. As such, the 4GL widget hierarchy is partially available via WIN32 but the 4GL state is not directly available. The deep dependency on WIN32 makes automated testing approaches overly complicated AND very fragile.

We can provide a much more stable and simple approach that appears to be directly accessing the 4GL widget hierarchy. But we can only do this for FWD. Although it would be nice to be able to test before (OpenEdge) and after (FWD) for comparison, we aren't going to try to solve OpenEdge's automated testing problems.

I think this approach requires the least effort to implement for obtaining the corresponding benefit. All the hardest parts of the work are under our own control so I am confident that this approach will succeed.

Let me know if there are any thoughts.

#6 - 08/29/2018 10:25 AM - Hynek Cihlar

Greg Shah wrote:

OpenEdge implements its GUI via direct usage of WIN32 controls and APIs. As such, the 4GL widget hierarchy is partially available via WIN32 but the 4GL state is not directly available. The deep dependency on WIN32 makes automated testing approaches overly complicated AND very fragile.

Even worse, some of the widgets (like browse), don't have WIN32 counterparts.

To gather reference test data, we could create dedicated 4GL code that would be injected into the native 4GL app. This would be used to gather screen data (state) usable by the testing framework. Creating reference screen data would be matter of executing this injected 4GL for the key screens.

#8 - 12/04/2018 03:19 PM - Greg Shah

A customer has found a tool called SikuliX. It works based on image recognition and OCR for text recognition. They talk about how they can handle GUI testing/automation on Flash applications. This would be exactly like the FWD approach.

I suspect this will already be close to working. Although I worry that the slight differences in our font rendering compared to OpenEdge will possibly break the tool, the use of OCR to grab text may work. **GES_UPDATE: Sikuli already provides OCR support for text to handle this exact issue. If it works, the font differences won't be a problem.**

A tool like this could be used for certification testing (running it both on the OpenEdge version as well as in FWD to prove it is correct).

It is a Java-based open source tool, though the original project site at MIT seems dead. It looks like this is the newer project:

<http://sikulix.com/> (main information site)

<https://launchpad.net/sikuli> (issues/technical project site)

<https://github.com/RaiMan/SikuliX1> (source code)

<https://www.simpliv.com/developmenttool/show-and-tell-sikuli-pattern-matching-and-automation> (some intro classes)

The downside is that it does not fit into the Harness approach. But if we can get good results without changes to FWD, then it may be a reasonable approach for now.

#9 - 12/04/2018 03:20 PM - Greg Shah

Another testing framework is [Cypress](#). A customer mentioned that this is becoming a popular framework and has advantages over Selenium.

#10 - 02/04/2019 06:41 PM - Greg Shah

Greg Shah wrote:

Another testing framework is [Cypress](#). A customer mentioned that this is becoming a popular framework and has advantages over Selenium.

I've done some review of their project and taken a quick look at their API. They are a pure JavaScript, all-in-one end-to-end testing system. Their code runs in the browser, in-line with your application code. This means that you can easily expose integration points that can be called to get things done. Tests are just JavaScript code so if you can provide a clean interface to handle all the common interactive needs then it might be a pretty straightforward thing to use this with the FWD web client.

The only downside I see right now is that this tooling is very much assuming a DOM approach that exposes individual widgets. I wonder to what extent we need to have DOM elements that shadow the widget hierarchy and what limits may exist in doing this. For example, we could implement a non-visual DOM "shadow tree" but this would not really provide a mechanism to handle the interactive stuff. For example, focus processing won't work and although state could be read out of an element, we might have to do something tricky to intercept other calls to these elements such that changing state would be reflected into FWD. And I'm not sure if there are more serious problems with the non-visual nature of the shadow DOM.

A customer is asking us to ensure Cypress can be used as a testing tool for FWD, so I need to have a plan for the kinds of changes that would make sense.

Hynek/Constantin: Can you each review the details of the [Cypress](#) project and consider how we could make testing our web client into a good experience for customers?

#11 - 02/04/2019 09:17 PM - Eric Faulhaber

Another thing to consider with Cypress: what are the performance implications of maintaining a DOM "shadow tree" that exists only for the purpose of testing? Surely there must be some added overhead for this, if it is something that is otherwise not needed for normal operation...

#12 - 02/05/2019 08:45 AM - Greg Shah

Eric Faulhaber wrote:

Another thing to consider with Cypress: what are the performance implications of maintaining a DOM "shadow tree" that exists only for the purpose of testing? Surely there must be some added overhead for this, if it is something that is otherwise not needed for normal operation...

If we need this at all, we would only need to do it when in testing. I don't see the requirement to do it during normal usage.

#13 - 02/06/2019 04:52 PM - Constantin Asofiei

Greg Shah wrote:

Hynek/Constantin: Can you each review the details of the [Cypress](#) project and consider how we could make testing our web client into a good experience for customers?

There are a few issues which I don't think we could add easily:

- recording support: this records what you do in the DOM; we would need to emulate in the hidden DOM every action taken by the user. And somehow 'trick' Cypress to record the DOM action, and not the canvas action. But we might be able to write our own recording, to automate the test creation.
- the timemachine - cypress takes a snapshot of the DOM (not sure if it uses a screenshot of the page, too) - after each test step. Again, this would not work easily if the snapshot doesn't include the canvas image save.

The hidden DOM issue I think is already supported by passing { force: true } to the command. The usual commands (mouse, keys) can be emulated on our JS driver easily (as we don't need an actual 'mouse click' to post a click from the JS side to the Java runtime) - we just need to identify the widgets uniquely and post this command to the JS driver.

If the test is written manually, another tricky part is to uniquely identify a widget - we could use its name (if is unique), but what if the same widget is part of multiple frames? Or multiple windows exist on screen? We would need to identify each widget via its tree path, starting with the window root node. And in case of dynamic widgets, is not necessary to have an explicit NAME. But we still need to have consistent widget IDs to identify them. So we will need a consistent and unique identifier for any widget, dynamic or not. If we make the recording possible, then this ID can be anything, as long as it's consistent.

Why we need to easily and uniquely identify a widget: because Cypress requires to select that widget before performing any action on it. As I don't think we can use `cy.focused()`, unless we override this to query our JS driver and get us the widget's ID. But then again, we need to uniquely identify this widget to select it.

At this time, I think the recording and consistent widget ID (which remains the same on server restart) are the biggest issues.

#14 - 02/06/2019 07:01 PM - Greg Shah

If the test is written manually, another tricky part is to uniquely identify a widget - we could use its name (if is unique), but what if the same widget is part of multiple frames? Or multiple windows exist on screen? We would need to identify each widget via its tree path, starting with the window root node. And in case of dynamic widgets, is not necessary to have an explicit NAME. But we still need to have consistent widget IDs to identify them. So we will need a consistent and unique identifier for any widget, dynamic or not. If we make the recording possible, then this ID can be anything, as long as it's consistent.

Why we need to easily and uniquely identify a widget: because Cypress requires to select that widget before performing any action on it. As I don't think we can use `cy.focused()`, unless we override this to query our JS driver and get us the widget's ID. But then again, we need to uniquely identify this widget to select it.

This one can be resolved fairly easily. The existing 4GL WIDGET-ID option was used for this purpose. Although we could consider using this, I actually don't see a good reason to do so. It would be messy and could conflict with other 4GL usage of that value.

Instead, we can simply add a new FWD extension attribute/option called DOM-ID or INTERNAL-ID or something else that is unambiguous. Instead of an integer value, I would recommend a string which could be any unique text. We then would push that data down to the JS side and expose it in the shadow DOM as an element id. In this way the selectors would work as expected. Or if we don't implement the shadow DOM, then we could provide helper functions that return a JavaScript object representing that widget. For any code the customer wants to write an interactive test for, they would have to make edits to the 4GL code to add these identifiers and then reconvert. That seems a reasonable approach and the names can be suitably descriptive.

#15 - 02/06/2019 07:03 PM - Greg Shah

In regard to the recording and timemachine support, I agree these are pretty tricky.

#16 - 02/15/2019 03:05 AM - Hynek Cihlar

Cypress doesn't seem to support shadow DOM. See <https://github.com/cypress-io/cypress/issues/144>.

#17 - 06/17/2021 05:21 PM - Roger Borrello

- File 3704_hotel_login_demo.mkv added

A little proof of concept using SikuliX.

Here is what I was able to create using the IDE.

```
# You must be logged into the OS manually

wait("os_logon.png",5)

# Wait for the application login screen
wait("hotel_gui_login.png",60)

# Click in the User Name box
click(Pattern("hotel_gui_login.png").targetOffset(-99,150))

# Enter signon
type("hotel\thotel\n")

# Wait for main screen
wait("main_window_available_rooms.png",10)

# Click through the tabs
click(Pattern("main_window_available_rooms.png").targetOffset(-399,47))

wait("main_window_guests.png",5)

click(Pattern("main_window_available_rooms.png").targetOffset(-297,51))

wait("main_window_reservations.png",5)

click(Pattern("main_window_available_rooms.png").targetOffset(-167,51))

wait("main_window_rates.png",5)

click(Pattern("main_window_available_rooms.png").targetOffset(-74,55))

wait("main_window_rooms.png",5)

# Logout

click(Pattern("main_window_available_rooms.png").targetOffset(467,6))

wait("hotel_gui_login.png",5)
```

```
# And exit

click(Pattern("hotel_gui_login.png").targetOffset(774,-334))

wait("os_logon.png",5)
```

The movie attached is what occurs when I press Play, then login to the OS manually. I don't want to script my password, but am still looking how to do this. From there, it takes off.

I tried changing the UI Theme to something other than Windows 10, and it failed at the first wait on the main window:

```
[log] CLICK on L[501,888]@S(0) (521 msec)
[log] TYPE "hotel#TAB.hotel#ENTER."
[error] script [ login ] stopped with error in line 15
[error] FindFailed ( main_window_available_rooms.png: (1257x148) seen at (1999, 1797) with 0.74 in R[0,0 3840x
2160]@S(0) )
[error] --- Traceback --- error source first
line: module ( function ) statement
65: Sikuli ( wait )      return SCREEN.wait(target, timeout)
15: main ( <module> )    wait("main_window_available_rooms.png",10)
[error] --- Traceback --- end -----
```

There are some parameters which control how "strict" the comparison is, so I'll look at that to see if that can be used.

Another test I performed was to use Firefox instead of Chrome using the same script. It never found the initial os_logon.png. I'll need to understand their errors, to determine if there is a way to have both work. Perhaps just OCR recognition of the text on the screen would work.

#18 - 06/17/2021 05:34 PM - Roger Borrello

Newer documentation for SikuliX is here: <https://sikulix.github.io/docs>

Sikuli relies heavily on OpenCV template matching, which is documented here:
https://docs.opencv.org/2.4/doc/tutorials/imgproc/histograms/template_matching/template_matching.html

#19 - 06/21/2021 01:14 PM - Roger Borrello

Once I delved into trying to use the same screen grabs as templates for both the OS logon screen in Firefox and Chrome, it was obvious that there was more needed. I had to install both OpenCV and a newer Tesseract than was in my distro.

OpenCV

https://docs.opencv.org/4.5.2/d7/d9f/tutorial_linux_install.html

This installed and build easily, except the last install needs to be performed with sudo, which wasn't mentioned in the main documentation.

Tesseract

<https://launchpad.net/~alex-p/+archive/ubuntu/tesseract-ocr-devel>

So far I need to get the leptonica-progs library as a pre-req

#20 - 06/22/2021 08:14 AM - Greg Shah

This is the list of known issues to resolve, as far as I know:

- Make a single test work on both Chrome and Firefox.
- Move to Java instead of Python for the test encoding. This will allow us to embed arbitrary Java logic/control flow to enhance the tests.
- Make OCR work in the tests. This will allow a more flexible test.
- Enable a single test to work across themes. If this is possible, the OCR support is probably critical.
- Automate the startup of the browser and the OS logon in virtual desktop mode.
- Design an approach to execute these tests in a batch, capturing a report of the results. This should be checked in to a customer's application cfg project in bsr and we should be able to use a single command to execute the test suite against a running server.
- Create real examples of full business scenarios for a customer application.
- Document how to write and execute tests, including how to avoid the most common problems/pitfalls.

Roger: Let me know if I am missing anything. Also, please post updates as you resolve the above items.

#21 - 06/22/2021 09:28 AM - Roger Borrello

That looks very comprehensive, and will keep this post updated.

#22 - 06/22/2021 04:56 PM - Roger Borrello

Some notes on building leptonica-1.81.1: <http://www.leptonica.org/source/README.html>

1. mkdir build && cd build
2. cmake ..
3. make
4. sudo make install

After running sudo apt-get install libleptonica-dev I got past the build issue noted for tesseract.

Some notes on building tesseract-4.1.1: <https://tesseract-ocr.github.io/tessdoc/Compiling.html>

1. mkdir build && cd build
2. cmake ..
3. make
4. sudo make install

Some notes on building opencv-4.5.2: https://jdhaio.github.io/2020/12/07/build_opencv_cpp_library_linux/

1. mkdir build && cd build
2. cmake ..
3. make
4. sudo make install

sudo apt install xdotool and sudo apt install wmctrl are required for using the App class.

Info on using in Java: <https://sikulix-2014.readthedocs.io/en/latest/faq/030-java-dev.html>

The javadoc: <https://raiman.github.io/SikuliX1/javadocs/allclasses.html>

#23 - 06/25/2021 11:06 AM - Roger Borrello

The API is so very, very granular and very expansive. There is a separate project for this: <https://code.google.com/archive/p/sikuli-api/>

#24 - 06/25/2021 12:44 PM - Roger Borrello

Roger Borrello wrote:

The API is so very, very granular and very expansive. There is a separate project for this: <https://code.google.com/archive/p/sikuli-api/>

Unfortunately, that project seems to have ended with an older version of the OpenCV libraries (2.4 instead of latest 4.5.2). My Hello World won't run without building the older version.

The newer Sikuli project incorporates the API into it, but there isn't much sample documentation.

#25 - 06/25/2021 01:31 PM - Roger Borrello

There is some courseware on <http://researcherstore.com>. 55 lessons in all. It's located here: <https://researcherstore.com/courses/sikuli/>

Also a sample finding text in an image: <https://www.pyimagesearch.com/2018/09/17/opencv-ocr-and-text-recognition-with-tesseract/>

#26 - 06/30/2021 09:09 PM - Roger Borrello

In trying to utilize OCR, I was following the path of the python linkages for OpenCV and tesseract, such as <https://www.pyimagesearch.com/2018/09/17/opencv-ocr-and-text-recognition-with-tesseract/>

They mention importing the below libraries, which facilitate text detection and recognition.

```
import cv2
import pytesseract
import imutils
```

I tried adding them into the sikuli python script, but it cannot find the module.

#27 - 06/30/2021 09:20 PM - Roger Borrello

The previous script that found and opened/closed the about screen:

```
setShowActions(True)

Region(2616, 329, 971, 453)
# Bring up help -> about
wait("main_menu_bar.png", 5)
click(Pattern("main_menu_bar.png").targetOffset(67, 14))
wait("main_help_dropdown.png", 1)
click(Pattern("main_help_dropdown.png").targetOffset(-14, 47))
wait("help_about.png", 5)
click(Pattern("help_about.png").targetOffset(-4, 145))
```

was functional. However, I had to get past the initial OS logon screen. I added the findText call via a method:

```
def waitText(text, time):
    while time > 0:
        match = findText(text)
        if match: return match
        time -= 1
    return null
```

When I added calls:

```
waitText("User Name:", 10)
waitText("Password:", 10)
```

It would find the User Name: but stop on the Password: call. It is not clear how to determine why it won't find it. Documentation is scarce.

#28 - 07/01/2021 07:35 AM - Greg Shah

I tried adding them into the sikuli python script, but it cannot find the module.

The best way to install Python modules is using the normal apt-get repo approach, if they exist there. Or try `sudo apt search python3-<package>` (for Python v3) or `sudo apt search python-<package>` (for Python v2.7). Google can help you find the right package names.

For example, I tried `sudo apt search python3-cv2` and `sudo apt search python-cv2`. Both report this:

```
Sorting... Done
Full Text Search... Done
```

So that should do the trick.

If that doesn't work, then use pip which is the Python-specific installation manager (you may have to install it using `sudo apt-get install python3-pip` for Python v3 or `sudo apt-get install python-pip` for Python v2.7). `python3 -m pip install <package-name>` (for Python v3) or `python -m pip install <package-name>` (for Python v2.7) installs Python modules from its own Python repos but the result bypasses normal package processing.

#29 - 07/01/2021 12:56 PM - Roger Borrello

When I built these 3 tools (leptonica, tesseract, and opencv) I was confused by the fact that I built 1.8.1.1 of leptonica, but when I run `tesseract -v` it shows leptonica at 1.75. That's not germane, except that in the rebuild, I now have a situation where I cannot run the `sudo apt search python3-cv`. It returns:

```
aptitude: error while loading shared libraries: libncursesw.so.5: cannot open shared object file: No such file
or directory
```

I'm glad this is the old laptop I'm still working on!

I had to post a question to the launchpad for Sikuli (<https://answers.launchpad.net/sikuli/+question/697864>) as there is something incompatible going on such that I cannot start the IDE. This is in both laptops. All components are built, and I was trying to build the sikuli project, which is having issues unrelated.

New question #697864 on SikuliX:
<https://answers.launchpad.net/sikuli/+question/697864>

```
"main" #1 prio=5 os_prio=0 tid=0x00007f2a9c00a800 nid=0x176c3 in Object.wait() [0x00007f2aa08ac000]
  java.lang.Thread.State: WAITING (on object monitor)
    at java.lang.Object.wait(Native Method)
      - waiting on <0x000000071dcedfa8> (a java.lang.UNIXProcess)
    at java.lang.Object.wait(Object.java:502)
    at java.lang.UNIXProcess.waitFor(UNIXProcess.java:395)
      - locked <0x000000071dcedfa8> (a java.lang.UNIXProcess)
    at org.sikuli.script.runners.ProcessRunner.runBlocking(ProcessRunner.java:187)
    at org.sikuli.script.support.RunTime.start(RunTime.java:193)
    at org.sikuli.ide.Sikulix.main(Sikulix.java:13)
```

```
"main" #1 prio=5 os_prio=0 tid=0x00007fee8400b000 nid=0x176e8 waiting on condition [0x00007fee89b5b000]
java.lang.Thread.State: WAITING (parking)
    at sun.misc.Unsafe.park(Native Method)
        - parking to wait for <0x00000005d59e4040> (a java.util.concurrent.FutureTask)
    at java.util.concurrent.locks.LockSupport.park(LockSupport.java:175)
    at java.util.concurrent.FutureTask.awaitDone(FutureTask.java:429)
    at java.util.concurrent.FutureTask.get(FutureTask.java:191)
    at org.GNOME.Accessibility.AtkUtil.invokeInSwing(AtkUtil.java:68)
    at org.GNOME.Accessibility.AtkObject.hashCode(AtkObject.java:234)
    at org.GNOME.Accessibility.AtkWrapper.emitSignal(Native Method)
    at org.GNOME.Accessibility.AtkWrapper$5.propertyChange(AtkWrapper.java:545)
    at java.beans.PropertyChangeListener.fire(PropertyChangeListener.java:335)
    at java.beans.PropertyChangeSupport.firePropertyChange(PropertyChangeSupport.java:327)
    at java.beans.PropertyChangeSupport.firePropertyChange(PropertyChangeSupport.java:263)
    at javax.accessibility.AccessibleContext.firePropertyChange(AccessibleContext.java:768)
    at javax.swing.JScrollPane$AccessibleJScrollPane.stateChanged(JScrollPane.java:1506)
    at javax.swing.JViewport.fireStateChanged(JViewport.java:1369)
    at javax.swing.JViewport.reshape(JViewport.java:839)
    at java.awt.Component.setBounds(Component.java:2261)
    at java.awt.Component.setBounds(Component.java:2405)
    at javax.swing.ScrollPaneLayout.layoutContainer(ScrollPaneLayout.java:890)
    at java.awt.Container.layout(Container.java:1513)
    at java.awt.Container.doLayout(Container.java:1502)
    at java.awt.Container.validateTree(Container.java:1698)
    at java.awt.Container.validateTree(Container.java:1707)
    at java.awt.Container.validateTree(Container.java:1707)
    at java.awt.Container.validateTree(Container.java:1707)
    at java.awt.Container.validateTree(Container.java:1707)
    at java.awt.Container.validateTree(Container.java:1707)
    at java.awt.Container.validateTree(Container.java:1707)
    at java.awt.Container.validateTree(Container.java:1707)
    at java.awt.Container.validateTree(Container.java:1707)
    at java.awt.Container.validateTree(Container.java:1707)
    at java.awt.Container.validate(Container.java:1633)
        - locked <0x00000005d5680170> (a java.awt.Component$AWTTreeLock)
    at java.awt.Container.validateUnconditionally(Container.java:1670)
        - locked <0x00000005d5680170> (a java.awt.Component$AWTTreeLock)
    at java.awt.Window.show(Window.java:1033)
    at java.awt.Component.show(Component.java:1671)
    at java.awt.Component.setVisible(Component.java:1623)
    at java.awt.Window.setVisible(Window.java:1014)
    at org.sikuli.ide.SikulixIDE.initSikuliIDE(SikulixIDE.java:335)
    at org.sikuli.ide.SikulixIDE.main(SikulixIDE.java:88)
```

I have Java 8 as the default jre, and Java 11 is also installed, but is not primary, as you can see from the command line of the 2nd java process:

```
/usr/lib/jvm/java-8-openjdk-amd64/jre/bin/java -Dfile.encoding=UTF-8 -Dsikuli.IDE_should_run -cp  
/home/rfb/projects/sikuli/lib/sikulixide-2.0.5.jar:lib/sikulixide-2.0.5.jar org.sikuli.ide.SikulixIDE -d3 -c
```

The debug output is:

```
[38 debug] RunTime:temp folder ok: /tmp/Sikulix_1143884152
Options: *** options dump
Options: testing =
Options: OptionsFile = /home/rfb/.Sikulix/SikulixStore/SikulixOptions.txt
Options: *** options dump end
***** show environment for 2.0.5 IDE
user.home: /home/rfb
work dir: /home/rfb/projects/sikuli
running on Linux 5.11.0-22-generic from a jar
running Java Private Build 1.8.0_292-8u292-b10-0ubuntu1~20.04-b10
java.io.tmpdir: /tmp
app data folder: /home/rfb/.Sikulix
executing jar: /home/rfb/projects/sikuli/lib/sikulixide-2.0.5.jar
*** classpath dump sikulix
 0: file:/home/rfb/projects/sikuli/lib/sikulixide-2.0.5.jar
 1: file:/home/rfb/projects/sikuli/lib/sikulixide-2.0.5.jar
*** classpath dump end
***** show environment end
[134 debug] IDE: running with Locale: en_US
[DEBUG]startUp: IDE: --- Sikuli parameters ---
[DEBUG]startUp: IDE: 1: -d
[DEBUG]startUp: IDE: 2: 3
[DEBUG]startUp: IDE: 3: -c
[139 debug] RunTime:loadlib: trying JXGrabKey
[142 debug] RunTime:libsExport: folder exists: /home/rfb/.Sikulix/SikulixLibs (2.0.5 - 202103030910)
[143 debug] RunTime:loadLib: libJXGrabKey.so (success)
[144 debug] HotkeyManager: add Capture Hotkey: CTRL+SHIFT 2 (50, 3)
[147 debug] HotkeyManager: add Abort Hotkey: ALT+SHIFT C (67, 9)
[148 debug] IDE: Init ScriptingSupport
[152 debug] IDESupport: enter
[1500 debug] Runner: added: PackedSikulix [zip] application/zip
[1501 debug] Runner: added: SikulixExecutableJar [executablejar] text/jar
[1501 debug] Runner: added: NetworkRunner [] NET
[1502 debug] Runner: added: PackedSikulix [skl] text/skl
[1502 debug] Runner: added: Sikulix [] directory/sikulix
[1503 debug] Runner: added: Text [txt] text/text
[1504 debug] Runner: init: class org.sikuli.script.runners.AbstractLocalFileScriptRunner: warning: not possible
[1504 debug] Runner: added: SikulixJar [jar] text/jar
[1509 debug] Runner: added: Jython [py] text/jython
[1510 debug] IDESupport: added: Jython
[1510 debug] IDESupport: added: Text
[1511 debug] IDESupport: exit: defaultrunner: Jython (py)
[1513 debug] IDE: Reading Preferences
[1513 debug] IDE: Adding components to window
[1521 debug] IDE: [I18N] menuRecent
[1541 debug] IDE: [I18N] menuToolAndroid
[1545 debug] IDE: creating tabbed editor
[1662 debug] IDE: creating message area
[1715 debug] IDE: creating combined work window
[1718 debug] IDE: Putting all together
[1718 debug] IDE: Putting all together - after main pane
[1756 debug] IDE: Putting all together - after toolbar
[1759 debug] IDE: Putting all together - before layout
[1787 debug] IDE: Putting all together - after layout
[1788 debug] IDE: Putting all together - Restore last Session
[1800 debug] EditorPane: created 1625523921045
[1810 debug] JythonRunner: starting initialization
[1839 debug] EditorPane: InitTab: (text/jython)
[1845 debug] ImagePath: new BundlePath: /tmp/Sikulix_1143884152/SikulixIDETempTab1625523921045
[1854 debug] Screen: initScreens: starting
[1855 debug] Screen: Accessing: GraphicsEnvironment.getLocalGraphicsEnvironment()
[1855 debug] Screen: Accessing: GraphicsEnvironment.getLocalGraphicsEnvironment().getScreenDevices()
[1855 debug] Screen: Monitor 0: (0, 153) 1920 x 1080
[1857 debug] Screen: Monitor 1: (1920, 0) 1920 x 1080
[1857 debug] Screen: No ScreenDevice has (0,0) --- using 0 as primary: java.awt.Rectangle[x=0,y=153,width=1920
```



```
,height=1080]
[1861 debug] Screen: initScreens: monitor mouse check
[1866 debug] Screen: *** checking: R[0,153 1920x1080]@S(0) center: (960, 693) --- OK
[1869 debug] Screen: *** checking: R[1920,0 1920x1080]@S(1) center: (2880, 540) --- OK
[1870 debug] Device: Mouse: moved externally: now (2873,533) was (2880,540) (mouseMovedResponse 0)
[1872 debug] Screen: initScreens: ending
[1873 debug] IDE startup: 1.9 seconds
[debug] Jython: added as Jython::sys.path[0]:
/home/rfb/.Sikulix/Lib/site-packages
[debug] Jython: ***** sys.path
0: /tmp/Sikulix_1143884152/SikulixIDETempTab1625523921045
1: /home/rfb/.Sikulix/Lib/site-packages
2: /home/rfb/.Sikulix/Lib
3: /home/rfb/projects/sikuli/lib/Lib
4: /home/rfb/projects/sikuli/lib/sikulixide-2.0.5.jar/Lib
[3179 debug] JythonRunner: ready: version 2.7.2
<<< Control-C Pressed at this point >>>
[debug] RunTime:***** final cleanup at System.exit() *****
[debug] HotkeyManager: reset: removed all SikuliX hotkeys.
[debug] FileManager: deleteFileOrFolder:
/tmp/Sikulix_1143884152
```

I tried cleaning off the ~/.Sikulix folder and restarting, but no change in the behavior.

The ~/.Sikulix/SikulixLibs/ directory contains
2.0.5_202103030910_MadeForSikuliX64L.txt libJXGrabKey.so libopencv_java430.so

I copied in the libopencv_java452.so that I had built (which I also built tesseract and leptonica):

```
rfb@rfb:~/projects/sikuli$ tesseract -v
tesseract 4.1.1
leptonica-1.79.0
libgif 5.1.4 : libjpeg 8d (libjpeg-turbo 2.0.3) : libpng 1.6.37 : libtiff 4.1.0 : zlib 1.2.11 : libwebp 0.6.
1 : libopenjp2 2.3.1
Found AVX2
Found AVX
Found FMA
Found SSE
```

Any help getting this running would be appreciated!

New question #697865 on SikuliX:

<https://answers.launchpad.net/sikuli/+question/697865>

I am not 100% sure I am building correctly, but I was running mvn compile from the SikuliX1-2.0.5-final directory. It results in:

```
rfb@rfb:~/projects/SikuliX1-2.0.5-final$ mvn compile
[INFO] Scanning for projects...
[INFO] -----
[INFO] Reactor Build Order:
[INFO]
[INFO] SikuliX API [jar]
[INFO] SikuliX IDE [jar]
[INFO] sikulix [pom]
[INFO] -----
[INFO] -----< com.sikulix:sikulixapi >-----
[INFO] Building SikuliX API 2.0.5 [1/3]
[INFO] -----[ jar ]-----
[INFO]
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ sikulixapi ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Copying 1 resource
[INFO] Copying 76 resources
[INFO]
[INFO] --- maven-compiler-plugin:3.8.1:compile (default-compile) @ sikulixapi ---
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 211 source files to /home/rfb/projects/SikuliX1-2.0.5-final/API/target/classes
[INFO] /home/rfb/projects/SikuliX1-2.0.5-final/API/src/main/java/org/sikuli/script/Env.java: Some input files
use or override a deprecated API.
[INFO] /home/rfb/projects/SikuliX1-2.0.5-final/API/src/main/java/org/sikuli/script/Env.java: Recompile with -X
lint:deprecation for details.
[INFO] /home/rfb/projects/SikuliX1-2.0.5-final/API/src/main/java/org/sikuli/script/Region.java: Some input fil
es use unchecked or unsafe operations.
[INFO] /home/rfb/projects/SikuliX1-2.0.5-final/API/src/main/java/org/sikuli/script/Region.java: Recompile with
-Xlint:unchecked for details.
[INFO] -----
[ERROR] COMPILATION ERROR :
[INFO] -----
[ERROR] /home/rfb/projects/SikuliX1-2.0.5-final/API/src/main/java/org/sikuli/script/support/Observer.java:[125
,26] incompatible types: PSC cannot be converted to int
[INFO] 1 error
[INFO] -----
[INFO] -----
[INFO] Reactor Summary for sikulix 2.0.5:
[INFO]
[INFO] SikuliX API ..... FAILURE [ 3.602 s]
[INFO] SikuliX IDE ..... SKIPPED
[INFO] sikulix ..... SKIPPED
[INFO] -----
[INFO] BUILD FAILURE
[INFO] -----
[INFO] Total time: 3.736 s
[INFO] Finished at: 2021-07-05T18:44:56-04:00
[INFO] -----
[ERROR] Failed to execute goal org.apache.maven.plugins:maven-compiler-plugin:3.8.1:compile (default-compile)
on project sikulixapi: Compilation failure
[ERROR] /home/rfb/projects/SikuliX1-2.0.5-final/API/src/main/java/org/sikuli/script/support/Observer.java:[125
,26] incompatible types: PSC cannot be converted to int
[ERROR]
[ERROR] -> [Help 1]
[ERROR]
[ERROR] To see the full stack trace of the errors, re-run Maven with the -e switch.
[ERROR] Re-run Maven using the -X switch to enable full debug logging.
[ERROR]
[ERROR] For more information about the errors and possible solutions, please read the following articles:
[ERROR] [Help 1] http://cwiki.apache.org/confluence/display/MAVEN/MojoFailureException
```

Perhaps I missed some initial setup, but I cannot find instructions for the source build.

#33 - 07/07/2021 01:05 PM - Roger Borrello

I used a fresh install on my VM.

1. I downloaded the sikulixide jar from <https://launchpad.net/sikuli/sikulix/2.0.5/+download/sikulixide-2.0.5.jar> to ~/projects/sikuli/lib
2. I installed tesseract-ocr using sudo apt install tesseract-ocr
3. I ran java -jar lib/sikulixide-2.0.5.jar -c -d 3

Debug statements were the same, and it hangs without displaying the IDE.

RaiMan provided this answer to my question:

Your question #697864 on SikuliX changed:

<https://answers.launchpad.net/sikuli/+question/697864>

Status: Open => Answered

RaiMan proposed the following answer:

Running Ubu20 in VirtualBox on Windows 10 I had no problem to start and use the 2.0.5 IDE with

```
java -jar sikulixide....jar -v -c
```

using "sudo apt install" I installed the default jre (which is 11).

I only had a startup problem with some canberra-gtk stuff, which could be solved by installing the stuff.

I think that this is a problem with your GUI-manger in your environment.

... but I have no idea how to help you

#34 - 07/07/2021 07:18 PM - Roger Borrello

I'm not sure what else I can try to get Sikuli working. I've tried on a Kubuntu VM, as well.

#35 - 07/08/2021 06:49 AM - Greg Shah

Are you running it with Java 11?

Can you get it running without the OCR nonsense?

#36 - 07/08/2021 09:34 AM - Roger Borrello

Greg Shah wrote:

Are you running it with Java 11?

I was running with the default java 8. I ran this: `/usr/lib/jvm/java-11-openjdk-amd64/bin/java -jar lib/sikulixide-2.0.5.jar`

...and the IDE starts. So I went back to the old laptop, and it starts with the default java 8. So it should work.

I compared output of the ide debug, and it comes down to:

```
executing jar: /home/rfb/projects/sikuli/lib/sikulixide-2.0.5.jar
*** classpath dump sikulix
  0: file:/home/rfb/projects/sikuli/lib/sikulixide-2.0.5.jar
  1: file:/home/rfb/.Sikulix/Extensions/jruby-complete-9.2.0.0.jar
  2: file:/home/rfb/projects/sikuli/lib/sikulixide-2.0.5.jar
*** classpath dump end
```

from the old laptop, and from the new:

```
*** classpath dump sikulix
  0: file:/home/rfb/projects/sikuli/lib/sikulixide-2.0.5.jar
  1: file:/home/rfb/projects/sikuli/lib/sikulixide-2.0.5.jar
*** classpath dump end
```

For whatever reason, the jruby extension was deployed to my old laptop, but not my new.

#37 - 07/08/2021 03:55 PM - Roger Borrello

Answers from RaiMan are basically, "Use Java 11".

So I created `/usr/local/bin/java11` with links to Java 11 items. I'll put that in the front of my path for these things. I can run the IDE, and I can actually build the source code now.

#38 - 07/08/2021 05:36 PM - Roger Borrello

Validation of opencv python3 binding:

```
sudo apt install python3-opencv
python3 -c "import cv2; print(cv2.__version__)"
4.2.0
```

#39 - 07/12/2021 05:56 PM - Roger Borrello

I have a pretty good reliable logon script, except for the password being echoed. I've posted a question: <https://answers.launchpad.net/sikuli/+question/697957>

I'm working on building up some functions that can be reused:

```
import getpass

def maximize_chrome(image):
    if not exists(image):
        type(Key.SPACE, Key.ALT)
        type("x")
    return

def waitText(text, time):
    while time > 0:
        match = findText(text)
        if match: return match
        time -= 1
    return null

myApp = App("chrome")
myApp.open()

type("l", Key.CTRL)
type("https://localhost:7443/gui\n")
maximize_chrome("chrome_maximized.png")

#Region(1920,0,830,391)
#click(existsText("Maximize",5))

u = input('Userid: ')
p = input('Password: ')
```

The region control is very tricky, setting it up properly so that it could run on multiple resolutions. I'll be posting a question about that, too.

#40 - 07/14/2021 05:47 PM - Roger Borrello

I added sikuli capabilities to my Windows system, in order to determine if there is any hope for cross-compatability with testcases. Sikuli lived up to it reputation:

- I was able to load the IDE just fine with Java 8 on that Windows desktop. This is contrary to what I was told by RaiMan... that I need to use Java 11
- I copied over the c-act_demo_about testcase, which currently uses OCR to determine if **Userid:** is on the region.
- Switching to Chrome and maximizing it, requesting userid/password worked. Then it fails with an error indicating that the same call I was using find text (findText) did not have proper parameters, because the : cannot be used in a filename. So the interface to the OCR isn't found in the same manner on Windows.
- I introduced tesseract-ocr on the Windows box, and configured the path so it could find the .exe and .dll
- At this point, sikuli no longer loads, just hangs in the same manner as on my Linux when I was told I would need to use Java 11. I tried several things to get it going... now trying a reboot.

#41 - 07/15/2021 09:20 AM - Roger Borrello

Question #697943 on SikuliX changed:

<https://answers.launchpad.net/sikuli/+question/697943>

Roger F Borrello Jr posted a new comment:

Updated the Java on the Windows 10 desktop to OpenJDK 11, and the same problem of the IDE hanging after the splash screen.

I tried removing the appdata\roaming\sikuli directory, and still hangs. tried relocating the sikulixapi-2.0.5.jar and sikulixide-2.0.5.jar to a different directory (C:\sikulix instead of C:\sikuli) and still hangs at the splash screen.

I didn't mean to hijack Parsawit's thread, but there is something that hangs up the IDE waiting for something to initialize and multiple people have reported this. If we cannot get past the loading of the IDE, we won't be able to really explore the capabilities of this.

#42 - 07/15/2021 07:25 PM - Roger Borrello

Subsequent to that post, I learned from looking at the output that the existing project was causing issues. So renaming the directory allowed the IDE to start. It most likely is a bug in the IDE, because the project is actually working.

I was able to get the same logon steps to work on Windows and Linux... hooray! A small victory, because there is OCR involved, but the API behaved differently on Windows. I was able to modify it to work on both, but things like that create a slow development process.

#43 - 07/16/2021 02:26 PM - Roger Borrello

I had to switch to using an old VM because my test systems are now devoid of functional databases. I was successful in at least getting my Windows system connecting to the new URL, but now it's gotten finicky again with respect to determining if the User name field is on the screen in the region specified. That is one of the frustrations I am seeing, in that the region of the screen I am concerned with looking at seems to move around, based upon the variables of OS, resolution, font, where the browser is positioned on the screen (which is why I want to make sure to maximize the browser before proceeding with testcases).

If I make the region too generous, I believe other pieces of text or graphics, may contribute to not making a proper determination of a matching screen. Overall, it is a challenge, and I am discouraged this methodology would work. If a test is setup properly, and there is uniformity in the location where the testcase is executed versus captured, and the data is uniform, there could be a case for browser/os-specific screen captures. This would work, and there would be effort around driving the testcases, based upon configuration of the wrapper script.

#44 - 07/19/2021 08:16 AM - Greg Shah

I was successful in at least getting my Windows system connecting to the new URL, but now it's gotten finicky again with respect to determining if the User name field is on the screen in the region specified. That is one of the frustrations I am seeing, in that the region of the screen I am concerned with looking at seems to move around, based upon the variables of OS, resolution, font, where the browser is positioned on the screen

Please ask the developers of SikuliX for their recommendations on these issues. If they have a solution, we want to know about it. And if they don't have a solution we want to know about it (so we can stop wasting time searching for something that may not exist).

#45 - 07/21/2021 08:56 AM - Roger Borrello

Posted question: <https://answers.launchpad.net/sikuli/+question/698070>

Hi. I have looked over some questions/answers like <https://answers.launchpad.net/sikuli/+question/675303> to determine if our usage of capturing testcases on one environment, and running them on another is even possible. It sounds like image captures will need to be specific to the system under test. In other words, captured Windows Chrome images should only be used on Windows Chrome tests. If we run the same tests on Ubuntu Chrome, they will need to use Ubuntu Chrome images to validate the functions.

If my above assumption is correct, what about utilizing more text-based comparisons? Given that we would know what text should appear in certain regions, I would expect that we could achieve a level of independence. But I am not seeing consistent results when I attempt this, and I am trying to determine what is at fault. Maybe I am choosing a region and the location of the browser on the target may be offset somewhat, so the same text is not found?

I am a beginner at this, and would like to know more about why text is not found, so I can make corrections and improvements.

Posted answer:

--- same image in other environments

The image search used in SikuliX is based on pixel content. An image can only be found at all, if width and height in pixels of the sample image is the same as it is shown at time of search on the screen.

The more pixels differ, the lower the match score. The lower the match score, the higher the risk for false positives.

Differences in pixel content in different environments are mainly produced by differences in the rendering behavior (how is a given graphic element shown on screen).

To minimize all the risks (besides taking care for same size) is to concentrate the snapshot on the key elements of the image, that makes it unique on the current screen. Have as little surrounding background (< 10-20%).

If you have the need to run your stuff in different environments, be prepared, to have different image sets (ImagePath feature, naming convention, re-capture on the fly, ...)

--- using text based on OCR

... is surely something you can use additionally in special situations.

But as an overall replacement for image search in my experience it is not really useable.

In the end it also has many quirks, that you have to accept or have to implement solutions/workarounds.

#46 - 10/15/2021 09:47 AM - Roger Borrello

I have been trying all morning to get sikulixide to run. It's back to it's hanging situation. This was working last week. The main difference is I don't have an external monitor connected. It is a very fragile tool.

#47 - 10/15/2021 10:03 AM - Roger Borrello

Roger Borrello wrote:

I have been trying all morning to get sikulixide to run. It's back to it's hanging situation. This was working last week. The main difference is I don't have an external monitor connected. It is a very fragile tool.

I ran this command, and it started working again:

```
rm -fr ~/.java/.userPrefs/org/sikuli/
```


#48 - 06/13/2022 06:29 PM - Roger Borrello

A new article on Sikuli was posted on Software Testing Help: <https://www.softwaretestinghelp.com/sikuli-tutorial-part-1/>

#49 - 10/19/2022 01:55 PM - Greg Shah

Roger: Please post your wishlist of features, fixes and/or answers that would make our work with Sikuli more successful/productive.

#50 - 10/19/2022 02:08 PM - Greg Shah

Here is the list of items that I know of:

- font differences between the legacy Windows/OpenEdge system and the new Linux/FWD/Web system means that we cannot use images captured on OpenEdge to test FWD
- window sizing and positioning problems lead to hard coding tests to a specific static configuration of OS platform + video resolution + video color depth + browser (e.g. these tests only work on Windows with Chrome and a resolution of 1920 x 1080 pixels and 24-bit color depth)
- OCR does not work out of the box and/or may require some specific training for the OCR engine which we have not yet accomplished

Roger: Did I misunderstand any of this?

#51 - 10/19/2022 03:50 PM - Roger Borrello

Greg Shah wrote:

Here is the list of items that I know of:

- font differences between the legacy Windows/OpenEdge system and the new Linux/FWD/Web system means that we cannot use images captured on OpenEdge to test FWD
- window sizing and positioning problems lead to hard coding tests to a specific static configuration of OS platform + video resolution + video color depth + browser (e.g. these tests only work on Windows with Chrome and a resolution of 1920 x 1080 pixels and 24-bit color depth)
- OCR does not work out of the box and/or may require some specific training for the OCR engine which we have not yet accomplished

Roger: Did I misunderstand any of this?

There are some things I have learned in the past few months (ok, year)... Let me give a narrative, since they won't match up one-to-one with your bullets.

Platform Independence

I have had some measure of success utilizing the screens I captured on Windows Chrome for testing under Linux Chrome. More testing is needed in this area to determine pitfalls and workarounds. I did have to make sure the testcase was running on the laptop monitor, and not the external monitor. This might be due to the coordinates of the regions starting on the laptop monitor, and stretching to the external. If that is a requirement, perhaps the testcase could do some self-configuration that SikuliX could accommodate. This isn't to say that we'd need any significant SikuliX work to get that working 100%. Things like launching the browser or other OS-specific operations could utilize the coding techniques SikuliX already has for determining OS. I just haven't focused efforts in that area past the initial demo.

The OE versus FWD differences are numerous such that I doubt the images captured from OE could every be used for comparison on our WebClient. The "look-and-feel" aspects of a pure Windows render-er would never match up bit-by-bit. In terms of OCR use to overcome those differences, well you wouldn't be using the original screens in that case anyway.

I have not attempted to run any tests cross-resolution, since both my laptop and external monitor are 1920x1080. My Windows desktop is also 1920x1080, and the Remote Desktop Connection to the customer's Windows desktop has an unknown resolution. Windows doesn't let you see that information when you are connected via RDC.

Scripting and Usability

The IDE has "grown" on me once I understood how to utilize the image thumbnails to determine regions and target offsets (and rename image files). I know you'd rather the script that is built from it be Java code, but the fact that most testcases are just a series of **1) data entry 2) actions, 3) comparison of resultant screens, and 4) repeat** wouldn't leverage much of Java's language features at all. It's really a series of steps to follow.

The process of creating the testcases can be tedious. You follow a script, and perform data entry and capture screens along the way. The only thing that could speed it up is if SikuliX IDE had a "record" button, and just stored the screen captures, mouse clicks, and data entry into the script. This is what you would imagine it *should* do, but doesn't.

A good feature would be if the IDE let you select numerous areas to "mask" out of the comparison. Right now I have to capture multiple areas of the screen, and perform multiple comparisons in the script. They let you mask, but <https://sikuliX-2014.readthedocs.io/en/latest/tutorials/masking/masking.html> is not really addressing the same use case of matching everything *except* a few areas on the screen.

OCR

We definitely need this to be bullet-proof and usable. Many places in SikuliX documentation it is still deemed "experimental". An in-depth understanding of tesseract itself is needed to know the features and limitations. It doesn't look like tesseract has been getting much attention for the last few years.

As far as I know, the usage via SikuliX out-of-the-box is, but not that accurate. You are required to define a specific region to pull the text out of. If you span multiple columns of data, the returned text won't be usable, so this is time consuming when a screen has data all over it that you want to verify (and may change from when a screen capture was taken). Think of a browse. You cannot use OCR against that control, as there isn't any determination of columns. All the columns would be pressed together. If there were a way for it to see "grids" and substitute commas or tabs, at least you'd get back some usable data to parse.

#52 - 10/20/2022 04:43 AM - Hynek Cihlar

Have we considered comparing widget trees?

By comparing a widget tree I mean to take a snapshot of all the widgets on the screen, the hierarchy, and their attributes and compare them between OE and FWD. I assume snapshots are possible to be created in OE, there are runtime tools that allow to inspect widgets on the screen.

The advantages of this approach:

- Effortless creation of the test cases.
- Good comparability of the snapshots.
- Easy to read reports.
- 100% precision for 80% of the cases
- Perfect for confirming screen content and the application behavior.

The disadvantages:

- Useless for some visual differences (colors, fonts).
- Will not work for 20% of the cases (OCXes, more complex widgets (browse?)).

I think this approach would be worthy even for the disadvantages. The functional side of the application is more important than the visual side. For the cases where comparing widget trees would not work other means of testing could be employed (human, SikuliX, or something else). For the type of applications we deal with the majority of the screens are simple widgets and frame layouts.

#53 - 10/20/2022 05:04 AM - Greg Shah

The "look-and-feel" aspects of a pure Windows render-er would never match up bit-by-bit.

Other than the fonts, they should.

In terms of OCR use to overcome those differences, well you wouldn't be using the original screens in that case anyway.

Why not?

I know you'd rather the script that is built from it be Java code

What I want is to be able to write Java code that does something useful (possibly all the way into FWD or CounterAct) and be able to call it from a test. Python is useless there. I'm fine to use Python otherwise, but we really would want to be able to call arbitrary Java code from inside the Python script (which we know is already running in Jython in the JVM).

#54 - 10/20/2022 05:12 AM - Greg Shah

Have we considered comparing widget trees?

By comparing a widget tree I mean to take a snapshot of all the widgets on the screen, the hierarchy, and their attributes and compare them between OE and FWD. I assume snapshots are possible to be created in OE, there are runtime tools that allow to inspect widgets on the screen.

Yes, this is a very good idea. It is what I was referring to in [#3704-1](#) by this:

I am OK with exposing an API in our client to allow the test suite to feed input and read output/state. I'm OK with using the widget hierarchy and widget state to confirm as much as possible rather than using only bitmap comparisons with the 4GL. On the other hand, I think it is inevitable that we do some non-trivial amount of bitmap comparison to ensure that we are exactly matching the look/feel of the 4GL.

As you say, it could get us 80% coverage and would be very reliable.

I think this approach would be worthy even for the disadvantages. The functional side of the application is more important than the visual side. For the cases where comparing widget trees would not work other means of testing could be employed (human, SikuliX, or something else). For

the type of applications we deal with the majority of the screens are simple widgets and frame layouts.

I agree, though I do think we could (and should) use this technique even from SikuliX tests which have the advantage of testing real input events.

It also seems to me that we could open up an API for feeding the low level input events in and this would allow us to do the 80% tests outside of SikuliX, fully automated, quickly.

What is the work effort to build this?

#55 - 10/20/2022 05:25 AM - Hynek Cihlar

Greg Shah wrote:

What is the work effort to build this?

The high level tasks for this would be:

- Create driver (accessible by the test creation tool/process) for enumerating widgets in OE.
- Create driver (accessible by the testing engine) for enumerating widgets in FWD.
- Identify the state to uniquely identify widgets in the tree.
- Identify the key attributes to use for comparison.
- Implement the comparison logic.
- Implement delta reporting.
- An integration in the testing engine.

Effort 8 man weeks.

#56 - 10/20/2022 06:30 AM - Greg Shah

Create driver (accessible by the test creation tool/process) for enumerating widgets in OE.

Would this include both of these?

- widget state (read from 4GL attributes/methods and other 4GL UI statements)
- WIN32 state

The 4GL widget state seems like the most important part. It seems to me that such code will need to be integrated with the application itself so that the 4GL enumeration code can execute inside OpenEdge.

I guess this must at least allow activating a "record" mode for a specific application, executing the scenario and then "stop" recording. All of this state

would need to be written into a standard format. I think XML would probably be preferred for the snapshot.

Perhaps we should just implement a snapshot tool that gets executed as a one shot capture of state. Then this could be integrated into other recording tools like SikuliX. If the code was fully 4GL based, we could register a trigger for an obscure key combination using "global" ANYWHERE and then take snapshots while running the application.

Create driver (accessible by the testing engine) for enumerating widgets in FWD.

The same 4GL enumeration code could be converted and included in FWD itself, with a registration feature to activate it. We could also use hand coded Java but then it would have to be edited whenever the original enumeration code is edited.

An integration in the testing engine.

We would want to support usage from the following test engines:

- Harness
- ABLUnit/OEUnit
- Sikuli

How to implement this in a manner that can easily be surfaced in multiple test engines? If the core enumeration is built into FWD, we at least must be able to invoke the capture from the test engine.

I also think we really need to include support for feeding key/mouse input.

#57 - 10/20/2022 11:01 AM - Hynek Cihlar

Greg Shah wrote:

Create driver (accessible by the test creation tool/process) for enumerating widgets in OE.

Would this include both of these?

- widget state (read from 4GL attributes/methods and other 4GL UI statements)
- WIN32 state

The 4GL widget state seems like the most important part.

I think so. The majority of the state will be the legacy state. There will be some Win32 stuff, too, but very limited. I am currently aware about system focus handle , which is tracked at Win32 layer.

It seems to me that such code will need to be integrated with the application itself so that the 4GL enumeration code can execute inside OpenEdge.

I guess this must at least allow activating a "record" mode for a specific application, executing the scenario and then "stop" recording. All of this state would need to be written into a standard format. I think XML would probably be preferred for the snapshot.

In order to have the least impact we should implement this by some other means than adding new triggers. For example as a simple socket server. It would listen for commands and would be capable to return state through the socket.

I'm not sure whether we can detect the wait-for input state, i.e. when application waits for user input. The most natural check points when to take snapshots would be the times when the execution stops waiting for user input. It would be very straight forward if the user just hit a "record" button, ran the scenario and the snapshots would be generated for the whole UI flow. Perhaps we could leverage the 4gl to 4gl conversion FWD feature and emit a "take snapshot" call before any of the "wait-for" statements.

We should also give the user full control and allow him to take a snapshot at any time.

Another random idea. We could hook the Win32 window procedures handling Win32 messages, identify the user input events and take snapshots before the input events were dispatched.

Perhaps we should just implement a snapshot tool that gets executed as a one shot capture of state. Then this could be integrated into other recording tools like SikuliX. If the code was fully 4GL based, we could register a trigger for an obscure key combination using "global" ANYWHERE and then take snapshots while running the application.

Create driver (accessible by the testing engine) for enumerating widgets in FWD.

The same 4GL enumeration code could be converted and included in FWD itself, with a registration feature to activate it.

This wouldn't work for any of the native code in the 4GL driver (fetching Win32 state, hooking Win32 events, etc.). For this I would tap directly in our Java runtime.

An integration in the testing engine.

We would want to support usage from the following test engines:

- Harness
- ABLUnit/OEUnit
- Sikuli

How to implement this in a manner that can easily be surfaced in multiple test engines? If the core enumeration is built into FWD, we at least must be able to invoke the capture from the test engine.

Again a simple socket server comes to my mind. We can hide any of the snapshot processing stuff in helper libraries.

I also think we really need to include support for feeding key/mouse input.

Can't we just do this through the standard system facilities to post input events to the Java client process or browser? I guess this should be already part of the test engines.

#58 - 10/20/2022 11:28 AM - Hynek Cihlar

Hynek Cihlar wrote:

Greg Shah wrote:

The same 4GL enumeration code could be converted and included in FWD itself, with a registration feature to activate it.

This wouldn't work for any of the native code in the 4GL driver (fetching Win32 state, hooking Win32 events, etc.). For this I would tap directly in our Java runtime.

On a second thought having one set of legacy code for this isn't a bad idea. Any native part of the driver could be conditionally split into system calls vs Java calls into FWD runtime.

I was thinking about potential false positives when using legacy API, which would be itself subject to testing. But in the sheer size of the testing application any false positives would eventually reveal as legitimate faults.

#59 - 10/20/2022 11:42 AM - Greg Shah

Can't we just do this through the standard system facilities to post input events to the Java client process or browser? I guess this should be already part of the test engines.

This is how Sikuli does it, but it limits us greatly because we must ensure that the system level input focus is on exactly the correct application (and 4GL window in GUI) which means there is a lot of extra test code written to make that work. And that extra code is fragile.

Instead, I'd like a scenario where we can have a rock solid, simple approach to input that can be used to automate most cases and get that 80% coverage.

#60 - 10/20/2022 12:47 PM - Hynek Cihlar

Greg Shah wrote:

Can't we just do this through the standard system facilities to post input events to the Java client process or browser? I guess this should be already part of the test engines.

This is how Sikuli does it, but it limits us greatly because we must ensure that the system level input focus is on exactly the correct application (and 4GL window in GUI) which means there is a lot of extra test code written to make that work. And that extra code is fragile.

Interesting, I would expect this to be the easy part. How GUI terminal apps like VNC do this?

Instead, I'd like a scenario where we can have a rock solid, simple approach to input that can be used to automate most cases and get that 80% coverage.

So how would this work? Do we create a browser plugin to allow interaction with JS client?

#61 - 10/20/2022 01:39 PM - Greg Shah

How GUI terminal apps like VNC do this?

These clients are operating at a lower level. They are probably pumping input into the GUI system's event queue and letting the system figure out focus and routing.

We can't do that.

So how would this work? Do we create a browser plugin to allow interaction with JS client?

No, I'm talking about a case where the test engine is interacting with the JVM directly. It can feed input and we can stuff it into our event queue directly.

#62 - 10/20/2022 01:43 PM - Hynek Cihlar

Greg Shah wrote:

How GUI terminal apps like VNC do this?

These clients are operating at a lower level. They are probably pumping input into the GUI system's event queue and letting the system figure out focus and routing.

We can't do that.

Why not? :-)

So how would this work? Do we create a browser plugin to allow interaction with JS client?

No, I'm talking about a case where the test engine is interacting with the JVM directly. It can feed input and we can stuff it into our event queue directly.

Some of the UI state is kept in JS client. Feeding events directly in the JVM process could have unexpected side effects.

#63 - 11/08/2022 01:03 PM - Roger Borrello

- Related to Bug #6920: Ins and outs of Sikuli for GUI testing added

#64 - 11/08/2022 01:04 PM - Roger Borrello

For ins and outs of Sikuli, please refer to [#6920](#) which will focus purely on Sikuli

#65 - 04/12/2023 09:58 AM - Greg Shah

[Automating GUI Testing](#) is our guidance on how best to write Sikiuli tests.

#66 - 06/07/2023 10:05 AM - Roger Borrello

It would be advantageous to have a graphical environment in a Docker container, so that the variability of graphics could be restricted, making testing infrastructures like Sikuli more feasible.

Asked this question to ChatGPT:

Can I have a graphical desktop in a Docker container?

Answer:

Yes, it is possible to have a graphical desktop environment within a Docker container. Docker is primarily designed for running isolated, command-line-based applications, but with some additional setup and configuration, you can run graphical applications as well.

To set up a Docker container with a graphical desktop environment, you would typically follow these steps:

1. Choose a base image: Start with a base image that includes the necessary components for running a graphical desktop environment, such as X11 server and a window manager. Examples of such base images include `ubuntu`, `debian`, or specialized images like `jess/x11-desktop`.
2. Install necessary packages: In the Dockerfile, install the required packages for the graphical desktop environment, such as X11 server, window manager, desktop environment (e.g., GNOME, XFCE), and any additional tools or applications you need.
3. Set up X11 forwarding: Enable X11 forwarding to allow the graphical applications within the container to display on the host machine. This involves sharing the X11 socket and setting the `DISPLAY` environment variable in the container.
4. Run the container: When running the container, ensure that you mount the necessary volumes and set up the network configuration to allow X11 forwarding.

It's important to note that running a graphical desktop environment within a Docker container can be more complex and resource-intensive compared to running command-line applications. Additionally, there might be limitations depending on the host operating system and the availability of hardware acceleration.

There are also alternative solutions, such as using tools like VNC or accessing the host's X server directly, but they come with their own considerations and setup requirements.

It's recommended to search for specific guides or Docker images that cater to running graphical desktop environments to get more detailed instructions tailored to your specific use case.

#67 - 06/07/2023 10:09 AM - Roger Borrello

Does [#3](#) really disqualify it as "running graphics inside the container"? Because Sikuli would, again, need to run on the host.

#68 - 07/12/2023 11:53 AM - Roger Borrello

Today we saw the Cucumber-Sikuli infrastructure and how the Gerkin "feature files" drive the testing into custom Java classes, which then perform Sikuli interfaces.

Marian indicated it is very much the same approach he was planning on. Questions:

1. Is this the best approach for GCD?
2. Should we implement our own version? Perhaps a POC for Hotel GUI?

#69 - 07/12/2023 03:44 PM - Greg Shah

Is this the best approach for GCD?

You and Marian will have to answer this. If this approach makes it easier/less work/more reliable to write tests, then I think we should use it.

Should we implement our own version? Perhaps a POC for Hotel GUI?

Marian's team is already implementing UI test suites in [Testcases](#). I think we would use those.

Of course, it would be nice to be able to regression test Hotel GUI using automation, but I think the testcases should have priority since that is what needs to be automated first.

#70 - 07/13/2023 02:38 AM - Marian Edu

Greg Shah wrote:

Is this the best approach for GCD?

You and Marian will have to answer this. If this approach makes it easier/less work/more reliable to write tests, then I think we should use it.

For automated testing I don't see any other solution, there are many other tools beside Sikuli that can be used but since there is already some experience with the tool and based on what we've seen yesterday I think we can probably build something with the Sikuli Java API to cover those tests that require user interaction (wait-for, update editing...).

Should we implement our own version? Perhaps a POC for Hotel GUI?

Roger, you're probably thinking of application testing much like what we've seen for <large GUI app>, in which case using Cucumber/Geckio BDD on top of whatever we are implementing using Sikuli might make sense so a business analyst can write those test scenarios without a need to write any Java code. However, for that to work we must have the core functionality implemented and the requirements for testing a full application are somehow a bit different from what we're looking at for automating our UI tests.

Marian's team is already implementing UI test suites in [Testcases](#). I think we would use those.

There are some UI tests in there that does not require user interaction so those are plain ABLUnit tests, for anything that requires user interaction we need to use something like Sikuli but what we are looking at is a somehow simpler user interface where we only test a specific widget behaviour. Right now I think all we need is the ability to work with only one screen and maybe a simple grid-layout, I would like to avoid using images to identify widgets if possible although I'm not really sure this is possible at all.

I would say we can start with implementing functionality on as-needed basis, in the end it will probably offer all/most of functionality needed to be able to automate application testing and then we can use that on Hotel GUI and other customers.

Of course, it would be nice to be able to regression test Hotel GUI using automation, but I think the testcases should have priority since that is what needs to be automated first.

#71 - 07/19/2023 05:52 PM - Roger Borrello

Vladimir,

Can you describe your process for capturing images of glyphs in this task and provide some examples?

#72 - 07/20/2023 09:22 AM - Vladimir Tsichevski

I am currently working on automating the GUI testing with Sikuli, and I have problems with OCR (Tesseract):

1. Tesseract results are very unstable, at least when recognizing FWD fonts, which are usually small
2. Tesseract cannot tell similarly looking glyphs (for example, 5 from S, 0 from O etc).
3. If the text is in a FILL-IN, and there is a cursor visible, then OCR is unusable. Besides, in unit tests we often need to check the cursor position, which is impossible with Tesseract.
4. When recognizing empty text, Tesseract returns some non-empty string ("ne") instead of an empty string.

So I created a solution to this problem, which is **not** based on Tesseract.

In unit tests, in most cases, we do not need to **read** the screen text, but check that the screen text literally matches the expected one. So instead of reading arbitrary text from the screen, we compare the screen text with the expected one character by character.

The solution is based on the fact that we know the character font, and can find the known character glyphs in screen region.

The solution has two parts:

1. Record all font glyphs as image files, cautiously preserving glyph sizes. Record two sets: with and without cursor visible.
2. Find the glyph in screen region.

Both parts of the solution are based on Sikuli.

Glyph recording

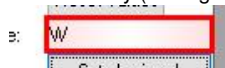
To record glyphs, I use a small 4gl test application, which displays a character FILL-IN. We will refer the FILL-IN inner screen region, where the edited text is displayed, as canvas.

The steps:

1. Measure the FILL-IN text left boundary in the canvas (we need to know where the text begins):

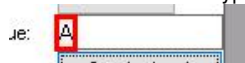


For this task (and some other tasks later) we will use a character, which glyph has no white space from the left. I used the W letter for that purpose. To be sure, type in the letter twice and assure there is no gap between the two. Then we can measure the left W boundary on the screen manually or automatically (the algorithm is quite simple, so I will not describe it here):



2. For each character *c* in the font (we will use the character A in this tutorial with the cursor on it), do:

1. clear the FILL-IN and type the character in, so it is the only character in the FILL-IN:



2. remove focus from the FILL-IN to hide the cursor

3. capture screen region with the character. The region coordinates are:

1. X is the text start measured in the previous step
2. Y is canvas top
3. height is canvas height
4. width should be wide enough for any character in the font

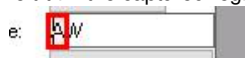
4. right-trim the region, the result is trimmedRight region variable:

5. type in the W letter without deleting the c character:

6. capture the region containing the W letter, use the trimmedRight region right boundary as the capture region left:



7. left-trim the captured region, use the resulting region left as the right boundary of the c glyph:



8. write the region to a file.

Screen text comparing

The algorithm is quite simple:

2. For each character *c* in the expected string, do:

1. Find the character in the region
2. Move the region left boundary left by the character glyph width.

#73 - 07/20/2023 11:31 AM - Vladimir Tsichevski

Vladimir Tsichevski wrote:

2. For each character c in the expected string, do:
 1. Find the character in the region
 2. Move the region left boundary left by the character glyph width.

Addition: the glyph image must be searched exactly in the glyph area. Otherwise, the glyph can be found at more than one location, which makes next glyph searching problematic. Example: the space character glyph.

#74 - 07/20/2023 11:58 AM - Vladimir Tsichevski

Vladimir Tsichevski wrote:

Screen text comparing

...

TODO: support selected text recognition. We can either record another image set for selected character, or mimic the FWD selection algo and convert unselected character glyph image to the selected one at runtime.

#75 - 07/20/2023 12:56 PM - Roger Borrello

Vladimir, do you have any images that you could capture of the steps you are performing? I am having challenges imagining what this looks like without seeing something in action.

#76 - 07/20/2023 02:15 PM - Vladimir Tsichevski

- File 3704-algo-5.png added
- File 3704-algo-4.png added
- File 3704-algo-3.png added
- File 3704-algo-2.png added
- File 3704-algo-1.png added

Images attached to illustrate the [#3704-72](#).

#77 - 07/20/2023 02:23 PM - Vladimir Tsichevski

Roger Borrello wrote:

Vladimir, do you have any images that you could capture of the steps you are performing? I am having challenges imagining what this looks like

without seeing something in action.

I updated [#3704-72](#): added some images.

#78 - 07/20/2023 02:25 PM - Vladimir Tsichevski

Greg Shah wrote:

This is the list of known issues to resolve, as far as I know:

- Move to Java instead of Python for the test encoding. This will allow us to embed arbitrary Java logic/control flow to enhance the tests.
- Make OCR work in the tests. This will allow a more flexible test.

Done

#79 - 07/20/2023 03:14 PM - Roger Borrello

Roger Borrello wrote:

Added embedded images. Vladimir, what is highlighted? The region that will be searched?

Sorry... disregard.

#80 - 07/21/2023 05:15 PM - Vladimir Tsichevski

- *File sikulidemo.mp4 added*

Sikuli-based test example (type in values into a date editor) in sikulidemo.mp4.

#81 - 07/26/2023 12:15 PM - Vladimir Tsichevski

I am working on recognizing application errors with Sikuli.

I'm going to discover message error boxes, and I want to be able to close these boxes with Sikuli to continue with other tests, and I bumped into a strange thing: I cannot close an error box by clicking any of its buttons: neither the OK button nor the window close buttons work.

Fortunately, I still can close boxes by typing the ENTER key.

I use Swing to run FWD applications.

Roger, did you suffer from anything similar?

#82 - 07/26/2023 12:24 PM - Roger Borrello

Vladimir Tsichevski wrote:

Roger, did you suffer from anything similar?

Are you saying the mouse clicks are not recognized by the application? I have not seen anything like that.

#83 - 07/26/2023 12:26 PM - Vladimir Tsichevski

Roger Borrello wrote:

Vladimir Tsichevski wrote:

Roger, did you suffer from anything similar?

Are you saying the mouse clicks are not recognized by the application?

Exactly so.

#84 - 07/26/2023 04:39 PM - Roger Borrello

Did you turn targeting on so you could see the "crosshairs"?

```
setShowActions(True)
```

#85 - 07/26/2023 04:40 PM - Vladimir Tsichevski

Roger Borrello wrote:

Did you turn targeting on so you could see the "crosshairs"?
[...]

No. Why?

#86 - 07/26/2023 04:50 PM - Roger Borrello

With that on, you will be able to see the mouse movements and where the click occurs. It might give you a clue about why it isn't working with the mouse.

#87 - 07/27/2023 12:34 PM - Hynek Cihlar

Vladimir Tsichevski wrote:

I'm going to discover message error boxes, and I want to be able to close these boxes with Sikuli to continue with other tests, and I bumped into a strange thing: I cannot close an error box by clicking any of its buttons: neither the OK button nor the window close buttons work.

Vladimir, I believe this is a regression. I've spotted this, too. ALERT-BOX unable to close with mouse. I still haven't figured out the exact conditions when this happens, this only occurred a couple of times to me. If you have a good recreate, can you open new issue and post it there? Also, do you see this in Web GUI, too?

#88 - 07/27/2023 12:44 PM - Vladimir Tsichevski

Hynek Cihlar wrote:

Vladimir Tsichevski wrote:

I'm going to discover message error boxes, and I want to be able to close these boxes with Sikuli to continue with other tests, and I bumped into a strange thing: I cannot close an error box by clicking any of its buttons: neither the OK button nor the window close buttons work.

Vladimir, I believe this is a regression. I've spotted this, too. ALERT-BOX unable to close with mouse. I still haven't figured out the exact conditions when this happens, this only occurred a couple of times to me. If you have a good recreate, can you open new issue and post it there?

OK, with Sikuli this always happens.

Also, do you see this in Web GUI, too?

My program can detect Swing error boxes, I have not tried it in WEB yet. Will try this either.

#89 - 07/31/2023 07:42 AM - Vladimir Tsichevski

- File *TestYearNormalizationWithSikuli.mp4* added

In the *TestYearNormalizationWithSikuli.mp4* you can see testing year normalization in date FILL-IN.

In the video 260 tests are performed in 125 seconds.

#90 - 07/31/2023 08:25 AM - Roger Borrello

Vladimir Tsichevski wrote:

In the *TestYearNormalizationWithSikuli.mp4* you can see testing year normalization in date FILL-IN.

In the video 260 tests are performed in 125 seconds.

That is very impressive! Can you post some of the code involved?

#91 - 07/31/2023 02:04 PM - Vladimir Tsichevski

Roger Borrello wrote:

Vladimir Tsichevski wrote:

In the *TestYearNormalizationWithSikuli.mp4* you can see testing year normalization in date FILL-IN.

In the video 260 tests are performed in 125 seconds.

That is very impressive! Can you post some of the code involved?

I've already asked Greg for creating a bzr repository at GCD for the project, so you can vote for it either :-)

#92 - 07/31/2023 02:29 PM - Greg Shah

The code should go into Testcases.

#93 - 07/31/2023 03:27 PM - Vladimir Tsichevski

Greg Shah wrote:

The code should go into Testcases.

This project is a pure Java with no FWD dependencies. It also has nothing to do with 4gl unit testing. So it has no any relation to the testcases project :-)

#94 - 07/31/2023 04:46 PM - Greg Shah

What is the purpose of the project?

#95 - 08/07/2023 04:37 AM - Vladimir Tsichevski

Greg Shah wrote:

What is the purpose of the project?

The short-term purpose of the project is to create automated testing suite for FILL-IN widget for all kinds of edit format, based on Sikuli. The suite should support both GUI and ChUI.

The long-term purpose is to create a framework for automated FWD testing based on Sikuli.

I need this FILL-IN testing suite to close the #7019, [#7515](#), #7606 issues. The changes I made are significant and may cause regressions.

I do not think this project should be a part of any other project right now, since it does not depend on any other GCD code, and is on early alpha state. Merging it into anything other is not necessary and would imply additional work.

At the moment the following implemented project features can be learned/used by other team members, which want to use Sikuli:

1. reliably comparing screen text to the expected one for known fonts. Automatic capturing new font characteristics.
2. automatically recognizing error message boxes from a registered set of error numbers, semi-automatic registering new box types
3. automatically recognizing widget locations and sizes for bordered widgets like FILL-IN and BUTTON (with some restrictions)

#96 - 08/07/2023 08:40 AM - Greg Shah

In [#6856](#), we already have a suite of editing tests planned for all widgets (not just FILL-IN). These will be 4GL tests and the interactive parts will be automated with Sikuli.

I don't want a separate project, with a one-off set of non-4GL tests. That just means we have duplication of testing. Non-4GL tests may be

convenient for some purposes but are ultimately not as useful because we cannot test a range of important functionality like events processing, default event behavior, interaction with focus processing and much more. Likewise we cannot test conversion without there being 4GL tests. So we must have the 4GL tests and thus I really prefer not to implement non-4GL tests since otherwise we are going to waste a huge amount of time replacing them.

#97 - 08/16/2023 03:31 PM - Greg Shah

I've spoken with Marian, whose team is leading the effort to write tests for the full range of UI processing. This will include both interactive and non-interactive testing. Details can be seen in [#6856](#).

For the interactive tests, we need to use Sikuli (or something like it). Sikuli seems to be the best option but:

- The test code is hard to implement.
- The results are not always reliable unless you do a lot of extra work to code the test.
- The methods to match screens and the OCR tools are especially tricky.

In addition to that, we already know that the majority of our test code is going to be written in 4GL and we will use ABLUnit to execute and report on the results. There are many advantages to this, including good reporting facilities, the ability to test conversion and runtime code using the same tests and much more. The biggest advantage of writing 4GL code for tests is that the test code can inspect the state of running program and very easily report failures or successes. This is something that non-4GL test tools could not do. In this case, using Sikuli for the interactive testing would not be able to verify the internal state of the converted code (e.g. check the SCREEN-VALUE for a given widget after a button is pressed). Any checking done by Sikuli requires one use some kind of visual matching that is less than reliable or if we implement enough processing around it it is just tricky and hard to implement.

For the above reasons, we plan to use Sikuli in the following way:

- All of our test code will be written in the 4GL and will be executed with ABLUnit.
- Sikuli will be used to drive the user input on interactive UI tests however it will not be the primary mechanism to detect if the test passes/fails.
- Sikuli may be used (when needed) to wait for a given screen to become available, but as much as possible we do not want to use it to check the test results themselves. Instead, the plan is to write 4GL code to check on the results and report pass/fail.
- The tests can be written to use OS-COMMAND to start Sikuli from the 4GL test code itself. If this works, then we don't need anything specific added to integrate Sikuli into ABLUnit.

There may be some cases like FILL-IN editing where between every keystroke it makes sense to check on the visual result of the editing, which would need to be written in Sikuli. We should try to minimize this but it is OK to do it when absolutely needed. In this case, we need to decide how we get the results back into the 4GL code so that they can be integrated into the ABLUnit results.

#98 - 08/22/2023 09:42 AM - Vladimir Tsichevski

Greg Shah wrote:

For the interactive tests, we need to use Sikuli (or something like it). Sikuli seems to be the best option but:

- The test code is hard to implement.

What do you mean by that? After learning some tricks and coding them into a small library, adding new tests is easy.

- The results are not always reliable unless you do a lot of extra work to code the test.

Yes, you do need some extra works, but you do the work only once, and pack the results into a library.

- The methods to match screens and the OCR tools are especially tricky.

Do **not** use Sikuli OCR whatsoever. Do match the screen text to the expected instead.

In addition to that, we already know that the majority of our test code is going to be written in 4GL and we will use ABLUnit to execute and report on the results.

Yes, this works OK for testing non-GUI stuff.

There are many advantages to this, including good reporting facilities,

ABLUnit implementation is based on JUnit5, and it is very convenient to write Sikuli-based unit tests in JUnit5 directly. So, reporting facilities are basically the same for both approaches.

The biggest advantage of writing 4GL code for tests is that the test code can inspect the state of running program and very easily report failures or successes.

Again, for non-GUI stuff, it is impossible to inspect internal state of FILL-IN, while the user interacts with it.

This is something that non-4GL test tools could not do.

In a limited way, they can. In the test application I can force the app to print internal state and compare it with the expected one.

In this case, using Sikuli for the interactive testing would not be able to verify the internal state of the converted code (e.g. check the SCREEN-VALUE for a given widget after a button is pressed).

In my Sikuli-based program I do exactly this.

Any checking done by Sikuli requires one use some kind of visual matching that is less than reliable or if we implement enough processing around it it is just tricky and hard to implement.

Yes, we implement enough processing around it to make it reliable. We need to do this only once.

For the above reasons, we plan to use Sikuli in the following way:

- All of our test code will be written in the 4GL and will be executed with ABLUnit.
- Sikuli will be used to drive the user input on interactive UI tests however it will not be the primary mechanism to detect if the test passes/fails.
- Sikuli may be used (when needed) to wait for a given screen to become available, but as much as possible we do not want to use it to check the test results themselves. Instead, the plan is to write 4GL code to check on the results and report pass/fail.

- The tests can be written to use OS-COMMAND to start Sikuli from the 4GL test code itself. If this works, then we don't need anything specific added to integrate Sikuli into ABLUnit.

We need to devise and implement a 4gl RMI protocol for this first.

There may be some cases like FILL-IN editing where between every keystroke it makes sense to check on the visual result of the editing, which would need to be written in Sikuli. We should try to minimize this but it is OK to do it when absolutely needed. In this case, we need to decide how we get the results back into the 4GL code so that they can be integrated into the ABLUnit results.

So, we will probably need an **extensive** RMI protocol to interact with Sikuli.

#99 - 08/22/2023 10:26 AM - Greg Shah

For the interactive tests, we need to use Sikuli (or something like it). Sikuli seems to be the best option but:

- The test code is hard to implement.

What do you mean by that? After learning some tricks

We don't want all of our UI developers to have to be Sikuli experts. Our experience with it shows that one must know quite a bit to do it properly.

and coding them into a small library, adding new tests is easy.

I can't comment on your library of helpers because it is not yet checked in to the testcases project. It would surprise me if that library solved all of our Sikuli issues. I'm sure it is very nice for fill-in editing tests. I'll reserve judgement until I see more detail.

In addition to that, we already know that the majority of our test code is going to be written in 4GL and we will use ABLUnit to execute and report on the results.

Yes, this works OK for testing non-GUI stuff.

Instead of non-GUI, think "non-visual". Sikuli is good for 2 things:

- entering user input
- capturing visual output for comparison

A huge amount of GUI state can be accessed from the 4GL. And we can provide even more access with our our custom 4GL attributes if we need

them. All of that can be written in the same language as the rest of our testing is written. That is a big factor in its favor.

We already have everyone on the team being able to write 4GL code. The same cannot be said for Sikuli. On top of that, anything in 4GL code automatically becomes a conversion test as well as a runtime test.

The biggest advantage of writing 4GL code for tests is that the test code can inspect the state of running program and very easily report failures or successes.

Again, for non-GUI stuff, it is impossible to inspect internal state of FILL-IN, while the user interacts with it.

I don't understand this. We can do that from triggers quite easily. Again, there are big benefits to doing that because it ensures that our event processing is the same as in the 4GL.

This is something that non-4GL test tools could not do.

In a limited way, they can. In the test application I can force the app to print internal state and compare it with the expected one.

We can do this from the 4GL without adding any custom access. To achieve what you have done (running these tests outside of FWD) you have to spend lots of time making our FWD functionality work in this external environment. This is wasted time. And I don't think it is only done once. We will constantly be changing our internal implementation of the UI. I don't want to have that work be limited by special purpose hidden APIs that are needed to enable external testing.

In this case, using Sikuli for the interactive testing would not be able to verify the internal state of the converted code (e.g. check the SCREEN-VALUE for a given widget after a button is pressed).

In my Sikuli-based program I do exactly this.

You don't do that from 4GL code so it is not a useful test.

Any checking done by Sikuli requires one use some kind of visual matching that is less than reliable or if we implement enough processing around it it is just tricky and hard to implement.

Yes, we implement enough processing around it to make it reliable. We need to do this only once.

We will do this, but my point is that this is something to limit. I expect that it will need maintenance over time, so this is not likely to be only once.

- The tests can be written to use OS-COMMAND to start Sikuli from the 4GL test code itself. If this works, then we don't need anything specific added to integrate Sikuli into ABLUnit.

We need to devise and implement a 4gl RMI protocol for this first.

Please help me understand why you think this is needed. Sikuli can execute as a child process of the 4GL code.

If this is needed, we would not use RMI. Perhaps you mean RemoteObject?

There may be some cases like FILL-IN editing where between every keystroke it makes sense to check on the visual result of the editing, which would need to be written in Sikuli. We should try to minimize this but it is OK to do it when absolutely needed. In this case, we need to decide how we get the results back into the 4GL code so that they can be integrated into the ABLUnit results.

So, we will probably need an **extensive** RMI protocol to interact with Sikuli.

If Sikuli can be integrated with JUnit5, we can launch JUnit5 to execute Sikuli and read the JUnit5 results as XML. I don't see why we need an RMI connection.

#100 - 08/22/2023 12:17 PM - Vladimir Tsichevski

Greg Shah wrote:

For the interactive tests, we need to use Sikuli (or something like it). Sikuli seems to be the best option but:

- The test code is hard to implement.

What do you mean by that? After learning some tricks

We don't want all of our UI developers to have to be Sikuli experts. Our experience with it shows that one must know quite a bit to do it properly.

You have to understand the basics of interactive testing. For example, to learn that you need to know the location and geometry of your test application, do not move or resize the windows which are being tested, do not use mouse or keyboard while the testing is in progress, that the delays in interaction may vary for the same operation (because the Java garbage collection may occur in any moment) and so on.

The Sikuli-related stuff can be learned in one day, it is not much to learn.

and coding them into a small library, adding new tests is easy.

I can't comment on your library of helpers because it is not yet checked in to the testcases project. It would surprise me if that library solved all of our Sikuli issues.

No program ever solves all possible issues :- (And, as I wrote above, most of the issues are not related to Sikuli itself.

I'm sure it is very nice for fill-in editing tests.

Yes, and must be used to testing thing like this.
And, there is a **huge** number of tests which need to be implemented in this field.

In addition to that, we already know that the majority of our test code is going to be written in 4GL and we will use ABLUnit to execute and report on the results.

Yes, this works OK for testing non-GUI stuff.

Instead of non-GUI, think "non-visual". Sikuli is good for 2 things:

- entering user input
- capturing visual output for comparison

Correct

A huge amount of GUI state can be accessed from the 4GL.

Before using Sikuli I tried to imitate user input by sending 4gl events programmatically. It was a complete fail, and you explained me the reason once: many of the original OE widgets are based on some Microsoft controls, which may know nothing about 4gl events.

And we can provide even more access with our our custom 4GL attributes if we need them. All of that can be written in the same language as the rest of our testing is written. That is a big factor in its favor.

So, you propose to extend FWD by custom API to access the internal program state while testing?

We already have everyone on the team being able to write 4GL code.

Once I've read a confession of a 4gl developer with 19 year experience of writing in 4gl. He wrote he does not feel proficient enough yet :-)

The same cannot be said for Sikuli.

Everyone on the team is able to write Java code, and Sikuli is Java.

The biggest advantage of writing 4GL code for tests is that the test code can inspect the state of running program and very easily report failures or successes.

Again, for non-GUI stuff, it is impossible to inspect internal state of FILL-IN, while the user interacts with it.

I don't understand this. We can do that from triggers quite easily.

How can we read the FILL-IN cursor position from 4gl?

This is something that non-4GL test tools could not do.

In a limited way, they can. In the test application I can force the app to print internal state and compare it with the expected one.

We can do this from the 4GL without adding any custom access. To achieve what you have done (running these tests outside of FWD) you have to spend lots of time making our FWD functionality work in this external environment. This is wasted time. And I don't think it is only done once. We will constantly be changing our internal implementation of the UI. I don't want to have that work be limited by special purpose hidden APIs that are needed to enable external testing.

I did not get this. We do **not** need any changes in FWD to test with Sikuli. It was you who wrote "we can provide even more access with our our custom 4GL attributes".

In this case, using Sikuli for the interactive testing would not be able to verify the internal state of the converted code (e.g. check the SCREEN-VALUE for a given widget after a button is pressed).

In my Sikuli-based program I do exactly this.

You don't do that from 4GL code so it is not a useful test.

I do not understand this. The test 4gl program gets the SCREEN-VALUE for a given widget and prints the result somewhere I can read it with Sikuli.

Any checking done by Sikuli requires one use some kind of visual matching that is less than reliable or if we implement enough processing around it it is just tricky and hard to implement.

Yes, we implement enough processing around it to make it reliable. We need to do this only once.

We will do this, but my point is that this is something to limit. I expect that it will need maintenance over time, so this is not likely to be only once.

So does any peace of software :-)

- The tests can be written to use OS-COMMAND to start Sikuli from the 4GL test code itself. If this works, then we don't need anything specific added to integrate Sikuli into ABLUnit.

We need to devise and implement a 4gl RMI protocol for this first.

Please help me understand why you think this is needed. Sikuli can execute as a child process of the 4GL code.

You have to tell Sikuli what to do and how to return the results. So you need to devise some protocol. Since you will call Sikuli from 4gl, you will have to write the implementation in 4gl. I called this 4gl RMI.

If this is needed, we would not use RMI. Perhaps you mean RemoteObject?

By the RMI I mean general synchronous protocol. We will run Sikuli in different OS process, and we will need to communicate with this process from

converted 4gl code somehow.

There may be some cases like FILL-IN editing where between every keystroke it makes sense to check on the visual result of the editing, which would need to be written in Sikuli. We should try to minimize this but it is OK to do it when absolutely needed. In this case, we need to decide how we get the results back into the 4GL code so that they can be integrated into the ABLUnit results.

So, we will probably need an **extensive** RMI protocol to interact with Sikuli.

If Sikuli can be integrated with JUnit5, we can launch JUnit5 to execute Sikuli and read the JUnit5 results as XML. I don't see why we need an RMI connection.

I was using RMI as a general term. Sikuli is just a Java library, we do not need JUnit5 to access it. We can write any command-line Java application and run it with OS-COMMAND from 4gl. We can pass the data to this application as command-line arguments or make the application be a server and use any protocol accessible from 4gl to communicate.

What I wanted to say is that we will have to devise, implement, test and maintain another piece of code just to connect 4gl with Sikuli.

Note also that Sikuli process needs to run on the same display as the FWD client.

#101 - 08/23/2023 05:15 AM - Marian Edu

Vladimir Tsichevski wrote:

The Sikuli-related stuff can be learned in one day, it is not much to learn.

Vladimir, this really isn't about learning Sikuli or not - learning the API is one thing, actually being able to get some consistent results when running tests we write in 4GL when running on both 4GL and FWD (various environments) is a bit different, regardless of how many tricks one can learn :)

This thread is about automated testing of UI, we're only concerned about the unit-testing part not application testing. When you need to test an application then that is a black box and all you can do is to perform actions on it and evaluate the results **visually**. For unit-testing on the other hand we have full control of the code we're testing and we're more interested on how 4GL events/attributes are affected by those actions so all we need is to simulate the user interaction - simple things like pressing a key or basic mouse events. We expect a certain result once all those actions are completed and we compare the succession of events and end result at the end, we just need to use 4GL triggers for that.

That being said I fully agree there are visual aspects that we can't test with this - like when setting the background color of a widget or the fact that when a widget is focused there is that visual representation that highlight it, how many pixels have the lines of the focus box or what is color they have, for some things manual testing will still be required and I don't think all the tricks in Sikuli will be able to solve everything :)

Before using Sikuli I tried to imitate user input by sending 4gl events programmatically. It was a complete fail, and you explained me the reason once: many of the original OE widgets are based on some Microsoft controls, which may know nothing about 4gl events.

It just doesn't matter what controls they are using, once you enter in the WAIT-FOR loop you can't actually send anything from 4GL anymore and the event-driven behaviour only kicks in inside the WAIT-FOR loop.

Once I've read a confession of a 4gl developer with 19 year experience of writing in 4gl. He wrote he does not fill proficient enough yet :-(

"Proficient" is an abstract term and it can mean different thing to different peoples I guess, it's all subjective... and relative :)

How can we read the FILL-IN cursor position from 4gl?

CURSOR-OFFSET, if this doesn't let you sleep at night :)

I was using RMI as a general term. Sikuli is just a Java library, we do not need JUnit5 to access it. We can write any command-line Java application and run it with OS-COMMAND from 4gl. We can pass the data to this application as command-line arguments or make the application be a server and use any protocol accessible from 4gl to communicate.

For our basic needs - again I'm only concerned with unit-testing of the "4GL" and how that is implemented in FWD - we do not need any protocol, the Sikuli script will just start, wait for the application screen to show and then perform a list of actions (last of which will be to close the screen) so the 4GL unit-test will be able to continue and evaluate the results.

How we will start Sikuli from our "4GL" tests is probably just OS-COMMAND for the 4GL but I don't see why we can't use something else for FWD - there is one pre-processor directive that we can use and that give us the chance to use some FWD extensions when running in that environment - the point being we can use the same ABLUnit tests.

Hope that make sense, maybe we can divide that thread in separate ones - what Vladimir is trying to accomplish will definitively have value for any application testing and can probably be used against hotel sample application for more integration testing.

And we can provide even more access with our our custom 4GL attributes if we need them. All of that can be written in the same language as the rest of our testing is written. That is a big factor in its favor.

So, you propose to extend FWD by custom API to access the internal program state while testing?

It is possible if needed. We don't yet know if it is needed or not. Either way, it can be easily exposed to the 4GL code level.

We already have everyone on the team being able to write 4GL code.

Once I've read a confession of a 4gl developer with 19 year experience of writing in 4gl. He wrote he does not feel proficient enough yet :-(

The same cannot be said for Sikuli.

Everyone on the team is able to write Java code, and Sikuli is Java.

This is not the case. We have a significant number of people on the team that only write 4GL code and they are writing 4GL testcases as I write this response. Our primary objective is to replace OpenEdge with a fully compatible 4GL implementation. As such, 4GL testing is the primary way we will do things. Adding 4GL developers to the team to help with this is a big benefit because they understand the original environment and can write these tests faster than Java developers that have only recently learned the 4GL. We do not expect these 4GL devs to learn Java. Some will, but not all of them.

Conversely, we do require all of our Java devs to learn the 4GL. And they must learn it well enough to write new testcases. This means that the 4GL is a common language for all people that have to write tests but Java is not common.

In addition, writing our tests in the 4GL itself greatly enhances our ability to test our primary objective. Writing pure Java tests does none of that and we will not write tests that way.

This is something that non-4GL test tools could not do.

In a limited way, they can. In the test application I can force the app to print internal state and compare it with the expected one.

We can do this from the 4GL without adding any custom access. To achieve what you have done (running these tests outside of FWD) you have to spend lots of time making our FWD functionality work in this external environment. This is wasted time. And I don't think it is only done once. We will constantly be changing our internal implementation of the UI. I don't want to have that work be limited by special purpose hidden APIs that are needed to enable external testing.

I did not get this. We do **not** need any changes in FWD to test with Sikuli. It was you who wrote "we can provide even more access with our our custom 4GL attributes".

You'll have to explain how you get internal access from pure Java without writing any scaffolding or custom code to execute our Java classes outside of a FWD environment.

You don't do that from 4GL code so it is not a useful test.

I do not understand this. The test 4gl program gets the SCREEN-VALUE for a given widget and prints the result somewhere I can read it with Sikuli.

We don't want to report **to** Sikuli. The 4GL code will be the test code and Sikuli will just drive the UI and do the minimum checking of the output.

Files			
3704_hotel_login_demo.mkv	1.6 MB	06/17/2021	Roger Borrello
3704-algo-5.png	674 Bytes	07/20/2023	Vladimir Tsichevski
3704-algo-4.png	705 Bytes	07/20/2023	Vladimir Tsichevski
3704-algo-3.png	733 Bytes	07/20/2023	Vladimir Tsichevski
3704-algo-2.png	969 Bytes	07/20/2023	Vladimir Tsichevski
3704-algo-1.png	692 Bytes	07/20/2023	Vladimir Tsichevski
sikulidemo.mp4	109 KB	07/21/2023	Vladimir Tsichevski
TestYearNormalizationWithSikuli.mp4	816 KB	07/31/2023	Vladimir Tsichevski