

CircuitSolver Test Plan

(Validation/Verification Documentation)

Introduction

CircuitSolver is a mobile app targeted for engineering students that allows one to analyze the voltage drop and current across components in a hand-drawn electrical circuit. It aims to eliminate the inconvenience (and time) of using in-app circuit builders and printable components kits when trying to analyze circuits. Our app consists of various components and uses multiple technologies and frameworks (OpenCV, TensorFlow, NgSPICE engine, Android Studio). Hence, our app is particularly vulnerable to issues of performance and integration complexity. Our testing plan addresses this vulnerability and contains strategies to mitigate their effects. Furthermore, key to our product's success is a smooth user experience, since it's goal is to save time and effort in solving circuits. Thus our testing strategy also incorporates feedback from users in our target group.

Verification Strategy

This product is designed for engineering students with the purpose of simplifying the analysis of hand-drawn circuits. Thus, our verification strategy stresses importance on user experience feedback from peer engineering students. Correctness of logic, which is also vital to the product's success, will be tested thoroughly by unit and integration tests to ensure a robust system. Key points of our verification strategy are included below.

- **External feedback on UI mockups/prototypes/iterations:** As we prototype our UI, we will seek feedback from at least 5 other engineering students and note any common issues they report. Each new iteration of the UI design, we will receive feedback from at least 1 other engineering student.
- **Internal review of UX:** Our own team members are part of the target audience, so our feedback is meaningful for this product as well. All team members will test the UX regularly through the emulators (at least 3 times a week), and at least 2 people with will be testing on their Android devices (at least once a week).
- **Automated UI tests:** Once enough of our UI is built, we will write automated UI tests for happy path scenarios (and exception/error scenarios if time) which should be run at least 3 times a week. This would ensure new code changes do not break our minimum viable product.
- **Unit and integration tests:** The above points are focused on UX, but this point is the foundation for ensuring our circuit analysis logic itself is correct. See functional testing strategy for details.

Non-functional Testing and Results

Portability testing: Test our app on both “virtual” devices on the emulator and physical devices with different resolutions/screen sizes. Make sure everything is still usable. Tests defined below will be used across different devices using a fast emulator such as genymotion or the default emulator for Android Studio.

Performance testing:

Goals and guidelines for the performance testing

Test our app with circuits of varying complexities. Ensure that image recognition and circuit analysis can be performed in reasonable time (10 seconds max for a small circuit). It is important that the time taken by Circuit Solver is faster than the time taken by a human to solve a circuit by hand. In [Donald Knuth's](#) paper "[Structured Programming With GoTo Statements](#)", he wrote: "Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We *should* forget about small efficiencies, say about 97% of the time: **premature optimization is the root of all evil**. Yet we should not pass up our opportunities in that critical 3%." Since we are currently at first trying to get our app working we have not looked into tracking tools to measure system time and plan on using simple approaches such as `System.nanoTime()` to measure execution time of particular identified bottlenecks (i.e. such as a triple for loop we currently have in our OpenCV service). Results will be documented in the associated issue on Github (see organization under Functional Testing Strategy)

Results of non-functional testing

After running the app multiple times on multiple inputs, we come to the conclusion that indeed the image processing is the bottleneck of Circuit Solver in terms of time performance. More precisely, deduced from empirical source, the recognition of individual components through TensorFlow takes a linear amount of time in terms of component (2-3 seconds per component), while the detection of the position of the corners and components takes polynomial time but is much faster (~0.2 seconds per component when fewer than 6 components). Also, the editing part is instantaneous (which is required for a comfortable user interface), and the solving part takes less than 1 second.

This achieves the main performance goals fixed for our app: while by hand we would use exponential time to solve a circuit in terms of number of components, our app can achieve it in polynomial time. Also, when the user wants to do a quick drawing on the app to solve a simple circuit, the drawing and the solving part are close to instantaneous what is required for this part of our app.

Functional Testing Strategy

Given time constraints and how our code will likely change drastically in the coming weeks as we realize prior designs aren't as feasible as we initially thought (as happened in the JPacman undo assignment) we seek a testing strategy with a low learning curve, that is easy to set up, maintain and is focused more on testing features rather than code design and expectation. Functional testing will involve unit testing (testing components in isolation - doesn't require an emulator) coupled with a mocking framework to make stubs to test them and all dependencies. We applied this testing strategy all along the program with resource stubs along the program. Secondly we will require instrumentation testing (driving the activity lifecycle manually - requires an emulator) to test the UI itself through simulated user interaction. With this in mind we have the following strategies

Framework choice and reason

- For Unit testing options available include Robolectric and JUnit. We chose JUnit due to our familiarity with the library and its wide usage. While some regard long-term maintenance with JUnit more difficult than other options, the short timeline for this product makes this not much of an issue. Furthermore, typical practice transfers over to JUnit in production anyway. [\[Source\]](#)
- Finally, for Instrumentation testing options available include Espresso and UIAutomator. Here we pick Espresso because it is considered best for small scale apps and easier to set up. UIAutomator on the other hand has a steeper learning curve and is focused more on external OS integration and testing across multiple applications at the same time. None of these features are required by us. [\[Source\]](#), [\[Source\]](#).

Strategy

- Unit tests will be built for for each method in classes that performs reasonable work (not getters and setters). Since we were able to reach a 90+% threshold for code coverage (for class, method and line coverage) for non-ui elements in our assignment with the undo functionality working fine we will aim for this coverage threshold. Aiming for as close to 100% is ideal, however code coverage approaches 100% asymptotically. Consequently, the last 10% is likely more effort than it is worth, offering small returns for the effort expended.
- Automated UI testing will be performed for each of our use-cases specified in our [Requirements Documentation](#). This will be done using the Espresso library. These end-to-end tests would ensure smooth integration of all our components. More rigorous testing of back end logic will be done in unit tests, so that these tests only need to test integration. Unit and Instrumentation tests should run on dev machines before each merge, ensuring iterative changes do not change feature requirements.

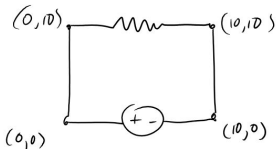
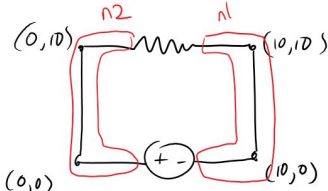
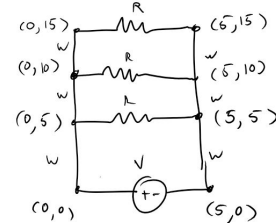
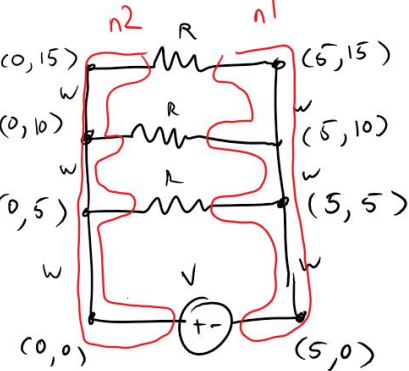
- Every significant feature, bug fix, refactoring, etc is given an issue using [our GitHub repo's](#) issue tracking. Each issue is assigned to at least one person and attached to at least one of the following labels:
 - OpenCV
 - TensorFlow
 - Circuit-Logic
 - UI
- Issues will be given labels based on which item they are tied to on our [product backlog](#) and what that item's priority is (very high, high, medium, low priority). Github issue tracking is well known by all of us meaning no learning curve is required and we became even more familiar with it through the JPacman assignment.
See <https://github.com/Frikster/CircuitSolverApp/issues> for our issue tracking.

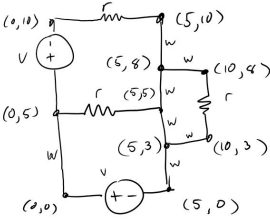
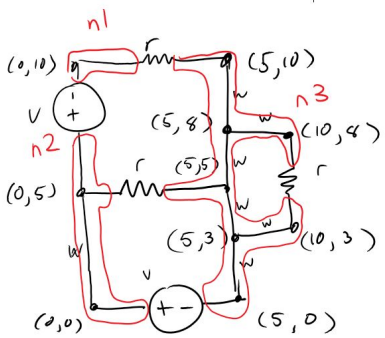
Unit Tests

These tests ensure our separate components work correctly in isolation. They should all be contained in JUnit test suites to allow for automated unit testing. These tests are part of the validation strategy of the Circuit Solver App.

Circuit Logic

These tests are in the `AllocateNodesTest.java` test suite. The `AllocNodes` component is triggered whenever solving a circuit, as node identification is required for NgSpice. These tests are designed to demonstrate confidence that the `AllocNodes` class correctly allocates circuit nodes.

Test #	Requirement Purpose	Action/Input	Expected Result	P/F	Notes
	Test circuit nodes are correctly allocated for a simple two node circuit.	<p>A List of CircuitElm with coordinates that represents the following circuit:</p> 	<p>A List of CircuitNodes, with each CircuitElm's CircuitNode attributes connected to the correct CircuitNode.</p> 	P	Refer to AllocNodesTest for details.
	Test circuit nodes are correctly allocated for intermediate circuit (as in some elements in parallel, not just single loop)	<p>A List of CircuitElm with coordinates that represents the following circuit:</p> 	<p>A List of CircuitNodes, with each CircuitElm's CircuitNode attributes connected to the correct CircuitNode.</p> 	P	Refer to AllocNodesTest for details.
	Test circuit nodes are correctly	A List of CircuitElm with coordinates that	A List of CircuitNodes, with each CircuitElm's CircuitNode	P	Refer to AllocNodesTest

	allocated for complex circuit (unconventional circuit with more than 8 elements)	represents the following circuit: 	attributes connected to the correct CircuitNode. 		st for details.
--	---	--	--	--	-----------------

Circuit Definition File I/O

These tests are contained in the CircuitDefParserTest test suite. This component, contained in the CircuitDefParser class, is triggered each time circuit is being saved or loaded. Circuits are saved when moving from the processing activity to the draw activity, the draw activity to home activity, and any time the app is paused in the draw activity. A circuit is loaded whenever going from the home activity or processing activity to the draw activity.

Test #	Requirement Purpose	Action/Input	Expected Result	P/F	Notes
	Verify component can correctly parse a circuit definition text file into a list of CircuitElm with correct type and coordinates	Circuit definition file text with resistor, wire, and DC voltage source types.	A list of circuit elements with the correct coordinates and type according to the circuit definition file specs.	P	Refer to CircuitDefParserTest for the details.
	Verify that, given the width and height to fit circuit onto, the component can parse CircuitElm with scaled coordinates, but maintain its relative position	Circuit definition file text with specified width and height of screen when circuit was written.	A list of circuit elements with scaled coordinates that maintain its relative position.	P	Refer to CircuitDefParserTest for the details.
	Verify component can correctly parse a list of CircuitElm into a circuit definition String.	List of CircuitElms with resistor, wire, and DC voltage source types.	A String following the circuit definition file specs that represents each CircuitElm in the list appropriately.	P	Refer to CircuitDefParserTest for the details.

NgSpice

These tests are designed to demonstrate confidence that the NgSpice instance can receive input and output expected results.

Test #	Requirement Purpose	Action/Input	Expected Result	P/F	Notes
1	Correctly invoke NgSpice from within our application	Call NgSpice within our application with a simple netlist at input.	The expected voltage/current information about the netlist as a string.	P	
2	Use NgSpice to populate our circuit instance with the correct voltage/current values.	Input a valid circuit instance into the NgSpice interface instance.	Our circuit instance has the correct voltage/current values.	P	

OpenCv

These tests are designed to demonstrate that beside the OpenCv internal functions such that HoughLines and Canny, the processing of the image input is done correctly

Test #	Requirement Purpose	Action/Input	Expected Result	P/F	Notes
1	Take a picture with the camera and process it for displaying	A list of lines detected by the OpenCv houghLines function on the input image.	A list of coordinates indicating the corners of the circuit	P	Done in processCornersTest()
2	Take a picture with the camera and process it for displaying	A list of coordinates of components and corners detected	A list of wires (containing any component or not) of the circuit	P	Done in findWiresTest()
3	Take a picture with the camera and process it for displaying	A list of wires detected from the circuit	A normalized version of the wires, that are ready to be displayed by the UI	P	Done in processWiresTest()

Integration/System Tests

We have developed UI tests using Google's Espresso framework with tests organized based on our Use Cases:

USE CASE 1: Edit a previous saved circuit in order to change some components or change numerical values.

USE CASE 2: View component values of a previous analyzed circuit

USE CASE 3: Solve a circuit from an uploaded or taken circuit picture

USE CASE 4: Create virtual circuit by directly drawing in the app

USE CASE 5: Delete saved circuit

This is the main part of the verification strategy for the Circuit Solver App. These tests runned with Espresso are an excellent way to automatically perform all the use cases related to our software and thus provide a good verification feedback.

We do not automate every incremental step in the Use Case documentation. Instead, we have 5 Classes based on the use cases and modularized tests related to that use case in each class.

This modularization is meant to facilitate debugging and keep ideas organized.

Test #	Requirement Purpose	Action/Input	Expected Result	P / F	Notes
UC 1.1	Test Selecting a project and opening it	Click first project on list and click floating action button to open it	Intent to activate DrawActivity is verified	P	
UC 1.2	Test selecting a circuit component	Click first project on list, click floating action button to open it, click midpoint of first available component	componentMenuButton has "Change" text. The component_value of the selected component is not null. The current and voltage text of the selected component is null (hasn't been solved). The units displayed correspond to the circuit component selected (including null for wires)	P	UC1.2 calls UC1.1
UC 1.3	Test changing a circuit	Click first project on list, click floating action button to open it, replace first	Circuit contains one more DC source and one less resistor after first swap. Circuit contains one less DC source and one	P	UC1.4 calls UC1.3

	component	found dc source component with a resistor and first found resistor with a dc source	more resistor after second swap. A toast message appears in either case informing the user which component was selected. componentMenuButton has "Add" text following component change. The current and voltage text of the selected component is null.		
UC 1.4	Test changing components and subsequently solving the circuit	Click first project on list, click floating action button to open it, replace first found dc source component with a resistor and first found resistor with a dc source, and then select a component and click solve	A toast message appears informing the user the circuit has been solved. componentMenuButton has "Change." The current and voltage text of the selected component are <i>not</i> null.	P	UC1.4 calls UC1.3. Ergo, if this test passes all passed. However modularization helps localize problems
UC 1.5	Test changing components and attempts to solve an unsolvable circuit	Click first project on list, click floating action button to open it, replace all resistors in circuit with DC sources to create an illegal circuit. Click a component. Click solve.	A toast message appears informing the user the circuit is invalid. componentMenuButton has "Change" text. Selected component value, current and voltage are all not null	P	Invalid circuits will still have values associated with them upon solve
UC 2.1	Test returning to HomeActivity from drawActivity	Process bitmap data through openCV+tensorflow. Press back.	Intent to DrawActivity is verified upon processing being finished. Intent to return to HomeActivity is verified once back button is pressed. Assert that the HomeActivity CircuitProject ArrayList has one more CircuitProject and that the LinearLayout view that lists the projects on the home screen has one more child	P	See Espresso functions Intending and Intended
UC 2.2	Test going back to HomeActivity from drawActivity and subsequently picking a new project	Process bitmap data through openCV+tensorflow. Press back. Pick a new project. Click it. Click the arrow action button to open the project	Intent to go to DrawActivity for the new project is verified	P	UC2.2 calls UC2.1

UC 2.3	Test returning to the Processing Activity from an external application	Process bitmap data through openCV+tensorflow. During processing pause the application and returns to it. (simulate user going to an extraneous app)	Verify data integrity. Verify sme output regardless of pause or no pause	-	Incomplete. Using Espresso to pause and restart an activity has proven difficult. Will hopefully be done by December 13
UC 2.4	Test returning to the DrawActivity from an external application	Process bitmap data through openCV+tensorflow. At DrawActivity pause the application and returns to it. (simulate user going to an extraneous app)	Verify circuit is unchanged following pause/unpause	-	Incomplete. See above.
UC 3.1	Test erasing all circuit components	Process bitmap data through openCV+tensorflow. At DrawActivity, erase each component of the circuit	Assert that all component counts are 0 in the project ArrayList.componentMenuButton has "Add" text. Component value, current and voltage text are all null	P	
UC 3.2	Test modifying the type and value of multiple (but not all) circuit components	Process bitmap data through openCV+tensorflow. At DrawActivity, change every other component of the circuit into a resistor. Change each one's resistance to 23.4 (arbitrary). Click solve for each component	Verify toast message for changing a component into a resistor occurs for each "Resistor" button click. Verify solve toast message appears for each solve button click. ArrayList.componentMenuButton has "Add" text after initial solve click. Component value, current and voltage text are all non-null following each solve click	P	
UC 3.3	Test the auto-connect that occurs when two endpoints of two components are placed close	Process bitmap data through openCV+tensorflow that has a discontinuity between components. Press componentMenuButton to draw a wire between these two, however, do	Verify that auto-connecting works: i.e. when two ends of two components are close enough to each other, they automatically connect	-	Incomplete. Too much time was spent on other test cases leaving no time for this one

	enough	not join ends exactly but instead stop short from one to the other			
UC 3.4	Test pinch zoom in and out	Process bitmap data through openCV+tensorflow that has a discontinuity between components. Press componentMenuButton to draw a wire between these two, however, do not join ends exactly but instead stop short from one to the other. Zoom in on DrawActivity. Connect ends of two components. Zoom out. Connect ends of two other components	Verify that auto-connecting works at different magnification	-	Incomplete - Testing Pinch in Espresso has proven to be far more complicated than we thought: to the point that we anticipate omitting this test entirely as it appears not to be worth the added work
UC 4.1	Test drawing a simple circuit and solve it from scratch	Press the floating action button and press the draw option. Draw a connected square circuit with one side a capacitor, one a resistor and two wires. Press solve. Iteratively click on each component. Change each one's component value to 23.4 (arbitrary value)	The intent to go to DrawActivity from HomeActivity is verified. Check that a toast message appears informing the user that the circuit has been solved upon solve. componentMenuButton has "Change" text for each component clicked. Verify units align with circuit component type for each selected and that each selected circuit component does not have null for any of their component values or current/voltage text. Verify however that all wires have null for all three and also for its units. Verify that the CircuitProject ArrayList has 1 Voltage source, 1 Resistor and 2 wire elements. Verify changing component values causes no crash.	P	This use case can potentially see expansion as new types of circuits need to be drawn from scratch and tested separately. Hence it makes sense to modularize it, despite it being by itself.
UC	Test	If there are less than 4	Verify that the deletion and arrow floating	P	

5.1	deleting multiple, but not all, projects	circuit projects on the home screen add 3 more by clicking the draw button from the floating action button menu and then immediately returning to HomeActivity, thereby creating empty projects. Delete the first 3 projects.	action buttons appear once you select a project. Verify that the size of the CircuitProjects array list as well as the number of children in the LinearLayout view are both 3 less		
UC 5.2	Test deleting all projects	Iteratively delete each project	Verify that both the CircuitProject array and LinearLayout for the project views are zero	P	

Non-functional Tests

Test #	Requirement Purpose	Action/Input	Expected Result	P/F	Notes
	Verify that performance time of image processing is within a reasonable threshold (under 10 seconds)	Take or upload a photo of a circuit (with 4 or less components) with the app and tap check mark to proceed.	The user should remain on the load screen under 10 seconds before the processed circuit displays.	P	Manually tested on 5 different Android devices.