QA Best Practices: GUI Test Automation for EDA Software

Ritu Walia, John Casey
ritu_walia@mentor.com, john_casey@mentor.com

Abstract

Many software failures can be traced back to incomplete, inadequate, or poorly-designed quality assurance (QA) testing. Designing and implementing an effective QA process is critical to the delivery of software that performs as intended and delivers the expected results. A graphical user interface (GUI) is often the primary or single point of interaction between the user and the underlying software. No matter how well-planned a GUI and its functionality may be, any failure of the GUI to operate correctly, either from the user’s or the owner’s perspective, can result in significant loss of market reputation for the entire product. We define and examine ten critical processes in relation to GUI testing and test automation. These include traditional QA testing techniques requiring experience/knowledge on the underlying software and its applications, for effective test execution and software analysis:

- Error identification: determine the common user mistakes likely to be made when using the GUI
- Character risk level: determine which characters may create problems and when/where (e.g., using reserved characters incorrectly)
- Operation usage: determine if/when operations are used incorrectly in the application (e.g., loading an invalid rule file)
- Element relationships: determine if/when different settings or combinations of related elements create problems
- Limitations and boundary values: determine what issues are created when limits are exceeded, or boundary values not observed
- Performance and stress testing: typically observing time and memory consumption performance under extreme conditions
- Smoke testing: finding fundamental instability within builds, to prevent superfluous testing
- Real-world data: using actual data (e.g., customer data) that is not refined or limited, to ensure adequate coverage of customer-critical issues
- Exploratory testing: when bugs are found, performing random testing in the general area, or of elements created by the same developer, to look for additional bugs.
- Efficient bug reporting: giving back a clear bug report that can drive the efficiency of the bug fix

Biography

Ritu Walia: Software QA/Test Engineer for Siemens Digital Industries Software, Integration's team, EDA tools for IC Physical Verification. Previous positions include Corporate Marketing Engineer and Systems Engineer in different industries. Received an M.Sc. in Electrical and Electronics Engineering, specializing in VLSI and MEMs, from University of Massachusetts, USA.

John Casey: Senior Software Quality Assurance Lead for Siemens Digital Industries Software, leading the QA and Testing infrastructure, Integration's team, EDA tools for IC Design Physical Verification, holding 35 years of experience in the Electronic Design Automation industry. Previous positions include Software Engineer, Senior Product Engineer, Senior QA Engineer with leading companies. Received a B.Sc. in Computer Science from the University of Oregon and M.Sc. in Applied Mathematics from the University of Virginia.
1. Introduction:

Technology in every sector largely revolves around the Integrated Circuit (IC) industry, which is continuously advancing; we can safely say nearly all of us now live in a semiconductor-centric world. The semiconductor industry is historically propelled by a solid correlation between technological scaling and performance improvement of ICs. To support that scaling, the functional complexity of electronic design automation (EDA) tools is constantly expanding as well. In turn, this expansion of EDA technology increases the challenges in the road-mapping processes that include parameters and requirements related to new functionalities.

For advancement in the semiconductor industry, where even a single IC project has a few hundred million dollars at stake, EDA tools are constantly challenged to avail the most efficient and top quality utilities to their users. When quality is of paramount importance, risk of failure must be reduced using every means possible, and productivity is essential. To ensure the most productive tool in hand, testing techniques derived from years of experience play an important role. Various traditional methods not only save significant testing effort, but also contribute towards architecting modern software testing techniques. As we know, the key benefit of quality assurance (QA) is the identification and subsequent elimination of errors in a tool's usage. This process must not only evaluate the reliability of the software behind the tool, but also ensure it operates safely and accurately when used in real-life scenarios. Moreover, the global pandemic accelerated digital transformation, which in turn increased the demand for testing to ensure the best quality outcomes for software and its users.

Traditional QA techniques provide a strong background for the effective implementation of these goals. We demonstrate the applicability of these practices using the tools provided by Calibre RealTime interface from Siemens Digital Industries Software, a platform intended to find errors in the layout of an IC during physical design & implementation processes, enable engineers to check the validity of error fixes applied to that layout, and provide design validation of an IC, prior to delivering the layout for manufacturing.

1.1. Automated, Manual & Exploratory testing

For a tester, it is crucial to understand the application design even as the development code is being written simultaneously. With this understanding, arises the need for an efficient and complete testing plan that must include manual, automated, and exploratory testing:

- Test automation provides more efficient code coverage to ensure all required scenarios are covered for most projects.
- Manual testing saves QA time on smaller projects where test automation might not be needed.
- Exploratory testing may play an important role as a precursor to test automation- for discovery, investigation, and learning new test scenarios.

All of these test strategies contribute to effectively tighten release cycle schedules and minimize the risk of test failure & product performance degradation.

1.1.1. Black-Box Testing:

The implementation of the above-mentioned test strategies broadly encompasses black box testing techniques. Black box testing is an opaque-testing scheme, meaning a tester mutes any working knowledge about the software while testing. This style is advantageous because a tester thinks differently than a developer, and can discover unusual software behavior that a developer might overlook. Black box testing makes a tester analogous to a tourist, whose exploratory senses highlight all the aspects and software functionality from a different perspective. Of course, the difference between a tourist and a tester is that the former just witnesses, while the latter evaluates as well.
Functional, non-functional and regression testing fall into the black box testing category. Black box testing can be performed using existing methods like [2],

- Syntax-driven testing
- Equivalence class partitioning
- Cause-effect graphing
- Boundary value analysis
- Requirement-based testing
- Compatibility testing

1.2. GUI Testing:

The graphical user interface (GUIs) is event-driven and consists of one or more dialog boxes, each of which usually contains multiple controls. The GUI operates on a response to stimuli basis, meaning, the initiation of an event is a response to the click of a button or selection of an option. The user must interactively wait for the task to happen, which differentiates GUI tests from testing of its command line interface (CLI) counterpart, where testers control exactly what happens and when. The interface and user experience play a significant role in the application success when it is released to the market. A GUI testing team pays closer attention to the details of the visual dynamics to ensure customer satisfaction and better usability. They test the various aspects of the user interface, such as visual design, functionality, security, compliance, usability, and performance.

What does GUI testing entail?

Most GUI elements are typically developed using instances of pre-compiled objects stored in a library. The source code of these elements may not always be available for coverage evaluation. Input to a GUI consists of a sequence of events, which are generated when a user interacts with the GUI elements. The number of possible permutations of these events may lead to a large number of GUI states. To ensure adequate testing coverage, a GUI event may need to be tested in a large number of these states. Hence, GUI testing demands validation of every bit of logic, GUI feature, or flow of actions to ensure the application works as expected. Not to forget, this type of testing is also closest to the utility and users’ perception of the application, making it extremely valuable.

1.3. GUI Test Automation:

Automation of GUI testing plays an important role by incorporating reusable tests parallel to the development phase, allowing more efficient generation and evaluation of the test results. Automating the tedious and repetitive portions of the task not only reduces costs, but improves the accuracy as well.

Automation of GUI testing also involves the execution of a sequence of instructions that are performed for QA testing, under different conditions (for example, different GUI states). This is considered one of the toughest forms of test automation as it involves setting up a communication interface between the test software and its GUI, which can get very challenging. A complicating factor is that a GUI may be highly subject to change in the form of enhancements and fixes. Such a change drives an increase in the need for repeatability for verification of these tasks. This makes automation an integral part of the testing process. Automation here provides repeatable actions to test functionality as a part of any continuous or agile development process. For example, if a user uses the mouse and keyboard, automated GUI tests would mimic the same behavior by making use of mouse and keyboard clicks or writing to objects present on the user interface, all of which can be incorporated as a part of reusable test script.

When building an automated GUI testing strategy, a tester must first verify the elements of a GUI:

- Validate the font size and readability of font
- Check the alignment of the text
- Check the quality and clarity of images
- Check the alignment of images
- Check the positioning of all GUI elements relative to different screen resolutions
- Verify usability conditions, navigation, data integrity, etc.
- Ensure error messages are displayed accurately [3]

This strategy, if implemented correctly can free up substantial QA resources, both human and computing ones. Automation for some test activities improves the efficiency, and is a necessity for implementing others. While a greater degree of automation can never substitute for a rational, well-organized quality process, considerations of what can and should be automated play an important part in devising and incrementally improving a process that makes the best use of human resources. [5]

The goal is to continue with the process of identifying and deploying automation to best effect, as the organization, process, and available technology evolve.

2. QA challenges in EDA Industry

Some Background:

Design rule checking (DRC) is the process engineers use to determine if the physical layout of a chip design complies with the manufacturing requirements, which is defined in the process design kit (PDK) provided by the foundry [8]. The Calibre RealTime platform, developed by Siemens Digital Industries Software, comprises the Calibre RealTime Custom and Calibre RealTime Digital tools that perform DRC to determine the validity of IC design & physical implementation against the design rules. In our case, a rule file contains these design rules.

This platform allows on-demand Calibre sign-off design rule checking in analog and digital design environments during the physical implementation process, enabling engineers working on ICs to optimize their manual DRC fixes and focus on meeting their power, performance and area (PPA) goals in far less time. [4][5][6][7]

Calibre RealTime interfaces apply rules for manufacturing of the circuit layout, such as minimum spacing of each area where the photomasks will be applied to the silicon wafer, to the proposed circuit design, and report any rule violation to the engineer designing that layout. Finding flaws in an IC design & layout prior to manufacturing can save the design company and manufacturer billions of dollars. If the photomasks used to create an IC layout are faulty, then the whole manufacturing process becomes faulty. This means the manufacturing yield (number of working devices made for each silicon wafer) is lower. Lower yields not only waste area on the silicon wafer, but also the time required to test the final circuit layout on each chip.

Therefore, performing quality assurance on EDA tools, Calibre RealTime platform tools in our case, poses various challenges, some of which may be unique based on the type of EDA tool under test:

2.1. Tool integration

The Calibre RealTime interfaces take entry of design data from a specialized editor tool for IC designing/Placement & Routing (P&R) [9], and display DRC errors in the editor window, both of which require tight integration with the editor. There are many IC design and P&R tools available from different EDA companies. To be competitive, Calibre RealTime interfaces must be tightly integrated with each of these third party tools, both in terms of functionality and performance.

Also, the display of rule violations on the circuit layout canvas must be close to instantaneous. This means, Calibre RealTime tool testing must involve these editors in automated functional, flow, and performance tests. Often, integrating these editors through their testing APIs and driving them through regression testing scripts is challenging due to differences in their overall architecture and the scripting language.
2.2. DRC integration

The Calibre RealTime platform is also integrated with its own circuit DRC tool, so communication must be established between the platform tools, their underlying DRC application, and the external layout editor in regression tests. All of this must happen synchronously for the automated regression tests to be reliable. Because of the existence of so many circuit design rule files from different manufacturers, Calibre RealTime products must have a large comprehensive test suite to cover all of these rule files.

2.3. Data formats

Design data formats differ between the editors for IC designing/P&R, so test-cases must be created for all of these different editor data formats, while keeping them functionally identical.

2.4. Software versioning

While the operating system for running the layout editors and P&R tools is restricted to Linux, each version of Linux supported by the editors (multiple RedHat and SusE versions) must be tested on, and regression tests must be compatible.

Creating regression tests create the following challenges:

- Operating the IC editor and P&R tool with their different forms and testing APIs
- Operating inside the Linux operating system with its different scripting syntaxes and OS levels

2.5. Performance and stress testing

Performance and stress tests must measure the CPU time, the wall clock time, and process memory consumption by both the Calibre RealTime process and the layout editor process. Tools that measure all of these quantities while a substantial circuit design is being processed, must be employed.

Traditional QA testing techniques play a significant role to overcome all of the above challenges. Let’s see how.

3. Traditional QA testing techniques

Traditional QA testing techniques incorporate a set of quality attributes that maximize test coverage. If any of these attributes are missing in a functionality, that functionality should be subjected to bug detection and quality analysis.

3.1. Error identification

Error identification is commonly known as the error guessing technique. Inputs are chosen by the tester based on both intuition and experience. While random guessing is acceptable, this technique is most productive when some understanding and analysis is applied to find common mistakes that programmers might introduce into a form or GUI element. This method follows no set rules and is instinctively brought to use. Below are some of the key paradigms of the error guessing technique:

3.1.1. Reserved characters

Testers must understand the underlying software package used to create the GUI, and from that knowledge, determine which characters may introduce problems. For example, with Tcl/Tk programming languages, the characters ‘$’, ‘{’, ‘}’, ‘[’, and ‘]’ are all reserved. Testers should enter these
specific characters when entering information into the form, to determine if the form accurately detects the improper use of a reserved character. Likewise, if the GUI element is written in Java, testers must use the reserved characters in Java when testing data entry in the form.

Testers must also be aware of special use cases involving reserved characters. For example, in the Tcl programming language, a custom command prints out a string of characters enclosed in double quotes. Testers must be aware of reserved characters that can be used within this string as an input. Figure 1 shows results from Calibre RealTime interfaces, when the reserved characters are included in different cases within the enclosed string, which are perceived by the Tcl interpreter differently that resulted in error:

![Figure 1](image)

Figure 1. The left screenshot shows the error caused by using ‘{’ in the string. The right screenshot shows the error caused by using the ‘$’ sign in the string.

### 3.1.2. Operation usage

Operation usage is understanding the underlying concept of data under test and determining if/when operations are used incorrectly in the application. Testers must first understand the application of the data being entered. For example, some operations are intended for use on objects of a specific drawing layer of an IC layout design, such as a polygon on layer Metal2. Does the GUI element work when operated on a non-metal layer? What happens if no such element is selected, or if there are no such objects of that type in the design, or if nothing is selected? While programmers in general strive to test their work and understand IC layout design, there may be inexperienced programmers for whom it is necessary to check their work when applied to different data types than what is specified in the original testing requirements. This approach is often more of an “educated guess” as to what might break a certain piece of functionality. Having knowledge of the different types of drawing layers in an IC layout provides testers with the intuition to try different layer types.
Element relationships and cross-functionality testing

This technique determines if different settings or combinations of related elements create problems. If there are relationships between elements of the GUI, and you are testing one of those elements, do different settings of the related elements result in the correct behavior?

For example, when an IC layout design is sent to Calibre RealTime interface, there is an option to include external files of polygons (in relation to IC layout designing, ‘polygons’ are used to draw complex orthogonal shapes of metal or polysilicon) to the design, and there is also a polygon limit that determines whether the layout design can be processed at all. Here, it’s a good idea to be mindful of potential scenarios such as whether or not the polygons are contained in the external files included in the polygon limit check.
3.2. Limitations and Boundary Values

The technique of boundary value analysis (BVA) identifies issues that can exist when limits are exceeded, or boundary values not observed. Limitations and boundary value testing is one of the most effective software testing techniques that validates input value ranges, where behavior is expected to change the most. The basis of boundary value analysis is testing the inputs at the boundary limits. It also entails the concept of Equivalence Class Partitioning, meaning that if the input conditions are in the range of $x$ and $z$ values, then a test case should be created with sample data “$x-1$, $x$, $x+1$, $y$ (mid point), $z-1$, $z$, $z+1$” that covers the boundaries (below boundary, boundary value, above boundary) at the two ends of the data set spectrum. Early errors are often found at these extremes. If the program can survive testing at the extremes, it is likely to survive less extreme results.

What kind of data type is being used is handy information in this case. For example, if a developer assigns a variable to be an 8-bit unsigned integer to store input value in the text box, the value of this variable should range between 0-255. Testers must be mindful of cases that may contain out-of-range values (such as an input like -1) or text/input field limitations for strings (where too large a string may cause buffer overflow/data corruption), to detect any unexpected behavior in the application during early stages of testing.

Within the EDA industry, there are often limitations imposed by the design & implementation process or tool being used when entering information into a form. These limitations can usually be found in the process or tool manual. For example, a mask layer number is typically a positive integer in most IC layout tools. Can the entry box of this layer number accept negative numbers, non-integers, or numbers beyond the tool or process limits, when entered? If the entry should be a number, what happens when a common name used in the IC layout design, such as “metal1,” is entered instead of a number? Covering the corner cases is essential for valid limitations and boundary value testing.

Testers however, must take care not to rely on this technique alone, as boundary value analysis and equivalence class partitioning do not explore combinations of input conditions, and may be restricted for an astronomical combination of inputs. For example, each IC layout design can contain hundreds of layers, and hundreds of design rule checks operating on each layer. Different IC layout designs can use different ways to map each layout editor layer to each physical layer, each rule check for each foundry process uses different minimum spacing rules for each layer, etc. Using ‘educated’ guessing of boundary values can expose the most important flaws in the tool, which is vital for test validity and efficiency.
3.3. Functional Performance and Stress Testing

Performance problems in code or functionality that are most critical to customer’s success must be identified in performance tests, such as the time needed to identify rule violations and highlight them on the layout editor canvas. Applications must be consistently run and reported on supported hardware so that any degradation in performance can be easily identified and fixed.

Performance tests are usually executed to determine how quickly the program runs to decide whether the optimization is needed or not. These tests can also expose many other bugs. A significant change in performance from a previous release can indicate the effect of an introduced coding error. For example, if testers investigate how long a simple function test takes to run today, and then discover that the same test on the same machine runs much faster or slower tomorrow, they’ll probably check with the programmer or investigate for a bug. Either case is suspicious, because something fundamental about the program has changed.

Functional performance tests must include layout designs that are larger, complex, and contain all elements that customers typically have in their layout designs. Asking relevant questions is essential—are forms opening and closing in a timely manner, and are other operations still showing adequate performance at the end of a stress test? Is the memory consumption acceptable, or are there signs of a memory leak?

Stress tests are also important to test prolonged usage of the application the way a customer would. Stress testing involving multiple designs and multiple layout editors must be performed. For example, testers can simulate a prolonged user session if that is typical customer usage. Another option is to keep inputting information to a form in fast succession, to see if the application can handle this scenario. Since customers of layout editor EDA tools often load their designs and work over prolonged periods of time while a design is in progress, the Calibre RealTime tools must be stress tested over prolonged periods of time as well.

Figure 6 illustrates a design divided up into tiles that has a script performing repetitive DRC runs, then a highlight operation on each tile. The performance would be acceptable if output of the cycle equals the number of DRC checks finished and total memory consumed is within the threshold.

A performance test suite must measure execution/processing time and memory consumption to completely cover all aspects of testing the key features of the application. Underlying third-party API changes may result in significant performance degradation that might be difficult to track, making it highly important to run performance tests as often as feasible. On the other hand, there can be significant performance up-ticks with continuous software improvement that can help the team track progress using the performance charts.
Performance and stress tests are equally important as functional tests. If these test conditions are not covered completely as well, negative customer experiences can occur.

![Figure 6. Stress testing: Script that tests the software in random areas of a complex design for a prolonged period can uncover performance slowdowns and memory mismanagement](image)

### 3.4. Smoke Testing

Smoke testing is the build verification testing (BVT), which consists of a standard suite of tests applied to a new build. In testing processes, the aim of smoke testing is to detect major issues early on—the tests look for fundamental instability, or key features that are broken or missing. If these tests fail, tester can abort the testing altogether, knowing that this software build is essentially unstable. Instead, one can continue testing the old build, or wait for the next one. In the simplest terms, smoke testing verifies that the important features are working and there are no showstoppers in the build. It is a rapid mini regression test of major functionality. Smoke tests qualify the build for further formal testing. Smoke tests establish system stability and conformance to the requirements.

### 3.5. Using Customer Design Data

The largest and most complex design flows and data from the customers, should be prominently used in the set of test cases. If the test cases and data are too simple, testers will not cover all the aspects that are important to customers, which can lead to inadequate testing of the product.

When testing the individual functional and performance aspects, it is better to use customer databases as the basis for these tests instead of creating a contrived design. Creating a smaller version of the design is often required to keep the regression run time as low as possible.

Most importantly, including customer designs in regression tests may require non-disclosure agreements (NDAs) between a test team and the customer. When required, NDAs should always be completed before testing begins.

![Figure 7. Testing of simple layout design versus complex customer layout design](image)
3.6. Exploratory Testing

When a problem is found for one GUI item, there is likely to be a problem with related items in it, or the items added by the same developer. Testing that found a bug, can be leveraged to create test cases that have a scope of finding bugs in another GUI element. Example, cross functionality testing between two different windows.

![Cross-functionality testing](image)

Figure 8.(a)(b). Cross-functionality testing for cross-highlighting between DRC results in the layout viewing window and the Realtime-Results Viewing Environment Window.

3.7. Efficient bug reporting:

Testers should be able to advocate for bug fixes they consider high priority. How testers write and present test outcomes can impact the perception of the reader. An informative, concise, and clear bug report can drive the efficiency of the bug fix, while a weak one may generate extra unnecessary work for the developers. Taking the time to create a value-added bug report can significantly improve overall productivity.
4. Conclusion

The traditional QA techniques described in this paper, when executed correctly, play a significant part in successful development of an IC design & implementation project, and finally to the ROI of the organization.

We discussed the importance of implementing appropriate testing methods for different conditions and operations, and explained how each technique might be evaluated in relation to GUI testing and its automation. We examined the elements of an EDA tool, employed the traditional testing techniques for overcoming QA challenges, and thereby demonstrated the development of an effective quality process. By correctly employing essential techniques, test engineers can ensure the overall quality and productivity of their testing processes. In turn, by providing a thorough and effective QA evaluation, they help ensure that products deliver premium quality and maximum profit when released to market.

References


