Improving R&D Productivity by Triangulating Failures

Richard Léveillé, Synopsys Inc.
Richard.Leveille@Synopsys.com

Abstract

Software developer productivity is vital to an organization's ability to meet its customers' needs. This presentation focuses on an innovative solution to optimize the regression testing cycle. Using the principles of triangulation to the software domain, we are able to identify the cause of the regression failures, shorten the test and debug loop, and enable quicker failure resolution. This paper presents background on the code integration method, describes our use of the Triangulation system of failure analysis, and provides insights of the impact of the solution in developing predictable and high quality software releases.

Biography

Richard Léveillé is a professional with over 20 years of experience in commercial software development, Software Quality Management Systems, and Software Release Management. Over the years, he has held positions as developer, Project and Program Manager in various organizations, as well as Quality Assurance Manager with experience managing ISO Certification programs. Currently, he is Senior Manager Corporate Quality and Program Management at Synopsys. In this role, he has had direct involvement in process design, Customer Satisfaction Programs, and promoting software development best practices in the organization.

Richard graduated as an Electrical Engineer from Sherbrooke University, Canada. He holds certification from Stanford University as a Stanford Certified Project Manager (SCPM).

He is a member of the American Society for Quality (ASQ) organization.

1 Introduction

With constant pressure for faster development and additional functionality, a software developer's productivity is paramount to an organization's ability to meet the needs of its customers. A product must be kept current with customer needs and adapted to emerging technologies.

In the context of EDA (Electronic Design Automation) applications, there are numerous challenges in delivering on the promises of predictable and high quality releases:

- Applications range in size from ½ MLOC (millions Lines of code) to more than 20 MLOCs
- Size of development teams range from 20 to more than 250 individuals
- Worldwide teams perform around-the-clock development.

To address these challenges, an organization has a responsibility to constantly evaluate opportunities to best meet expectations from the customers while keeping a highly productive development environment. A structured and methodical approach is necessary to create conditions for effective and efficient development leading to a successful release.

As we consider the entire development life cycle, we realize that testing still represents a major portion of the development schedule. Actually it is estimated at ~30% for multiple stage testing [1]. Despite best efforts to design-in quality, software development still relies on testing of all sorts to validate the implementation. Opportunities for productivity improvements are numerous in that space and many avenues of investigation can be pursued.

By examining the various tasks involved in testing, we have found that the Test and Debug activity a developer performs is repetitive, time consuming, and most conducive to automation. Reproducing an error is always the first step in debugging. The task requires recreating proper conditions, extracting results, and comparing with a baseline. There is one set of tests in particular, regression tests, that always run on a known configuration and are totally under the control of the software development team.

This is where the concept of Triangulation comes into play. "Triangulation" [2] is a method used to locate an object by correlating information from different angles. In the context of software development, and specifically when we consider failure analysis, triangulation can help locate the cause of the failure to speed up defect remediation.

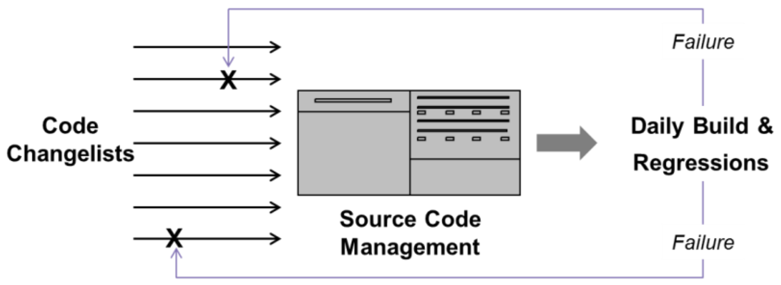

Figure 1 - Build and Regression Failure Tracking

In order to accelerate defect location and resolution, a triangulation system, consisting of software process analysis tools and methods, is used to identify the "bad code changelist" that may cause build failures or regression failures, as illustrated in Figure 1. A "changelist" [3] is a set of code changes made in a single checkin the Source Code Management System (SCM). An internally developed tool, called

Tracer, was developed to perform failure triangulation and thus improve the efficiency of the daily Build & Regression cycle.

This paper outlines how the development environment ties to productivity, describes the triangulation system, its flow and implementation, and ultimately, how it tangibly contributes to higher productivity for the developers.

2 Conditions for High Developer Productivity

As we indicated above, the focus is to improve developer productivity. To do so, the organization must provide the right environment and tools to enable it. This is accomplished by providing favorable conditions and by removing obstacles for efficiency. We are focusing here on two conditions supporting high productivity: **Stable code** and a **quick feedback loop**.

2.1 Continuous Code Integration Model for Stable Code

Business needs demand faster releases and the capability to be more responsive to customer requests and issues. This requires high predictability for delivery schedules and, therefore, constant monitoring of the state of the code.

With many developers aggregating code changes in the same development branch, it is important to ensure stability of the code at all times. The changes from one developer might affect the whole team and slow down progress. For that reason, and to minimize the impact of large code merges, the Continuous Code Integration Model [4] is adopted.

Every developer performs development in their own “sandbox.” As soon as the development is complete, they check the code into the main code branch. Before a developer checks in code, a set of strict checkin criteria must be met. The earlier the changes are integrated with the rest of the code, the sooner its effect is factored in and any issues addressed. It is the goal to minimize large code changes being checked in at once because it usually imposes a clean-up phase and affects the productivity of the entire team.

At the product level, daily build and regression runs are performed. Results are posted and the status of the branch is constantly assessed. The goal for the product team is to keep the branch very stable as measured by a regression pass rate between 98% to 100%. In some cases, the team will allow passing rates to slip down a bit lower prior to the alpha release date milestone, but the criteria go up quickly for the Beta milestone.

Those are challenging goals but it has enabled product teams to release stable software quickly if required. The release criterion is 100% regression passing rate. If there is a need to release, a focused effort to close on fewer regression tests makes the task manageable and achievable within a reasonable and predictable time. This is a great contributor to improved responsiveness.

The success of this approach depends on a developer’s discipline to fix regression failures as soon as they are discovered. As part of the development discipline, the expectation is that a developer’s first task of the day is to review assigned regression failures and take action. Test results are made available to the developers via a web application that consolidates all assigned issues for each developer.

2.2 Short Feedback Loop for Developers

One condition essential for high productivity is to provide a short feedback loop on any task so that remediation, if necessary, can take place before any ripple effect can be triggered. In a development environment with a quick feedback mechanism, developers can immediately see the effect of the changes they make. With feedback in minutes instead of days, developers can confidently address any remaining issues and proceed to the next assignment without having to constantly revisit their previous work.

A multipronged approach is taken to shorten the feedback loop and support the developers:

- Early validation of the checkin on the code build (part of triangulation mechanism)
- Validation of the effect of each checkin on the integrated code base (part of triangulation mechanism)
- Predictable and fast turnaround time for the daily Build & Regression cycle

The daily Build & Regression is closely managed to ensure all developers have complete results before the start of their day. A central code build takes place on a snapshot of the code and the complete set of regressions, consisting of thousands of test cases, are run against the various development platforms. This operation establishes a new and fully qualified baseline for the next day of work for the entire development team.

The Tracer tool described herein, which performs Build triangulation and Regression triangulation, is designed to further shorten the feedback loop for developers.

3 Effective and Efficient Triangulation

As mentioned, the software development progress relies heavily on the stability of the code and the reliable outcome of the Build & Regression process. Any failures must be quickly dispatched and resolved in order to keep all members of the development team productive. Therefore, the goal of the automated tool is to locate the point of failure to notify the developer who can best address the issue.

To achieve this objective, the Tracer tool interacts with the Source Code Management system (SCM) and the Build & Regression system. As illustrated in Figure 2, Tracer essentially monitors the code changes submitted throughout the day to improve success of the official daily build. Tracer also performs analysis of the regression test results to identify the source of failures.

Tracer includes two components: Build Tracer and Regression Tracer.

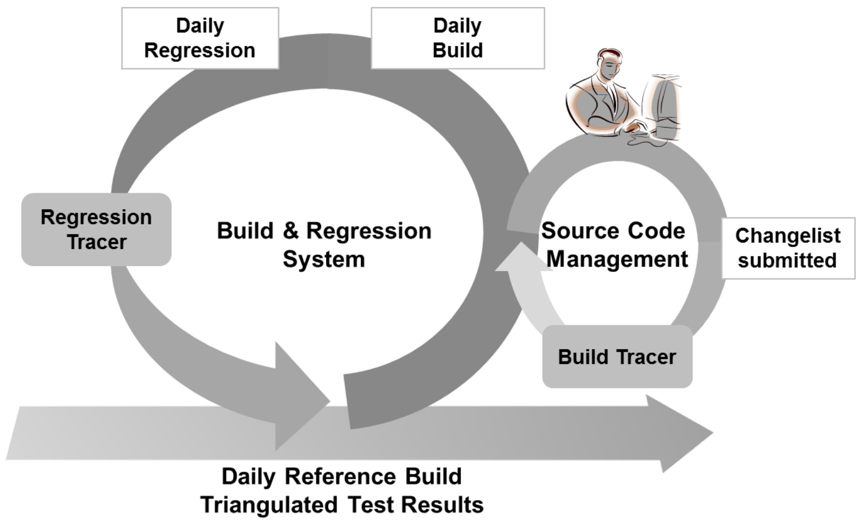

Figure 2 – Tracer Solution as part of Continuous Integration Model

Build Tracer builds every changelist as they come in to make sure it won't break the daily build on the Integration branch. If a failure is detected, Build Tracer sends email to the person who checked in the code.

Regression Tracer uses two sources of input:

- Daily regression test results from regression tool
- Executables for every changelist saved by Build Tracer

Regression Tracer compares today's daily regression test result with yesterday's result. For every new failure detected, Regression Tracer invokes the test cases that failed and runs the tests against all executables built from each changelist of the day. Regression Tracer can then point out suspicious changelists and notify owners.

Let's examine in more detail the logic of the Build Tracer and Regression Tracer as key components of the Continuous Integration process, and the respective results.

4 Build Triangulation

Build Triangulation supports two main objectives:

- Achieve 100% daily build success
- Detect build errors early on and notify the developer who checked in the code

The Build Triangulation workflow, illustrated in Figure 3, is essentially a four-step process. Build Tracer performs the following functions:

- Starts an incremental build at every changelist as soon as the code is checked in
- Detects any build errors
- Notifies success or failure to the developer who checked in and other stakeholders (e.g. integrator, branch manager)
- Maintains record posted on web pages

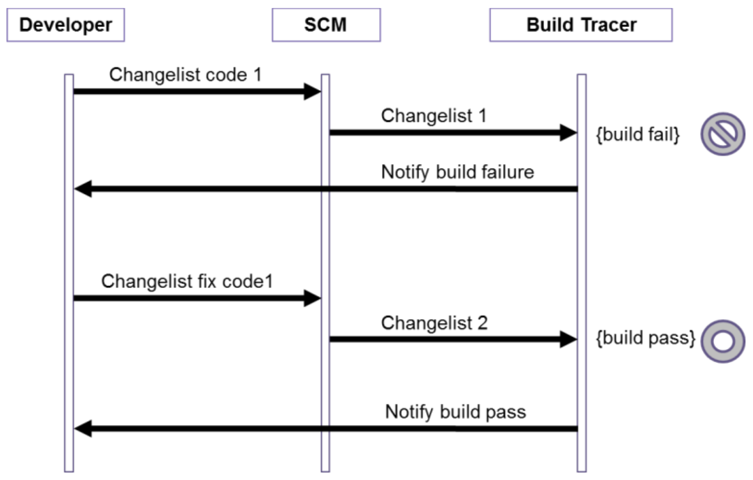

Figure 3 - Build Triangulation Workflow

The code branch is constantly monitored. An email notification is sent as soon as the incremental build fails. The developer can then take immediate action to correct the failure before the daily build cycle takes place. Build Tracer contributes to higher daily build success rate by ensuring all changelists have been validated prior to the daily build.

Each successful incremental build is kept for later use by the Regression Triangulation system.

4.1 Results from Build Triangulation

The effectiveness of Build Tracer can be measured and its impact observed. Table 1 below profiles a sample of products with key indicators to assess effectiveness:

- Volume of changelists in the code branch
- Number of changelists identified as faulty by Build Tracer
- Percentage of daily builds prevented from failure by use of Build Tracer

Table 1 - Build Triangulation Effectiveness

	3-Month Period		
	# Changelists	**% Faulty Changelists**	**Prevented Daily Build from Failure**
Product 1	592	4%	26%
Product 2	1675	1%	21%
Product 3	5771	2%	70%
Product 4	1826	2%	33%
Product 5	1398	1%	19%
Product 6	1121	4%	35%
Product 7	1121	2%	28%
Average		**2%**	**33%**

Looking closely at the results, it is reassuring, first, to observe we only have a small percentage of bad changelists. We note that the number of bad changelists does not necessarily match the volume of changelists. Some of the large teams with hundreds of developers actually have a smaller percentage of faulty changelists than smaller teams with ~30 developers. The rigor of the pre-checkin practice is a better indicator of the frequency of bad changelists.

The other observation is the fact that, even with a small percentage of bad changelists, the potential for disruption and destabilization of the code branch is significant. It only takes one bad changelist, without the Build Tracer in place, to make the official daily build fail. On average, Build Tracer actually prevents 33% of daily builds from failing. In other words, if Build Tracer was not in use, 1 out of 3 daily build would have failed. This is a significant gain in productivity for all developers.

5 Regression Triangulation

The objective of the Regression Triangulation is to ensure maintaining a high Regression Pass rate on the code branch assuring continuous readiness for release. In order to do so, the Regression Triangulation must cope with complex test environment and the large number of regressions running on various platforms.

To put things into perspective, the typical product runs over 5k -10k test cases/platform. Some of the products exceed 35,000 test cases. With a goal of keeping the pass rate above 98%, it means that 200 test cases might be failing at any given time for a product with 10,000 test cases.

The Regression Triangulation workflow, illustrated in Figure 4, is initiated as soon as the Regression test results against the daily build are available. Regression Tracer performs the following functions:

- Compares today's daily regression test result with yesterday's result
- Selects test cases to triangulate
- Invokes these test cases upon all executables built from each of the day's changelists
- Notifies developer(s) who made the suspicious checkins

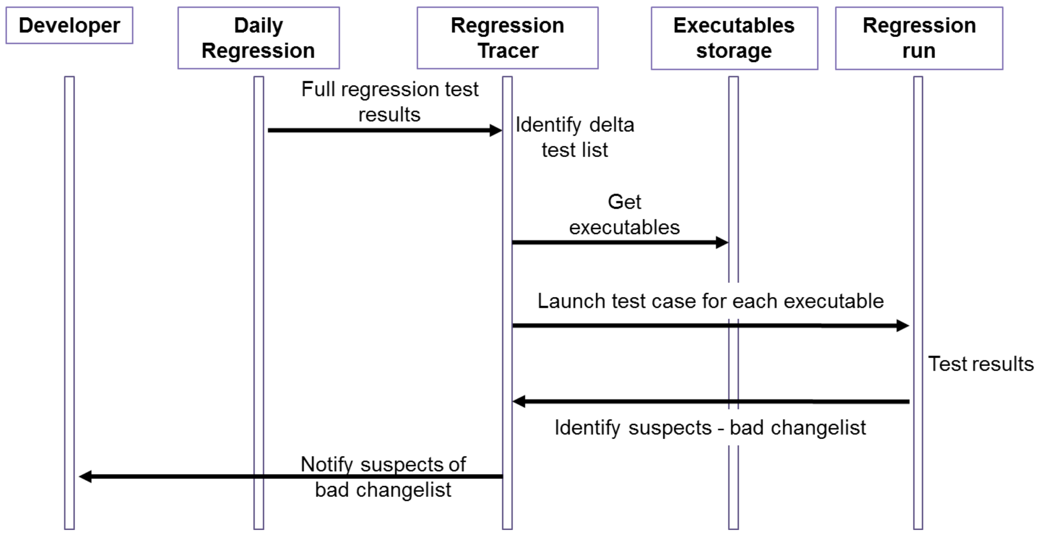
Developer
Daily Regression
Regression Tracer
Executables storage
Regression run
Full regression test results
Identify delta test list
Get executables
Launch test case for each executable
Test results
Identify suspects - bad changelist
Notify suspects of bad changelist

Figure 4 - Regression Triangulation Workflow

The first step is to determine the list of tests to triangulate. Only the new test failures for the day and test cases whose result worsened are analyzed; previously failed tests are already assigned and their resolution is tracked as part of the daily monitoring the product team performs. Table 2 below illustrates the logic to determine the list of tests, or deltas from the previous day's results, requiring triangulation. Applying this logic reduces the number of cases to triangulate for any given day.

Table 2 - Logic for Tests to Triangulate

Daily build / Test Case Results	T1	T2	T3	T4	T5	T6	T7	T8	T9
Yesterday's results	Pass	Pass	Pass	Fail	Fail	Fail	Fatal	Fatal	Fatal
Today's results	Pass	Fail	Fatal	Pass	Fail	Fatal	Pass	Fail	Fatal
Cases to Triangulate		√	√			√			

Once the list is available, Regression Tracer runs each test against each of the builds of the day. The Regression Tracer utilizes the incremental builds from the Build Tracer. By analyzing the Pass/Fail/Fatal patterns, Regression Tracer can locate the suspect changelist and notify the appropriate developer. The following Table 3 illustrates three (3) typical cases.

Table 3 - Pass/Fail Triangulation Patterns

Regression Tracer launches test case for each executable
(One Executable per Changelist)

	Exec2	Exec3	Exec4	Exec5	Exec6	Exec7
Test2	Pass	**Pass**	**Fail**	Fail	Fail	Fail
Test3	Pass	Pass	**Pass**	**Fatal**	Fatal	Fatal
Test6	Fail	Fail	**Fail**	**Fatal**	Fatal	Fatal

Inference: (One Executable per Changelist)

- Test2's failure is found within Executable4 corresponding to Changelist4
- Test3's failure is found within Executable5 corresponding to Changelist5
- Test6's failure is found within Executable5 corresponding to Changelist5

In the case of Test2, we observe that the failure appears when running against Exec4. This indicates Changelist #4 as the suspect. Test3 goes from Pass to Fatal with Exec5 (Changelist #5), identified as suspect. "Fatal" is a third state of the regression test outcome as it terminates the application as opposed to "Fail" which might, for example, just return the wrong results for that operation. For Test6, the result worsened from Failed to Fatal with Changelist #5 identified as the suspect.

From an operational standpoint, instrumenting and running these combinations can be onerous. Continuous refinement of the Tracer algorithm helps to manage the operation within acceptable resources. It is key to realize though, that if the Regression Tracer is not in use, the same tasks would need to be performed by each developer and using a less optimal compute environment, therefore introducing more variability and delays in the debug cycle.

5.1 Results from Regression Triangulation

In order to assess the effectiveness of the Regression Triangulation, we consider four (4) indicators:

- Volume of regression tests
- Volume of regression tests to be triangulated
- Proportion of suspect Changelist found – Found Rate
- Accuracy in identifying the correct Changelist suspect

The results for the sampled products are summarized in Table 4.

Table 4 - Regression Triangulation Effectiveness

	12-month		
	Volume of regression tests	**Volume of regression tests to triangulate**	**Found Rate**
	Daily Average	Daily Average	12-month Average
Product 1	8200	22	42%
Product 2	7900	26	51%
Product 3	7000	3	54%
Product 4	4200	5	67%
Product 5	22000	35	58%
Product 6	4200	8	63%
Product 7	5100	18	70%
Average			58%

The first observation is the volume of transactions to be triangulated is manageable, with an average daily load of new test failures at relatively low levels over 12 months of results. There are days where the volume of regression to triangulate is high. If the regression pass rate drops drastically from one day to the next, for example, the Regression Tracer will analyze only a portion of the results. Our experience shows that generally, this is caused by a single changelist, or by a code merge. At that point, there is no basis to triangulate every single issue individually. Limiting the triangulation to a maximum daily number allows better management of the compute farm resources.

One of the indicators, the accuracy of the findings, is not provided in the table. This is because we found that very rarely does Regression Tracer identify the wrong changelist. There are some anecdotal cases, but typically very few are reported. This is due, in part, to the logic applied to determine the suspect which is “conservative”, meaning that if there is a doubt in identifying a single suspect or because the test environment appears unstable, the case is classified as “Suspect Not Found.” The accuracy of the findings was favored in the implementation to establish high confidence in the results.

That last point, also explains, in part, the lesser than 100% Suspect Found Rate, but only in part. The Found Rate averages 58%. In general, reproducing failures involves many moving parts. To get a more comprehensive understanding as to why some failures cannot be triangulated, we have to consider some of the variables.

- **Nature of the application**: Within the domain of the EDA (Electronic Design Automation) software application, not all tests are deterministic [5]. For example, an EDA application compiles instructions from the programming language for two equivalent but slightly different electronic circuits.
- **Brittleness of the tests**: In many cases, the results must be interpreted or are acceptable within a range of values. For some of these tests, the range is too strict or requires constant revision. The design of the tests must then be revised.
- **Modifications in the test environment**: Changes in the test environment may also prevent Regression Tracer from singling out the suspect. For example, the test passes for all changelists during the Tracer runs. This might be because the test environment during the daily regression run was unstable.

Root cause analysis of cases where the checkin was not identified by our regression triangulation system leaves a path for further improvement initiatives to refine how the tests are designed and how to improve control of the test environment.

6 Impact on Productivity

The argument for a developer's high productivity is based upon two aspects: a short feedback loop and a stable code base. The results of the Build Tracer and Regression Tracer demonstrate significant gain in the goal to shorten the feedback loop.

To assess the gain on stabilizing the code, we ultimately need to look at the Build Success Rate and Regression Passing Rate. Table 5 provides the figures for a 12-month average period.

Table 5 - Impact on Productivity

	Daily Build Success Rate	Daily Regression Passing Rate
	(12-month Average)	
Product 1	98%	95.6%
Product 2	97%	98.3%
Product 3	95%	99.6%
Product 4	100%	99.2%
Product 5	99%	98.0%
Product 6	96%	99.4%
Product 7	96%	93.7%
Average	97%	98%

The results definitely show, for the majority of the products, high predictability of the Daily Build, as well as a code branch stable enough to allow daily code integration from many developers. The Build Success Rate does not yet achieve the goal of 100%. The root cause analysis has identified few improvement opportunities in the Build process that will close the gap to 100%.

The Tracer tool has been in use for more than two years by many teams. Its use has had many positive outcomes:

- Reduced effort required to dispatch failures to all stakeholders
- Reduced effort by each developer in parsing reports and running multiple tests to locate the point of failure
- Shortened the defect resolution cycle
- Improved stability of the code development branch

Many of these points are difficult to quantify but product teams within our company using Tracer agree that it is now essential to their operation. For one particular team, Tracer is credited in saving considerable effort by the QA team dedicated to reproducing failures.

One side effect in deploying Regression Tracer is on the assigned responsibility for taking action for a failed test. The accountability now lies with the person who makes the regression test fail as opposed to, in the past, the person who owns the regression test. This is gradually forming a more disciplined culture within the organization.

7 Future Development

As we have presented in this paper, the solution is not optimal in all aspects. There are further avenues of development to pursue to increase the value proposition of the Tracer tool. Among those identified are methods and processes needed to do the following:

- Characterize cases where regression triangulation cannot determine a unique changelist for the failure
- Shorten turnaround time to obtain regression triangulation results
- Triangulate other test suites beyond the ones for regression tests

8 Conclusion

R&D productivity has been at the forefront of Synopsys development strategy. Automation through tools like Tracer is one successful approach towards optimizing the development resources. The point of improving productivity for developers is to alleviate tasks that are not contributing to the value of our products for our customers. All these gains result in more time to develop new features and solutions that solves our customer's challenges.

The Triangulation System has proven effective at providing predictable and high quality software releases. The benefits of the system are tangible to the development team on a daily basis and it is now part of the standard tool set in the Synopsys development environment.

References

Acknowledgments

I would like to sincerely thank Wei Gui and Sangeeta Aggrwal for their patient support in providing the data necessary for this paper. It is much appreciated.

Reference

[1] Jones Capers and Bonsignour Olivier, "The Economics of Software Quality", Addison-Wesley 2012

[2] Wikipedia, "Triangulation definition", http://en.wikipedia.org/wiki/Triangulation

[3] Wikipedia, "Changelist", http://en.wikipedia.org/wiki/Revision_control

[4] Fowler, Martin (1 May 2006). "Continuous Integration". *martinfowler.com*. (Accessed 9 January 2014)

[5] Wikipedia, "Nondeterministic algorithm", http://en.wikipedia.org/wiki/Nondeterministic_algorithm