www.spkaa.com
Ph: 888-310-4540

SPK and Associates
900 E Hamilton Ave, Ste.100
Campbell, CA 95008

Static Analysis: Improving Quality by Finding Coding Issues As Soon As Possible

Introduction

Many companies utilize static analysis tools in some fashion. However, we often find they are not leveraging them to their fullest potential. At SPK and Associates, we believe in driving quality up front, at the Engineering Desktop, and not relying solely on finding issues during QA verification and validation testing.

In this article we will review one of our customer experiences with respect to static analysis for early defect detection/correction.

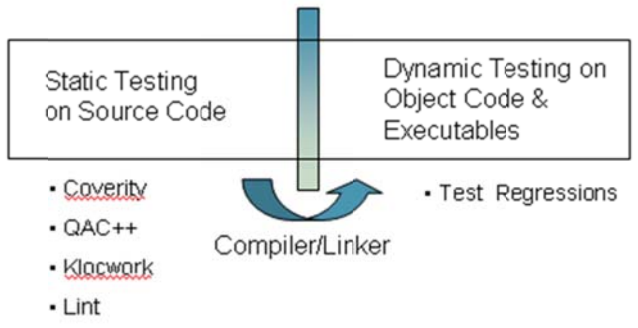
Review of Static Analysis Concepts

High levels of quality cannot be "tested into" software. It needs to be aggressively managed/maintained at all phases of development. And preventing flaws is far more cost effective then trying to remove them.

Static Analysis focuses on inspection of source code to locate defects without actually executing it. The oldest and most widely known static code analyzer is lint, the basic C & C++ code scanner bundled with UNIX.

Dynamic testing (the counterpart to static analysis) is the traditional verification/validation process involving the execution of software. Example: The running of regression tests utilizing replay scripts/makefiles and golden data for comparison. See Chart 1

Static Analysis vs. Dynamic Testing
Chart 1 [1]


Static Testing on Source Code
Dynamic Testing on Object Code & Executables
• Coverity
• QAC++
• Klocwork
• Lint
Compiler/Linker
• Test Regressions

Some of the more common code issues identified by Static Analysis Tools include [2]:

- Memory & Resource issues: e.g. dynamically allocated memory which is not freed, files, sockets etc. which are not properly deallocated when no longer used;
- Illegal operations: Division by zero, calling arithmetic functions with illegal values, over- or underflow in arithmetic expressions, addressing arrays out of bounds, dereferencing of null pointers, freeing already deallocated memory;
- Dead code and data: Code and data that cannot be reached or is not used. This may be only bad coding style, but may also signal logical errors or misspellings in the code;
- Incomplete code: This includes the use of uninitialized variables, functions with unspecified return values (due to e.g. missing return statements) and incomplete branching statements (e.g. missing cases in switch statements or missing else branches in conditional statements).

Looking At Tools

A quick search online will return over three dozen static analysis tools. Their capabilities range from local syntax and metrics collection to being able to analyze programs at a semantic level.

We narrowed down the evaluation to 3 Tools - Coverity, Programming Research's QAC/QAC++, and Klocwork.

Programming Research's QAC/QAC++ tools easily supported incremental analysis of source code at the engineer's desktop. This enabled the benefit of early detection/correction, before the engineer checks their code into CM for system builds.

PRQA Programming Research

However, at the time of the evaluation, we felt that QAC/QAC++ did not provide deep enough inter-procedural (whole-program) analysis across function, file, and module boundaries to achieve 100% path coverage. Additionally QAC/QAC++ generated more false positives and negative errors than some of the other tools.

Coverity, our second tool in the top 3, was viewed to be the industry leader based on the number of checkers used in analysis and a lowest number of false positive and negative errors.

coverity™

However, at the time of the evaluation, Coverity did not easily support desktop incremental static code analysis, requiring all code to be built in the context of the system. And the existing turn-around time for the system build and analysis was taking longer than an overnight run. This lengthy feedback loop resulted in an inefficient and reactive development behavior. Preventable bugs are allowed into code, not caught and reported till much later. And, the engineer was faced with remembering the context of the error of code that may have been updated multiple times

Our goal was to provide the best of both: Quick feedback via incremental/desktop analysis along with excellent coverage and precision (low false positives). Knowing that neither Coverity nor PRQA provided this (at the time), we included a third vendor for consideration (Klocwork) which did provide both incremental desktop and system level coverage.

Also, in addition to quick turn-around and excellent precision, our client was requesting the ability to have architectural visualization and code optimization. The purpose was to aide debugging, code reviews, architecture discovery, and the understanding of system complexity. These tools are often bundled with (or have as a prerequisite) static analysis tools. Both Klocwork and Coverity offered excellent architecture visualization applications.

Requirements

Our final static analysis requirements list included the following:

- High Degree of Completeness
  System level deep inter-procedural (whole-program) analysis across function, file, and module boundaries.

- Excellent Precision
  - High Quality/Meaningful Errors Found
  - Low False Positives
  - Filtering/Tuning/Prioritization of results

– Custom Checker Functionality

- Robust Performance
  Incremental/Desktop analysis without loss in precision in under 30 minutes System (level 0) non-incremental build, analysis/reporting completed overnight.

- Aligns with Clients Infrastructure
  – Supports C, C++
  – Runs on Linux/Solaris OS
  – Runs on Virtualized Machines
  – Farm runs must support LSF

- Lowest cost of ownership possible
  – License support of floating licenses
  – Minimal IT support
  – No dedicated internal expert required
  – Low Training Impact (Intuitive)
  – Low storage requirements

- Must be a Proven Solution

– Industry top 3 established company
– Extensive Customer References

Evaluation Score Card
We compared the three static analysis tools against the requirements. Evaluations were done on two separate products at the client site. The findings were:

Requirement	**Coverity**	**Klocwork**	**QA C/C++**
Completeness	System	Local & System	**Local**
Precision	Excellent	Excellent	**Average**
Performance	Average	Excellent	**Excellent**
Infrastructure Alignment	Achieves	Achieves	**Achieves**
Cost of Ownership	High	Medium	**Medium**
Proven Solution	Yes	Yes	**Yes**
Best Overall Solution		**Winner**	

Klocwork distinguished itself as being the most complete solution at the time, offering excellent precision and completeness while providing immediate feedback for R&D as a desktop engine. Developers exposed to the evaluation were uniformly positive on Klocwork.

Summary Statement
Static Analysis is a part of the verification testing flow. There are many tools out there which can do a reasonable job. However, the tool with the highest positive impact to reducing defects is the one that R&D folds into their daily coding process to catch errors before they are submitted to CM.

Vendor Links:
– www.klocwork.com

– www.coverity.com

– www.programmingresearch.com

References
[1] Giesen, D. Philosophy and Practical Implementation of Static Analyzer Tools, QA Systems Technologies BV, (1998), pp. 4.

[2] Emanuelsson, P. and Nilsson, U, A Comparative Study of Industrial Static Analysis Tools Analysis, Electronic Notes in Theoretical Computer Science 217

Carlos Almeida
SPK and Associates
Architect, Software Engineering