Software Quality: Static Code Analysis and dynamic Tests – Complementary Procedures for Quality Assurance

By Klaus Lambertz (Verifysoft Technology GmbH)

To ensure good software quality, during software development two complementary procedures are used: Static Code Analysis and Dynamic Tests associated with Code Coverage measurements. This paper shows the advantages and limitations of both methods and explains why Static and Dynamic Tests are complementary and both necessary to guarantee high quality software.

Static Analysis is a verification of the internal structure of an application. It is not an evaluation of the functionality of a software.
During Static Analysis the code is accessed without executing it. Such analysis can be done manually as part of walkthroughs or inspections. Although useful, the quality of such manual assessments depends on the employees and their wakefulness during the time of the inspection.
For this reason, walkthroughs and inspections should be completed by automatic checks with Static Code Analysis tools.
With Advanced Static Analysis Tools, it is possible to get cross-module analysis and analysis for large code bases with millions of lines of code.
Roughly speaking static analysis is about checking for errors and compliance with programming guidelines. For this there is no need for test cases – the analysis is done automatically by the tool.

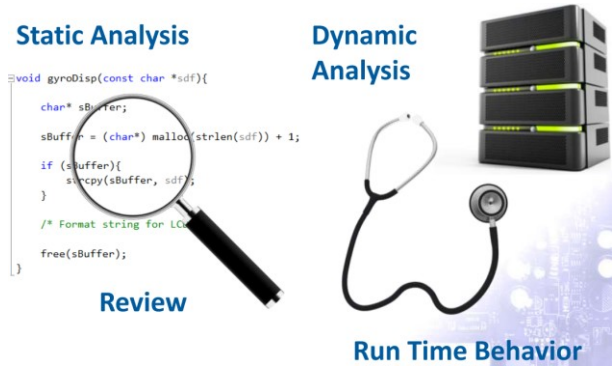

Fig. 1: Static Analysis and dynamic Tests

Dynamic tests on the other hand adopt the opposite approach. They are executed while a program is in operation. During run time is checked if the application works as expected. Test cases are used to verify whether the software works according to the requirements. This means that it is determined if the software does what it should do.

For the cost of software testing, it is interesting to look at Barry Boehm's research. Although the exact figures vary depending on the product and the project, the cost of eliminating errors increases always exponentially over the duration of the software development process. When a problem is detected in an early development phase it can be fixed at a relatively low cost. If a bug is detected during the implementation, it can be fixed even by the developer himself.
In the worst case a problem is detected after delivery of the software. This can result in expensive recalls, production stoppages, financial losses and personal injury.

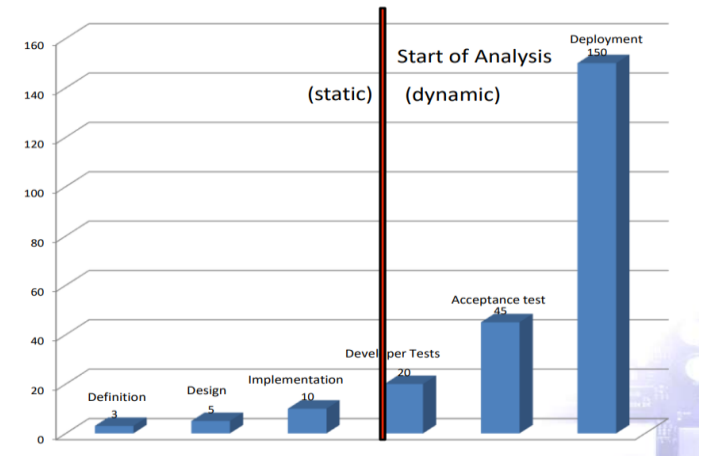

Fig. 2: Cost of error handling (according to Barry Boehm)

For the testing, this means: the earlier the tests are done, the earlier errors are detected. They can then be fixed earlier when the correction is still relatively inexpensive.
As for dynamic testing at runtime, the software needs to be executed, those tests can only be done after implementation.
Static analysis on the other hand can be done earlier during the implementation and already with small pieces of code.
With Static Code Analysis errors can be detected before dynamic tests are performed. This is known under the term “shift left”, a practice of moving testing, quality, and performance evaluation early in the development process.

Once a Static Code Analysis tool is set up, it works “by itself” and without efforts. It is a good approach to have the tool available for every developer, so that the code is only checked in after it has been analyzed.

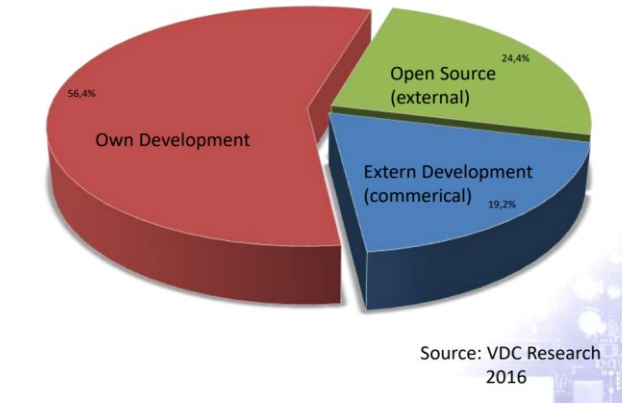

Fig. 3: Own and externally developed code

Nowadays only a part of the software is written within the company. According to a study of VDC Research in 2016, only 56% of the code is developed in-house. The rest is open-source and externally developed third party software for which usually only the binaries are available and not the source code. In this context, it is good to know that good Static Analysis tools can analyze not only source code but also binary code (even if the quality of binary analysis does not always match that of source code analysis).

Today, static analysis tools check for the following:

- Syntax
  This is done in a far better quality and depth compared to the syntax checks of compilers.

- Semantic
  For example, the sentence “It’s colder at night than outside” is syntactically correct, but the semantic is not, as this sentence is nonsense. However, Static Code Analysis check the semantic only to a limited extend, as the tool must understand what is meant in the code.

- Control Flow Problems
  Such as unreachable code, jumps into or out of loops.

- Data Flow Problems
  Such as uninitialized variables.

- Concurrency Problems
  Such as race conditions (for example data races), synchronizations which were forgotten or done in an incorrect manner, and dead locks.
  Concurrency problems are a key discipline of static analysis – often such problems cannot be detected during dynamic analysis at runtime.

- Programming Rules and Coding Standards
  The most known are MISRA rules, but there are a lot more. A lot of safety standards require those checks. Many tools offer also the option of implementing own rules.

- Maintainability / Metrics
  Such as Halstead and McCabe metrics. There are a lot of other metrics, even metrics like “Estimated number of bugs in a software” and “Time to understand a code”.
  Even though code does not necessarily have to be bad if certain metric thresholds are exceeded, it is a fact that there is a correlation between code complexity and error-proneness, testability and maintainability.

- Architecture
  How is the structure of the software? This is of particular interest for legacy code that no one is familiar with.

- Security
  Due to the connectivity of modern software, checks for security vulnerabilities are becoming more and more important. Static Code Analysis detects whether the code contains structures that enable attacks from outside.

Static Code Analysis is mandatory for safety critical Software

Standards for the development of safety critical software like ISO 26262 in the automotive sector require Static Code Analysis. Control flow analysis and data flow analysis are “highly recommended” for ASIL C and higher. Static analysis is mandatory for ASIL B and higher and recommended for the low level ASIL A. The recommendations of other standards are similar.

Methoden	ASIL			
	A	B	C	D
Walk-through	++	+	o	o
Inspection	+	++	++	++
Semi-formal verification	+	+	++	++
Formal verification	o	o	+	+
Control flow analysis	+	+	++	++
Data flow analysis	+	+	++	++
Static code analysis	+	++	++	++
Semantic code analysis	+	+	+	+

\+ recommended ++ highly recommended

Fig. 4: Static Analysis Methods according to ISO 26262

Static Code Analysis Tools provide many Checks

For the functional safety Static Code Analysis tools have many checkers, among them for memory overflows, null pointer dereferencing, division by zero, memory release of non-heap variables, memory leaks, missing return statements, and many more.
Checks for security include buffer overrun, unterminated C string, addition-, multiplication- and subtraction-overflows of allocation size, hardcoded DNS name, plaintext storage of password, use of system(), and many more.

To summarize, here are the main advantages of static code analysis:

- Can be performed early in the development process
- Provides a lot of different checkers
- The analysis is done automatically and usually without any effort

Code Coverage is mandatory to make sure that the complete Code is Tested

Complementary to the Static Code Analysis are the tests at runtime: the dynamic tests.
During dynamic testing the application is fed with test cases starting at the unit test level. The results of the tests are compared with the expected results, and thus it is checked if the application works as it is expected to work. For this the quality of the tests is important. The tests should be based on the requirements. Especially limit values and boundary conditions should be tested.
It is also important to measure the code coverage, to see, if the entire application has been tested.

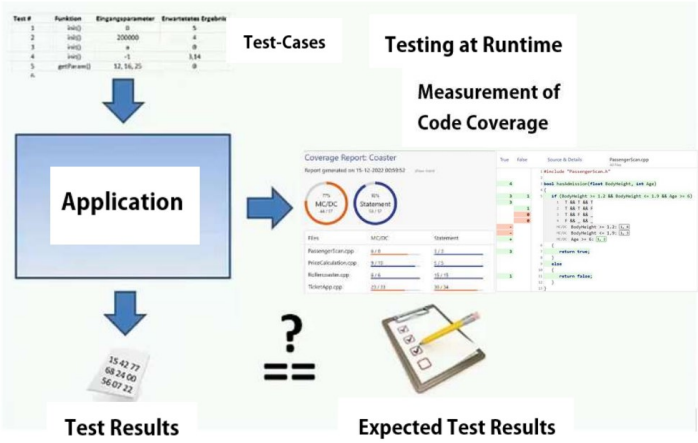

Fig. 5: Dynamic Testing at Runtime

Code Coverage shows, if the software has been tested completely. If there remain untested parts in the code, this code parts may contain dangerous errors which are hidden in the application – and this should be avoided.
However, the code coverage don't give information about the quality of the tests. Just doing "any" tests to get the coverage ratio increased, is not a good approach.

The question is always what "complete tests", or "100% coverage" means.
This depends on the chosen or required coverage level. As a starting point, one could assume that every statement has been tested once. This is Statement Coverage, a relatively weak coverage level.

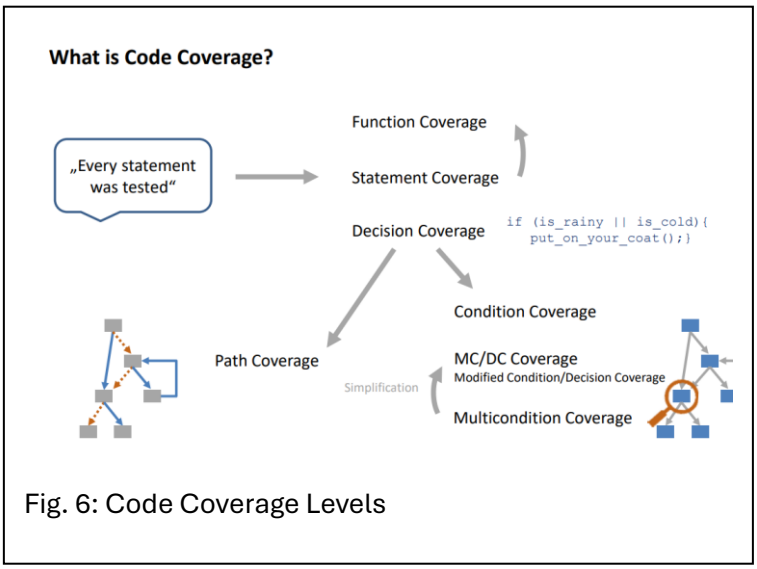

Fig. 6: Code Coverage Levels

Function Coverage is even weaker. Decision Coverage and Condition Coverage are already stronger, and Multicondition Coverage is very strict. Here each possible combinations of all atomic conditions have to be evaluated as “true” and as “false”. For a software with a lot of atomic conditions, this will lead to a high number of test cases. For this reason, there is a simplification of the Multicondition Coverage: the Modified Condition/Decision Coverage (MC/DC Coverage), which produces the same test results with fewer test cases. This MC/DC Coverage is the highest level requested by the safety standards.

All safety standards have in common: the higher the safety level, the higher the required coverage level. The ISO 26262 requires for ASIL D, the highest Automotive Safety Level, MC/DC-Coverage. The same applies to the standards for the other sectors, for example in aviation and railway industries as well as for the “general” standard IEC 61508. The highest required level is always MC/DC-Coverage.

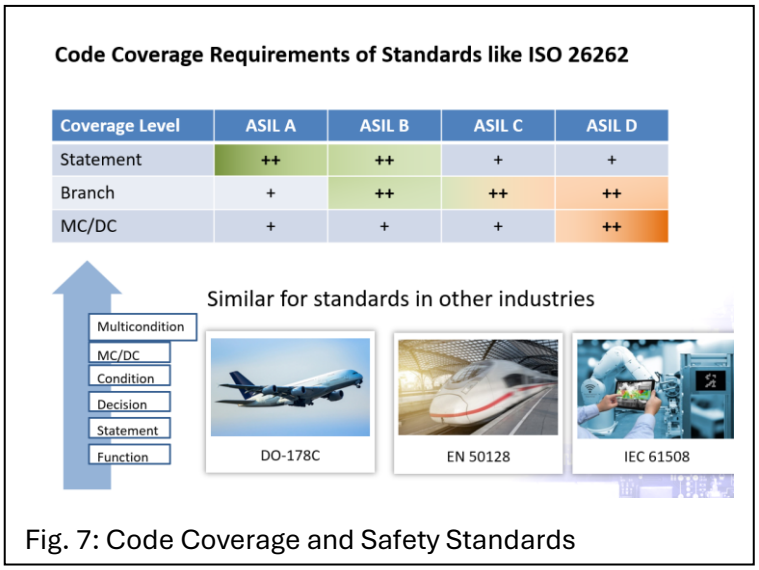

Fig. 7: Code Coverage and Safety Standards

Fig. 8: Static Analysis and dynamic Tests

Static Code Analysis and Dynamic Tests associated with Code Coverage

The following can be summarized:
The main application of Static Code Analysis is the detection of errors that lead to undefined behavior. A particular strength is the detection of concurrency problems.
When we talk about Static Code Analysis, we are

primarily talking about non-deterministic errors. In other words, errors that do not always occur. This is the reason why they are difficult to detect with tests during runtime. Sometimes those non-deterministic errors appear only after years, sometimes never, but which can occur at any time.
Although Static Code Analysis reveals serious problems in the source code, it cannot provide any information about the correct functionality of the software. This can be done by the dynamic tests as soon as a program (or a part of it) can be executed.
Static and dynamic analysis are both essential and must be used in a complementary way. It is not either or - it is both!

Selection Criteria for Tools

In the following I will describe some selection criteria for choosing tools. First, as in the development process, let's have a look at Static Code Analysis tools. We have seen that those tools have great advantages: deployment early in the development process, checking for programming guidelines, and detecting errors without writing test cases.

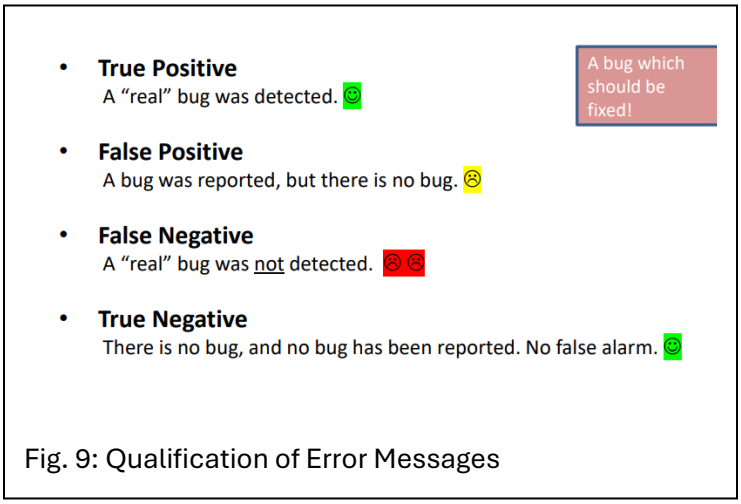

Fig. 9: Qualification of Error Messages

There is, however one thing, that makes uncovering errors a little bit difficult: Static Code Analysis does not always detect errors 100% accurately.
When the tools reports "True Positives" and "True Negatives" all is fine. True Positives are errors which are shown by the tool, and which are real errors which you should fix. True Negatives means, that the tool doesn't report an error and there is also no error in the code – this is fine as well.

Things are more difficult with "False Negatives" and "False Positives".
False Negatives are real errors which are not found by the tool.
False Positives are "errors" shown by the tool, but which are no errors: false alerts.
You will of course fix the True Positives, but you will also have a look to the False Positives, because you don't know that they are False Positives. That means you must review all error messages, regardless of whether they are actual faults or false alerts.
The problem of False Positives is that they take your time, and there is a risk that after five False Positives, you might be no more be attentive for the sixth, which might be a real error…
It is therefore important to have a Static Code Analysis tool which finds as many as real errors as possible (True Positives), which misses no (or very few) real errors (False Negatives) and which shows no (or very few) False Positives.

When choosing a tool, it is also important to evaluate which checks, programming rules and metrics are supported. The possibility to implement own rules is an advantage. It is important, that certain checks and rules can be deactivated in order to not get all error messages with the first run of the tool. Thus, the tester has the possibility to focus

on the most important errors first. For the same reason the possibility of excluding parts of the project during the analysis is advantageous.

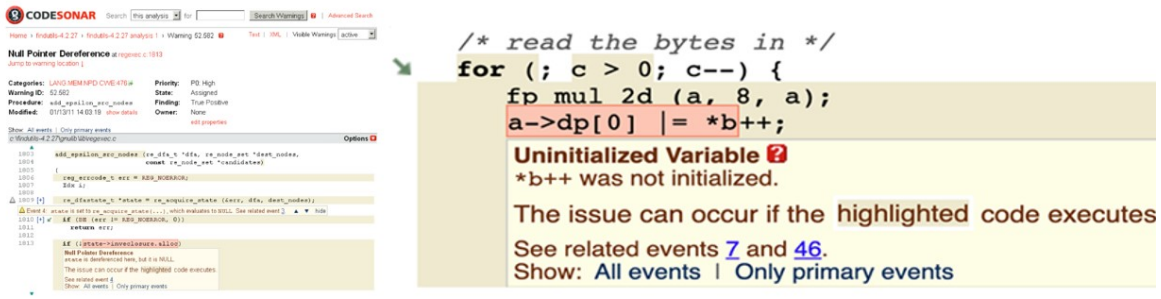

Fig. 10: Error Message of a Static Analysis Tool (CodeSonar by CodeSecure)

A good presentation of the error messages is important for the understanding. Thus, it helps to save time. There should be the possibility to classify the messages and there should be the necessary help to qualify the warnings. It is important that the messages can be assigned to individual team members for fixing. There should also be rules for accessing the results.

Especially for larger code bases, the performance of the Static Code Analysis tool is important. When choosing a tool, you should check if larger code bases with several million lines of code can be analyzed within a reasonable time frame. To achieve this goal, it will be helpful if the analysis can be distributed across several computers. An incremental analysis should be possible. The required hardware and memory space should be evaluated.

Concerning the selection of a Code Coverage analyzer it is evidence, that the tool must support the required code coverage levels. For a ASIL D project in the automotive industry, MC/DC coverage is mandatory. Even if you now have a lower safety level, you should consider a tool which can also analyze higher coverage levels, as you may later have projects with stricter requirements. Forward-looking decisions when acquiring a tool avoids that you need to buy a tool twice: a cheap one first and a good one later.

The tool must of course work with the compiler you use. But also, here – for the reasons mentioned above – it makes sense to choose a tool which works with several compilers or better with all compilers.

The Coverage Analyzer should be able to work with your embedded targets. For this reason, it is important that the tool has a low instrumentation overhead. The instrumentation is the adding of counters to your code (or a copy of your code) in order to measure the coverage.

As for the Static Code Analysis it is also important for the code coverage tool what the reports look like. They should be clear and easy to understand, showing both an overview of the coverage in your project as well as the detail of the coverage in your source code.

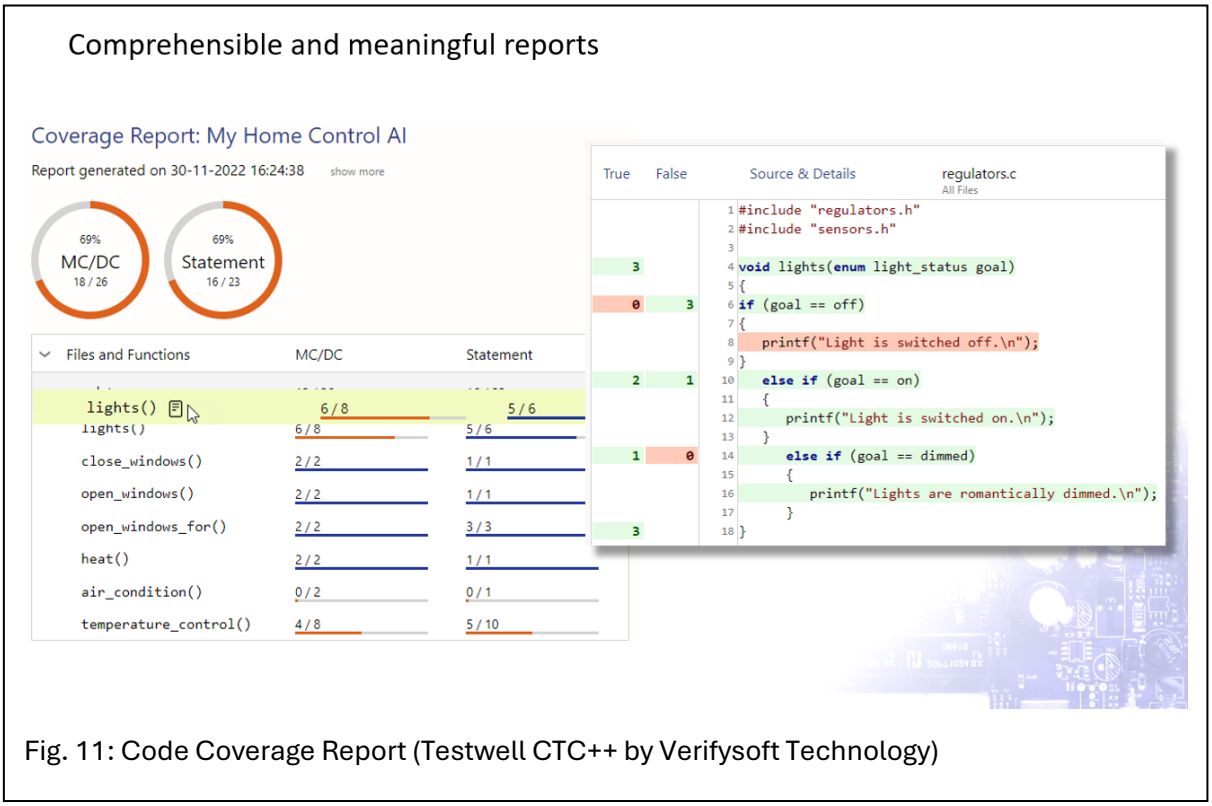

Fig. 11: Code Coverage Report (Testwell CTC++ by Verifysoft Technology)

Sometimes it is not possible to reach all code parts with tests – for example when you need to apply a defensive programming style. For this it is helpful when the tool offers the possibility to add “justifications” to explain why certain parts of the code have not been covered by tests.

When using a code coverage tool for safety-critical projects, you need to prove that the used tool is qualified. When the tool has a certificate (for example a TÜV certificate), the tool can be used for numerous safety standards without further qualification measures. For DO178-C projects in aeronautics it is a bit different. The tool needs to be qualified, but also here the tool certificate will be helpful as a starting point.

This qualification/certification is also needed for Static Code Analysis tools.
The following selection criteria concern both static and dynamic analysis tools:

Always important is the cost and the licensing model for a testing and analysis software. When speaking about cost, this in not only the tool price. It is also about cost of setting up the tool, teach the users, and the time you need to understand the reports. Here a “cheep” tool can become expensive… A “cheap” tool, which causes a lot of problems is ultimately not cheap…
It is also important to check what is the scope of use of a tool: one seat, the whole project, a location, or even would-wide usage?

When choosing a tool, you should also have a look to the quality of the support. During a tool evaluation you will already get an idea of the response time and the cooperation between user and tool provider.

In summary we have seen that Static Code Analysis and dynamic tests associated with code coverage are necessary and need to be used in a complementary way. There are very good tools on the market which support you with these methods. Before acquiring licenses, the tools should be evaluated to check if they meet the requirements of your current project and for projects you might have in future.

About the Author

Klaus Lambertz is founder and CEO of Verifysoft Technology GmbH www.verifysoft.com , a company that provides software testing and analysis tools. Verifysoft Technology has been founded 2003 in Offenburg (Germany) and has currently more than 750 satisfied customers in over 40 countries. In addition to the code coverage analyzer Testwell CTC++, the company provides complementary tools for Static Code Analysis. Verifysoft also offers seminars about software testing topics.
Parts of this paper are based on publications of Royd Lüdtke (Director Static Code Analysis, Verifysoft) and Dr. Sabine Poehler (Product Manager Testwell CTC++, Verifysoft).