

Static Code Analysis (SCA) Standardization Efforts & Integration in the Software Development Flow

Abstract

The complexity and reliability demands imposed on embedded systems have increased rapidly in recent years. Traditionally embedded software systems served one specialized task. Nowadays embedded software is often integrated in larger distributed systems that operate in a networked environment, potentially interacting via wireless sensor networks. This makes the software more vulnerable to programming flaws as well as threats due to illegal inputs received via the network.

This article is about “Static Code Analysis”. Static code analysis offers a means to improve the overall quality of embedded software. This article describes the ongoing standardization efforts carried out by government agencies and industry consortia and shows how these standards can be applied in the software development process. These quality improvements have technical and economic aspects and the relationship between these aspects are illustrated below.

What is Static Code Analysis & why is it needed?

Managing software complexity has become the main key to meet time to market, security and reliability demands. Traditional methods such as peer reviews and dynamic debug and test methods are valuable means to identify potential bugs but these fail to verify all possible paths and corner cases within a large code base. In addition the development costs of test cases that cover a significant part of all paths through the code base are high. Typically the costs of verification, testing and debugging exceed the initial development cost of the code base.

Static code analysis is a method to verify all possible paths within a software program without actually executing the program. A static code analysis tool can efficiently locate defects such as out of bound array accesses, memory allocation errors, arithmetic over and under flows, and inconsistent code fragments that go unnoticed during dynamic tests or peer reviews. Static code analysis can be applied early in the software development process, it can be applied on incomplete and incorrect code bases and when no test-cases have to be developed.

The basic reasons for applying static code analysis are two fold. The first is to reduce the time and costs of developing source code. The second is to increase revenue and reduce business risk by providing reliable software to customers.

Standardization Efforts

National governments as well as industry consortia have recognized the importance of software security, safety and reliability and have driven research and development of

standards that enforce secure, safe and reliable coding practices. The “CERT C/C++ Secure Coding Standard” and the “MISRA-C/C++ Guidelines for the use of the C/C++ language in critical systems” are today’s most well known and thorough standards that deal with the use of the C/C++ programming language in networked embedded environments.

CERT, the Computer Emergency Readiness Team was founded by the US government to address internet security risks and potential threats. CERT researches internet security vulnerability, identifies common programming errors that lead to software vulnerabilities, establishes standard secure coding standards, and educates software developers to advance the state of the practice in secure coding. CERT has observed, through an analysis of thousands of reports, that most vulnerabilities stem from a relatively small number of common programming errors. By identifying insecure coding practices and developing secure alternatives, CERT provides software developers with practical steps to reduce or eliminate vulnerabilities before deployment. These guidelines are collected into “The CERT C/C++ Secure Coding Standards”.

The MISRA-C Guidelines are the culmination of research into the worldwide automotive industry and other related sectors, such as the aerospace and process industries. The guidelines are developed by The Motor Industry Software Reliability Association (MISRA) which is a consortium of organizations formed in response to the UK Safety Critical Systems Research Programme. MISRA has recently completed work on the production of a set of guidelines for the use of C++ in critical systems, the output of which will be a set of guidelines similar to those that were produced for "C".

The quality and usefulness of these open standards is marked by the fact that many embedded compiler and static analysis tool vendors have adopted the above standards and upgraded their tool accordingly.

Software Development Process & Tools Support

There are many static analysis tools available on the market ranging from the simple lint tools to more sophisticated packages developed to prevent space craft launch failures. These SCA tools are either integrated within a compiler or are implemented as a dedicated tool with its own graphical user interface.

The main challenges for a static code analysis tool are:

- To find all violations, i.e. avoid “false negatives”.
- To analyze a large code base within an acceptable time frame.
- To provide real practical value to the engineer.
- To avoid “false positives”, i.e. avoid false alarms.

These challenges affect one another. Tying to find “all” violations typically results in notorious long execution times and excessive amounts of false alarms which reduce the practical value of the analysis. It is the task of the SCA tool developer to find a good balance between these challenges.

For embedded software development, the SCA tool needs to be aware about specific issues such as: the existence of special function registers, the use of in-line assembly language, C-language extensions such as pointer and memory space qualifiers to address multiple address spaces, and DSP specific data types such as circular buffers, and fixed point data types. General purpose SCA tools do not support these specific features to the same extent as SCA tools that are integrated within the embedded compiler.

Various embedded compiler vendors have integrated SCA with their compiler and they have many good reasons to do so. SCA algorithms are based on abstract syntax trees, structural analysis, data flow analysis, inter procedural control flow analysis and heuristics. These techniques fit well within a compiler front-end, can be executed fast, can be made aware of compiler specific embedded C language extensions, and can take the compiler’s ISO-C implementation defined behaviors into account. As a beneficial side effect the gathered information about a program is also available to the compiler’s optimizers, enabling further reductions in code size or providing opportunities to increase run-time performance.

Furthermore the output of the SCA can be intermixed within the compilers error and warning messages. This tight integration of SCA in the compiler eases the inclusion of SCA in an existing software development flow. SCA can be applied during the normal edit-compile-debug cycle. No modifications to tools, build scripts and procedures, work flows, or organizational structure are required. This prevents the “hidden cost” that occur when dedicated stand-alone SCA tools applied in an organization.

To many false positives i.e. false alarms rapidly lead users to ignore the output produced by the SCA tool, potentially missing serious issues. This is particular relevant when analyzing source code written in C/C++. The use of C/C++ “pointer types” complicates the analysis to such extent that it is often not possible to guarantee whether a software fragment will, or could potentially cause, a system failure. To mitigate the problem of false positive, it is necessary to clearly differentiate between a “guaranteed problem” and a “potential problem”.

SCA by Example

Consider the following code fragment:

```
void f(void)
{
    char buf[10];
    for (int i = 0; i <= 10; i++)
    {
```

```
    if (some_condition(i))
    {
        buf[i] = 0; /* subscript may be out of bounds */
    }
}
```

In this case the array “buf” may be accessed beyond its upper boundary. This could happen when the loop iteration counter “i” equals “10”. However if function “some_condition(10)” returns ‘0’ then all accesses to array “buf” are within legal range.

How should a SCA tool deal with such a case? Should it report a “guaranteed problem”, a “potential problem” or should it report “guaranteed no problem at all”?

If the file that contains the implementation of “some_function()” is passed to the SCA tool and the SCA tool can statically compute the return value for input parameter value “10” then the SCA tool can either guaranteed whether an illegal access will take place or not. In other cases, e.g. if “some_condition()” has not been implemented yet a “potential failure” will be issued. Today’s high-end SCA tools provide such application wide global analysis features.

Static Analysis Complements other Development Tools

Analyzing the “potential problems” reported by a SCA tool can take a significant amount of time. Dynamic, run-time error checking tools provide a cost-efficient means to analyze whether potential problems appear to be a real threat.

In opposition to SCA, the run-time checks never report “false positives”. A possible disadvantage is that a user has to develop a set of test-cases. This adds time and costs to the development process because the application must be in state where it is executable, i.e. all functions must be implemented and compile-able.

Run-time checks can be performed manually by inspecting the source code in a debugger. This inspection process can also be automated. Many embedded compilers can annotate the generated code with “run-time error check stubs”. These stubs typically validate all pointer accesses to uncover out-of-bounds accesses, null pointer dereferences, uninitialized automatic pointer variables, and illegal use of dynamic memory. Warning messages are issued whenever a stub triggers, typically the message identifies the type of error and indicates the line in the source that caused the error.

Economics of SCA

Application of SCA requires investments in tools, in education, and integration of the tools used in existing software development procedures. If the SCA tool is integrated in the compiler, then the education and integration costs are minimal.

The benefits of using SCA run-time checks are reductions in time and development costs, increased revenue and reduced business risk as result of providing reliable software to customers.

SCA enables engineers to detect and fix problems more efficiently and earlier in the development cycle, which reduces time in development and maintenance as well as reduce costs. This is a fact that has been proven in many studies.

As a consequence of a shortened development cycle, products can be brought to market earlier and stay on the market longer. This translates to higher sales and profits. Increased product quality reduces post sales cost and increased market position and reputation.

TASKING's Embedded software development tools support static code analysis techniques to validate code bases against aforementioned CERT and MISRA standards. Complementary tool such as automated run-time error checking are also available.

Relevant URLs:

<http://www.cert.org/>

<https://www.securecoding.cert.org/confluence/display/seccode/CERT+C+Secure+Coding+Standard>

<http://www.misra-c.com/>

<http://www.tasking.com/>

Author:

Gerard Vink, Associate Director, Core Embedded Technology
Altium BV, Altium Technology Centre The Netherlands