CAPABILITY OF TODAY'S PROGRAM VERIFICATION: A PRACTICAL APPROACH FOR BETTER QUALITY AND RELIABILITY IN INDUSTRIAL APPLICATIONS

Michael Bogner [(a)], Johannes Schiller [(b)], Franz Wiesinger [(c)]

[(a, b, c)]University of Applied Sciences Upper Austria – Embedded Systems Design,
Softwarepark 11, A-4232 Hagenberg, AUSTRIA

[(a)] michael.bogner@fh-hagenberg.at, [(b)] johannes.schiller@fh-hagenberg.at, [(c)] franz.wiesinger@fh-hagenberg.at

ABSTRACT
Software programs are an essential part of our everyday's life. Starting with large software programs on the PC, via complex control systems for the industrial area, to safety-critical software solutions for the automotive and aerospace industry; software is almost everywhere. Especially nowadays a high degree of reliability and security is essential. But due to the constantly growing size and complexity of such software programs the verification effort is increasing too. For this reasons, beneath dynamic testing and manual reviews, automatic verification methods became more and more popular. This paper deals with the expected benefits and the effectiveness of static code analysis and especially shows the limitations of this technique. Empirical tests have been developed and various code analysis tools employed. The paper discusses the obtained results. It becomes apparent that current code analysis tools can already find a variety of potential errors and weaknesses while critical cases are still undetected.

Keywords: static code analysis, software, testing, verification, Goanna Studio, PC-lint, Yasca, C++

1. INTRODUCTION

Current software programs become more complex from year to year. The test and verification effort for these software programs is constantly increasing and it becomes increasingly intricate to maintain these systems properly. Therefore, automatic static code verification became more and more popular in the last few years. Beneath dynamic testing methods and manual code reviews, static code analysis is another instrument to ensure the safety and reliability of future software programs. Even before the actual execution of the software program, the program code is checked against weaknesses and errors by a strict set of rules.

Therefore static code analysis can already be used in early stages of development to detect critical errors in software programs and eliminate them. This paper provides a brief introduction to the topic of static code analysis and shows the current state of the art respectively the power of technology in this area.

Static code analysis is a method for quality assurance of software programs. The underlying program code is statically checked for weaknesses and errors. At automatic static code analysis the analysis of the program code is performed using special software programs. To analyze the program code as effective as possible, these code analysis tools use a large number of different kinds of analysis methods and software metrics. The respective program code must neither be complete nor executable (Hoffmann 2008, Liggesmeyer 2009).

Static code analysis can be used everywhere where software is developed. Especially for large software projects or for safety critical applications the use of static code analysis is recommended. Some programming conventions, like MISRA-C++ (MISRA 2008) or the UK Defence Standard 00-55 (German 2003), explicit stipulate the use of static code analysis for safety-critical software. At the same time static code analysis cannot replace ordinary, dynamic testing methods.

The aim of this study was to determine the practical benefits of static code analysis and the current state of the art in this area. Therefore it was attempted to use real life examples and analysis tools which are preferably different.

2. RELATED WORK

Static code analysis has become increasingly important in the last years. For this reason there are many works which deal with the investigation of static code analysis (Muchnick 1981, Hoffmann 2008, Liggesmeyer 2009) and other with the evaluation and comparison of static analysis tools (Emanuelsson 2008, Hofer 2010, Almossawi 2006). Many works in this field are more specialized in the theoretical operating principles (Fehnker 2007, Miller 2007) or concentrate on a very specific area of computer programming (Cong 2009, Cooper 2002).

3. VERIFICATION METHODOLOGIES

In the context of software development, verification means to ensure that a given program code or algorithm meets its formal specification. In the case of static code analysis, the instruction set and the syntax of the programming language form the formal specification.

Static code analysis tools use a variety of different methods to verify the correctness of the respective program code. For example dataflow- and controlflow-analyses are used. Some tools, like Goanna Studio transform the respective program code into a finite state machine and use model checking techniques for the

evaluation. Beneath these hard criteria the tools often use software metrics to determine the quality of the respective program code. Therefore the program code will be converted into different quantifiable values. Among the most popular metrics are Halstead-, McCabe-, component- and structural metrics (Hoffmann 2008, Liggesmeyer 2009, Fehnker 2006, Fehnker 2007).

Static code analysis can not give any information about the functional correctness of a program code. Even if static code analysis couldn't find any errors or flaws in the program, there is no guarantee that the examined program delivers the correct result.

4. USED SOFTWARE SOLUTIONS

To determine the current state of the art of static code analysis, three different code analysis tools have been evaluated. To get a broad overview of the subject it was attempted to choose software solutions that are as different as possible.

The choice fell on the following software solutions: Goanna Studio, PC-lint and Yasca.

4.1. Goanna Studio

Goanna Studio is a C++ code analysis tool by the company Red Lizard Software. It follows the approach to use model checking for searching the program code quickly and effectively. Model checking is an automated process to analyze transition systems. Therefore, the C++ code needs to be converted into a context-free grammar. After that it will be analyzed through the model checker. Especially in large projects this can lead to performance advantages. Another advantage of this method is the simple and fast expandability of the analyzing-rules (Huuk et al. 2008, Fehnker et al. 2007, Red Lizard Software 2012)

Goanna Studio was used in the version 2.4.1 (trial version).

4.2. PC-lint

PC-lint is a C / C++ code analysis tool by the company Gimpel Software. It is one of the well-established code analysis tools on the market. It is a pure console application. This has affects on the clarity and usability of the tool. However there are some plug-ins available which can add GUI elements to PC-lint. This is advisable especially for large projects. One of the biggest advantages of PC-lint is that it supports a variety of different Compilers and programming conventions (Gimpel Software 2012).

PC-lint was used in the version 9.00.

4.3. Yasca

In addition to these two commercial solutions the open source solution Yasca was added to the evaluation. It is under the GNU General Public License and may therefore be used free of charge. Yasca is a simple code analysis tool which offers far away as much setting opportunities as the other two solutions. Thereby it combines several different code analysis tools like CppCheck, RATS, etc. in it. These analysis tools can be added to Yasca via plug-ins. It is also possible to add your own rules (Scovetta 2012, Cppcheck 2011, Fortify Software 2012).

Yasca was used in the version 2.21.

5. THE TEST CODE

The selected software solutions were evaluated using an extensive test code. This test code covers the major areas of the programming language and includes a broad range of errors typically found in industrial applications. By combining on the one hand, frequently occurring errors with on the other hand more complex errors, the suitability for daily use of static code analysis should be covered and the limits of the current technology demonstrated.

5.1. The programming language

C++ was selected as programming language for the test code. C++ is a very well known and widely used programming language. Beneath C and Assembler, C++ plays more and more a role in safety-critical areas, such as Embedded Systems. Especially in these areas a high degree of safety and reliability is essential. Through the use of so-called programming conventions, such as MISRA-C++, the programming language can gain additional security. However the programming language is therefore somewhat limited in their functionality. PC-Lint is the only one of the three software solution that supports the programming convention MISRA-C++.

5.2. Structure of the test code

The test code was structured in several projects. Each of them concentrates on a specific area of the programming language. To analyze the performance of the different code analysis tools, it was attempted to build as much errors as possible into the individual projects. A large part of these errors are typical errors from the daily practice. To verify the limitations of static code analysis these errors were supplemented by some more specific and complicated errors. Additionally in some cases deliberately messy code was used, to simulate real conditions and make it more difficult for the evaluated analysis tools.

The following sections of the programming language are covered by the test code:

- Bounds Checking
- Division by Zero
- Memory Leaks
- Over- / Underflows
- Out-of-Scope Errors
- Problems with Classes
- Problems with Threads: Deadlock
- Problems with Threads: Race Condition

5.3. Examples

Below are a few examples of errors which are included in the test code. The errors in the test code are on the one hand self-designed. On the other hand a large

number of errors came out of typical industrial situations or were inspired by relevant literature. Therefore, one should refer to the following titles: (Breymann 2005, Dewhurst 2002, Hoffmann 2008, Intel 2010, Klein 2003, Wolf 2009).

5.3.1. Example 1

Listing 1: Program code of Example 1.

```
1 struct Cont{
2       char name[3];
3       int number;
4 };
5
6 int main(){
7       Cont *c = new Cont;
8
9       strcpy(c->name, "Max");
10      c->number = 1234567;
11
12      cout << "name: " << c->name <<
        endl;
13      cout << "number: " << c->number <<
        endl;
14
15      strcat(c->name, "!");
16      cout << "name: " << c->name <<
        endl;
17      cout << "number: " << c->number <<
        endl;
18      delete c; c=0;
19      return 0;
20 }
```

Listing 1 shows a classical case of an out-of-bounds error. The code includes a struct *Cont* which contains a character array *name* and an integer value *number*. The character array *name* is limited to three digits.

Within the *main* statement the word "Max" will be copied into the character array *name*. Therefore the function *strcpy* is used. This function doesn't compare if the length of the committed string matches with the length of the target string. So in this case "Max" is a string. Therefore it ends with a terminating null (\0) and has consequently 4 digits. As a result, a text with 4 digits will be copied in a character array that can hold only 3 digits.

This error may remain undetected because of the memory alignment of the compiler. The memory is usually 4-byte aligned, which is the case on systems using natural alignment. Therefore between the character array *name* and the integer value *number* is a so-called padding byte. So instead of overwriting the integer value *number*, the padding byte will be overwritten. This critical side effect gets even worse if later on the alignment changes and so the number *value* gets suddenly modified.

For a better explanation of the problem screenshots from the Memory Window of Visual Studio have been added. Figure 1 shows the allocation of the string "Max". In Figure 2 the number 1234567 was added to the memory. Figure 3 shows the attachment of the exclamation mark behind the string "Max". The Memory Window shows that through this attachment the value of the integer variable *number* was changed too. Figure 4 shows a screenshot of the output of the program.

This function can be compiled and executed in Visual Studio 2008 without any errors. The analysis tools PC-lint and Yasca could find this error.

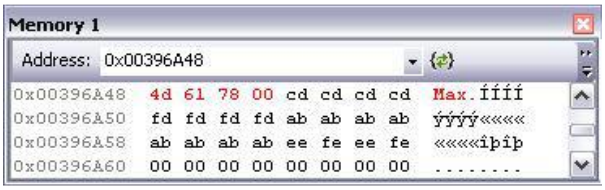

Figure 1: Screenshot of the Memory Window after the allocation of the string "Max".

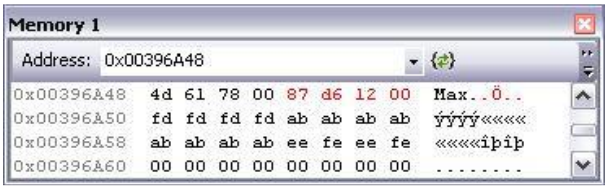

Figure 2: Screenshot of the Memory Window after the allocation of the number 1234567.

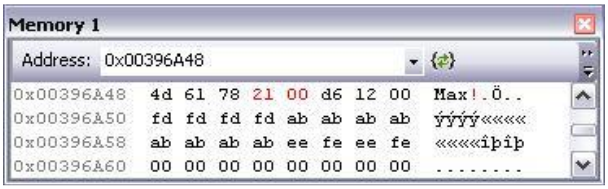

Figure 3: Screenshot of the Memory Window after adding an exclamation mark behind the string "Max".

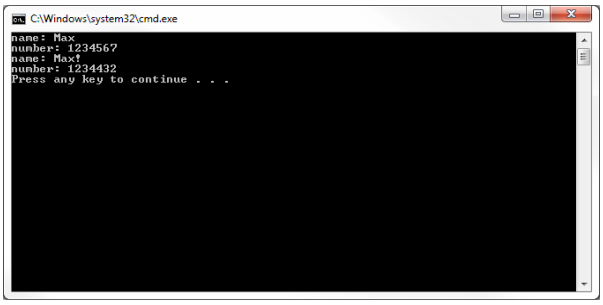

Figure 4: Screenshot of the output after the execution of the program.

5.3.2. Example 2

Listing 2: Program code of Example 2.

```
1 int main(){
2       int const n=10;
3       int *pa = new int(n);
4
5       for(int i=0; i<n; i++){
6             pa[i] = i;
7       }
8       delete [] pa; pa=0;
9       return 0;
10 }
```

In Listing 2 the programmer intended to create an integer array *pa* with the size of *n* (10). However a small error crept in. Instead of square brackets the programmer used round brackets. Therefore, instead of creating an array with the size of 10, an integer value with the initial value of 10 will be created. As a consequence, each access, except of the first one, results in a violation of the memory area representing a serious error. Furthermore the allocated memory for the integer value *pa* will be freed with the keyword *delete []* instead of *delete*, which normally will not lead to any serious problems. Anyhow there is no guarantee that it would cause unwanted side effects on some systems.

5.3.3. Example 3

Listing 3: Program code of Example 3.

```
1 class Number{
2 public:
3       Number(int val = 0):Num(val){}
4       ~Number(){}
5
6       int getNum(){
7             return Num;
8       }
9 private:
10      int Num;
11 };
12
13 int main(){
14      Number *n1 = new Number(5);
15      Number *n2 = new Number();
16      //…
17      n2 = n1;
18      //…
19      delete n1; n1=0;
20      //…
21      delete n2; n2=0;
22      return 0;
23 }
```

Listing 3 shows a good example how the default Copy Constructor respectively Assignment Operator can lead to problems. The class *Number* has a member variable *Num*. In the *main* statement two instances of the class *Number* were dynamically created, *n1* and *n2*. The programmer wants to assign the value of *n1* to *n2* and uses the Assignment Operator. However, instead of copying the value of *n1* to *n2*, the memory address will be copied. Therefore after this assignment both pointer point to the same memory address. In most of the cases this is not intended and can lead to unwanted behaviour. In this case the memory would be freed twice, which can lead to security flaws and crashes.

5.4. Test Criteria

For reasons of clarity and comprehensibility only faults with a security level of *error* or *warning* were considered in the evaluation. The code analysis tool Yasca finds on its own admission only errors which would not be found by a conventional compiler. The code analysis tool Goanna Studio is integrated in the IDE of the MS Visual Studio and shares therefore the same error-window with the compiler. For this reason, errors which were already found by the compiler are not included in the evaluation of Yasca and Goanna Studio. To have a comparable basis, MS Visual Studio 2008 (SP1) was used as compiler for all test cases. All test cases could be compiled without any errors.

6. RESULTS OF THE EVALUATION

This Chapter provides a compact overview of the various kinds of errors found by the individual code analysis tools. Therefore all found errors of a section are compared with the expected errors for the same section. In some cases additional errors were found by the analysing tools. This additional errors were also added to the evaluation and it was determined whether these errors refer to real problems (true positive) or not (false positive). No code analysis tool can find all faults without generating some false positives. An analysis tool can only be called “safe” if it really displays all found errors and warnings. Although this increases the number of false positives, but therefore the number of false negatives is as low as possible (Emanuelsson 2008).

6.1. Overview Goanna Studio

Table 1 shows an overview of the errors found by the code analysis tool Goanna Studio. Goanna Studio found 42 of 84 errors expected and therefore achieved a success rate of 50%. Furthermore it found six additional errors. These errors are indicators for unnecessary functions or assignments. Five of these additional errors are real errors (true positives) and one is a false report (false positive). This results in one false positive of 48 errors found (ca. 2%).

Table 1: Overview of errors found by Goanna Studio.

Results Goanna Studio				
Section	Expected errors	Errors found	Additional errors found	Errors not found
Bounds Checking	15	7	0	8
Division by Zero	4	4	1	0
Memory Leaks	16	7	1	9
Over- / Underflows	6	3	0	3
Out-of-Scope	6	5	1	1
Classes	29	14	0	15
Deadlock	6	2	3	4
Race Condition	2	0	0	2
Summary	84	42	6	42

6.2. Overview PC-lint

Table 2 shows an overview of the errors found by the code analysis tool PC-lint. PC-lint found 54 of 85 errors expected and therefore achieved a success rate of 63,6%. Furthermore it found 42 additional errors. These

are, however, 30 times the warning 586. This warning refers to an unauthorized function of the MISRA programming convention. This is mainly to functions which are used for dynamic memory management, as new, delete, etc. Admittedly these functions are in violation of the MISRA guidelines, but cause no direct errors. For this reason, these errors not considered for the calculation of the false positives. Therefore there are six correct errors (true positives) of a total of twelve additional errors found. This results in six false positives of 96 errors found (ca. 6,25%).

Table 2: Overview of errors found by PC-lint.

Results PC-lint				
Section	Expected errors	Errors found	Additional errors found	Errors not found
Bounds Checking	15	10	7	5
Division by Zero	4	2	0	2
Memory Leaks	16	13	15	3
Over- / Underflows	6	4	0	2
Out-of-Scope	6	6	0	0
Classes	30	14	16	16
Deadlock	6	3	1	3
Race Condition	2	2	3	0
Summary	85	54	42	31

6.3. Overview Yasca

Table 3 shows an overview of the errors found by the code analysis tool Yasca. Yasca found 18 of 78 errors expected and therefore achieved a success rate of 23,1%. Furthermore it found seven additional errors. These are, however, rather indications than real error messages. Therefore a further classification in false respectively true positives is unnecessary.

Table 3: Overview of errors found by Yasca.

Results PC-lint				
Section	Expected errors	Errors found	Additional errors found	Errors not found
Bounds Checking	13	5	1	8
Division by Zero	3	0	0	3
Memory Leaks	16	9	0	7
Over- / Underflows	5	1	1	4
Out-of-Scope	4	2	1	2
Classes	29	1	0	28
Deadlock	6	0	4	6
Race Condition	2	0	0	2
Summary	78	18	7	60

6.4. Summary

On average, the three evaluated code analysis tools found approximately 46% of all in the test code contained errors. It is striking that there are sometimes significant differences between the various code analysis tools. For example, the open source tool Yasca finds with about 23% by far the fewest errors. The two commercial software solutions find however at least 50% of all included errors. The evaluated code analysis tools differ not only in the number of detected errors, but also by the errors found themselves. Each of the evaluated code analysis tools found at least one error, which none of the other two solutions could find. Therefore it can be found about 75% of all in the test code included errors by sequential execution of all three analysis tools (Figure 5). In consequence it is advisable to use several different analysis tools for the analysis of safety-critical systems.

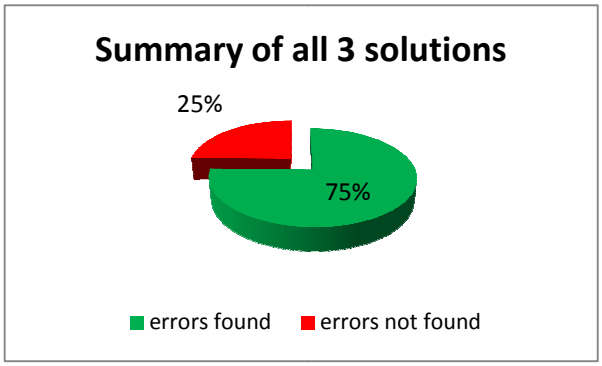

Figure 5: Summary of all 3solutions.

Furthermore, the evaluation has shown that for small projects, the false positive rate is already well below 10%. However for large projects this value can be in some cases significantly higher. This is largely because of the increased complexity and the resulting dependencies in large projects. For this reason the subsequent evaluation of large projects can be extremely time-consuming. In such projects it is therefore advisable to use static code analysis already at the beginning of the project.

7. CONCLUSION

The evaluation of the three software solutions has shown that static code analysis is already capable to find a variety of potential errors and weaknesses within software programs. It is noticed here that certain error classes can be found very well, while others can barely be detected or can't be detected at all. For example memory leaks and out-of-bounds errors can be recognized very reliable, while errors relating to threads remain in general unrecognized. Also the context in which an error occurs plays an important role, whether this error can be found by static code analysis or not.
The program code that needs to be analyzed doesn't have to be completed; neither does it to be executable. For this reason static code analysis can be used in software projects very early. Consequently, potential

errors and weaknesses can be identified and corrected as soon as possible. This can save time and money. At the same time the number of new errors can be reduced to a minimum and therefore the analysis can be kept manageable. This approach is advisable, especially for large projects.

Furthermore, it is advisable to invest enough time for the configuration of the code analysis tools and for the selection of the right programming conventions in such projects. The selection of the highest security level is in many software projects unnecessary and only leads to increased effort. The same applies to programming conventions such as MISRA-C++. In safety-critical software such conventions are essential, but in normal projects a subset of these rules is often more than sufficient.

Static code analysis can not replace dynamic testing. While dynamic testing methods are especially checking the correct function of the program code, static code analysis mainly ensures the correct and safe use of the respective programming language.

All in all, static code analysis is a simple and fast way to improve the quality of software programs without stealing too much of the developers time. Static code analysis can not replace traditional testing methods, but it provides a solid addition to these, which can already be used in very early stages of a software project. In addition static code analysis can find kinds of errors, which can't be found by other testing methods, like e.g. dead code.

Static code analysis offers a cheap and easy method to increase the security and reliability of software programs and should therefore be used in software development as common as conventional testing methods like dynamic testing or code reviews.

REFERENCES

The Motor Industry Software Reliability Association, 2008. *Guidelines for the Use of the C++ Language in Critical Systems*.

Emanuelsson, P. and Nilsson, U., 2008. *A Comperative Study of Industrial Static Analysis Tools.* Techn. Ber., Linköping University: Dep. of Computer and Information Science.

Hoffmann, D. W., 2008. *Software-Qualität*. Heidelberg: Springer-Verlag.

Liggesmeyer,P., 2009. *Software-Qualität: Testen, Analysieren und Verifizieren von Software.* Heidelberg: Spektrum-Verlag.

Breymann, U., 2005. *C++ - Einführung und professionelle Programmierung*. München: Carl Hanser Verlag.

Dewhurst, S., 2002. *C++ Gotchas: Avoiding Common Problems in Coding and Design.* Boston: Addision-Wesley.

Intel Corporation, 2010. *About Static Security Analysis Error Detection*. Available from: http://software.intel.com [accessed 10 April 2012].

Wolf, J., 2009. *C++ von A bis Z.* Bonn: Galileo Computing.

Klein, T., 2003. *Buffer Overflows und Format-String-Schwachstellen.* Heidelberg: Dpunkt.Verlag.

German, A., 2003. *Software Static Code Analysis Lessons Learnt.* Techn. Ber., QinetiQ Ltd.

Fehnker et al., 2007. Model Checking Software at Compile Time. *Proceedings of the First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering.* pp. 45-56. Washington, D.C. (USA)

Huuk et al., 2008. Goanna: Syntactic Software Model Checking. *Proceedings of the 6th International Symposium on Automated Technology for Verification and Analysis.* pp. 216-221. Heidelberg: Springer-Verlag.

Red Lizard Software, 2012. *Goanna Studio.* Available from: http://redlizards.com/index.php [accessed 10 April 2012].

Gimpel Software, 2012 PC-lint. Available from: http://www.gimpel.com/html/index.htm [accessed 10 April 2012].

Scovetta, M. V., 2012. *Yasca*. Available from: http://www.scovetta.com/yasca.html.http://www.scovetta.com/yasca.html. [accessed 10 April 2012].

Cppcheck, 2012. *A tool for static C/C++ code analysis*. Available from: http://sourceforge.net/apps/mediawiki/cppcheck/index.php. [accessed 10 April 2012].

Fortify Software, 2012. *RATS - Rough Auditing Tool for Security*. Available from: https://www.fortify.com/ssa-elements/threat-intelligence/rats.html. [accessed 10 April 2012].

Fehnker et al., 2006. *Goanna - A Static Model Checker.* Available from: http://www.ssrg.nicta.com.au/publications/papers/Fehnker_HJLR_06.pdf [accessed 5 July 2012]

Hofer, T., 2010. *Evaluating Static Source Code Analysis Tools.* Available from: http://infoscience.epfl.ch/record/153107/files/ESSCAT-report.pdf [accessed 10 April 2012].

Almossawi et al., 2006. *Analysis Tool Evaluation: Coverity Prevent.* Carnegie Mellon University, Pittsburgh, PA, USA.

Muchnick, S and Jones, N., 1981. *Program Flow Analysis: Theory and Applications*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey.

Cong et al., 2009. *Evaluation of Static Analysis Techniques for Fixed-Point Precision Optimization. Proceedings of the 2009 17th IEEE Symposium on Field Programmable Custom Computing Machines.* Pp. 231-234. Washington, D.C. (USA).

Cooper, K. and Xu Li, 2002, *An Efficient Static Analysis Algorithm to Detect Redundant Memory Operations.* Dep. of Computer Science Rice University Houston, Texas, USA.

Miller, D., 2007, *A Proof-Theoretic Approach to the Static Analysis of Logic Programs.* Dep. of Computer Science Rice University Houston, Texas, USA.