Static Analysis tools, a practical approach for safety-critical software verification

Rui Lopes, Diogo Vicente, Nuno Silva
Critical Software SA
Parque Industrial de Taveiro, Lote 48,
3045-054 Coimbra, Portugal
[rmlopes, dpvicente, nsilva]@criticalsoftware.com

Abstract

Static code analysis tools available today range from Lint-based syntax parsers to standards' compliance checkers to tools using more formal methods for verification. As safety critical software complexity is increasing, these tools provide a mean to ensure code quality, safety and dependability attributes. They also provide a mean to introduce further automation in code analysis activities. The features presented by static code analysis tools are particularly interesting for V&V activities. In the scope of Independent Code Verification (IVE), two different static analysis tools have been used during Code Verification activities of the LISA Pathfinder onboard software in order to assess their contribution to the efficiency of the process and quality of the results. Polyspace (The MathWorks) and FlexeLint (Gimpel) tools have been used as examples of high-budget and low-budget tools respectively. Several aspects have been addressed: effort has been categorised for closer analysis (e.g. setup and configuration time, execution time, analysis of the results, etc), reported issues have been categorised according to their type and the coverage of traditional IVE tasks by the static code analysis tools has been evaluated. Final observations have been performed by analysing the previously referred subjects, namely regarding cost effectiveness, quality of results, complementarities between the results of different static code analysis tools and relation between automated code analysis and manual code inspection.

1 INTRODUCTION

Static code analysers are used to uncover hard to find implementation errors before run-time, since they may be even more difficult or impossible to find and assess during execution. These tools can discover many logical, safety and security errors in an application without the need to execute the application. Static analysis tools do not require the application to be compiled nor executed; problems are found by analysing the source code directly and statically.

In this paper we will discuss the usage of static code analysis tools for the assessment of safety critical on-board software applications. We will present some results and lessons learnt from the usage of some of these tools, reasons to use them in order to find and correct code issues, their strengths and limitations and complementarities, and in what situations should static analysis tools be used. Finally, and according to the results obtained by using the tools in a safety-critical application, we will present some results and conclusions about their usage and usefulness.

2 STATE OF THE ART

First, static analysis tools have been used in the form of early versions of lint or pattern matching static analysis tools to enforce coding styles, or to discover simple programming errors.

The first tools were difficult to use, tedious, and limited in their ability to find real bugs (they could only find a list of functions known to be dangerous).

Static analysis tools have then evolved to discover more programmatic and complex errors. Currently, such tools can parse almost all the languages for many common coding problems within different categories, including complex error patterns.

More complex techniques have been added to the static analysis tools such as code metrics analysis (lines of code, lines of code / lines of comments, cyclomatic complexity and others). These techniques help understanding the complexity of the code and can lead to code changes. For example, the cyclomatic complexity or the number of paths, are a precise measure of the code complexity, and the more complex the code is, more likely it will contain masked bugs.

When context analysis was added to the static analysis tools it became possible to find bugs that required interaction between multiple function calls.

Another technique, semantic analysis, enables the discovery of the basic structure and relation of the functions within the application. This additional contextual information helps the analyser understand and report bugs that require knowledge of specific code paths through the application. Some static analysers use abstract syntax trees to provide the best possible bug finding capabilities. This way, a static analysis tool can run detailed simulations of suspicious code fragments to better predict how the code will react.

Some static analysis tools allow marking the code with comments or another form of metadata to describe rules and inter-function dependencies. This information allows the analyser to understand under what conditions a bug may occur as well as expectations each function has for

parameters passed in and values returned. The use of metadata reduces the number of false positives and helps the analyser follow code paths in a more controlled manner.

Some static analysis tools can model the source code into complex mathematical representation in order to effectively prove the reliability of the application.

Some of the recent static analysis tools allow the creation of new rules or modification of the existing rules. This helps in customising the analysis for specific target applications, since the analyser will look for issues specific to the operating environment, application needs and complexity, and applicable coding standards.

3 PURPOSE AND RATIONALE

Static analysis is the best way to find coding issues. However, uncovering a particular defect depends on whether it is generic or context specific, and whether it is visible in the code or only in the architecture [1]. This is particularly true for Independent Software Verification and Validation (ISVV) tasks.

	CODE LEVEL	ARCHITECTURE LEVEL
Generic findings	Naturally found by static analysis tools. Built-in rules make it easy for tools to find these issues without programmer guidance. *Example: buffer Overflow.*	Most likely to be found through architectural analysis only. • *Example: the application executes On-Board Control Procedures (code downloaded by a telemetry).*
Specific findings	Possible to find with static analysis tools (customisation required). • *Example: mishandling of housekeeping information.*	Requires both understanding of general security principles along with domain-specific expertise. • *Example: cryptographic keys kept in use for an unsafe duration.*

Static analysis tools are today used successfully as a complement to the safety or mission-critical developments. As an example nearly every major product at Microsoft must be tested rigorously with static analysis tools, NASA requires each code change to mission critical applications to undergo thorough static analysis. Each warning and suggestion the analyzer finds must be either fixed or a comment added to the source code describing why the warning is unnecessary to fix.

ESA requires assurance that the safety-critical software developed is also dependable and robust. This is why the usage of static analysis tools has been recommended (e.g. ECSS-E-40 2B) and extensively applied. In the specific case of this work, static analysis tools have been used to assess an on-board software application in the frame of an ISVV project. The usage of static analysis tools is also highlighted in the ESA ISVV Guide for IVE (Independent Verification) activities.

4 STATIC ANALYSIS TECHNIQUES

The static analysis tools implement different and very specific techniques to achieve the required assessment or verification objectives.

Static analysis tools verify the code without knowing or requiring any external states to be set. Since the static analyser does not know what the application or function is intended to do it will not make assumptions that a developer or code reviewer might.

The techniques used in the majority of static analysis tool include:

- Semantic checking
- Strong type checking
- Memory allocation checking
- Logical statement checking
- Interface and include problem checking
- Safety/Security checking
- Metrics analysis
- Simulator (data generation)
- Crawl source code

5 TOOLS EXPERIMENTATION (POLYSPACE AND FLEXELINT)

Testing using static analysis tools can drastically reduce a number of bugs which may be difficult to find in black box testing such as buffer overrun, encoding and dangerous function usage. A quick scan with a static analyzer may discover many bugs that are difficult to find by other means.

This section shall describe the tools used to analyse the on-board system.

Two tools have been used for this project, Polyspace Verifier R2008a (The Mathworks) and FlexeLint 8.0 (Gimpel).

FlexeLint is a static analysis tool that checks C/C++ source code and finds bugs, glitches, inconsistencies, non-portable constructs, dead code and much more. It looks across multiple modules, and so, enjoys a perspective the compiler does not have. FlexeLint is a highly customisable tool, allowing the selection of individual verification rules, the precision level of the analysis, the output format and detail level of the analysis reports. FlexeLint supports several coding rules, including some MISRA C, ANSI C and K&R rules. FlexeLint is a console application and is run from the command line, with the possibility to be integrated in several IDEs.

Polyspace® products verify C, C++, and Ada code for embedded applications by detecting run-time errors before code is compiled and executed. This advanced verification technology uses formal methods not only to detect errors, but to prove mathematically that certain classes of run-time errors do not exist. Polyspace is a client-server based application, where some verifications are performed on the client side (e.g. MISRA C checker) but most of the analysis is performed on the server side (e.g. static analysis). This architecture eases the tool usage by a large team. Both client and server applications provide a command line interface and GUI. Polyspace provides a dedicated application to analyse all the results generated by both client and server applications. This data can also be exported to other formats. Polyspace is highly customisable, allowing multiple configuration settings. Besides providing a static analysis based on a mathematical approach, Polyspace also provides a MISRA Checker that verifies subset of the rules defined in this standard, a data-flow and concurrency analysis.

FlexeLint and Polyspace have been used to analyse two of the three modules (Application Software and Basic Software) that constitute the Lisa Pathfinder on-board software. This analysis has been performed using the software versions used for the ISVV activity. The effort taken to complete the activities has been categorised and recorded for evaluation. The issues reported by the static analysis tools have been checked manually, filtered and categorised according to their problem type.

Both tools have detected several issues that need to be addressed by the software development team. All the results obtained by the static analysis tools have been checked manually in order to assess the ratio between true and false positives.

The sooner an issue is found and fixed the less it costs. Late discovered vulnerabilities can have a high cost. The same vulnerability, if found in early development phases, could be very inexpensive to fix. Most static analysis tools can point directly to the problematic lines of code so the developer doesn't have to track down the problem from a high level bug report.

The collected data allowed performing several conclusions regarding the usage of static code analysis tools, namely cost effectiveness, quality of results, usefulness and applicability of the tools, complementarities between the results of different static code analysis tools, relation between automated code analysis and manual code inspection and coverage of the traditional IVE tasks.

6 TOOLS COMPLEMENTARITIES

Static analysis tools, such as Polyspace and FlexeLint can complement other development tools, and can even complement each other.

As stated in [1], and defined by [2], the common areas where application security problems can occur can be described by:

- Input Validation
- API misuse
- Safety/Security features
- Time and state
- Error Handling and FDIR
- Code quality
- Encapsulation
- Environment effects

Since static analysis tools do not cover all these areas, there is a need to have complementary tools or techniques applied in order to have a better verification assessment.

Static analysis tools commonly report false positives. These often take time to find in source code and to decide if the problem is a serious bug that needs to be fixed. Thus, an expert shall review these false positives and decide if they are related with problems or not. These tools shall be complemented with other tools and techniques:

- Manual code reviews allow a fresh pair of eyes to look at a specific code to find errors or conditions not thought of by the original coder.
- Dynamic analysis, such as code stepping with a debugger, is a great white box testing technique that can be used to find bugs before release time that may not be discoverable using static analysis.

Static analysis tools will not find all the problems in the code, nor can they ensure there will be no problem at all. Like the tools used in the development, static analysis tools can only support the discovery of issues.

Table 1, presents some examples of the verifications provided by Polyspace and FlexeLint. Although both tools share some verification types, they are complementary and, therefore, not redundant.

Verification Type	Polyspace	FlexeLint
Illegal pointer access	Yes	Yes
Non-Initialized Variables	Yes	Yes
Divisions by Zero	Yes	Yes
Unreachable code	Yes	Yes
Indentation not respected	No	Yes
Customize analysis at type level	No	Yes
Tasks and concurrency analysis	Yes	No
Data Flow analysis	Yes	No
Shift Operations checks	Yes	No

Table 1: Some examples of Polyspace and FlexeLint coverage

In fact, FlexeLint and Polyspace outputs are distinct. FlexeLint presents a list of errors, warnings and information messages, very similar to a compiler, while Polyspace static analysis results are presented as a list of lines with errors, lines with possible errors and lines with no errors at all.

7 LESSONS LEARNT

This section describes the lessons learnt extracted from the usage of the static analysis tools on a safety-critical on-board software system, as well as some generic lessons learnt.

Specific lessons learnt from the usage of the static analysis tools:

- Polyspace and FlexeLint are not redundant tools.
- In average PolySpace raises issues with higher severity.
- In average FlexeLint raises more issues than Polyspace.
- Polyspace is very sensitive to the lack of libraries and system files, which results in an increased setup time before the analysis able to be run.
- FlexeLint is very easy to configure and straightforward to use in terms of setup and analysis repetition.
- Polyspace is very hardware dependent and the analysis can be very time consuming.
- FlexeLint raises an enormous amount of messages (hard to verify manually). The verifications performed must be filtered by a proper configuration.

Generic lessons learnt from the tools usage:

- Tool users should learn the tool extensively, understand how the tool works and to get the most out of it, a professional training is recommended.
- Developers should use static analysis tools during the project life-cycle, as a supporting tool and to enforce coding standards.
- Testers should static analysis tools to find sections of the application with high complexity and where bugs may be exist.
- To take the most out of the static analysis tools, these should be periodically updated (database of known coding problems, fixing tool problems, performance issues, etc).
- Define appropriate project specific static analysis tools, some tools allow definition of coding rules, customisations to fit a specific project, filtering of issues, etc. More relevant issues can be found before compilation and less false positives.
- Promote the correction of warnings that will help make a better product only. Changing code extensively will create newer problems.
- Static analysis tools can be complemented with dynamic analysis tools (white box testing) and manual code reviews. Code reviews are still very important and can catch errors that a static analysis tool cannot, it adds a fresh set of eyes to look at the code. With an experienced code reviewer, some functional issues can be detected manually, which would not be possible using static code analysis tools. Dynamic analysis tools check the code at runtime to discover problems within the source that may be related to the environment. This can be very important since the static analysis tool can not predict system state or specific hardware configurations.
- Other dynamic analysis tools can be used to greatly increase the value of application analysis; these tools can include, code coverage tools, fault injection tools, performance monitors, schedulability tools, etc.

8 RESULTS ANALYSIS

This section presents the main results obtained from the usage of the static analysis tools.

The effort spent to perform the activities has been measured in order to evaluate their cost effectiveness. The results are presented hereafter:

Category	Description	% Hours
SETUP	Tool /Environment setup	18%
VERIFY	Tool results verification	31%
CMGMT	Configuration management	1%
COORD	General coordination	5%
ANALYSIS	Results compilation, pre-processing	21%
PMGMT	Project management	13%
REPORT	Work-package reporting	11%
Total		100%

Table 2: Effort categorisation

The SETUP category corresponding to tool & environment setup and configuration, and can be further detailed for each tool as presented in Table 3:

Tool	Task	% Hours
FlexeLint	SETUP	35%
Polyspace	SETUP	65%

Table 3: Tools setup and configuration times

Setup time is far from being negligible, taking 18% of the total spent effort. Polyspace requires extra information (besides the source code) to perform any kind of verification. The gathering of data (e.g. libraries, OS system files, compiler, etc) resulted in an increased setup time.

Nonetheless, the configurations performed for both tools can be re-used for future projects, decreasing the setup time ratio in the overall effort and, therefore, increasing the cost effectiveness of using static code analysis tools for subsequent utilisations.

The total amount of RIDs (Review Item Discrepancies) identified by both tools, including their severity and problem type is presented in Table 4 and Figure 1**Error! Reference source not found.**:

Problem Type[1]	Number of RIDs
Improvement	14
External Consistency	17
Correctness	55
Completeness	0
Internal Consistency	4
Superfluous	8
Technical Feasibility	0
Readability & Maintainability	13
Total	111

Table 4: Problem Types of the RIDs generated by both tools

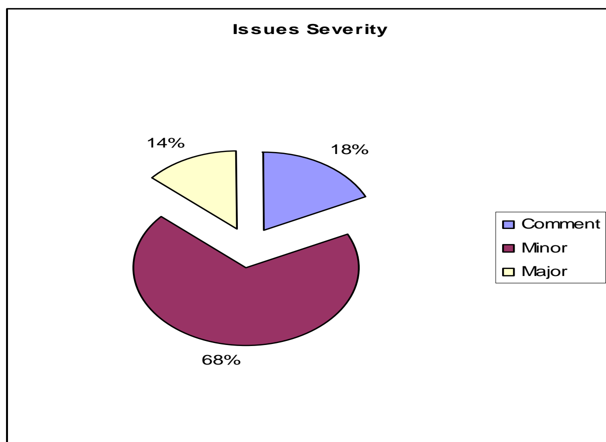

Figure 1: Severity[2] of the RIDs generated by both tools

The source code under verification by the two static analysis tools was constituted by approximately 28800 physical lines of code. 111 RIDs have been raised according to the static verifications performed with the tools. The majority of the RIDs raised are code standards non-compliances and have been classified with "minor" severity. Most of these RIDs have been classified with "Correctness" problem type. Nonetheless, 15 RIDs have been classified as "major" and require immediate attention from the development team.

[1] *The presented problem types categories are defined in [3].*
[2] *The presented severity categories are defined in [3].*

9 CONCLUSIONS

Static analysis tools are beneficial to the software production and verification processes, although the way they are deployed and used may be different. They help keep development costs down by finding issues as early as possible in the development cycle.

Static analysis tools are very good at code level discovery of problems and help enforce coding standards and keep code complexity low. Besides helping developers finding issues the tools can also ensure future readability (maintainability) by making sure all coding standards are followed. Metrics can be generated to analyse the complexity of the code and to discover ways to make the code more readable and less complex.

Static analysis tools are also very important in IVE activities since they are able to automatically detect several issues that are usually difficult to detect by manual inspection (e.g. the fulfilment of several coding standards, verification of possible buffer overflows, etc). Nonetheless, they do not replace completely the manual code inspection since these tools are not able to detect functional issues (e.g. arithmetic formula erroneously implemented, acknowledge packet sent after receiving a communication that does not require acknowledge, etc). Nevertheless, static analysis tools allow the manual code inspector to focus on functional aspects instead of trying to detect semantic issues. By applying manual and automatic code verification, thus spending effort for each verification, the quality and completeness of the results are increased.

The usage of more than one tool is also recommended, although with careful planning. The selection shall focus on tools which take different approaches to code verification. Different approaches often provide different results which will bring added value to the verification activity.

Throughout this paper static analysis tools have been discussed in depth. These tools are today irreplaceable in the development cycle and very important in verification activities. Each phase of the development cycle can benefit from the use static analysis tools, ultimately resulting in a better and safer product.

10 REFERENCES

[1] Chess B., West J., Secure Programming with Static Analysis, ISBN: 0-321-42477-8, June 29, 2007, Addison Wesley Professional.

[2] Tsipenyuk, Katrina, Brian Chess, Gary McGraw. "Seven Pernicious Kingdoms: A Taxonomy of Software Security Errors." Proceedings of the NIST Workshop on Software Security Assurance Tools, Techniques, and Metrics (SSATTM) (Long Beach, CA, 2005), 36–43.

[3] ESA ISVV Guide, version 1, revision 0c, November 22, 2005.