Evaluation of Verification Tools Continued: More Tools, More Software, More Aspects

Ralf Gerlich, Rainer Gerlich
Dr. Rainer Gerlich BSSE System and Software Engineering
Immenstaad, Germany
e-mail: Ralf.Gerlich@bsse.biz, Rainer.Gerlich@bsse.biz

Jens Gerlach, Jochen Burghardt
Fraunhofer-Fokus,
Berlin, Germany
jens.gerlach@fokus.fraunhofer.de,
jochen.burghardt@fokus.fraunhofer.de

Sergio Montenegro, Frank Flederer
Julius-Maximilians-University, Informatik VII
Wuerzburg, Germany
sergio.montenegro@uni-wuerzburg.de,
frank.flederer@uni-wuerzburg.de

Christian R. Prause
Deutsches Zentrum für Luft- und Raumfahrt e.V. (DLR)
Bonn, Germany
e-mail: Christian.Prause@dlr.de

***Abstract*—In a previous study six software verification tools have been applied to a representative space software package. The findings reported by each tool have been compared in order to derive footprints regarding fault identification. In a continuation three more tools were applied to the previously selected application software and to another application together with two tools previously used in order to broaden the base of evaluation. More aspects were considered regarding the evaluation of results: an additional evaluation criterion was added and a comparison of reported defects with the outcome of unit tests was performed. Due to a higher degree of formalization and automation the manual evaluation effort could be decreased while extending the number of considered reports and the number of tools. The encountered evaluation and verification issues are discussed in detail. All results together shall provide a detailed view on the defect identification capabilities of the considered tools w.r.t. current software base. Altogether, the high quality of reports as obtained in the previous study was not obtained again: in context of a different set of tools and another (object-oriented) language a lot of trivial reports were observed.**

Keywords: verification tools, unit test, C / C++ software, false positives, false negatives, software faults, fault identification, fault coverage, fault report evaluation, software verification, verification efficiency

I. Introduction

In [1] results of a first step towards evaluation of verification tools were presented and discussed. In this paper we provide results of a continuation of the tool evaluation activities.

The conclusions at the end of the previous study suggested broadening the base of the evaluation by considering more tools and more application software. Therefore another representative software package was selected and three further tools were added to the set: Frama-C, PC-lint and a commercial third one, for the disclosure of which no permission was given yet. Due to the results of the earlier activity regarding tools' capabilities and applicability to the space domain, three tools from that earlier evaluation were not considered to be of further interest and were not part of the present evaluation.

The recently obtained results are quite different from the previous ones. While the amount of reports and their classification as true and false positives was rather straightforward in the first study, the number of reports was significantly higher and their classification was rather challenging. Not only the amount increased as such, but a high number of trivial reports was observed not contributing any value, but compromising heavily the recognition of true positives in the large set of reports.

However, this observation shall not be considered as a counter argument regarding the benefit of analysis tools. It should be understood that a careful selection of tools is required in order to maximize defect identification and minimize the related effort.

The deeper analysis of the defect identification and reporting mechanisms led to the conclusion, that just to buy and apply a tool at the end of development is not sufficient. In fact, a developer also can contribute a lot to reduce the amount of reports on suspicious code, implying fault potential, but not causing a risk in the current context.

A major point is the continuous use of such a tool over the development period, which is not a new message, but it has been confirmed again.

For comparison of results of analysis with unit testing the tool reports were correlated with the results achieved by unit testing to investigate how complementary or overlapping both aspects of software verification are.

This paper is structured in the following manner:

In Ch. II principal terms are explained required to understand the evaluation process and the results. In Ch. III the evaluation context is described, as well as the software and the tools used. The evaluation process is explained in Ch. VI. Verification issues are discussed in Ch. V. The evaluation process is described in Ch. VI. The evaluation results are presented in Ch. VII. Lessons learned are provided in Ch. VIII, and in Ch. IX conclusions are drawn.

II. Definition of terms

The terms relevant for understanding the evaluation process are defined in this chapter.

Paper presented at DASIA'2017, 30.05.2017
Eurospace Symposium DASIA'2017 "Data Systems in Aerospace", 30 May – 1 June, 2017, Gothenburg, Sweden

A. *Tool Report*

In the context of this paper a “tool report” is a message issued by a tool indicating that one of its verification rules is violated.

B. *Defect, Fault, Error, Failure*

A *defect* commonly refers to troubles with a software product, with its external behavior or its internal features (e.g., its maintainability). This includes consideration of the risk of faults by potential changes of the context, which could invalidate previous verification results. For details please refer to [1].

C. *Defect Types*

Defects can be grouped into “defect types”, so that a defect is considered as an instance of a defect type. Many defects of the same defect type may be reported.

Defects of the same type may be called differently by different tools. In consequence, for matter of comparison, the terms used for a defect type in a tool report must be mapped onto a standard defect type. This mapping may be automated by mapping tables.

D. *Criticality of Defect Types*

Four criticality classes were introduced for the defect types: critical, warning, uncritical and ignore.

“critical” means that the defect always manifests itself as an error, “warning” that it may manifest case-by-case, and “uncritical” that it is a software engineering issue only, not manifesting during execution and impacting runtime behaviour. Finally, “ignore” collects all other reports which are not considered as useful at all, e.g. providing additional explanation to another report, or highlighting a trivial case which would not be subject of corrective maintenance at all.

E. *Classification of Tool Reports*

In general, a report is a message issued by a tool on a supposed defect found in the software according to its defect identification approach, usually based on violation of verification rules.

A tool may fail to report a defect or may report a defect where no defect is present. There are 4 distinct cases depending on whether a defect exists or not and whether a tool reports a defect or not (Fig.II-1).

		Code	
		Defect present	*Defect NOT present*
Result	*Defect Reported*	true positive, TP	false positive, FP
	Defect NOT reported	false negative, FN	true negative, TN

Fig.II-1: Classification of Tool Reports

The characteristics of a tool regarding its capabilities to correctly report defects shall be described by two figures:

Sensitivity: it is defined as the quotient TP / (TP+FN).

Precision it is defined as the quotient TP / (TP+FP)

Sensitivity represents the portion of confirmed defects (TP) in relation to the overall number of defects. As the overall number of defects remains unknown, it is approximated by the set of confirmed defects found by all tools or by analysis in the context of manual assessment of the reports.

Precision represents the portion of reported defects that are actual defects compared to the number of issued reports.

F. *Complementarity of Tools and Tool Combinations*

A result of previous evaluation is that no tool can cover all defect types: tools may be complementary or overlapping. To maximize defect identification in the context of verification in a project, especially for identification of the tools being made mandatory in the Software Verification Plan (SVP), it is essential to know which combination of tools increases the sensitivity and how much.

The more tools are complementary, the higher is the portion of unique contributions by tools.

G. *Report Classification*

For classification of tool reports as TP or FP two main criteria were applied with two sub-criteria each (Fig. II-2):

- Criterion 1: tool criterion
- Criterion 2: state criterion
- Sub-Criterion 1: without context
- Sub-Criterion2: with context

The “tool criterion” was applied in the previous study, and there it was the only one. The classification is purely performed by answering the question “Is the tool right or not”.

Now, in addition the “state criterion” was introduced to consider whether an undesired state could result from the reported defect. If so, the report is classified as TP, otherwise as FP.

Cases may exist where the tool is right, but the resulting state is still valid. A typical example is “while (1)”. In a task body, this construct is frequently used and the non-termination of the loop is intended.

Classification Category	Criterion	Applied Condition	Applied to Application
validity	tool	Is the tool message formally correct?	1,2
	state	Can an undesired state be reached?	2
context	with context	Input domain may be constrained by callers	1,2
	without context	Maximum input domain can be used	1,2

Fig. II-2: Evaluation Criteria

Similarly, release of a resource may not be intended, because the application will never terminate, so that the state resulting from the endless loop or the not intended release of the resource will either not be of relevance or not matter at all.

These examples show that the state criterion is not an objective criterion: the decision may depend on a supposed intention or consideration of an extended evaluator-defined scope not seen by a tool. Whether an undesired state will occur, may also depend on the platform and the algorithm implemented in the code.

Therefore it does not seem to be a suitable criterion to compare the tools.

The decision on FP or TP may also depend on the context as explained in the following section.

H. *Context and Platform Dependency*

The constraints imposed on the input domain of a function, as spawned by the type ranges of its parameters together with conditions or constraints imposed by its call-context, is called “the context” of a function call. In case of the sub-criterion “without context” the full input domain and no other constraints are considered.

Different results may be derived depending on whether considering the context is activated or not. The context may constrain the input domain, so that a defect cannot be activated or cannot manifest as error or failure. A report may be considered as TP in the context of the full input domain, but as FP in the context of an application imposing a limited input domain.

The number of considerations can be minimized for context-sensitive defect types due to the following conclusions:

- In case of dead code and invariant conditions a TP for the case “without context” implies a TP for “with context”.
- Vice versa, for the other context-sensitive defect types a TP for “with context” implies a TP for “without context”.

The evaluation result also may be affected by the properties of the platform (compiler, linker, processor, other hardware), and the contents of the data when the suspicious code is executed. So it may happen that an overflow in a byte-operation is masked by the processor because it always applies 32bit-operations, or a linker silently maps data with same, but from different compilation units onto each other.

III. CHARACTERIZATION OF THE EVALUATION CONTEXT

A. *Overview*

Three out of the six tools used in the previous ESVW study [1] were no longer included in the activity (they were called “1/xxx”, “4/zzz” and “6/gcc” in that study). Instead, three other tools were included in the current FSVW study: FramaC, PC-lint and a tool called www for the time being, yielding five tools in total to be considered. Tab. III-1 gives an overview on the use of tools in both studies. Unfortunately, not all names can be disclosed at this point. Currently, we have not received a confirmation from the vendor of Tool 5 / www in the current set to disclose its name.

Further, another application software package (written in C++) was selected. In order to achieve full coverage for the current set of tools regarding the two software packages, all current tools were applied to the Package 2, except for FramaC, which currently does not support C++, and the new ones also were applied to Package 1, yielding the matrix shown in Tab. III-2.

As already done in [1] a subset of functions had to be chosen to limit the manual effort for evaluation of the tool reports.

For Application 1 26 functions were selected by fault distribution (the ones with highest number of defects), 34 randomly. In case of Application 2 a first evaluation of defect distribution vs. functions yielded no significant accumulation of defects for certain functions like it was observed for Application 1. Therefore it was decided to take the cyclomatic complexity (CC) for selection. CC varied from 1 to 16. Five groups were built according to CC and 60 functions were selected randomly from these groups.

Study	Standard Defect Types					Tool-specific Defect Types
	Criticality					
	total	critical	warning	uncritical	ignore	
ESVW	20	11	8	1		▨
FSVW	40	16	17	6	1	371

Number of Observed Tool-Specific Defect Types				
Frama-C	yyy	DCRTT	PC-lint	www
28	43	42	185	73

Tab. III-1: Overview on Tools and Studies

Appl.	Tool								
	1		2	3	4		5		6
1 / C	xxx	Frama-C	yyy	DCRTT	zzz	PC-lint	QA/C	▨	gcc
2 / C++	▨	▨	yyy	DCRTT	▨	PC-lint	▨	www	▨

Tab. III-2: Overview on Tools and Software Packages

B. *The Application Software*

Tab. III-3 shows the characteristics of both software packages.

Property	Application 1 C	Application 2 C++
Size / KLOC, total h+c	42	20
Functions, total	610	611
c-Files, total	49	55
with functions	39	
without functions	49	
h-files	96	104
Functions, manually evaluated	60	60

Tab. III-3: Characterization of the Software Packages

C. *The Tools*

The spectrum of analysis approaches as listed in Tab. III-4 applied by the tools is quite broad, and defect identification by the different tools is based on a number of independent methods and implementations (Tab. III-5).

Analysis Approaches	
static	abstract interpretation
	dataflow
	symbolic execution
	analysis based on dedicated checking and tracking
dynamic	auto-stimulation / automated testing

Tab. III-4: Spectrum of Analysis Approaches

Only those tools are listed there which are still in the set of Study 2 / FSVW.

Tools 1, 2, 4 and 5 are static analysers, Tool 3 applies dynamic analysis (automated built of the test and stimulation environment).

Abstract Interpretation is used to approximate the semantics of a computer program in order to soundly prove certain characteristics of the program, e.g. the absence of certain defect types.

Tool		Type	Analysis Approach	Appl.
1	xxx	static	abstract interpretation	1
	Frama-C			2
2	yyy			1,2
3	DCRTT	dynamic	auto-stimulation	1,2
4	zzz	static	symbolic execution, dataflow analysis	1
	PC-lint		Analysis based on dedicated checking and value tracking	1,2
5	QA/C		Symbolic execution, dataflow analysis	1
	www			2
6	gcc	compiler	syntax, semantic, type checking	1

Tab. III-5: Characteristics of Tools

For *Automated Testing* / auto-stimulation the software is automatically stimulated with inputs and its behaviour is monitored, e.g. by instrumentation. As not all possible combinations of inputs can be provided, the method may miss present defects, leading to FNs. However, any input that leads to an error is a witness for the presence of the respective fault in the code. FPsare only possible if representativeness of the test platform is not ensured.

Symbolic Execution is a method used for analysis where the software to be analysed is executed symbolically: Instead of concrete values, symbolic variables are used. Similar to actual execution, only a specific path through the software is executed. In order to prove absence of a defect at a given point in the code, all paths by which this point is reachable have to be enumerated, similar to testing. As a consequence, if complete enumeration is not possible, the method may miss present defects, leading to FNs.

D. *Tool Configuration*

Every tool provides its own and specific set of configuration options. Of course, the chosen set of such options impacts the issued reports.

The applied configuration options are briefly described in the following sub-sections.

1) *Tool 1: FramaC*

The value-analysis-plugin of FramaC (version Silicon) was subject of evaluation.

In case of FramaC several attempts were required to find a suitable configuration.

There seems to be no knowledge on trade-offs between execution time and accuracy of results. Therefore the configuration parameters *slevel*, *plevel* and *ulevel* were reduced in three steps from the highest value down to a value where the tool terminated its run within three days.

2) *Tool 2: yyy*

The same optimized configuration as in the previous study (especially regarding a 32-bit application) was applied to the C++ application with the following additional decisions.

Reports on non-initialized class members were turned off as they lead to errors which block further code analysis.

The tool failed initially and the software was provided to the tool supplier. According to the feedback one function was stubbed as work-around.

3) *Tool 3: DCRTT*

The same optimized configuration as in the previous study was applied to the C++ application. However, three runs were executed: the first one under consideration of constraints on function parameters and global variables regarding data ranges and size of arrays, collected automatically, and with call of suitable initialization functions, while for the second and third run such constraints were removed stepwise.

The reason for execution of the additional runs was that the constraints were also present for the “without context”-case, and may hide reports, while intentionally the constraints were inherently considered to reduce the number of FP-reports for the “with context”-case.

4) *Tool 4: PC-lint*

The standard configuration of PC-lint was applied. Only the options for the maximum width for integer and float were set to 32-bit *(-si4 -sp4*), as the application was written for a 32-bit processor.

Although PC-lint offers the opportunity to switch off report types on a case-by-case basis, this capability was not applied. Instead in the course of the mapping of tool report types onto standard defect types, report types which were considered as irrelevant were mapped onto an additional type “DefectTypeIgnored” in order to get rid of such reports in the course of manual evaluation..

5) *Tool 5: www*

The standard configuration for this tool was used, except for raising the dataflow analysis level to the maximum.

IV. TOOLS VS. UNIT TESTING

The intention of a comparison between results of tool analysis and unit testing should clarify what the benefit of each of the approaches is, and whether they are complementary or overlapping, and if so to which degree.

A. *Overview on the Approach*

For each (executable) line of the source code a marker was added indicating

- whether the line was covered by a unit test at all, and in detail,
- by which unit test out of the whole set,
- whether the line includes a “normal” statement / expression or a condition, and
- if available, information was added whether an exception or a defect was detected during a unit test.

Cross-module coverage was considered, i.e. if a unit test did not only generate coverage in the function-under-test, but in the call tree as well.

The stream of lines of an application, either for the whole set of functions, or the selected subset, formed the basis for correlating the tool reports with coverage results from unit tests. This way, contributions from unit tests and analysis tools were compared.

If a tool reports a defect for a covered line, obviously the defect was not detected during the test. Vice versa, if a defect was detected during unit testing, but not by analysis, then a tool is not sensitive for this defect.

In consequence, the more either a tool reports or a test highlights a defect, but both do not for the same defect and the same line, the higher the complementarity of analyses and unit testing.

As (additional) contribution by tools the following 2 cases are considered:

- If a line was covered and a TP was reported for this line, then the defect can be assumed not to be detectable by a unit test.
- If a line was not covered and a TP was reported for this line, then the analysis brings an added value by reporting a defect for a location, not addressed during unit testing.

Vice versa, as additional contribution from a unit test with respect to tools the following case is considered:

- If a line was covered and a defect or an exception was detected during a unit test, but no tool issued a report for this line, then this is a valuable contribution by unit testing.

Further, coverage as such can be compared:

- Is missing coverage confirmed by a “dead code” report or not?
- Does a tool report “dead code”, while coverage was achieved?

Different scenarios may be applied for these considerations taking TP from tool analyses, then possibly leading to different results: was the TP a matter of “with context” analysis, from “without context” analysis, or both, and which context was considered during unit testing.

B. *Application 1*

The unit tests were already performed for 362 of 610 functions, and detected defects were removed successively. For 248 functions no unit tests were executed as these functions were auto-coded, and test of a few of such functions was considered as sufficient. However, by the tool reports principal issues in the code generator were detected, mainly related to fault handling, suggesting that more or even all auto-coded functions should have been subject of unit testing.

As the idea of correlating unit tests and analysis reports came up after completion of unit testing, the information about detected defects was not recorded.

Therefore in case of a covered line, either a defect already detected during a unit test was not fixed, not detected or not detectable.

The tool used for the unit tests was VectorCAST [2]. It was also used for retrieval of the unit test information in a format suitable for the merge with report information

C. *Application 2*

The unit tests were established as part of the study, but due to budget limitations only for the subset of (60) selected functions. The information on defects and exceptions was recorded, but only a few defects were detected, actually. In all cases at least one of the tools reported the issue, too.

The tools used for unit testing are cppunit [3] and gcov [4].

V. Verification Issues

The results of the previous activities were discussed with tool suppliers and software developers. The contents of such discussions are briefly listed below, followed by conclusions on the evaluation criteria. The discussion highlights principal issues of report classification, also driven by the high number of trivial reports observed in the recent study.

The issues related to tool vendors are mainly a matter of FNs, while the discussion with developers focus on FPs.

In addition, pro’s and con’s regarding unit testing, verified-by-use and analyses are discussed in order to get a clearer picture about which verification approach may be appropriate regarding required dependability and implied costs.

A. *About False Negatives*

The general point of discussion is under which condition(s) a missing report may lead to an FN for a tool.

In the previous study, the criterion 1 / tool criterion was applied to investigate which defects can be reliably found by a tool, focusing on whether a report is justified or not.

If a tool is right, the lack of a report from another tool must be considered a FN.

Tool suppliers argued that the FN originated in conditions imposed by the tool that would preclude the respective fault to be activated. While these conditions were not present in the original code, the vendors pointed to the documentation of such assumptions justifying the lack of a report. If such documentation was missing, this was an issue of the documentation, but not of the tool, the vendor argued, and as such should not be considered an actual FN.

Our position is that if a tool does not report a fault because, although the error can be activated under the circumstances imposed by the code itself the tool imposes additional constraints, then an actual fault is not being reported due to aspects solely to be blamed on the tool. As such, the lack of a report is to be considered a FN. Whether the reason for that FN can be explained or even is documented or not is irrelevant.

This discrepancy in opinions led to the vendor blocking publication of the tool name.

However, if the tool can be configured not to impose the restriction, then care is to be taken before marking down the lack of a report as a FN, because a user can modify the configuration so that the fault could be detected.

Still, if the additional assumption normally blocking detection of the fault is enabled by default and must be explicitly disabled by the user, this needs to be considered in a critical manner. The question arises whether a user would be sufficiently aware of such a default assumption. The documentation for such tools is often already very extensive so that realistically it is not useful to assume that a user will understand and keep in mind all the consequences such assumptions would have.

It may be essential for a tool supplier to distinguish whether the source of the FN is related to the environment built by the tool, constraining detection, or the algorithm applied for detection, especially when the tool supplier claims that the algorithm can ensure absence of FN. For the user, the reason is irrelevant.

Also, a case was observed in the recent study in which a report was not issued although it ought to be according to documentation. Obviously, there is discrepancy between documentation and implementation.

To summarize: any fault not being reported must be considered a FN, independent of the reason for the lack of a report.

B. *About False Positives*

The main concerns about FPs are coming from developers, claiming that the reported defects would not cause a failure of the software – even if the tool is formally correct.

Cases were observed – not in the reference studies, but for other applications, where the implemented logic was completely wrong, but incidentally for the few parameter sets given the results was correct.

The big challenge is that the number of reports related to suspicious code is – as a matter of experience – much higher than the ones related to clear TP. This implies a high overhead for the analysis, which would not occur if suspicious code were not present or was even avoided, e.g., by using the tools right from the beginning and considering their feedback.

However, it has to be admitted that also a tool may be originator of a significant number of trivial reports (see the discussion in Sect. C below and in Ch. VIII.B), which result in TP according to the tool criterion, but could be considered unjustified. Then proper measures have to be undertaken to avoid an overhead.

In part, such reports may be put in a separate category "ignore". But this is not possible in every case. "Loss of precision" turns out as uncritical in most cases. However, amongst such a set one report may evaluate to a critical TP (the reader is reminded of the incident during Ariane Flight 501, the maiden flight of Ariane V – coincidentally the trigger for the creation of some notorious static verification tools). If the whole set would be ignored, then the critical report would be lost. This is not acceptable. Therefore other measures need to be considered.

The comments of developers to TP according to the tool criterion were mixed, ranging from immediate acceptance (considering it as a violation of best practices) to rejection because the probability of the system state being compromised was assumed to be sufficiently low or even zero, although not being compliant with best practices.

However, the essential point is: usually it is not known in advance whether the result does not lead to unwanted states, while a negative impact is possible in general. A valid conclusion can only be drawn *after* – manual – analysis.

Therefore the principal options are:

- to ignore/drop a report and take the potential risk,
- to do the analysis and decide after whether to fix or not, or
- to do the analysis and to fix the issue, and/or to try to avoid similar issues in future.

C. *System/ Context-immannent False Positives*

In some cases it is reasonable that a tool frequently produces a FP if it does not have – sufficient – information on the context. In many cases it is even impossible to provide this information on language level. Amongst such cases are: resource leaks and loss of precision.

However, a tool may support provision of meta-information to suppress such FP-reports.

D. *Verified-By-Use vs. Verification by Analysis*

In the discussion with developers frequently the issue of a high number of FPs – in the sense of the state criterion – is addressed, doubting the added value coming from analysis tools like the ones under consideration, and claiming that most of the reports issued by such tools would actually result in FPs.

In the discussion it is important to understand that the state criterion does not deal with probabilities: If it is possible to enter an undesired state as per the state criterion, the report has to be considered a TP, independently of the probability of occurrence of such an event. For the study, the reasons for this are pragmatic: Neither is the probability distribution of the inputs known nor was a limit probability specified below which events can be considered seldom enough not to be considered.

However, the same problems occur in practice as well: Typically, at least one of these items of information is not formally available.

Still, low probability of occurrence can only be a valid defence against a fault report if proof can be provided that the probability is small enough.

According to the experience obtained so far – not only in the course of the ESVW and FSVW studies, but also in context of analysis activities in other projects – the essential point raising such discussion is that the verification goals are not precisely defined, if at all.

Then – as a consequence – the use of analysis tools is not harmonized with the development process, leading to overhead and missing acceptance of the tool reports.

When a decision is made towards use of analysis tools, an integrated approach needs to be defined prior to any tool usage, addressing

1. definition of criteria for TPs,
2. definition of the fault removal process,
3. selection of suited tools considering required criticality criteria,

4. continuous tool usage in the development process,
5. definition of report processing to reduce the manual effort, possibly including pre-processing and filtering of raw reports.

All these steps are a pre-condition for successful tool usage. Dropping any of them suggests that either

- use of tool analysis implies an overkill regarding the real needs,
- (possibly) unacceptable risks are tolerated, or
- efficiency of verification suffers.

The more of above process steps are dropped, the closer the envisaged verification process comes to verified-by-use, while the costs of verification remain much higher as for verified-by-use.

To support the clarification process regarding what is the proper approach, the approaches are briefly characterized. The provided arguments may be used as a checklist.

As unit testing is later compared to analysis it is also put into the list.

- unit testing
  - o demonstration of compliance with requirements focusing on functional aspects
  - o limited subset of input domain, coverage-driven
  - o verification goal is to pass the (possibly requirements-based) tests
  - o currently requires major effort at limited predictability on future defect rates
- verified-by-use
  - o demonstration that software does properly work for a given, but probably unclear scenariofocusing on functional aspects
  - o implies that software was sufficiently exposed to such a scenario
  - o possibly enhanced compared to UT due to extended set of conditions
  - o lean approach at limited predictability on future defect rates
- static and dynamic analysis
  - o aiming to demonstrate presence or absence of faults
  - o considers large set of conditions
  - o applies increased capability to detect defects, but still not perfect
  - o provides capability to look beyond scenarios as used for UT and verified-by-use
  - o may imply overhead if improperly applied.

To summarize:

If you want to know that the implementation is correct, i.e. that you can expect always correct results under arbitrary conditions, then do apply a rigorous verification approach like static and dynamic analyses do support.

If you just want to know that you will get correct results under current conditions, although only partially or fully unknown, then unit testing or verified-by-use should be sufficient.

VI. The Evaluation Process

A. The Overall Process

Due to the experience obtained in the previous study a simplified and slightly modified process flow (Fig. VI-1) was applied.

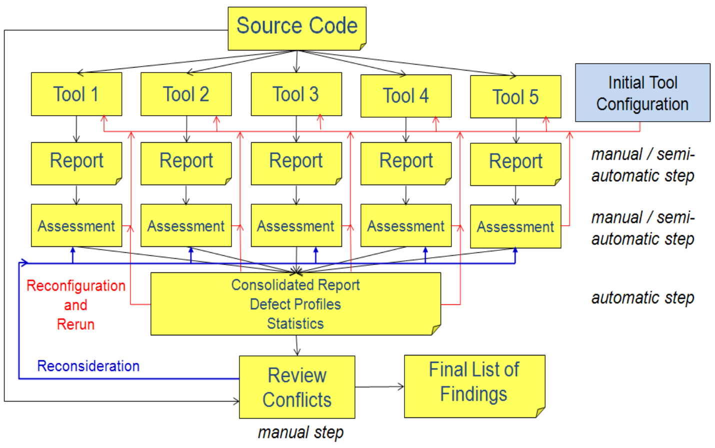


Fig. VI-1: Logic Flow of the Evaluation Process

Now, in a first step every tool is applied to the software and the reports are extracted and immediately classified as either true or false positive, not trying to correlate them with reports from other tools, thereby allowing parallel evaluation of tool reports. Then in a second step all reports are merged into a single stream, correlating reports from different tools about the same alleged defect, automatically, while this step was previously done manually.

After the merge – automatically – FNs are identified for tools not reporting a TP in contrast to other tools.

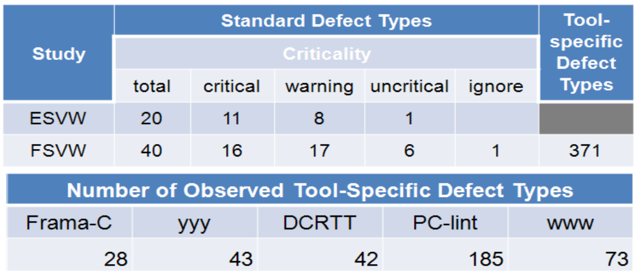
Then evaluation scripts are applied on the consolidated list to derive statistical figures regarding similarities or differences.

The tool and state criteria were applied for all new analyses, i.e. to all analyses related to Application 2, and all analyses performed with FramaC and PC-lint on Application 1.

B. Standard Defect Types

Due to use of another programming language – C++ for Application 2 instead of C for Application 1, a different programming style, and two additional tools, more standard defect types (onto which all the specific messages from all the tools are mapped) were identified: 40 (plus an ignore category) instead of 20 before.

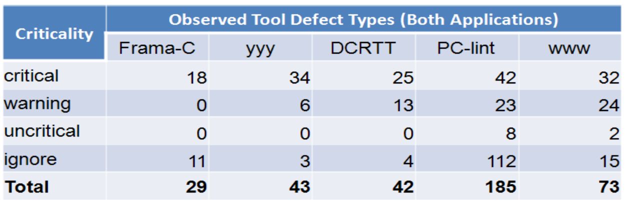
Tab. VI-1 and Tab. VI-2 show the previous and current distribution of defect types vs. criticality and the number tool-specific defect types. Tab. VI-3 provides the list of current standard defect types together with the criticality. Yellow rows indicate new defect types. The total number of considered tool messages is 371, i.e. about 74 messages per tool on the average and nearly 2 specific defect types per tool and standard defect type.



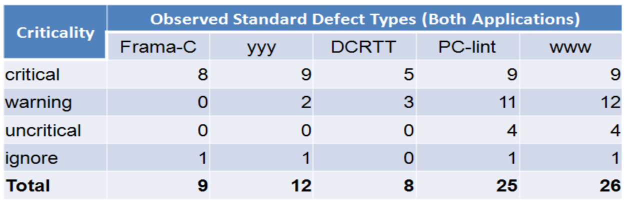
Study	Standard Defect Types					Tool-specific Defect Types
	Criticality					
	total	critical	warning	uncritical	ignore	
ESVW	20	11	8	1		
FSVW	40	16	17	6	1	371

Number of Observed Tool-Specific Defect Types				
Frama-C	yyy	DCRTT	PC-lint	www
28	43	42	185	73

Tab. VI-1: Identified Defect Types vs. Criticality, Summary



Criticality	Observed Tool Defect Types (Both Applications)				
	Frama-C	yyy	DCRTT	PC-lint	www
critical	18	34	25	42	32
warning	0	6	13	23	24
uncritical	0	0	0	8	2
ignore	11	3	4	112	15
Total	**29**	**43**	**42**	**185**	**73**



Criticality	Observed Standard Defect Types (Both Applications)				
	Frama-C	yyy	DCRTT	PC-lint	www
critical	8	9	5	9	9
warning	0	2	3	11	12
uncritical	0	0	0	4	4
ignore	1	1	0	1	1
Total	**9**	**12**	**8**	**25**	**26**

Tab. VI-2: Identified Defect Types vs. Criticality, Detailed

Tab. VI-2 gives detailed figures on the distribution of tool defect types and standard defect types vs. criticality for each tool.

C. *Automation*

The results of the previous study suggested that a higher degree of automation is urgently needed for processing of the large amount of tool reports, either to complement missing information, to harmonize reports from different evaluators or to merge and compare contents of reports.

For example, in part, function names are provided by tools, to the other part file names and line numbers. The missing information can be added automatically, and all reports can be put on the same contents of information.

In case of C++, the full signature of a function may be provided. Other tools may provide mangled names for unique identification of C++ functions. Such differences can be harmonized automatically, complementing the missing part.

For some defect types the result is identical for with and without context cases, for the tool and state criterion alike, so that the justification can be shared between both cases, automatically, either filling in the fields or – if already filled in – checking manually inserted decisions for consistency. More such rules have been identified and applied.

Further, the determination whether a function is called in context of the application or not was previously done manually. Knowledge is required about whether context has to be considered or not. Due to available parsing information the provision of this information also could be automated.

Defect Type	**Criticality**
Array Index Out-of-Bounds	critical
Dangling Pointer	critical
Dereference of Invalid Pointer	critical
Dereference of NULL-Pointer	critical
File Access Error	critical
Invalid function pointer	critical
Invalid Return Statement	critical
Loss of Precision	critical
Macro Use with Unintended Consequences	critical
Non-terminating Loop	critical
Passing Invalid Argument to Standard Library Routine	critical
(Possible) Recursion	critical
Resource Leak	critical
Undefined Result	critical
Uninitialized Variable	critical
Unintended Use of Implicit Member Function	critical
Arithmetic Operation on NULL Pointer	warning
Arithmetic Overflow	warning
Cast to pointer of incompatible types	warning
Comparison of floating-point values	warning
Conflicting Declarations	warning
Incomplete List of Cases for enum-Type w/o default	warning
Intended Change of Invariant Data	warning
Invariant Condition	warning
Invariant Expression	warning
Loss of Update	warning
Name overloading	warning
Parameter Type Mismatch in Function Call	warning
Timeout during Execution	warning
Unnecessary Loop Construct	warning
Unnecessary Operation	warning
Unreachable Code	warning
Unused Result	warning
Change of Data expected, but missing	uncritical
Incomplete List of Cases for enum-Type with default	uncritical
Inconsistent Overloading	uncritical
Multiple return paths	uncritical
Security Issue	uncritical
Unintended Change of Data	uncritical
Ignore	ignore, don't care

Tab. VI-3: List of Standard Defect Types

And there are still more steps which were automated.

The implemented functionality on automation should not only be useful for tool evaluation, but should also be beneficial for (real) projects needing support for analysis of tool reports.

The challenges are the same for tool evaluation and tool usage: identification of the critical issues from a probably large stream of reports.

The formalization of a number of steps of the evaluation process – a pre-condition for automation – also allows to get a clearer view on the tool reports while limiting and reducing the amount of manual effort. This work still can and shall be extended in future. It is a pre-condition for detailed evaluation of larger quantities of code and tools.

VII. The Evaluation Results

Remark: The results presented here strongly depend on the application. Defects which do not occur in the chosen applications will not be considered. Therefore the results may not generalize to any other context.

In contrast to the previous paper no figures on sensitivity and precision are provided here, for reasons already mentioned. Instead, information is provided which sufficiently characterize the tools.

The tables and graphics provide a lot of information, so that a reader can get an idea on a tool's capabilities. A detailed discussion of all aspects related to this information would go far beyond of the scope of this paper. Therefore the most interesting and important aspects are discussed, only.

A. *Overview on Reported Defects*

Tab. VII-1 provides an overview on the number of reports per tool.

Function Set	Tool Reports Application 1				
	Frama-C	yyy	DCRTT	PC-lint	QA/C
all, raw	10124				
all	1913	948	1480	5245	4976
selected	107	165	187	43	232
ignored, all	39	0	5	3100	2870
ignored, selected	0	0	0	0	0
critical, all	1874	616	942	146	393
critical, selected	107	137	102	6	93

Function Set	Tool Reports Application 2						
	Frama-C	yyy	DCRTT			PC-lint	www
			1	2	3		
all, raw							
all		2132	365	366	370	11999	798
selected		508	73	78	80	107	39
ignored, all		182	0	0	0	8614	510
ignored, selected		41	0	0	0	0	0
critical, all		1155	193			737	141
critical, selected		376	14			55	10

Tab. VII-1: Overview on Tool Reports for both Applications

The rows of the tables show

- all, raw:
  the initial number as issued by a tool, without having applied any steps for reduction
- all:
  the number after having applied tool-specific measures for reduction
- selected:
  the number relevant for the selected 60 functions derived from the all-figures,
- ignored:
  the number of ignored reports

In case of Application 2 and DCRTT three runs were executed to see the impact on different configurations:

1. *without* injection of NULL-pointers, but *with* call of suitable initialization functions,
2. *with* injection of NULL-pointers, but *with* call of suitable initialization functions,
3. *with* injection of NULL-pointers and *without* call of suitable initialization functions.

The impact on the number of reports as such is not so high. However, there are differences in the reports. E.g. when an index-out-of-bounds was reported in run 1, in run 2 a NULL-pointer dereference was reported, excluding the report on index-out-of-bounds, yielding still one report, only.

In case of Application 1 the raw figure of FramaC was reduced by mapping equivalent messages from different call-paths onto one entry, using file, line and report text for compressing. However, different independent reports related to the same triple will be mapped on the same entry, too, e.g. in case the same message was issued for the left and right part of an expression.

For the selected case this impact can be compared and yields about 6 missed entries (~6% of the total number).

In case of Application 2 the raw number of PC-lint was reduced by dropping all reports which are classified as negligible by the mapping tables established for every tool.

B. *Profiles*

Tab. VII-2 shows the distribution of the reports (not of the TPs) over the criticality classes for both applications and the full set and the subset of 60 functions.

A reader should bear in mind that the figures show the percentage regarding the set of standard defect types a tool is supporting (as shown in Tab. VI-2). E.g. all supported standard defect types of FramaC are either critical (8) or to be ignored (1). Therefore the percentage shown for FramaC amounts to ~89% for critical defect types.

These figures just give an impression on the distribution of supported defect types per tool over criticality w.r.t. the overall number they are supporting. They should not be used for direct comparison of tools.

Tab. VII-3 and *Tab. VII-4* show the profiles regarding the standard defect types for both applications and the full set and the subset. For Application 1 more critical defect types are covered than for Application 2. Vice versa, it is for criticality "warning".

The figures also show that the spectrum of the subset is not representative for the full set. This result led to a reconsideration of the selection process based on functions performed prior to report analysis. Due to automation of the process it shall be possible in future to select samples by defect types according to the overall profile.

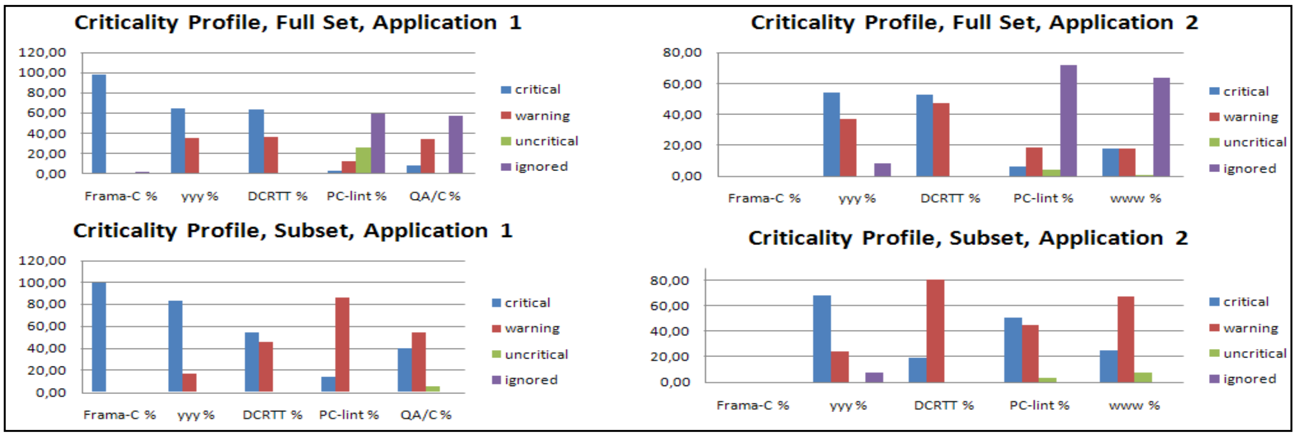
Tab. VII-5 and Tab. VII-6 compare the distribution of TPs between the tool and the state criterion. It is obvious that some defect types remain at nearly the same amount, while others disappear for the state criterion.

Please note that the decisions derived for the state criterion were based on different interpretations of the state criterion by the evaluators as discussed below.

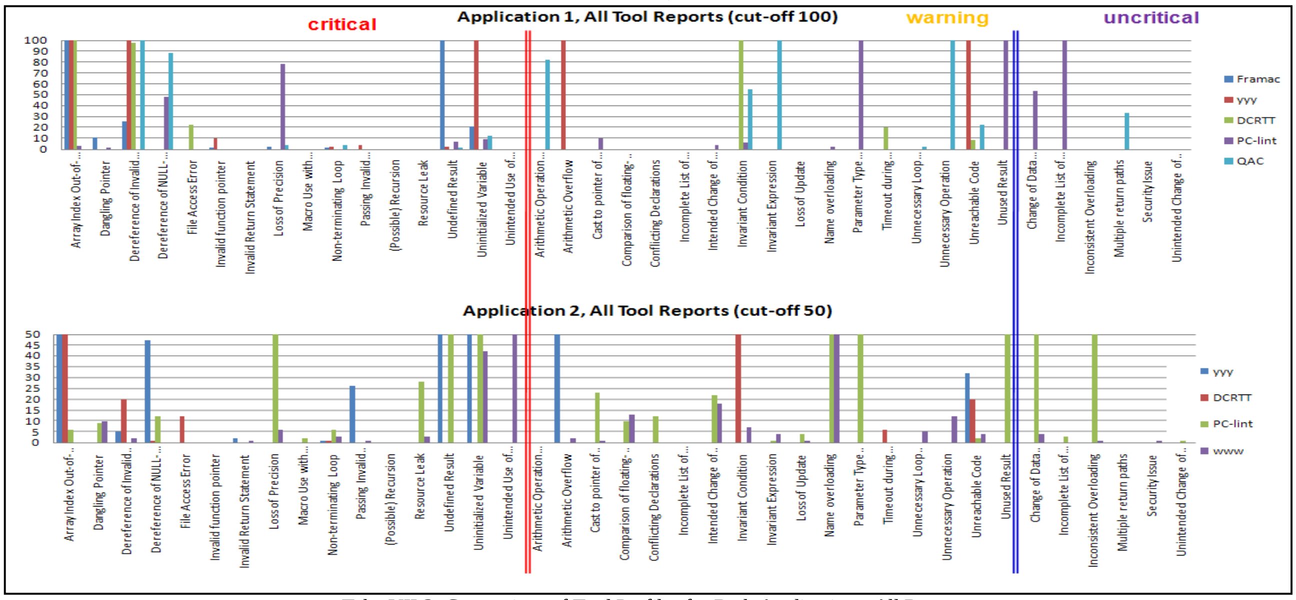
Tab. VII-7 gives average figures on the four different transitions for both applications. As an FP for the tool criterion implies an FP for the state criterion, a transition FP/tool⇒TP/state is not possible.

The most interesting transition from a developer's point of view – worrying about unjustified tool reports – is TP/tool⇒FP/state, which is highlighted in yellow colour. While the percentage for TP/tool⇒FP/state is nearly the same for both applications, it is quite different for the two remaining transitions.

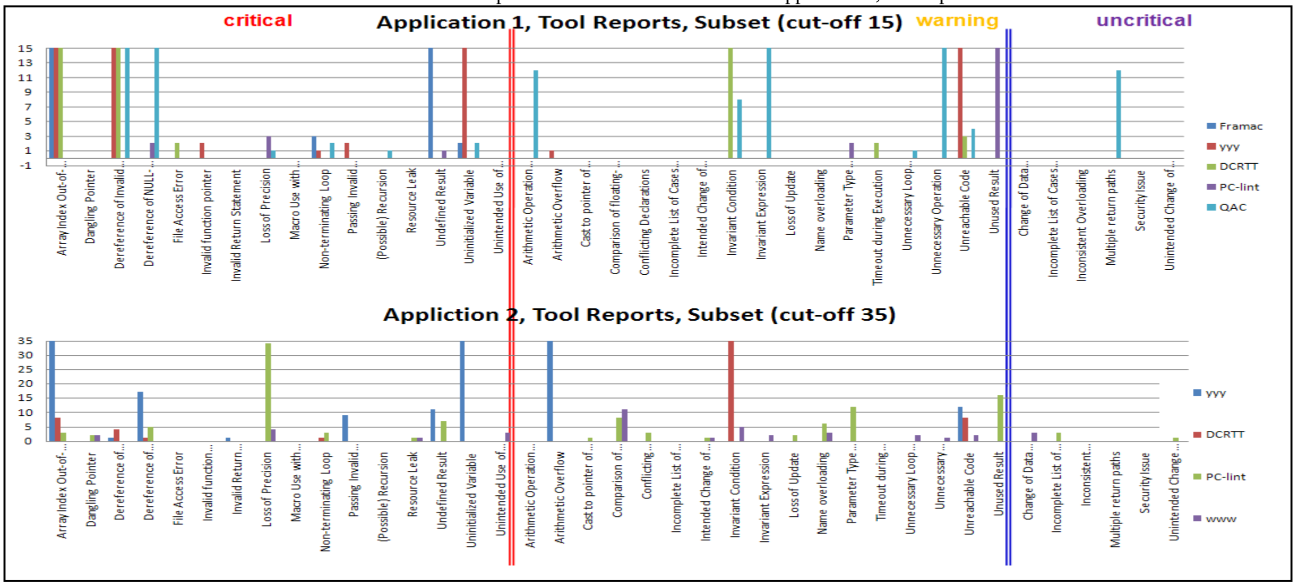
The reason for the big differences needs further investigation. As already mentioned, this may be a matter of individual interpretation, but it may also depend on the application. *Tab. VII-8* gives an impression on the broad range of individual decisions, ranging from about 13% to 80% for TP/tool⇒FP/state.



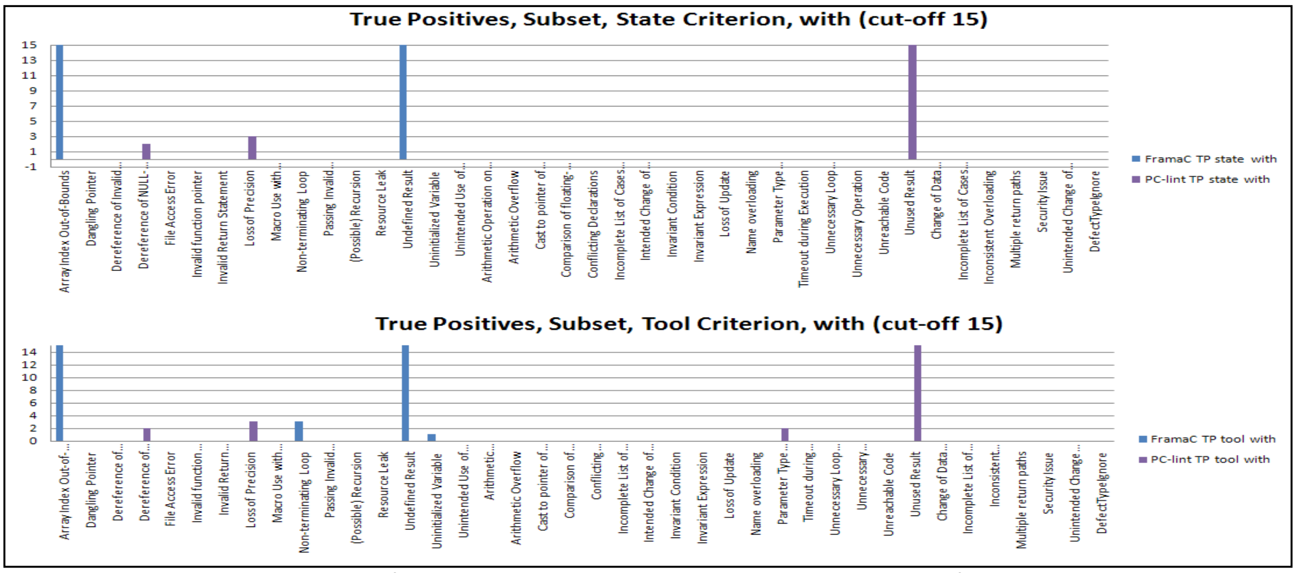
Tab. VII-2: Criticality Profiles of Tools



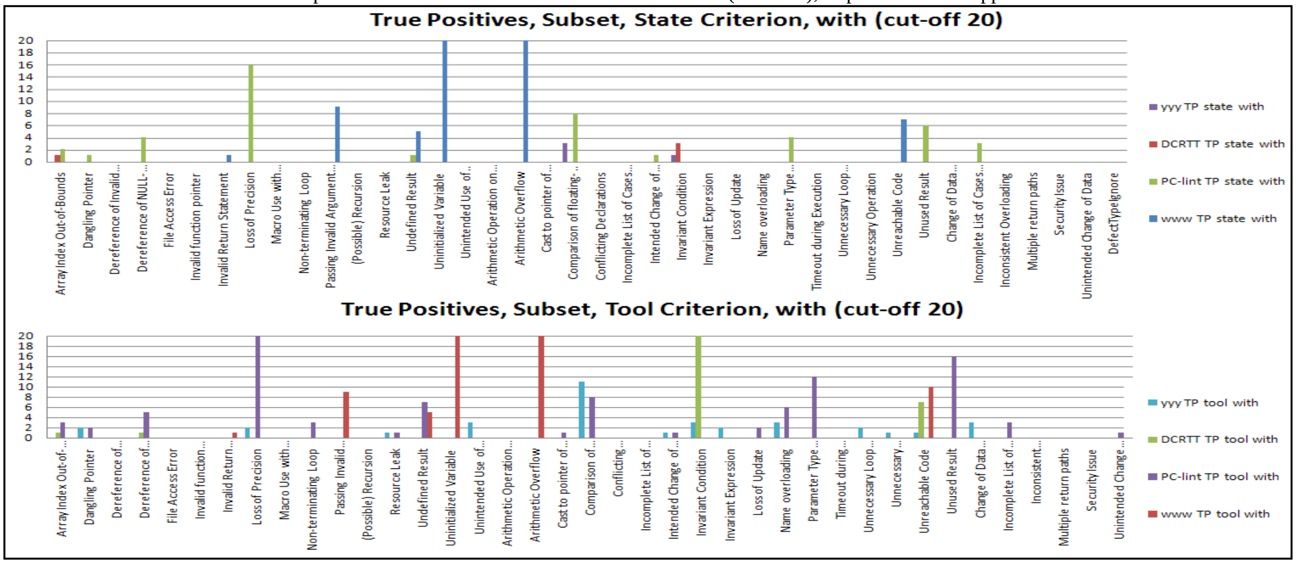
Tab. VII-3: Comparison of Tool Profiles for Both Applications, All Reports



Tab. VII-4: Comparison of Tool Profiles for Both Applications, Report Subset



Tab. VII-5: Comparison of True Positives and Tool/State Criterion (with ctxt), Report Subset of Application 1



Tab. VII-6: Comparison of True Positives and Tool/State Criterion (with ctxt), Report Subset of Application 2

Application 1				
Criterion Transition	**With**	**With %**	**W/O**	**W/O %**
Tool TP / State TP	81	34.32	83	39.34
Tool FP / State TP (impossible)	0	0.00	0	0.00
Tool TP / State FP	7	2.97	7	3.32
Tool FP / State FP	148	62.71	121	57.35
Total	236	100.00	211	100.00
Application 2				
Criterion Transition	**With**	**With %**	**W/O**	**W/O %**
Tool TP / State TP	112	39.30	123	43.16
Tool FP / State TP (impossible)	0	0.00	0	0.00
Tool TP / State FP	137	48.07	102	35.79
Tool FP / State FP	36	12.63	60	21.05
Total	285	100.00	285	100.00

Tab. VII-7: TransitionsTool⇒State, Average, Both Applications

Tool	Transition	With	With %	WO	WO %
yyy	Tool TP / State TP	63	69.23	63	68.48
	Tool FP / State TP	0	0.00	0	0.00
	Tool TP / State FP	12	13.19	12	13.04
	Tool FP / State FP	16	17.58	17	18.48
	Total	91	100.00	92	100.00
DCRTT	Tool TP / State TP	4	6.45	10	15.87
	Tool FP / State TP	0	0.00	0	0.00
	Tool TP / State FP	44	70.97	16	25.40
	Tool FP / State FP	14	22.58	37	58.73
	Total	62	100.00	63	100.00
PC-lint	Tool TP / State TP	42	42.42	47	47.96
	Tool FP / State TP	0	0.00	0	0.00
	Tool TP / State FP	54	54.55	48	48.98
	Tool FP / State FP	3	3.03	3	3.06
	Total	99	100.00	98	100.00
www	Tool TP / State TP	3	9.09	3	9.38
	Tool FP / State TP	0	0.00	0	0.00
	Tool TP / State FP	27	81.82	26	81.25
	Tool FP / State FP	3	9.09	3	9.38
	Total	33	100.00	32	100.00

Tab. VII-8: TransitionsTool⇒State, tool-specific, Application 2

In order to understand what the reasons, the evaluators were asked on details of their decision. While in case of FramaC only the “while(1)”-case was considered to turn a TP/tool to an FP/state, in case of www an extended context was considered, based on the knowledge that in the application pointers or data are initialized in a context, not visible to a tool.

This feedback indicates a need to refine the definition of the state criterion. However, a deeper analysis of the data – not presented and discussed here – suggests that both criteria still do not cover extreme cases of reports, which do invalidate the overall evaluation if they occur at a high rate compared to other reasonable reports.

C. *Uniqueness and Complementarity of Tools*

In the previous study the possibility to increase sensitivity due to combination of two tools was demonstrated. Due to questionable reports which for the time being cannot be removed from the criticality categories “critical” and “warning” because this would have to be done manually case-by-case, the results for sensitivity would be questionable, too.

The current conclusion on this dilemma is that possibly an earlier separation on the level of tool defect types may help, i.e. to map questionable defect types immediately into the ignored-group. Later, having mapped them already on standard defect types, it is not possible, because also reasonable reports would be moved, too. As removing reports is a very sensitive decision regarding evaluation and comparison of tools, a deeper and more careful consideration is required.

Appl.	Number of Tools	Coincidences
1	0	0
	1	407
	2	61
	3	33
	4	4
2	0	0
	1	548
	2	11
	3	0
	4	0

Tab. VII-9: Coincidence Profile for Both Applications

The content of Tab. VII-9 may help to understand the issue. While in case of Application 1 about 25% of the reports are shared with 2 or more tools, for Application 2 the equivalent figure is less than 2%, i.e. the difference amounts to about one order of magnitude.

In consequence, in case of Application 1 about 75% of reports are unique contributions of a tool, while the equivalent figure for Application 2 is about 98%.

This latter figure suggests (and this is confirmed by other data not shown here), that most of the many unique contributions may not be considered as useful, supposing that a higher percentage of reasonable reports should be shared.

Tab. VII-10 provides the list of observed combinations for which tools share the same report, i.e. they report the same standard defect type for the same file and line.

While in case of Application 1 up to 4 tools shared a report, the equivalent figures amounts to 2 tools only, in a very few cases.

Appl.	Cnt	%	Unique Contributions and Tool Combinations
1	4	0.79	FramaC yyy
	2	0.40	FramaC yyy QAC
	29	5.74	FramaC
	4	0.79	FramaC yyy DCRTT QAC
	7	1.39	FramaC yyy DCRTT
	1	0.20	FramaC DCRTT QAC
	4	0.79	FramaC DCRTT
	23	4.55	yyy DCRTT
	23	4.55	yyy DCRTT QAC
	79	15.64	yyy
	18	3.56	yyy QAC
	92	18.22	DCRTT
	12	2.38	DCRTT QAC
	43	8.51	PC-lint
	164	32.48	QAC
	505	**100.00**	**Total**
2	9	1.61	yyy DCRTT
	362	64.76	yyy
	56	10.02	DCRTT
	1	0.18	DCRTT www
	98	17.53	PC-lint
	1	0.18	PC-lint www
	32	5.72	www
	559	**100.00**	**Total**

Tab. VII-10: List of Tool Coincidences

D. *Unit Tests vs. Tool Reports*

In order to compare the impact of unit tests to the results of analyses regarding defect detection, both data streams have been synchronized using filename and line number.

Coverage information per line together with additional information on observed exceptions or defects found is shown for unit testing. In the unit test block at the bottom of

Fig. VII-1 three are three rows:

- The lowest row represents the line type: gray/normal or black/conditional expression.
- The middle row indicates the coverage: red/no coverage, green/full coverage, blue/false covered, yellow/true covered.
- The upper row indicates whether an exception occurred or a fault was detected: red/exception, magenta/fault.

Then the 5 tools follow bottom up with 4 traces each related to the 4 combinations of criteria and context.

From the analyses the false and true positives related to a line are shown for each of the 4 combinations resulting from with/without context and tool and state criterion using 3 colours:

- yellow/false positive
- red / true positive
- blue /true and false positive are reported for a line

This allows to seeing where tools reported FP or TP for a covered or non-covered line.

As in case of Application 1 all defects found during the unit tests were already fixed, neither defects nor exceptions occurred, and 4 exceptions in case of Application 2 (3 of them shown at the bottom trace of

Fig. VII-1).

Tab. VII-11 provides information on the unit tests for both applications. In addition, figures were added for robustness testing as done by DCRTT. While unit testing primarily addresses coverage and compliance with requirements, robustness testing aims to provoke activation of defects by a large number of test stimuli.

Tab. VII-12 provides figures on the distribution of TP as reported by the tools over covered and non-covered lines, for the 4 combinations of criteria and context cases and for both applications. Surprisingly, the probability to find a TP in a covered line is about two times higher than to find a TP in a non-covered line. This needs further investigation. It may be a matter of complexity of the code.

Also, more detailed figures on the defect profile regarding covered and non-covered lines should be derived to get a better understanding why the TP were not detected by a unit test.

Overview on Number of Unit Tests and Functions						
Test Mode	Application	Functions under Test	Performed Unit Tests	Test / Function	Average Coverage	
					stmt	cond
manuallly	1	368	954	2,59	94,14	89,63
	2	60	164	2,73	85,03	60,41
DCRTT	1	610	1042420	1708,89	87,35	95,78
	2	466	216348	464,27	76,89	76,08

Application	Files		Functions	
	total	affected by test	total	affected by test
1	39	25	610	368
2	40	24	557	60

Tab. VII-11: Overview on Figures of Unit and Robustness Testing

Appl.	Description	TP in non-covered lines	TP in covered lines	Total TP	% TP in non-covered / total TP	% TP in covered / total TP	TP per non-covered line	TP per covered line
1	tool / with ctxt	25	302	327	7.65	92.35	0.1656	0.3471
	tool / without ctxt	23	312	335	6.87	93.13	0.1523	0.3586
	state / with ctxt	7	104	111	6.31	93.69	0.0464	0.1195
	state / without ctxt	7	105	112	6.25	93.75	0.0464	0.1207
2	tool / with ctxt	36	233	269	13.38	86.62	0.1593	0.2852
	tool / without ctxt	33	215	248	13.31	86.69	0.1460	0.2632
	state / with ctxt	14	131	145	9.66	90.34	0.0619	0.1603
	state / without ctxt	15	145	160	9.38	90.63	0.0664	0.1775

Tab. VII-12: TP Reports vs. Coverage

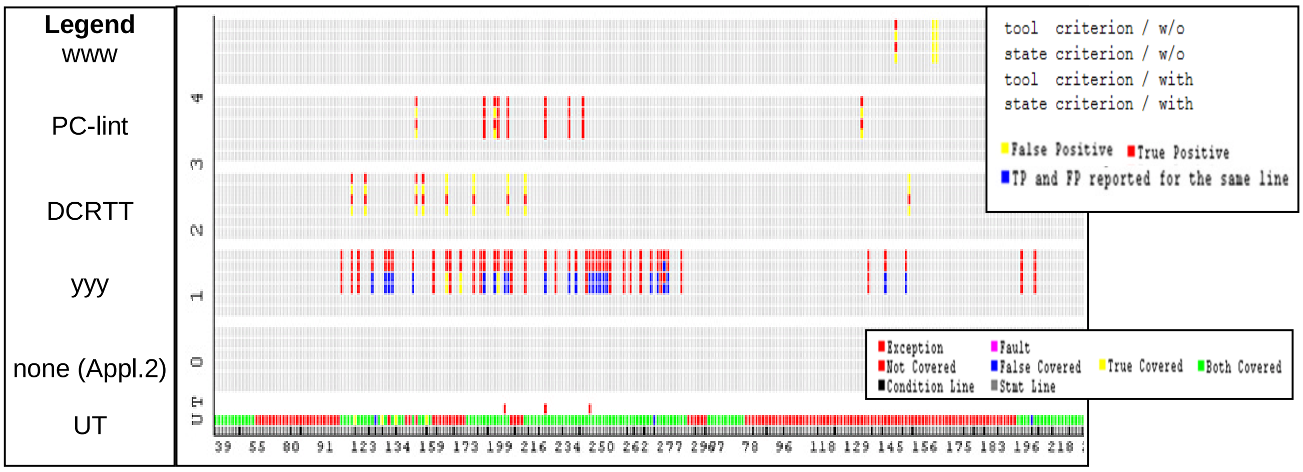


Fig. VII-1:Merge of Tool Reports with Coverage Information from Unit Testing

The presence of TP in covered lines leads to the conclusion that to a major degree unit tests and analyses are complementary. This applies to static and dynamic analysis.

As usually the goal of unit testing is to prove fulfilment of requirements (in a positive manner) – apart from requirements requesting fault injection – and the goal of analysis is to positively determine whether defects are present or not – valid for static and dynamic analysis, this result is not surprising.

Whether the absence of defects for lines for which the tools issued reports while no defect was detected or left as a result of unit testing, indicates an overhead induced by tools needs to be subject of further investigations.

A user of Application 1 reported that the application behaves quite stable. As some of the critical defects detected by analyses are related to the error handling parts, this seems to be reasonable. Also, it indicates the added value of analyses, pointing to critical locations not yet detected, compared to unit testing and verified-by-use as discussed in Ch. V.D.

VIII. Lessons Learned

Due to the additional application software with different programming styles and another programming language and due to the additional tools more issues had to be tackled to get a common view on the evaluated tools. Compared to the ESVW study the evaluation was highly challenging and several issues of evaluation could not be closed. In case of Application 2 a fair view on the tools could not be achieved due to the heterogeneity and – in part – poor quality of the reports due to a high number of trivial reports.

A. *Defect Types*

New defect types were found by the new tools, the new language and the new applications, because the list of defect types as outcome of the previous study did only reflect the status of Application 1 and the applied tools.

In fact, the number of defect types was doubled from 20 to 40 and it can be expected that it will grow further if the set of tools and the set of software applications will be extended, again.

The mapping between tool-specific messages and standard defect types had to be automated to be more flexible in the mapping process and to save effort. Automation of this step also supports a consistent redefinition of the mapping regarding already existing tool reports.

At the beginning – in ESVW – there was nearly a 1:1 mapping between tool-specific defect types and the standard defect type. Although the naming of defect types differed from tool to tool, only one tool-specific type was mapped on a standard defect type per tool in most cases. However, by the new tools this changed a lot.

B. *Evaluation Criteria*

An analysis of the obtained results yields that still the two applied criteria are not sufficient to get a clear and fair picture regarding the position of the tool supplier and the developers:

The tool criterion is an exact criterion regarding whether the tool report complies with the content of the source code.

However, it may also cover trivial cases, yielding TPs which may be considered as unjustified. Especially when many such reports are issued by one tool but not by others, a fair comparison is not possible. Counting only the TPs would give such a tool a significant advantage compared to a tool not reporting such a trivial case, or even intentionally reducing the amount of reports for such a case.

The state criterion attempts to exclude trivial cases of getting an unjustified TP, but it turned out that there is a wide range of interpretations by individuals possible.

This disqualifies the criterion for comparison of tools, unless it is guaranteed that all involved persons have the same understanding. In part, misinterpretations can be detected by conflicts during the consolidation phase. But if all persons came to the same wrong conclusion, only additional manual and thus costly checks can exclude such wrong decisions.

The principal issue is to accurately flag TPs which are critical regarding the system state. However, the current rules are not such conclusive that only reports are issued which are really critical in the given context.

Further, there are messages like “Loss of precision”, which in part cannot be avoided by a developer because they are related to the limitations of the representation of numbers in context of a computer, implying that

a. arithmetic operations cannot be interpreted in the classical mathematical sense, i.e. a result may exceed the range of the data type,
b. not all real numbers in mathematical sense can be represented as float or double,
c. not all float or double numbers can be represented as integers.

This weakness leads to a large number of reports which have to be classified as FPs at the end regarding the system state, while still a few ones may be TPs for both criteria. But the final conclusion can only be done manually.

In fact, high numbers of reports resulting in false positives effectively lead to FNs, because not all reports can be manually analysed, implying to miss reported TPs. This is a fact which has already been identified in the course of the ESVW study, so it is not really new, but this issue occurred again in this study.

Currently, three approaches (most probably non-exhaustive) have been identified which support reduction of the manual effort for such cases:

1. A tool reduces the number of reports related to the same origin of a defect.
2. A combination of tools support to reduce automatically the number of reports related to the same origin of a defect by comparison.
3. A developer does apply a programming style by which the number of reports resulting in FPs is reduced or even 0.

C. Tools

1) FramaC

The analysis with Frama-C required additional effort due to

- missing support for verification of top-level functions, (currently only one entry point can be specified),
- missing configuration support, and
- poor reporting capabilities at generation of a large unstructured data stream.

In order to make the Frama-C results comparable to the other tools already applied to Application 1, an artificial entry point was established manually, calling all top-level functions. To build such an entry point is non-trivial and required a lot of additional, unplanned effort, as the full environment for the call must be provided for each of the functions, implying declarations for every parameter of any such function. This task had to be performed manually.

There was no hint on what is the best set of configuration parameters regarding a certain size of an application. Starting with the highest precision and then cutting down in half the configuration parameters until a run terminated normally within a reasonable time, resulted in a number of non-terminating runs, starting with a duration of about a week, and then approaching step-by-step a configuration which terminated after some days. No indication on the achieved progress was provided during or the end of an aborted run.

The huge amount of unstructured reports required special handling. A list of reported defect types does not seem to exist in the documentation, especially regarding critical ones, and there is no conclusive description how the relevant reports can be filtered.

2) yyy

For Application 2 an increased number of reports classified as critical was generated. This was much more than expected according to the experience with this tool in context of Application 1, and compared to the other tools.

To reduce the manual effort the use of heuristic rules was considered to decide on TP or FP automatically. But the results are not considered as sufficiently reliable, currently. More experience is required to consider such results for a comparison with other tools.

The major origin of the occurrence of such a high number of reports is as follows: as it is C++ code the tool expects that all object member variables are initialized in the constructor. But in nearly all cases of Application 2 this is not true: initialization is done in separate functions, of which the call is not directly visible.

The tool vendor was contacted for clarification. The given recommendation is to apply the tool as early as possible during development, and to modify the code by the feedback from tool. Then the tool would report more conclusively.

After all, the results for Application 2 / Tool 2 cannot be considered to be representative.

3) DCRTT

DCRTT was already applied to a platform-independent subset of Application 2, and critical defects were fixed. This may have introduced bias regarding the found defects, but not only for DCRTT but also for tools addressing similar defect types.

DCRTT generated in case of Application 2 a significant number of reports on “unreachable code” and “invariant condition” which mainly result in an FP regarding the state criterion or in case of “w/o context”. This is a consequence of insufficient coverage due to either defects preventing to reach certain locations, or specific conditions difficult to fulfill with the stimulation methods employed by DCRTT, such as filling queues related to global pointers.

4) PC-lint

The following defect types were issued frequently for the full set of reports at criticality “critical” and “warning”. If not explicitly mentioned, all reports were classified as TP according to the tool criterion. But most of these reports can be considered to be trivial.

- Dereference of NULL
  Only a few ones, some shared with other tools.
- Loss of precision
  Most of the reports result in an FP regarding the state criterion.
- Cast to pointer of incompatible types
  Really useful and nearly only reported by PC-lint (1 report by Tool www)
- Name overloading
  Most of the reports result in an FP regarding the tool and state criterion.
- Parameter type mismatch
  All reports were classified as FP regarding the state criterion.
- Unused result
  Most of the reports result in an FP regarding the state criterion.
- Unused enum-literal in switch with default
  For 6 switches about 1300 reports were issued, formally TP according to the tool criterion, but in principal unjustified and therefore FP according to the state criterion as the default was present and no impact on system state or logic.

The reporting approach of PC-lint could be improved regarding minimization of the number of reports, e.g. in case of unused enum-literals. There are 4 levels for activation / deactivation of reports, but the levels do not match with the criticality levels defined in Tab. VI-3.

PC-lint allows deactivating some message types, but then TPs could also be deactivated, e.g. regarding loss of precision, depending on whether the operation is intended or not, because potential TP and FP cannot be clearly separated without taking the risk of dropping real TP.

5) www

This tool was applied to Application 2, only. Very few reports on invalid pointers were issued. Several FP in case of “Name overloading” were detected because it reported overloading between a parameter or local variable with a member of struct or class.

6) QA/C

In case of Application 1 this tool achieved the best values for sensitivity. Please refer to [1] for details.

IX. CONCLUSIONS

As suggested at the end of the previous study, the additional tools and software brought in new aspects which enriched the knowledge on verification tools and applications and led to a number of conclusions and suggestions on improvements for the verification process.

The experience from both studies confirms that the success of tool usage heavily depends on the chosen verification tools and their integration in the development cycle. Selection of suitable tools implies sufficient knowledge on their efficiency w.r.t. the software to be verified.

The diverging results from both studies regarding the efficiency of the tools indicate that more is needed than just to buy and apply a tool to succeed at the end.

A number of new standard defect types had to be added to the database, the criterion for tool evaluation had to be reconsidered and complemented. A higher degree of automation allowed considering all the reports issued by the tools and to derive profiles on defect types reported by the tools, for the whole application.

A disappointing result is that the quality of the data obtained by the new tools for the previous and the new application is rather poor compared to that one of the ESVW study. However, the principal issues making evaluation difficult were identified and give guidance towards improvement.

The spectrum of issued reports is rather broad: the number of reports varies from 800 to 12.000 for full Application 2, and from 40 to 500 for the chosen subset. The value of the reports varies from trivial to of high value.

A major result is that the current evaluation criteria need to be improved and also the verifiability of the application has to be considered. Optimization of the whole verification process requires consideration of the defect identification and reporting capabilities of a tool and the degree to which an application supports the analyses. In addition to the evaluation criteria for a tool, also a metric should be defined characterizing the verifiability of an application.

If many reports result in FP this may not necessarily imply a weakness of a tool. It may also be an indicator for potential improvements in the application. A lot of FP could be avoided when considering the tool reports in a constructive manner to improve the quality of the code and not in a destructive manner only causing overhead. The more FP are issued the higher is the probability of missing critical reports. This should motivate to tune the code of an application.

A tool may be sensitive to certain defects. Their occurrence may block the tool to report more critical defects, or may cause an explosion regarding the number of reports. This observation confirms previous conclusions on this issue, demanding to apply a tool as early as possible in the coding phase to prevent by the given feedback that defects are multiplied in the course of development.

Many reports should not necessarily be the ultimate goal of a tool, but a minimized set of reports highlighting the essential issues for the given context, also implying minimized manual analysis effort. Duplication of reports and multiple reports with different defect types on the same origin of a defect should not be considered as of advantage for a tool.

The comparison of results from analyses and from unit testing confirmed the expectation that both verification types are complementary to a high degree. This is mainly a matter of the verification goals. While the main goal of unit testing is to demonstrate compliance between implementation and requirements in order to get acceptance, the main goal of verification tools is to demonstrate that defects still are present. Also, the results show that a high coverage figure as a result of unit testing does not necessarily imply that tools will not find defects any more in such lines, even if all defects found during unit testing have been fixed.

The experience achieved in the course of this study, but also parts of the automated process chain could be reused in real projects, regarding

- definition of the issues of verification and the contents of the verification plan,
- classification and priorization of reports,
- harmonization of report streams coming from different tools, thereby reducing the amount of reports to be manually analysed, and easing the use of more than one tool,
- derivation of figures on distribution of defects across the application.

ACKNOWLEDGMENTS

The results presented above are an outcome of contract DLR-50PS1606 of the Space Administration of the German Aerospace Center (DLR) on behalf of the German Ministry of Economics and Energy (BMWi).

We thank our colleagues Hans-Juergen Herpel, Mladen Kerep (Airbus Defence and Space, Friedrichshafen, Germany) and Anton Fischer and Mario Pinto (etamax space GmbH, Brunswik, Germany), as well as Thomas Boll (BSSE) for their highly valuable contribution in the study.

The authors also thank the suppliers of tools 2 (yyy) and 5 (www) who provided evaluation licenses for their tools.

Also, we would like to thank Sabine Philipp-May (DLR) for her valuable suggestions and contributions to the discussions in the course of the project.

REFERENCES

[1] Ch.R.Prause, R.Gerlich, R.Gerlich, A.Fischer: „Early Results from Characterizing Verification Tools Through Coding Error Candidates Reported in Space Flight Software", Eurospace Symposium DASIA'16 "Data Systems in Aerospace", 10 – 12 May, 2016, Tallinn, Estonia

[2] VectorCAST, https://www.vectorcast.com

[3] cppunit, https://de.wikipedia.org/wiki/CppUnit, cppunit.sourceforge.net

[4] gcov, https://gcc.gnu.org/onlinedocs/gcc/Gcov.html