Early Results from Characterizing Verification Tools through Coding Error Candidates Reported in Space Flight Software

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

Anton Fischer, Mário Pinto
etamax space GmbH
Braunschweig, Germany
e-mail: A.Fischer@etamax.de, m.pinto@etamax.de

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

***Abstract*—Six software verification tools have been applied to space flight software and the findings reported by each tool have been compared in order to derive footprints of the tools regarding capabilities of fault identification. Currently available results are provided in this paper: sensitivity and precision of individual tools and combinations of pairs of tools out of the set. A reader should bear in mind that the results as presented here depend on the spectrum of fault types as present in the reference software and on the configuration of tools towards real defects and fault types which are of interest for embedded systems and space flight software.**

Keywords: tool footprints, verification tools, false positives, false negatives, software faults, fault identification, fault coverage, fault report evaluation, software verification, verification efficiency, software verification plan

I. INTRODUCTION

In [1] the basics of tool characterization regarding fault identification were defined and discussed.

In this paper we provide early evaluation results having applied the 6 tools to the chosen flight software and having recorded and assessed the tools' reports.

The evaluation was performed in the context of software which can be characterized as embedded software with

- real-time operations,
- file system operations,
- use of communication channels such as a bus, and
- direct hardware interfaces, e.g., to sensors and actuators.

The tools were selected according to the following analysis methods and capabilities for automated analysis of software:

- symbolic execution,
- abstract interpretation, and
- automated testing.

The paper is structured in the following manner:

Sect. II provides a definition of the terms being part of the analysis and evaluation process. Sect. III addresses lessons learned. In Sect. IV the evaluation approach is described and the evaluated tools are characterized. The results of evaluation are provided in Sect. V. Final conclusions are given in Sect.VI.

II. DEFINITION OF TERMS

A. Defect, Fault, Error, Failure

To get a clear understanding on the terms used later we repeat briefly definitions already explained in [1].

In the context of this paper a message issued by a tool on a defect is called a "report".

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.

In this terminology, faults are considered as a subset of defects. A fault may cause an undesired, observable behavior of a system. A defect, which is not a fault, will not, but it still addresses issues to be considered[1].

Note that an error can be encountered either while abstractly reasoning about the software, e.g. in the context of the virtual semantics of a programming language, or during actual execution.

From these definitions the following chain of causality results as shown in Fig.II-1.

Term	Scope
Fault	Mistake in code
⇓	⇓
Error	Bad state of a system
⇓	⇓
Failure	Unexpected observed behaviour

Fig.II-1: Causality Chain Fault, Error, Failure

The term *fault coverage* describes the degree to which defects present in the software are or were detected, or are detectable in the course of the defined verification process. It is usually represented by the ratio of the number of defects recognised and the number of defects present.

Be aware that the number of defects present is an unknown figure. Therefore, a percentage cannot be derived, in principal, but approximated only.

[1] As the term "fault" is widely used, e.g. in context of "fault coverage", this term is kept, although "defect coverage" would be the proper term following the terminology of this paper. Similarly, „criteria for fault identification" need to be re-considered as "criteria for defect identification", and "fault types" as "defect types".

The term *code coverage* describes the degree to which the code of the software under test is or was executed during test. In the simple case, it is represented by the ratio of the number of code instructions executed and the total number of instructions. There are different kinds of code coverage such as statement coverage, block coverage, decision or branch coverage, or Modified Conditional/Decision Coverage (MC/DC).

B. Tool Reports

A report is a message issued by a tool on a supposed defect found in the software according to its defect identification approach.

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-2).

		Code	
		Defect present	*Defect NOT present*
Result	*Defect Reported*	true positive	false positive
	Defect NOT reported	false negative	true negative

Fig.II-2: Classification of Tool Reports on Defects

The following abbreviations are used for the 4 cases:

TP true positive, FP false positive
TN true negative, FN false negative

Sensitivity or *recall* – as used in information retrieval theory – is defined as the quotient TP / (TP+FN). *Fault coverage* can now be expressed by the sensitivity as the ratio of *true positives* and the sum of *true positives* and *false negatives*.

Precision is defined as the quotient TP / (TP+FP) and represents the portion of reported defects that are actual defects.

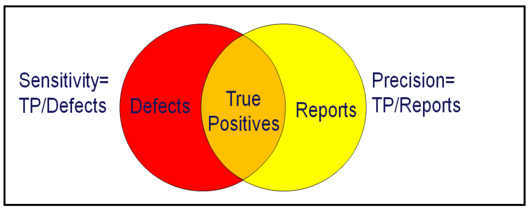


Fig. II-3: Sensitivity and Precision

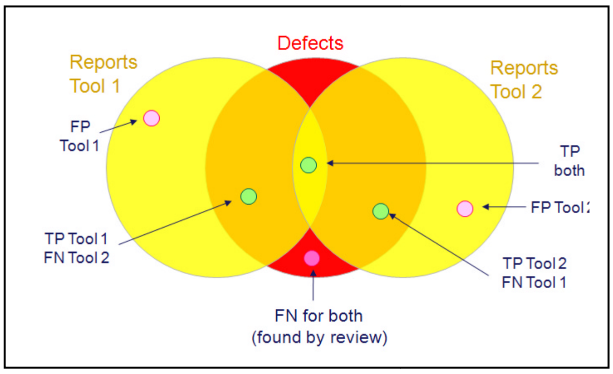


Fig. II-4: TP, FP and FN Examples for Two Tools

In practice, the number of defects is not known, only the subset spawned by the true positives found. Therefore sensitivity can only be approximated by taking the number of true positives found by all tools or by analysis, which is an upper bound of the real sensitivity (Fig. II-4). Fig. II-4 illustrates how FNs can be detected when applying more than one tool – or not.

C. The Evaluation Process

In a first step every tool was applied to the software and the reports were extracted. Then in a second step all reports were merged into a single stream, correlating reports from different tools about the same alleged defect (Fig. II-5).

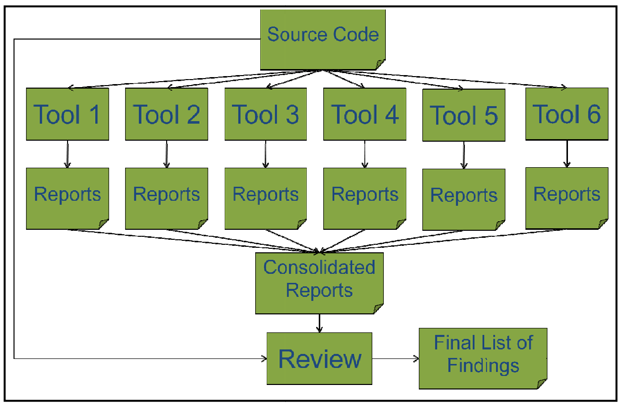


Fig. II-5: The Evaluation Process

The combined reports in this consolidated list were then analysed and classified as either true or false positives. This analysis was executed twice for each report. In one case, caller context was not considered ("without context"), while in the other case, caller context within the available code base was taken into account ("with context"). The details of these two analysis approaches are discussed in the following Sect. II.D.

D. Context

The possibility of a defect resulting in an error depends on the set of possible values for variables and parameters at the location of the defect.

This set may be constrained to an actual subset of the values allowed by the respective variable and parameter types due to initial conditions (e.g. values of global variables and parameters), conditional execution or operations yielding only a specific set of outcomes.

Thus, while an error may be possible in one context, it may be impossible in another context. As a consequence, a report about a possible error may be considered a false positive or a true positive depending on the context considered.

This also means that the performance of a tool in any given situation may strongly depend on its strategy regarding the context considered.

For example, tools may differ in whether they consider a function in the context of calls present in the code or only stand-alone. In the latter case, there are usually less constraints on the set of values possible at a given point, possibly leading to more reports.

Values of parameters and global variables may be further constrained artificially, e.g. by pre-defining allowed ranges which are real subsets of the ranges allowed by their formal types.

Initialization of global variables may be assumed to establish fixed initial conditions for their values, or they may be considered to be modifiable, including their full or pre-constraint value set in the set of values examined.

Even global variables declared to be constant may be considered variable, e.g. when analyzing the sensitivity to changes in these initial values representing possible differences in configuration of the software.

A variety of possibilities does exist for the assumptions a tool makes on the context when assessing the fault potential of a piece of code.

The assessment whether a finding is a TP or FP depends on the context considered. Therefore two principal cases were applied for assessment:

a. consideration of the function containing the alleged defect as standalone, allowing the full range of values for parameters and modifiable global variables ("without context"), and
b. consideration of the function within the context set up by its callers ("with context").

In the latter case, the whole call tree up to the main entry point into the software is considered, if necessary.

E. Fault Identification Strategies

Two different views on fault identification exist:

- the system view, and
- the library view or view of module testing.

The system view requires considering the context as it is formed during execution of the system. It avoids reports about pieces of code with fault potential which cannot be activated in context of the current configuration of the source code.

This implies intentionally ignoring faults because they cannot compromise system operations in the current version of the system. This approach also implies minimization of analysis and maintenance effort.

In contrast, the library view does consider any function as standalone. If a function may be put in any context, no assumption on the context would be valid. As the term is indicating this happens for a function of a library. It is also true in case of module testing, when interface functions are tested independently, intentionally ignoring the system context. This approach leads to a higher rate of detected defects, but may imply later a reduced effort for maintenance.

F. Analysis Approaches Applied

As a consequence of these two views, a tool may benefit if a report results in a TP depending on the conditions of the chosen assessment type (context or not), or it may be depreciated if its reports are classified as FN or FP, while the reason for upgrading or downgrading are not really visible for the reader of the final defect classification.

Therefore it has been decided to derive two sets of results, based on consideration of the context (again, whatever it is for a given tool) or not.

G. Analysis Approaches by Tools

The context as described in Sect. II.D is not the only aspect which may result in a disadvantage or an advantage for a tool. Another aspect is the dynamic modification of the context depending on previous faults or defects found.

This aspect shall be explained by Fig. II-6.

No context shall be considered here, i.e. the full spectrum of parameter values may be possible. The focus shall be put on the length of *src* and *dest* (excluding for sake of simplicity consideration of possible NULL-pointers).

```
void myFunc(char *dest, const char *src) {
  dest[3]=src[0];
  dest[2]=src[1];
  dest[1]=src[2];
  dest[0]=src[3];
  return;
}
```

Fig. II-6: Dynamic Modification of Assumptions on Context

Due to the missing context the lengths of *src* and *dest* are suspicious. Presumably, a tool should issue a report for any index>0 (if NULL is excluded, assuming a minimum length of 1 could be acceptable, although even a non-NULL-pointer may point to non-allocated memory).

During the evaluation one tool did issue a report for the first access *dest*[3], but not for *dest*[2] and *dest*[1], while the other tools did.

According to the process description in Sect. B, the lack of reports for the latter two would be counted as false negatives towards the first tool, thereby depreciating it in terms of sensitivity in relation to the other tools.

The other tools are not necessarily wrong: Although the result of the operation writing to *dest*[3] is undefined according to the C-Standard [2], there is no guarantee that the access will lead to abortion of execution.

The first tool is not necessarily wrong either: When it detects that index 3 may be invalid, it produces a report and discards the erroneous case for further analysis. In consequence, when it reaches the line with `dest[2]` it has made the assumption that the previous code is correct, i.e. that `dest` has 4 elements at least. Therefore no fault is flagged in the following lines as the indices 0 – 2 cannot be invalid under this assumption.

Another example of a defect where its status as FP or FN is debatable is shown in Fig. II-7.

```
myFunc(&myPtr[ind])      original source code

myFunc(myPtr+ind)        expression expanded by compiler
```

Fig. II-7: Views on Pointer Dereferencing

Although the expression used as parameter to `myFunc` in the first line includes an array subscript expression, the array is not actually accessed. The address operator ('&') indicates that only an address calculation operation is executed [2]. The functionally equivalent expression is shown in the second line.

Such an address calculation cannot directly lead to a memory access violation. That violation would only occur when the resulting address is actually accessed. Thus, it may be considered valid for a tool not to report a fault at this location but rather at the code location where the actual access occurs (here: inside `myFunc`). A report at this location could therefore be considered a false positive, as no memory access occurs.

However, the semantics of the additive operators ('+', '-') in context of pointer arithmetic state that unless the resulting address falls within the object pointed to by the base pointer (here: `myPtr`) or one past its end, the result of the address calculation itself is undefined.

Thus, from the perspective of the language standard, both expressions shown in the example would be faulty if the index would be allowed to be out of range.

As a consequence, it could be argued that failure to report this at the location of the address calculation implies a false negative.

Such aspects have to be considered when comparing tool reports in order to get a fair evaluation.

Of course, reporting the issue only at the location where the pointer is dereferenced may make it more difficult to detect the root cause of the error, which is ultimately the invalid address calculation at the point of call.

H. Analysis Methods

The analysis methods as applied by the selected tools are briefly discussed here.

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.

The immediate result of symbolic execution is a set of assignments and conditions in terms of the original input which represents the conditions under which the analysed path will be executed in the real system.

An analysis tool can use this information to determine whether a given condition can be fulfilled at the end of the given path. Such a condition could be, e.g., the presence of a NULL-pointer dereference or a division by zero.

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 false negatives.

Abstract Interpretation[4] 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. The set of possible program behaviours is conservatively approximated, i.e. all possible behaviours are included in the approximation, but not all behaviours included in the approximation are possible.

Thus, if a given (faulty) behaviour is not included in the approximation, the program does not have information on the behaviour. Thus it is possible to prove the absence of faults. The converse is, however, not possible. As a consequence, the method may produce false positives.

By introducing an additional optimistic approximation, some of the reports may be automatically pre-determined to be true positives. The optimistic approximation represents a subset of the possible behaviours of the program. Thus, if a faulty behaviour is present in the optimistic approximation, the presence of the fault in the program can be proven.

For *Automated Testing* the software is actually executed, either on the target, on a simulated target or in a version ported to a host platform. 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 to the software – either because that set is infinite or too large for practical consideration – the method may miss present defects, leading to false negatives.

However, any input that leads to an error occurring in the software is a witness for the presence of the respective fault in the code. False positives are only possible if the representativity of the test platform is not ensured. Some reports may be considered irrelevant in context (see Sect. D).

I. Tool Configuration

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

The selection of a proper set is driven by the following goals:

1. the reports shall be related to the defect types of interest, none such report shall be suppressed,
2. reports not of interest shall be suppressed, to reduce the evaluation effort and to avoid that reports of interest are not visible within a large set of issued reports.

As a remark to (2):

One of the tools is very strong in detecting non-conformances regarding lexical guidelines and standards. It supports a large variety of such checks. With all options activated about 95.000 reports were issued for 42 KLOC, while only a very small part of this set was related to non-lexical, dynamic issues. Therefore all options related to lexical non-conformances were deactivated.

J. Use of a Fault Database

In the course of the project a fault database was used to investigate the behaviour of tools on concrete examples, if a tool report was not sufficiently understood. When required, existing examples were varied to get different responses from the tool.

The fault database provides source code for about 100 defect types which were identified in the course of defect analysis in space flight software in the past years. It also contains counter-examples, i.e. the respective corrected source code, for a number of defect types, to assess whether no defect is reported in the correct case. The decision for providing a counter-example depends on some (felt) uncertainty whether a tool does really only report in case of a defect. The example described in Sect. III.A is part of this database. More examples follow in Sect. III to explain some observations by anonymized code.

III. Lessons Learned

A number of lessons had to be learned on the defect identification mechanisms and the reporting approaches of the tools.

To one part such lessons are related to unexpected behaviour of tools, regarding stability of results, validity of reported code coverage, and expressiveness of reported defect location.

To the other part it is a matter of the principal difference between getting a report and checking whether it is of relevance or not, and comparison of reports from different tools and classification of received reports as TP or FP and missing reports as FN.

Some of these issues are discussed below.

A. *Conflicting Tool Conclusions*

When possible, the depth of the analysis was varied. This led to an unexpected result in case of an example from the fault database, which represents a typical defect related to a mix of signed and unsigned expressions. *msgLen* is of type *signed char* and the signed bit is set., which is expanded in the call to 32bit. The resulting value is interpreted by *memcpy* as 4GB-128. One report on the validity of a memory range out of three reports (2 on memory range, 1 on initialization) is wrong in one case, It is in conflict with the other (correct) report on memory range, although the context is identical.

```
#define ESVW_MAXLEN 128
typedef struct TyESVWMsg23 {
  char msgLen;
  char msgData[ESVW_MAXLEN];
} TyESVWMsg23;
TyESVWMsg23 ESVWtheMsg23={128,{0}};
char ESVW_23_buf   [ESVW_MAXLEN];
memcpy(ESVW_23_buf,
        &ESVWtheMsg23.msgData[0],
        ESVWtheMsg23.msgLen);
```

Low:
Destination may be **out-of-range** of the area given by size
Source may be **out-of-range** of the area given by size
Size is correctly initialized

Medium to high:
Destination may be **out-of-range** of the area given by size
Source **is** in the area given by size
Size is correctly initialized

Fig. III-1: Unexpected Result Dependency on Analysis Depth

A systematic investigation over the verification levels yielded the results shown in Fig. III-2. Surprisingly, the report is correct for lower verification levels, but wrong for higher levels.

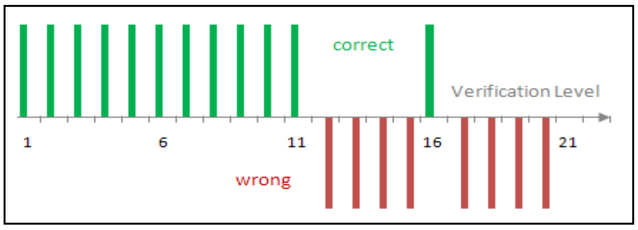


Fig. III-2: Occurrence of Correct and Wrong Reports vs. Verification Levels

B. *Classification of Criticality*

A tool may classify reports already as highly or less critical. But the classification details usually differ from tool to tool. Then good knowledge on the analysis and classification approach of the tool is required to fairly compare the reports.

Reports of higher criticality can be prioritized if a tool already does classification of criticality inherently. However, it happened twice that defect types of medium criticality were put in a set of uncritical defect types by a tool.

Excluding such sets which are considered as obviously uncritical, will result in FNs for this tool regarding defects which are expected in another set.

Also, the classification for either highest criticality (critical, error) or medium criticality (warning) may depend on the programming style and/or the chosen configuration options.

In case of the code shown in Fig. III-1 the defect related to the third argument may be reported as critical / error or warning.

If the contents of global data may be varied over the full range, the classification is “warning”, as the critical value 128 (or -128) of `msgLen` in `TyESVWMsg23` is only one out of 256 possible values.

If not, then the defect is classified as “error” / “critical”as it will always occur because no other value than the one of the initializer will be considered.

If the contents may be varied, but `const` is added to the declaration of `msgLen` in `TyESVWmsg23`, then the classification is “error” / “critical” again, because global data are still varied, but not if being declared `const`.

In consequence, the programming style and configuration options must be considered for the classification of a defect inherently performed by a tool.

C. *Percentage Figures Based on an Unknown Reference*

A tool may provide summary figures, also in terms of percentage. Then the essential question is, what is the reference figure?

We compared two cases with and without defects for the same function while preserving the code structure.

For the correct version 62 items are reported as reference figure (whatever an item is, possibly an applied check), but the reported reference figure for the faulty is only 38.

In case of the faulty version 36 items were considered as proven, 62 for the correct version. Therefore the report for the wrong case suggests that 94.7% of the checks were proven, referring to 38 checks. The report for the correct case referred to 62 proven checks.

For the wrong case the true percentage of checks with positive result is 58.1% only instead of 94.7% as reported.

In consequence, as far as defects are present reported figures may be misleading.

D. Deferred Reports

A report may not be issued at a location where it could be expected as already explained in Fig. II-7 of Sect. II.G. A more detailed example is shown in Fig. III-3. A report would be expected for *&buffer[start]* in the call of `read32`.

A number of such cases were observed and considered for comparison of tool reports.

```
errorCode_t enableMonitoring( const byte_t * buffer,
      const uint32_t buffer_size, enableMonitoring_t * tc ) {
   uint32_t start = 0;
   if ((buffer == NULL_POINTER) || (tc == NULL_POINTER))
     <error>
   else {
      if ( buffer_size < 1 )  // wrong check!!
        <error>
      else {
        tc->elemNo = buffer[start];
       start += 1;   // invariant expression !!
       upLim = tc->elemNo;
       if (upLim>PARA_MAX)
         upLim=PARA_MAX;
       ii=0;
       while (upLim > ii){                    // PARA_MAX elements
         read32(&buffer[start], &tc->para[ii].elem);  // buffer+start
         start += 4;
         ii++;
      }
   }
   return ret;
}
```

Performed Checks: Pointer is initialized , Local variable is initialized

Fig. III-3: Deferred Dereferencing of a Pointer

This example will also be used for further discussions.

E. TP or FP or None – Impact of Context

The context impacts a decision on the classification as TP or FP which will be explained by Fig. III-3.

If it is a library function which can be called in any context, then no assumption can be made on its parameters, in contrast to a function which is called from another function, possibly being a member of a call tree, defining a context, e.g. constraints on parameter ranges or pointer values.

If called on top-level – without context – then the values of its pointer parameters may become `NULL`. Therefore the error branch at the entry point of the function may be reached.

However, if the function is called within a context of which the source code can be analyzed and in all cases valid pointers are passed, e.g. pointers to an array, then the error handling part will never be reached, and a tool should report an invariant condition for the checks on `buffer` and `tc`.

The following, quite different conclusions for this case can be drawn regarding a potential report on an unreachable error handling branch:

If a tool issues

- a report "unreachable" while not considering context, it is a FP.
- a report "unreachable" while considering context, it is a TP.
- no report – meaning it is reachable – without considering context, it is a TN.
- no report – meaning it is reachable – while considering context it is a FN.

Considering the context usually requires significant effort for manual analysis of the constraints occurring along a call graph.

F. Grouping of Reports

Grouping may be applied by a tool to reduce the number of reports. Reports may be grouped when by a fix of one of the reported defects all (potential) defects (of that group) will disappear. An example for this case is given in Fig. III-4:

```
int myFunc(unsigned int ind, int arr [4] ) {
  int ret;
// if (ind<0 || ind>3) <error>  possible check
  if (arr[ind] == 0)  ret=0;
  else  if (arr[ind] == 1)  ret = 1;
  else  if (arr[ind] == 2)  ret = 2;
  else  if (arr[ind] == 3)  ret = 3;
  else ret = -1;
  return ret;
}
```

Fig. III-4: Grouping of Reports Possible

In this case an index check at the beginning – as shown as comment – will suppress all reports for the following index expressions: such reports may be grouped.

However, grouping is not possible for the example shown in Fig. III-5

All the reports on a potentially invalid index do not have a common origin. For every index expression an independent check is required to suppress a report.

```
int myFunc(int *arr) {
  int ret=-1, ind = 0, ii, jj, upLim;
  upLim = arr[0];
  jj = 0;
 for (ii==0;ii<upLim;ii++) {
   jj+=arr[jj];
   if (arr[jj] == 0) ret=0;
   jj+=arr[jj];
   if (arr[jj] == 1) ret=1;
   jj+=arr[jj];
   if (arr[jj] == 2) ret=2;
   jj+=arr[jj];
   if (arr[jj] == 3) ret=3;
  }
  return ret;
}
```

Fig. III-5: Grouping of Reports Not Possible

The exclusion of multiple reports as described in context of Fig. II-6 may also be considered as some kind of grouping, because a report is issued once only for a set of issues having the same root cause.

If one tool supports grouping and another one does not, the figures on TP, FN and FP may be not comparable. A tool issuing less reports may have a disadvantage when the suppressed reports may result in TPs. Vice versa, if they would result in FPs, it would get an advantage.

We tried to compensate such side effects by manual modification of the raw data, but decided at the end that the unmodified raw data should be more representative, because manual modification may introduce unfair conditions for one or the other tool, when trying to be fairer to a certain tool.

For completeness and to show the potential impact of grouping – affected by (partial) manual modification of raw data – we provide figures on the grouped and not grouped cases for some tables in Sect. V.

G. *File Contents*

As the application included file operations (`open`, `close`), examples in the fault database were established to explore the impact of file contents on defect identification directly.

None of the static analysers, but the test tool did consider the file contents. The missing context may lead to false positives or negatives.

H. *Defects Not Yet or Hardly Detectable by a Tool*

Two examples are provided referring to defects which cannot be detected by static analysis, because the wrong logic is not subject to their checks. In case of dynamic analysis invalid values may be detected. But it depends on the actual context.

Both cases refer to Fig. III-3. Therefore only code snippets are shown here.

1) *Wrong Check on Buffer Size*

The check on `buffer_size`

if (buffer_size < 1)

is not what is intended. This check shall ensure that the size of the total data stream in `buffer` is compliant with the number of elements extracted from `buffer[0]`. Therefore the correct check should be:

*if (buffer_size < (1+buffer[0]*4))*

The correlation between `buffer_size` and the contents of `buffer[0]` is not visible to a tool. What is only of relevance for the static analyzers is whether `buffer_size` may be less than 1, and 0 does fulfil this condition (as it is of type *unsigned* this is the only possible value).

In contrast, the test tool does have the information on the real size of `buffer`, and can issue a defect report if `buffer[start]` points to an invalid address.

However, in any case no tool can detect that the check does not do what is intended. The static analyzers can only highlight a potential risk, and the test tool can report an invalid index.

2) *Wrong Error Handling*

Fig. III-6, upper part, is an extension of the example shown in Fig. III-3. Instead of one loop for decoding of the data stream, two nested loops are executed. In this case the error handling mechanism of checking whether the upper limit `PARA_MAX` is exceeded and resetting upLim to the maximum value does not imply proper error handling for the inner loop.

Limiting the inner loop to `PARA_MAX` elements, while more elements will follow, implies that the outer loop will start its next cycle at the wrong offset. The best case is that decoding will fail in `read4byte_87` and an error is detected. But it is most likely – as in both cases the same structure is expected – that data are moved to the wrong location.

```
  tc->elemNo_2 = buf[ind];
  ind += 1;
  upLim2= tc->elemNo_2;
  if (upLim2>LENGTH_87)
    upLim2=LENGTH_87;
  idx2=0;
  while (upLim2 > idx2){
    read4Byte_86(&buf[ind], &tc->elem[idx2].elem0);
    ind += 4;
    tc->elem[idx2].elemNo_1 = buf[ind];
    ind += 1;
    upLim1= tc->elem[idx2].elemNo_1;
    if (upLim1>LENGTH_87)
      upLim1=LENGTH_87;
    idx1=0;
    while (upLim1 > idx1){
     read4Byte_87(&buf[ind], &tc->elem[idx2].elem_1[idx1].elem_1);
       ind += 4;
       indx1++;
     }
   }
  indx2++;
}
```

```
  upLim1= tc->elem[idx2].elemNo_1;
  idx1=0;
  while (upLim1 > idx1){
    if (idx1 < LENGTH_87)
    read4Byte_86(&buf[ind], &tc->elem[idx2].elem_1[idx1].elem_1);
      ind += 4;
      indx1++;
    }
  }
 indx2++;
}
```

Fig. III-6: Tail of Data Stream Not Skipped

None of the tools detected this (logical) defect. It was detected by review due to another report of the test tool on an invalid index for `buffer`.

I. *Possible False Negatives*

In many cases where initially an FN was presumed it could be proven at the end of the analysis that it is not an FN, at least according to the internal logic of the tool not reporting it.

However, so far we were unable to find a similar justification for the lack of a report in two cases.

In both of the following 2 examples the functions are called on top level, i.e. there are no constraints from context.

The relevant code is a variation of the code already shown in Fig. III-3 and Fig. III-6. Therefore a snippet is shown only in Fig. III-7.

```
start=0;
tc->elemNo = buffer[start];
start += 1;
read16(&buffer[start],&tc->elem);
start += 2;
tc->paraNo = buffer[start];
start += 1;
```

Fig. III-7: Possible False Negative / Case 1

At the beginning of this sequence, `start` is 0, so the value of `start` is 3 after `start+=2` in the fifth line. The array subscript expression in line 6 thus refers to index 3 of the array buffer.

A check not shown in the example code of Fig. III-7 (but in Fig. III-6) ensures that buffer cannot be NULL. The code also includes a check for whether a parameter called `buffer_size` – apparently intended to be set to the size of the area pointed to by buffer – has a large enough value, similar to the one seen in Fig. III-3.

However, besides the apparent intention, there is no direct semantic correlation between `buffer` and `buffer_size`, so that even with this check present, the minimum length of the area pointed to by buffer is not conclusively established.

For none of the array subscript expressions in line 2 and line 6 a report is issued. The lack of a report for the first could be due to a possible implicit assumption that dereferencing a non-NULL-pointer (i.e. at index 0) is always allowed. That assumption is not valid in general, but may be considered reasonable.

However, the same assumption does not explain the absence of a report for the access in line 6. There is no information available to the tool that would imply that the memory area pointed to by buffer contains at least 4 elements.

A similar issue arises for the accesses to `bin` shown in Fig. III-8.

It seems that the tool implicitly assumes any non-NULL pointer to have sufficient length even if there is no context to justify such an assumption. Whatever the reason for the lack of reports may be, in practice the respective array accesses may lead to an error under the respective circumstances. Consequently, the missing reports were classified as false negatives.

It should be noted, however, that the C-language does not provide any mechanism of checking the actual size of memory available at a location given by an arbitrary pointer. There also is no standard mechanism to establish whether a non-NULL pointer actually points to allocated memory.

As a consequence, it is not possible for a developer to provide the context required, e.g. in the form of appropriate checks in conditional statements.

It may be subject to debate whether that actually justifies implicit and possibly surprising assumptions on the side of the analysis tools. The alternative would be explicit annotations – in the source code or elsewhere – establishing the relationship between the pointer and its associated length parameter.

In contrast, the test tool has information on the size of every item explicitly allocated on heap and stack or by malloc and can conclude whether an address is valid or not – except for objects implicitly allocated in the context of initializers.

```
void getSize(const byte_t *buffer,
             const uint buffer_size, int *size)
{
  if (buffer_size > MAX_SIZE)
    <error>
  else {
    len= buffer[4] << 8 | buffer[5];
    *size=len + 1 + HEADER_SIZE;
    if (*size > MAX_SIZE)
      <error>
  }
}
```

The value returned for size remains unknown in binToAsc.
No conclusion on bin is possible that is not NULL and has sufficient size.

```
void binToAsc(byte_t *bin, uint bin_size,
        const byte_t *asc, const uint asc_size)
{
  getSize(bin,bin_size_&size);
  if (size > bin_size)
     <error>
  else {
    asc[0]='0'
    asc[1]='x';
    for (ii=0;ii<size;ii++) {
      low =bin[ii] & 0x0F;
      high=(bin[ii] & 0xF0) >> 4;
      merge(&asc[ii*2+2],low,high);
    }
  }
}
```

Fig. III-8: Possible False Negative / Case 2

J. Modification of the Planned Process

A major finding of this exercise is the difference between understanding of a tool report aiming to fix the defect and comparing it to reports from other tools, as was explained above by many examples. The increase in effort was considerable.

Also, more effort had to be spent on exercises with the fault database in order to better understand a number of reports sufficiently. The database had to be expanded continuously by specific examples to have clean conditions for a certain defect type to ease understanding of tool reports. Also, variations of the code in the database were required to get reports from different points of view.

For this reason the process described in [1] could not be executed as originally planned due to schedule and budget constraints.

IV. THE EVALUATION APPROACH

Tool evaluation requires configuration of the tools towards the identification of desired defect types and harmonisation of reports from the tools towards a common set of defect types.

A. Tool Configuration

We put the focus of our tool characterization in the context of embedded software, especially on space flight software, regarding defect types which will or may compromise mission goals. Therefore defect types were excluded – as far as possible – which are – "only" – related to violation of lexical rules or layout of source code files. Instead, priority was given to aspects of Reliability, Availability, Maintainability and Safety (RAMS) issues. Also, configuration options which are not relevant for C code were turned off as the application software analysed was written in pure C.

This limitation of configuration options may imply an essential cut-off of capabilities of a tool, possibly in areas where it is strong. This aspect has to be considered when interpreting the evaluation results. Such results are only valid for the chosen application area and the subset of functions considered during evaluation. They may not be extended to the full set of configuration options and software from other application areas.

At the start of the project two evaluation phases were planned:

- an initial phase for which the configuration options were chosen according to existing knowledge on tools, and
- an optimization phase for which the progress on knowledge about the tools and feedback from report evaluation and comparison of reports from different tools should be considered.

However, the second phase was dropped for three reasons:

Firstly, the unexpected higher effort for comparison of reports did not allow a second phase within budget and schedule constraints.

Secondly, the degree of variation for the configuration options was not as high as assumed initially, and thus the expected impact on the reports was considered as negligible.

Thirdly and finally, the driver for the second phase was the intention of a fair evaluation: no tool should have a disadvantage due to initially insufficient knowledge about it. However, as already explained in Sect. II.C – II.G, the increasing knowledge on the defect identification mechanisms of the tools was already achieved in the first step, also resulting in reruns for more specific analysis of the issued reports and the related background.

However, after composing of the results, a comparison of figures yielded unexpected low values for sensitivity of tool 4. A detailed check showed that reports on "unused results" (see Tab. IV-1) which may result in true positives were excluded as the related defect type was classified by the tool as a type which was excluded in general initially (see remarks in Sect. III.B), because most of these reports were considered as not relevant regarding the safety issues discussed above. Unfortunately, it was detected lately at the end of evaluation by a review of the results, that two FPs should have been considered for this tool.

Similarly, this also happened for Tool 2 regarding "recursion". Due to a high number of reports on MISRA-C rules such reports were not considered as (potentially) safety critical. However, rule 17 addressing recursion was also excluded, unintentionally.

Further analyses yielded however, that the impact of the missed TPs does not significantly impact the evaluation results.

B. Spectrum of Principal Defect Types

Harmonisation of all reports ended up in a set of common defect types onto which all reports can be mapped. In addition, a criticality level was assigned to each such defect type. Tab. IV-1 shows the common set of defect types and Tab. IV-2 the chosen criticality levels. Probabilities of defect activation or potential recovery were not considered.

Defect Type	**Criticality Level**
Array Index Out-of-Bounds	Critical
Dereference of Invalid Pointer	Critical
Dereference of NULL-Pointer	Critical
File Access Error	Critical
Invalid function pointer	Critical
Non-terminating Loop	Critical
Passing invalid argument to standard library routine	Critical
(Possible) Recursion	Critical
Resource Leak	Critical
Undefined Result of Arithmetic Operation	Critical
Uninitialized Variable	Critical
Arithmetic Operation on NULL Pointer	Warning
Invariant Condition	Warning
Invariant Expression	Warning
Parameter Type Mismatch in Function Call	Warning
Timeout during execution	Warning
Unnecessary loop construct	Warning
Unreachable Code	Warning
Unused Result	Warning
Multiple return paths	Uncritical

Tab. IV-1: Common Set of Defect Types

Criticality Level	**Comment**
Critical	The defect type does impact the correctness of system operations if being activated, i.e. it manifests to an error or a failure.
Warning	The defect type highlights a possibly unintended operation in the source code which may, but not necessarily does manifest as a critical defect.
Uncritical	The defect type is neither critical nor can it be considered a warning.

Tab. IV-2: Criticality Levels

C. Characterization of the Software

The software package chosen for this exercise is middleware for use on-board of a spacecraft. It consists of a common kernel and several dedicated additional application sets for data management including TM/TC handling, input-output handling, event handling, file operations, etc.

The middleware can be configured for 3 operating systems: Linux[5], Pike OS[6], VxWorks[7].

The largest application set (data management) together with the kernel was selected and configured for Linux.

Infinite loops – like "while (1)" typical for waiting on events in task bodies – were replaced by a finite number of iterations to enforce termination during test.

Property	Quantity	%
Size / KLOC	42	-
Functions, total	610	-
API Functions	376	61,64
c-Files	39 / 49	-
h-files	96	-
Functions, evaluated	60	9,84
Files of evaluated functions	22	56,41
Size of evaluated functions / KLOC	3	7,15

Tab. IV-3: Characterization of the Chosen Software Package

Tab. IV-3 shows its properties and Tab. IV-4 the spectrum of the fault types in terms of numbers as observed in the evaluated subset of functions. The software was provided in 49 files, but only 39 files provided function bodies.

The evaluated functions were selected in two ways:

- Subset 1 was chosen according to the number of reports per function, giving priority to the highest number, resulting in 26 functions.
- Subset 2 was chosen by random selection from the full set. By chance it contains 5 functions also contained in Subset 1. The size of this set was constrained by an upper limit on the evaluation effort, resulting in 39 functions.

The results from both sets were merged for final evaluation. In total, reports from 60 functions out of 610 were considered. Tab. IV-5 shows the number of reports for the different combinations which were subject of evaluation and Tab. IV-6 the number of reports issued for all functions.

The set of defect types marked by an "x" in Tab. IV-4 is called "weighted" and is later referred to when the results are presented in Sect.V. For this set four defect types were dropped because they are not considered as such critical as the other ones.

The Application Interface (API) of the middleware consists of 376 functions out of the full set of 610 functions. For evaluation no assumption on the context shall be made for the API functions, while for the low level functions the context as provided by the API functions may be considered. No contract constraining the input of the API functions is visible.

Defect Type	TP		Selected for Evaluation
	with	w/o	
Array Index Out-of-Bounds	120	126	x
Dereference of Invalid Pointer	31	47	x
Dereference of NULL-Pointer	3	8	x
File Access Error	1	1	x
Invalid function pointer	2	2	x
Non-terminating Loop	1	1	x
Passing invalid argument to standard library routine	1	1	x
(Possible) Recursion	1	1	x
Resource Leak	2	2	x
Undefined Result of Arithmetic Operation	2	2	x
Uninitialized Variable	14	15	x
Arithmetic Operation on NULL Pointer	0	3	
Invariant Condition	16	12	x
Invariant Expression	44	44	
Parameter Type Mismatch in Function Call	2	2	x
Timeout during execution	1	2	x
Unnecessary loop construct	1	1	
Unreachable Code	61	45	x
Unused Result	58	58	x
Multiple return paths	12	12	
Total	373	385	

Tab. IV-4: Spectrum of Observed TP in the Application Software (not grouped)

Report Grouping		Number of Reports (Set 1)				
		TP+FP	TP		FP	
			with ctxt	w/o ctxt	with ctxt	w/o ctxt
Non-Grouped	all defect types	500	369	381	131	119
	weighted	439	311	320	128	119
Grouped	all defect types	270	195	201	75	69
	weighted	231	159	162	72	69

Tab. IV-5: Number of Reports Considered

Tool	1	2	3	4	5	6
Reports	146	742	1481	4995	2106	9

Tab. IV-6: Number of Reports Issued by the Tools

In part, the API functions are checking the validity of the inputs, e.g. NULL-pointers and insufficient size of buffers. But to some part such checks do not fully reject invalid data.

About 96 functions (as far as could be identified by manual inspection), i.e. about 15%, were auto-coded, at least. During analysis of reports it was soon recognised that the code generator did not produce code as intended in some cases, i.e. the auto-code was faulty in part and the same fault patterns repeated in a subset of the auto-coded functions.

This had to be considered when defining the subsets for evaluation in order not to get too many faulty auto-coded functions disturbing the statistics.

D. *Characterization of the Tools*

The spectrum of analysis approaches represented by the tools is quite broad, and defect identification by the different tools is based on a number of independent methods and implementations (Tab. IV-7).

Tools 1, 2, 4 and 5 are static analyzers, Tool 3 applies dynamic analysis (automated built of the test and stimulation environment), tool 6 is the gcc – added for comparison.

Tool	Type	Analysis Approach
1	static	symbolic execution, data flow
2	static	abstract interpretation
3	dynamic	auto-stimulation
4	static	symbolic execution, dataflow
5	static	dataflow
6	compiler	syntax, type checking

Tab. IV-7: Characteristics of Tools

The tools were configured with focus on options regarding reporting of safety aspects. The gcc was run with " –Wall", but not with "-On".

V. Evaluation Results

A. *Complementarity of Tools*

Tab. V-1 gives summary information on the complementarity of the tools, i.e.,the unique contribution by the tool in terms of the number of TP as compared to the overall number of TP. For column 'all' contributions to all defect types were considered, for column 'weighted' 4 defect types were removed.

Tool	Unique TP Contributions							
	All (non-weighted)				weighted			
	ctxt		w/o ctxt		ctxt		w/o ctxt	
	TP	%	TP	%	TP	%	TP	%
1	1	0,27	1	0,26	1	0,32	1	0,31
2	29	7,86	29	7,61	29	9,32	29	9,06
3	65	17,62	56	14,70	65	20,90	56	17,50
4	27	7,32	27	7,09	27	8,68	27	8,44
5	126	34,15	131	34,38	69	22,19	71	22,19
6	2	0,54	2	0,52	2	0,64	2	0,63
Uniq	250	67,75	246	64,57	193	62,06	186	58,13
Total	369		381		311		320	

Tab. V-1: Unique TP Contribution to a Defect Type per Tool

It is surprising, that the sum of all unique contributions is in the range of 58% to 68%, i.e. only about 1/3 of the TP are reported by more than one tool.

B. *Sensitivity and Precision of Tools and Tool Combinations*

Figures on a single tool and combinations of two tools are provided in Tab. V-2 and Tab. V-3.

The rational for providing combined figures is: No tool does cover all considered defect types. Therefore a user may want to know: When I am already using tool A, what do I gain by adding tool B?

Tool	Criticality Level (with context, not grouped)								
	critical			warning			All (weighted)		
	TP+ FP	S	P	TP+ FP	S	P	TP+ FP	S	P
1	9	0,04	0,78	21	0,16	1,00	30	0,09	0,93
2	138	0,40	0,51	27	0,20	1,00	165	0,32	0,59
3	101	0,44	0,78	85	0,31	0,48	186	0,39	0,65
4	55	0,30	0,98	43	0,29	0,91	98	0,30	0,95
5	100	0,44	0,78	71	0,53	1,00	171	0,48	0,78
6	0	0	n/a	2	0,02	1,00	2	0,01	1,00

Tab. V-2: Sensitivity and Precision vs. Criticality Levels

In Tab. V-3 the figures of a single tool can be found on the diagonal of the matrix. In both tables, cells are highlighted for the best values regarding sensitivity.

Tool A in Use	Sensitivity if Tool B added (weighted, with context, not-grouped)					
	1	2	3	4	5	6
1	0,09	0,34	0,44	0,33	0,54	0,10
2	0,34	0,32	0,57	0,52	0,68	0,32
3	0,44	0,57	0,39	0,61	0,77	0,39
4	0,33	0,52	0,61	0,30	0,65	0,31
5	0,54	0,68	0,77	0,65	0,48	0,49
6	0,10	0,32	0,39	0,31	0,49	0,01

Tab. V-3: Sensitivity for Combinations of 2 Tools and Set "Weighted" (context, not-grouped)

Tool A in Use	Sensitivity if Tool B added (critical, with context, not-grouped)					
	1	2	3	4	5	6
1	0,04	0,41	0,46	0,33	0,46	0,04
2	0,41	0,40	0,69	0,67	0,65	0,40
3	0,46	0,69	0,44	0,71	0,74	0,44
4	0,33	0,67	0,71	0,30	0,54	0,30
5	0,46	0,65	0,74	0,54	0,44	0,44
6	0,04	0,40	0,44	0,30	0,44	0,00

Tab. V-4: Sensitivity for Combinations of 2 Tools and Criticality Level "critical" (context, not-grouped)

Sensitivity is increased for Tools 2 and 3 when considering the "critical" subset only, compared to "weighted", while sensitivity for Tools 1 and 5 decreases slightly, but increases for "warning". The sensitivity of Tool 4 does not (much) vary.

The figures vary slightly for other combinations of (context, grouping). Tab. V-5 and Tab. V-6 show the results for the grouped case.

Tool A in Use	Sensitivity if Tool B added (weighted, with context, not-grouped)					
	1	2	3	4	5	6
1	0,11	0,44	0,43	0,19	0,61	0,12
2	0,44	0,40	0,60	0,43	0,79	0,40
3	0,43	0,60	0,38	0,40	0,81	0,38
4	0,19	0,43	0,40	0,14	0,64	0,14
5	0,61	0,79	0,81	0,64	0,56	0,57
6	0,12	0,40	0,38	0,14	0,57	0,01

Tab. V-5: Sensitivity for Combinations of 2 Tools and Set "Weighted" (context, grouped)

Tool A in Use	Sensitivity if Tool B added (critical, with context, not-grouped)					
	1	2	3	4	5	6
1	0,07	0,72	0,46	0,15	0,47	0,07
2	0,72	0,69	0,88	0,72	0,81	0,69
3	0,46	0,88	0,42	0,47	0,63	0,42
4	0,15	0,72	0,47	0,11	0,49	0,11
5	0,47	0,81	0,63	0,49	0,46	0,46
6	0,07	0,69	0,42	0,11	0,46	0,00

Tab. V-6: Sensitivity for Combinations of 2 Tools and Criticality Level "critical" (context, grouped)

In the grouped case Tool 2 gets an advantage for the "critical" subset. However, as was already mentioned that grouping had to be done manually for all tools except Tool 5, to make the data comparable. But the modified data may not not exactly represent the basic properties of the tools (except Tool 5) due to manual intervention, thereby possibly overcompensating the figures for Tool 2. Therefore we consider the non-grouped case as more representative, as it refers to the raw data as delivered by a tool, and provide the figures on the grouped case for information only, indicating potential deviations and impact by manual modifications.

Tool 4 is rather strong regarding precision, but weaker for sensitivity. The gcc gets also good precision figures, but achieves very poor sensitivity, or because of the low sensitivity. This may be higher if higher optimization options will be activated, which may be done in another future exercise.

VI. Conclusions

A. *Sensitivity and Precision of a Tool*

Sensitivity of a single tool reaches nearly 50%, and 70% for the grouped case which however is considered as an artificial case due to manual modification of the raw data.

The precision of tools, i.e. the figure indicating an analysis overhead due to false positives, is in the range of 60% to 100% regarding all defect types.

The sensitivity figures vary slightly for different sets of defect types, indicating that a tool is stronger or weaker regarding certain defect types.

B. *Unique Contributions*

The unique contribution of a tool to the overall set of true positives may be considerably high. Surprisingly, the unique contributions from all tools amount to about 65%, i.e. only about 1/3 of the found true positives are reported by more than one tool, but about 2/3 are reported by one tool only.

C. *Tool Combinations*

When combining two tools the maximum sensitivity increases to about 80%, while precision is slightly depending on context and grouping.

But even for the best combination of two tools yielding a sensitivity of about 80% (not grouped), about 20% of defects will remain undetected.

D. *Dependencies*

Considering the sensitivity for a single tool, the impact by context – with and without – is marginal. In contrast, the impact by grouping – grouped or not grouped – is slightly higher (5% up to 20%) because the number of reports varies significantly.

In most projects one tool only is applied – for cost matters – together with functional testing. The open question is – to answer it was out-of-scope of the project – whether the remaining 50% defects could be found by functional testing.

In general, it seems, the lower the number of true positives, the better the precision. This is not surprising: the lower is the number of issued reports, the lower is the chance for a false positive.

The derivation of summary figures for all defect types depends on the defect profile of the application. Defect types with a high number of true positives get a higher weight compared to others with a low number.

We have chosen this approach because it seemed to be the most fair one, but any weights may be applied if desired.

As an example, resource leaks may be higher weighted due to criticality considerations compared to their occurrence in the application, which is rather low.

E. *Convergence and Completeness of Defect Identification*

A higher analysis depth does not necessarily imply a higher quality of the reports, and a monotonically increasing analysis depth does not imply that the true positives also converge to the final set.

Support of a certain defect type does not imply that a tool will find all defects of that type.

F. *Impact on Software Verification Plan*

Knowledge about profiles of tools regarding sensitivity, precision, complementarity of tools and the benefit of tool combinations should be useful when writing the verification plan.

By recent discussions with tool vendors we learned, that not all tool vendors put the focus on the sensitivity only, but on the response time, too. Therefore a user should be aware of different strategies.

A tool vendor may prefer a compromise between sensitivity and response time: being not complete in reporting, but allowing by a fast response fixing of a number of issues immediately (see Fig. VI-1).

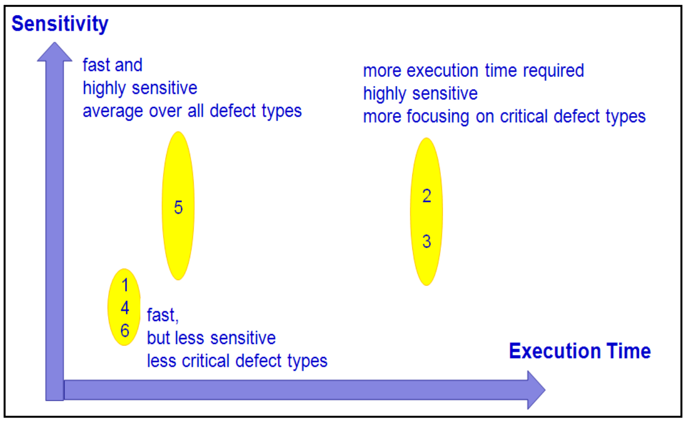
In contrast, high sensitivity may imply a longer response time, and address more complex, but fewer defect types.

Such aspects should be considered in advance by a user when selecting a verification tool or a set.

Believing that every tool will aim to achieve the highest sensitivity figure at low response times or not knowing which defect types are supported by a tool may result in a degradation of the verification approach.

In addition, the applied method or its implementation may already limit the number of defect types which can be detected by a tool. Therefore higher sensitivity should be achieved by diversification of tools.

Fig. VI-1 shows indicatively the areas covered by the tools in a sensitivity-response representation. There is no real scale, the axes just should give an idea on how the tools do cover sensitivity and response time.



Sensitivity
fast and
highly sensitive
average over all defect types
more execution time required
highly sensitive
more focusing on critical defect types
5
2
3
1
4
6
fast,
but less sensitive
less critical defect types
Execution Time

Fig. VI-1: Tool Classification by Sensitivity and Response Time

As Fig. VI-1 indicates, the performance of a tool depends on the spectrum of supported defect types and intended completeness of reports in combination with the response time.

G. *Final Assessment*

A user has to think about what is the best tool or tool combination for the intended application and the quality goal / gate.

A considerable part of defects was found in defect handling parts – as already observed during previous activities. In some cases even defects were induced for correct system states by the error handling part. Obviously, such parts are not subject of extensive verification. This raises the question whether defect handling is meaningful at all if not sufficiently verified at the end.

Two cases for consideration of context were analysed – with and without – to evaluate the impact on the sensitivity. The differences are not as high as expected.

The gcc was added to investigate how well a compiler already does verification. It seems that the focus of a compiler – code generation – covers only a small part of the scope of verification. However, its contribution may increase if optimization is switched on.

Activation of lexical checks is meaningful only if the corresponding rules were / should have been applied right from beginning of development. Applying the checks to software which was not developed under such constraints may result in a huge number of useless reports compromising the detection of critical reports – "useless" because at this time in a project it is usually impossible to apply modifications of this extent to the source code due to schedule constraints.

After analysing about 500 reports our conclusion is:

The percentage of false positives is not such high as expected and communicated. On the average it amounts to about 25%.

Amongst the true positives we saw many reports which would not have been issued if best practices would have been applied. The development effort related to an increase of quality by applying best practices is close to zero – in our mind. But not doing so multiples of the effort saved at development time need to be spent later for analysis of – valid – tool reports.

The best report is a report not being issued at all i.e. not giving any reason to issue a report. The programming style may heavily affect the number of true positives. If complaining about too many reports, the reason for the high number should be investigated.

Once the acceptance criteria have been defined – preferably before coding starts – checks on compliance with these criteria should be performed periodically and as soon as possible to obtain a feedback and being able to adapt coding to the given and accepted rules.

A high number of reports may not be a matter of the tools, but in many cases also a matter of coding and compliance (or non-compliance) with given rules.

The obtained results presented in this paper may not be valid in general. The focus was put on embedded systems, critical defects (as defined by the authors) and middleware foreseen for use in a space flight application. Therefore some valuable features of a tool, e.g., regarding security, lexical checks, might not have been considered during evaluation, which may be of high interest in the scope of another application domain.

H. *Outlook*

The evolution of results due to modification of the following parameters may be subject of future work:

- the amount of evaluated reports,
- the type and size of an application package,
- the defect profile of an application package,
- the tools considered,
- stability and convergence of results regarding analysis depth and configuration options.

Further, an extended evaluation of the database regarding more aspects and correlations, e.g. require effort, and derivation of graphics may be performed.

Figures on evaluation effort for true and false positives for single tools and tool combinations have been recorded or derived, but could not be considered due to schedule and budget limitations.

Another field of evaluation may be whether combinations of three or more tools provide any significant additional benefit in comparison to the effort added for analysis of the reports of these additional tools. Of course, one possibility for reducing the effort in general is standardization of reporting among tools so that automated consolidation of reports becomes possible.

In future the support software for evaluation and comparison of reports needs to be extended in order to reduce the amount of manual effort and to allow evaluation of more reports.

The obtained results shall also be discussed with tool vendors / distributors.

Future work may address an extension of the database and higher independency of the software applied for evaluation by

- same software, but other tools,
- other software, but same tools (in part) and other tools.

I. About the Tools

Tab. VI-1 gives the names of the tools as far as the disclosure was approved by the tool supplier at the time of finalizing this paper.

Id	**Status**	**Name of**	
		Tool	**Supplier**
1	disclosure not decided yet		
2	disclosure not decided yet		
3	approved	DCRTT	BSSE
4	no explixcit approval received		
5	approval received	QA•C	PRQA
6	open source	gcc	

Tab. VI-1: Evaluated Tools

REMARK

Details on the analysis approaches of the tools and the classification criteria as well as a detailed discussion of the results are out of scope of this paper.

ACKNOWLEDGMENT

The results presented above are an outcome of contract DLR-50PS150 of the Space Administration of the German Aerospace Center (DLR) on behalf of the German Ministry of Economics and Energy (BMWi).

The authors also thank the vendors / distributors of tools 4 and 5 who provided evaluation licenses for their tools.

REFERENCES

[1] Ch.R.Prause, R.Gerlich, R.Gerlich, A.Fischer: „Characterizing Verification Tools Through Coding Error Candidates Reported in Space Flight Software”, Eurospace Symposium DASIA'15 "Data Systems in Aerospace", 19 – 21 May, 2015, Barcelone, Spain

[2] C-Standard, ISO/IEC 9899:2011, Information Technology – Programming Languages – C, 3rd edition, 2011

[3] MISRA :2012, www.misra.org.uk/MC2012

[4] P. Cousot, R.Cousot: “Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints”, Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, ACM, 1977, pp. 238-252.

[5] Linux

[6] PikeOS, Sysgo, www.sysgo.com

[7] VxWorks, WindRiver, www.windriver.com/products/vxworks