BSSE System and Software Engineering

FASTII-Abstract-BSSE

Issue : 1 **Revision :** 0
Date : 29.10.2019

FAST II

Automated Source-code-based Testing, Improvement

under ESA Contract No. 4000116014/16/NL/AF/as

FAST II Abstract

APPROVAL	
Approval Authority :	BSSE
Date :	29.10.2019
Signature :	
Name :	Rainer Gerlich

Project Manager

Ralf Gerlich

FAST II	DISTRIBUTION LIST	FASTII-Abstract-BSSE Issue: 1 Revision 0	Page 2

N A M E S	C O M P A N Y	COPIES	PURPOSE		
			APPROV.	APPLIC.	INFOR.
Maria Hernek	ESA/ESTEC	1	x		
Rainer Gerlich	BSSE	1		x	
Ralf Gerlich		1		x	
Allan Pascoe	SCISYS	1			x
Glenn Johnson		1			x

BSSE System and Software Engineering

FASTII-Abstract-BSSE	**Issue** 1	**Rev.** 0	**Date :**	**29.10.2019**
			Page	3 / 15

	COMPANY		**PROJECT**		**ESA**	
CLASSIFICATION	BSSE unprotected	☒	Unprotected	☒	General Public	☒
	BSSE Reserved	☐	Restricted	☐	Industry	☐
	BSSE Confidential	☐	Confidential	☐	Restricted	☐
	BSSE Secret	☐	Secret	☐	Confidential	☐

CONTRACT	**Customer** ESA/ESTEC	**Contract number** 4000116014	**Project** FAST II
	Contractual document yes ☒ no ☐		**Work Package** WP6000

TITLE **FAST II Abstract**

ABSTRACT

This document summarises the work performed and results achieved in the FAST II project..

KEYWORDS

FAST process, test automation, test data generation, C, C++

File ref. FASTII-Abstract-BSSE-V1.0-2019-10-23.docx **Software :** WORD 2007	**Configuration management**	**Distribution Category**
Language code: EN	None ☒ Internal ☐ Customer ☐	Authorized ☒ Checked ☐

	Author		**Head of Company**
Name	Rainer Gerlich Ralf Gerlich		Rainer Gerlich
Signature			

FAST

FAST II	STATUS RECORD	FASTII-Abstract-BSSE	Page 4
Issue Revision	**Date**	**Reasons for document change**	
1 / 0	29.10.2019	First version	

FAST II

1. SUMMARY

The FAST process (**F**low-optimised Automated, **S**ource-code-based **T**est) aims to enter a further step in test automation starting already with test data generation including identification of test stimuli and building of the test environment and ending with provision of reports documenting the test results.

The evaluation of the test results, especially their comparison with requirements or the detailed design is not covered, except in the case when oracles are provided.

The tool DCRTT supports the FAST process. In the FAST II project improvements to DCRTT were implemented and benchmarked at the end. Goal was to achieve TRL 5.

Two improvements were directly requested by ESA:

- support of an open tool interface

  allowing to export test data to other test tools for re-execution
- support of Requirements-Based Testing (RQBT)

  aiming to close the gap between automatically generated test stimuli and requirements.

while other improvements should be identified in the project.

The following steps were executed:

1. Definition of an Open Tool Interface,
2. Definition of an approach to RQBT in the context of DCRTT,
3. Identification of other improvements,
4. Implementation of the improvements in DCRTT,

   implementation of the improvments was done on BSSE own investment
5. Benchmarking of the results with other verification tools, and
6. Preparation of TRL 5 assessment.

As an outcome of the project an Open Interface to Cantata[1] and VectorCAST[2] is now available. The interface may be extended to other tools as well.

A major driver for the other DCRTT improvements was the reduction of false positives.

For benchmarking Astree[3], BugFinder[4], CodeProver[4] and QAC[5] were chosen. DCRTT and the other tools were applied to representative flight software in an early and late version. The number of reports varied in a large range from tool to tool. Therefore only subsets of the reports issued by the tools could be manually analysed. Sources of false positives were identified and discussed. In context of the evaluated subsets a few true positives were found and forwarded to the development team. The intention of analysing an early and late version was to check whether more defects – supposed to occur in the early version – would impact the sensitivity of a tool to find true positives. Regarding the small subsets which could be evaluated manually no significant difference was found. Also, the number of issued reports did not differ much. However, for the late version some defect types disappeared suggesting that they were fixed.

[1] Cantata is a product of QA Systems GmbH, Stuttgart, Germany
[2] VectorCAST is a product of Vector Software Ltd., London, UK
[3] Astree is a product of AbsInt Angewandte Informatik GmbH, Saarbruecken, Germany
[4] BugFinder and CodeProver are products of The Mathworks Inc., Natick, Massachusetts, USA
[5] QAC is a product of Perforce Inc., Minneapolis, USA,

2. THE FAST PROCESS

Fig. 2-1 shows the principal logic flow of the FAST process from test preparation to result evaluation – as it is supported now by the implementation in DCRTT at the end of the FAST II project.

The process is driven by

- the source files,
- requirements or oracles[6],
- the test configuration files, and
- annotations or meta-information like constraints on type ranges or correlation of data items,

  thereby adding information to the application software which cannot be expressed in the C language itself, but necessary to tune testing, e.g. to reduce the number of false positives by providing information on the context.

Several runs may be executed under different test configurations.

The process starts with analysing the source code and ends with providing the test reports and test drivers with recorded input-output vectors of functions-under-test. The latter represent proposed test cases for regression testing with DCRTT or other test tools. Everything between is fully automated by the process. A user should check the reports and approve the proposed input-output vectors (test vectors) provided in the test drivers by checking compliance with requirements. The test drivers can be used for regression testing on host and target.

The test drivers are generated in an intermediate format. This format is the central part of the Open Tool Interface, from which the native test drivers of DCRTT is derived and the link to other test management software, such as Cantata and VectorCast is established.

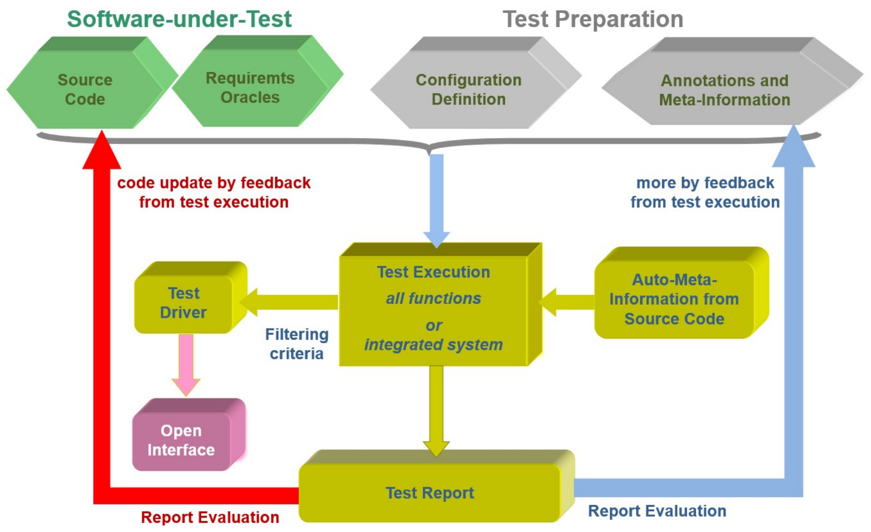
Software-under-Test
Test Preparation
Source Code
Requiremts Oracles
Configuration Definition
Annotations and Meta-Information
code update by feedback from test execution
more by feedback from test execution
Test Driver
Test Execution all functions or integrated system
Auto-Meta-Information from Source Code
Filtering criteria
Open Interface
Test Report
Report Evaluation
Report Evaluation

Fig. 2-1: The FAST Process – From Test Preparation to Result Evaluation

The manual check on compliance is automated when oracles are provided in the notation of DCRTT. Then the passed/failed result can be propagated bottom-up in the requirements hierarchy – provided the tracking information is available. An oracle may be derived automatically from a machine-interpretable requirement.

Stimulation of the functions and test driver generation is performed by auto-generation of input vector.

[6] An oracle is a function which can determine whether a test has passed or failed, i.e. which can automatically conclude on the correctness or compliance of expected and observed results. Apart from an exact comparison of results, an oracle may also apply heuristics from which an approximate pass/fail conclusion can be derived.

When the stimulation phase is completed, the reported anomalies are manually analysed. The data in the report provide information required for localizing the defect.

Explicit fault injection can be activated. Then constraints placed on the allowed values for input parameters and global variables in stimulation are ignored. Modification of global variables declared as constant, blanking of initializers for global variables and modification of return values from called functions are supported.

Together with massive stimulation, fault injection creates a harsh environment for the functions subject to testing in order to increase the probability of anomalies being raised which may point to defects in the code.

Fig. 2-2 shows the interfaces of the FAST process,

- at the input interface the various modes for stimuli generation, and
- at the output interface
  - the report(s) and
  - the test drivers

  where other tools can be attached. Currently, interfaces to Cantata and VectorCAST are supported.

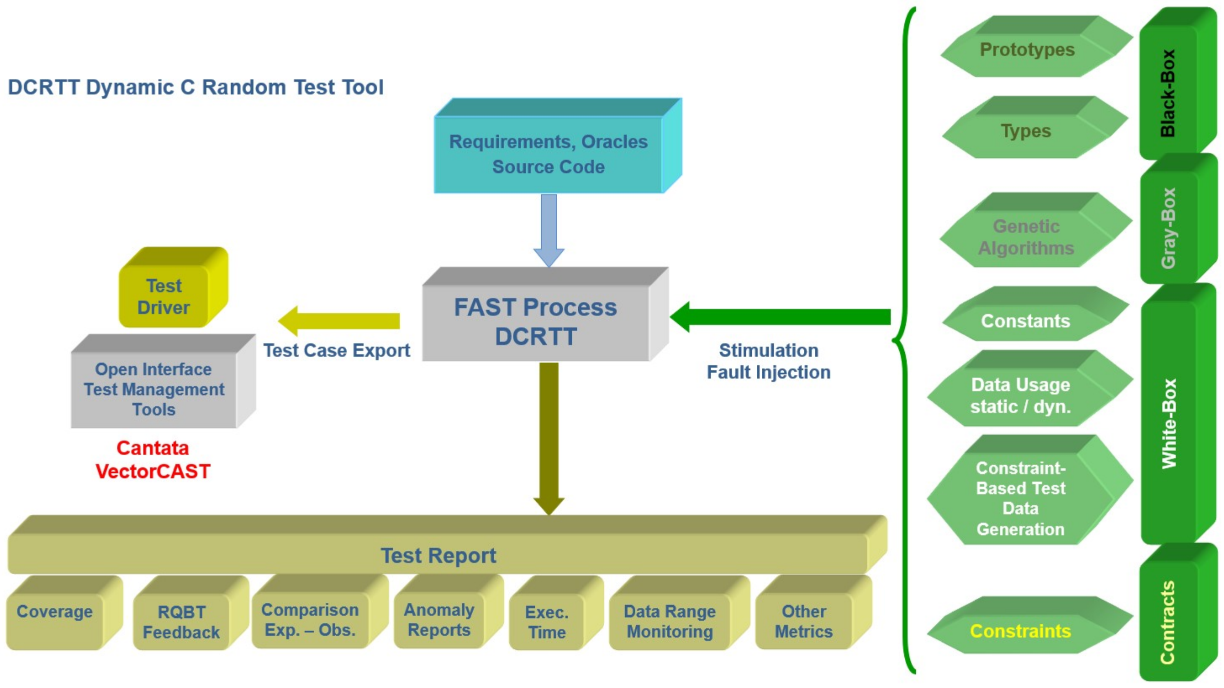

Fig. 2-2: Interfaces of the FAST Process

Four principal stimulation approaches of functions are supported implying different sources of information:

• black-box stimulation	based on a functions's prototype
• gray-box stimulation	by applying genetic algorithms
• white-box stimulation	based on information about a function's body,

- contracts
  providing additional information on parameters and data ranges constraining black-box and white-box stimulation.

Any stimulation approach may be applied to global data used in the function-under-test, too.

For robustness testing the information on type ranges is sufficient for deriving stimuli. It is possible to further restrict the ranges from which stimuli are selected. This can be used to approximate valid set of inputs for which function shall work correctly, or to approximate the context of the intended operational environment.

Not all information on the context can be found in the C code. Therefore, additional information on range constraints can be provided by a user or is obtained automatically to the degree possible by provided heuristic rules.

Missing functions or data are automatically substituted by stubs. The out- and return-parameter of the function stubs provide values randomly generated according to the type. Composite structures are supported.

The source code is instrumented to get information on coverage, data range, exceptions, resource consumption and other run-time anomalies.

Several reports on the test results are provided as text or graphics.

3. IMPROVEMENTS

The **Open Tool Interface** allows to establish a link between DCRTT and other test tools on the base of components separated from DCRTT. These components usually depend on the attached tool and have to be added to the DCRTT environment so that the other tool can be controlled for re-execution of the test drivers. A major component is the converter for the test drivers. Currently, converters exist for DCRTT, Cantata and VectorCAST.

The existing format of the native DCRTT testdriver was converted to an intermediate format from which specific format of tools – including the DCRTT format – can be supported. A tool-specific converter is required for every tool. Two converters were established: one for support of the classic DCRTT test drivers and Canatata and the other one for VectorCAST. For configuration of the interface information must be provided in a master-file from which a converter-templates is generated, which may be extended manually if required.

Tests with the external tool can either be executed step-by-step with every function test in DCRTT, or the complete set of tests can be re-executed separately after a complete DCRTT run.

Requirements-Based Testing is supported on the base of oracles. An oracle consists of two parts: the *pre- and post-condition.* If – and only if – the pre-condition is fulfilled, the post-condition checks a condition representing a requirement or a part of it. Requirements and oracles are correlated on 1:m relationships. Oracles and functions-under-test may have n:m relations.

DCRTT expects oracles in the notation

<short name> <pre-condition>?<post-condition

in a file where pre- and post-condition must be provided in valid C code and evaluate to a boolean expression, and short name represents the link to the requirement.

The complete infrastructure has been implemented for processing and evaluating oracles, and propagation of the pass/fail-result bottom-up in the requirements hierarchy. This way the gap between auto-generated test data and requirements is closed automatically.

Initially, the intention was to derive oracles from requirements automatically. To do so requires a machine-readable format for the requirements. However, all requirements found were provided in text format, and – as another obstacle – also were difficult to convert manually into a form suitable for oracles. The requirements would have to be re-structured completely. So only a few requirements could be derived from the existing requirements document. They were complemented by other requirements used all together for demonstration of RQBT. An interesting result is that in case of failed tests counter examples can be provided.

Other Improvements address reporting and compensation of missing information on the context to reduce the number of false positives, and integration and demonstration of existing features to increase coverage. This is the list of major improvements:

- reporting
    - provision of line numbers referring to the original soure file
- compensation of missing context information
    - auto-resizing of arrays
- integration
    - constraint-based test data generation
    - test data generation with genetic algorithms.

As the source files are instrumented, the **line numbers** determined at run-time in response to an anomaly have to be converted to line numbers in the uninstrumented, original file. The auto-conversion is a pre-condition for comparison of DCRTT reports with other tools at little effort.

The unknown **size of pointer-arrays and unconstrained arrays** passed as parameters is a major source of false positives. Their size may be determined dynamically at run-time. If too little memory is allocated in the test environment false positives will be generated for valid indices, and real invalid indices may not be recognised if too much memory is allocated. Dynamic resizing solves this issue. The size allocated at the end is recorded so that a comparison with requirements is possible in order to avoid missing of a true positive.

Integration of sophisticated test data generation approaches was performed to demonstrate their capabilities regarding increase of coverage for cases for which the type-range-based generation process cannot achieve high coverage. Both approaches are still in an expermimental stage and were not applied to the flight software.

4. BENCHMARKING

The goal of benchmarking was to assess the probability of finding an error in the analysed or tested software and to compare DCRTT for its capabilities after implementation of the improvments with the other selected verification tools. This probability is driven by the false and true positive rates, which are implicitly dependent on the supported defect types. A low number of false positives and a high number of true positives is desired. The challenge is to find the true positives in the set of issued reports. The challenge is higher the more reports are issued, of which it is not known a priori whether they will turn out as true or false positives by manual evaluation. A tool is the better, the more it can support this decision.

A false positive may be caused by a defect in the tool or by insuffient information on the operational conditions – the context.

Static analysers are often context-centric, i.e. they consider for analysis the constrained data range as occuring along the call path. Only, for the entry-point at top level issues may occur due to missing information on the context – this is the idea.

In contrast, in case of function-level unit testing every function-under-test occurs on top-level, and issues with context, too. In case of automated testing the tool (DCRTT) itself collects context information to the degree possible. As the context information may be incomplete for every function, the number of reports issued by static analysers, and in particular the false positive rate, should be lower than for DCRTT.

The tools based on abstract interpretation are considered as sound, while DCRTT is unsound. A tool is sound if its verdict represents reality under all possible conditions. In case of verification tools this means that for all supported defect types, all existing faults must be reported. Absence of any report implies that no defect exists. However, soundness does imply presence of false positives, possibly many. Further, by configuration additional assumptions may be provided to the tool, allowing the tool to not report defects which cannot occur under these assumptions. It is therefore of utmost importance to also properly consider the effects of tool configuration.

Two versions – early and late – of representative flight software were subject to analyses and tests. Its C source code amounts to about 170 KLOC and comprises more than 3400 functions spread over dozens of tasks. The tools chosen for comparison with DCRTT are all static analysers based on abstract interpretation or symbolic execution and dataflow analysis.

The code was slightly adapted to ensure that all tools can perform analysis and test. All tools received the same code. To obtain a common context for all tasks a single entry-point function embedding all the tasks was automatically constructed based on information extracted from the source code. Three different configurations were considered for the calls of the tasks for analysis by the static analysers: a deterministic sequence, a non-deterministic, random sequence and multi-tasking – as far as supported.

DCRTT executed unit tests for all the functions according to the FAST process. The entry-point function was executed in addition – although not originally planned – for the deterministic and non-deterministic cases with the full instrumentation for monitoring and recording as provided for unit testing. In contrast to unit testing the entry-point function provides a better approximation of the real operational context for all functions below the task entry. Due to the more representative context the number of false positives should be reduced.

The intended automatic comparison of the tools required to put the focus on the reports to make an automatic comparison line-by-line possible. Inspection on GUI level was only performed in a few cases together with the tool supplier.

The output from the static analysers was converted into a standardised format so that the reports from the different tools could be compared. Due to the rather high analysis time only a small subset could be evaluated to check for true positives. Several statistics were established on covered defect types and coincidences of reports from different tools. The manually evaluated subset was to small to derive representative figures on sensitivity and precision.

4.1 RESULTS

The number of reports varied largely, from ~800 to ~29.000. About 1.500 reports were issued by DCRTT. No conclusion is possible whether the number of reports would be much lower at the end and at intermediate quality gates if the tools would have been applied continuously over the development period in the sense of continuous integration, so that findings could have been considered in the course of further development.

The execution time varied between about 1 hour to about 10 hours in case of analysers and 7 days for DCRTT in case of unit testing. The latter figure is related to the time needed for execution of the 3.400 tests and ~350.000 injected stimuli. About 3 minutes per function were consumed on the average, including compilation and linking which took most of the time. In case of execution of the entry-point function, the duration varied from ~ ½ hour (early version) to ~2 hours (late version) for ~35.000 task calls.

The maximum memory consumption of the tools was 30GB. Memory consumption of DCRTT varied from 200 MB to 400 MB. As the memory limit of 30 GB of the computer was reached, the analysis options of one tool were constrained.

The number of false positives in case of unit testing with DCRTT results from insufficient information on the context at the function interface. The number of reports decreased drastically to ~30 reports in total for the entry-point functions – keeping the monitoring capabilities from unit testing. 4 true positives could be found at low effort for the early version and 1 for the late version, which represent about 6% for the early version and 1.6% for the late version w.r.t. the manually evaluated subset of ~60 reports.

4 reports were considered as relevant true positives for the late version and forwarded to the developer team. 2 out of the 4 reports were confirmed by the team, but 2 were classified as not relevant due to hidden checks (see the explanation below) which were not visible to the tools. For the 2 confirmed issues the team concluded that they have no impact on the current operational concept and mission performance.

The results of the tools were discussed with two tool suppliers. The discussion brought up interesting aspects regarding the soundness of the analysis chain and consideration of context during the analysis.

Firstly, due to resource constraints context information may have to be dropped. In consequence, the number of false positives increases, and the supposed advantage of static analysis diminishes with increasing complexity of the subject of analysis.

Secondly, soundness implies completeness of the report set. In the discussed case a report proven as true positive – reported by DCRTT – was not found in the generated report file. However, after a while it was found in the full log-file recording all analysis considerations and results. In consequence, a – theoretically – sound tool may turn out as unsound due to issues in reporting related to the information source.

4.2 LESSONS LEARNED

In the course of the evaluation, new insights on several aspects in use of software verification tools on the source-code level were gained.

These are the most important ones regarding the issued number of false positives:

- mismatch of verification scope
  If verification is context-driven the tool should (accurately) consider the context in the system to be verififed.
  If robustness analysis is the goal a tool should be chosen which can vary the context.
- non-representative environment
  function stubbing, insufficient modelling / consideration of scheduling and insufficient information on the context increase the number of false positives, in most cases significantly.
- broken control and data flow, hidden checks
  Information on protection against invalid data may be hidden to the tools, e.g. if checks are done in one task and the checked data are used in another task, but a tool cannot identify the flow of the verified data and reports an issue.
- certain code constructs
  Untyped byte streams – as used for telecommands – including dynamically varying data structures are challenging for both, static and dynamic analysis. In case of DCRTT / testing the issue can be solved by specific test data generators, while for static analysers the set of possible instantiations has to be explicitly described in the form of code. Most of the tools support constraining the set of inputs using value ranges, which may be useful if the set of valid inputs is completely or approximately orthogonal.

Arrays used for mapping of information are a challenge for static analysers and enforce approximation of the context – in most cases, even for small arrays. In contrast, DCRTT exactly takes the intended elements, no false positives are generated in this case.

- over-approximation in static analysers
  Due to resource constraints static analysers may have to approximate the context, i.e. they are losing context information, increasing the number false positives.

The number of false positives can be reduced by the following means:

- continuous application of the verification tool(s) during the development phase
  If code is developed without considering the quality gate at the end, a high number of false positives should be expected.
- contracts and annotations
  They can complement information on the context not provided in the source code.
- context independency
  The more the software-under-analysis is independent of external context, the less false positives are issued when exposing it to context-centric verification. Partitioning a system into context-independent parts decreases the complexity for static analysers and the need for approximation of the (now less complicated) context.
- coding style
  Provision of information on parameter constraints and correlations – relevant for static and dynamic analysers, which is already practiced in other domains like automotive.

The following experience was made regarding reporting of true positives:

- The number of reported true positives depends on the external and internal configuration of a tool. True positives may not be reported by one sound tool, while being reported by another sound tool due to different configurations.
- Reports related to true positives may be missing in the standard report file while present in the much larger log-file (of a sound tool).
- Identification of true positives can be improved by use of Requirements-Based Testing requiring however provision of machine-readable requirements.

Regarding the evaluation approach applied for tool comparison the following issues and challenges occurred:

- Standardisation of reports
  The individual tool reports had to be converted into a standard format to make them compatible. The individual text messages were mapped onto a standard message for every defect type, and the defect locations were converted into a standard format.
- Harmonsiation of reports
  Multiple reports on the same defect had to be mapped onto a single report.
- Report coincidences
  Correlation of reports from different tools on the same defect type and the same location were identified to find tool coincidences.
- Configuration issues
  The knowledge on the defect detection mechanisms of the tools increased during and after evaluation, in particular the discussions with tool suppliers yielded essential details, and brought up principal differences suggesting that the results are not directly comparable.

- Report set for manual evaluation
  The number of reports initially intended for manual evaluation had to be reduced to rather high effort per report. Meaningful results on precision and sensitivity could not be obtained due to few evaluated reports.

5. CONCLUSIONS

Conclusions are drawn on the future application areas of DCRTT according to the capabilities available at the end of the project and on benchmarking of the verification tools and comparison with DCRTT.

5.1 USE OF DCRTT

By the features added in the course of the project DCRTT was significantly enhanced.

The results of the project suggest use of DCRTT for the following cases:

1. Robustness testing

   All reports will be true positives, except for a small part possibly induced by stubbing.

2. Unit testing

   The number of false positives should be reduced by provision of information left for the user after auto-extraction of context information by DCRTT.
   Context-independent units will help to solve this issue.

3. Requirements-based testing

   The infrastructure for RQBT is provided based on oracles. It can be used in context of machine-readable requirements. However, the provision of requirements in such a format is still a challenge – however not an issue for DCRTT.

4. Open Tool Interface
   The interfaces provided for Cantata and VectorCAST and the support of a link of more test tolls by the open interfae open new application scenarios for DCRTT.

5. Test of the integrated system
   The use of the integrated system at presence of monitoring capabilities of DCRTT showed a drastic reduction of reports and allowed identification of true positives at low effort. Improvements for stimulation and task scheduling are needed.

5.2 BENCHMARKING

The following conclusions are drawn on benchmarking:

- Context

  Static and dynamic analysis tools both have problems with context.

  Though static analysers aim to consider and keep the context inside the software-under-analysis along the call-paths, but often they lose the context due to resource constraints. In consequence, context information is missing along the call-paths resulting in a significant number of false positives.

  In contrast, DCRTT as dynamic analysis tool, only may miss context information at the top-level of the function interface, but keeps the context along the call-paths. This should result in a reduced number of false positives.

- Code Coverage

  If statement coverage is reported from static analysers the figures are in the range of 70 – 95%. Higher coverage figures may be achieved due to over-approximation when values are applied which would not occur in the representative case, i.e. loss of context information increases coverage, but also false positives.

  DCRTT keeps the context and may not reach all locations by white-box or black-box stimulation over the interface. Therefore constrained-based test data generation or genetic algorithms are required.

- Reports

  Static analysers issue a report when they identifiy fault potential, which is possibly driven by loss of context information. The – possibly – incomplete context is propagated along the call tree and can raise more false positives on lower levels.

  DCRTT only issues a report when an anomaly is detected during real execution, where no approximation is required. Insufficient information may only occur at the top-level, but for every function-under-test.

- Coverage of defect types

  The static analysers considered cover a broader range of defect types compared to DCRTT.

- Evaluation support

  At presence of a large amount of reports it is essential to get hints to find true positives.
  In case of the static analysers all reports have the same probability to evaluate as false or true positive. DCRTT filters reports which have a high potential to evaluate as true positives. In fact, the majority of true positives was found following these suggestions.
  The few other TP were found by sampling, i.e. by random selection of a subset.

- Soundness

  DCRTT is not sound. The static analysers based on abstract interpretation are considered as sound. But TP may be missed due to imperfectness of reporting, the chosen configuration or by limiting the manual analysis to a subset.

5.3 OUTLOOK

The drastically reduced number of reports – and associated reduction of false positives – and the related identification of true positives at low effort suggest to further investigate the entry-point function approach. The environment should be complemented by telecommand injection, stimulation of external (hardware) interfaces and support for modelling of representative scheduling.

The following strategy looks promising regarding tests with valid data / representative context:

- Test with entry-point function and full instrumentation provided by DCRTT, with representative scheduling, telecommand injection and stimulation of external interfaces.
- Achieve missing coverage by use of constrained-based testing and genetic algorithms applied to a subset of functions (as for current unit testing, but without type-range based stimulation).
- Unit testing with full set of functions complemented by constraints and correlations not found automatically.

For tests extended to invalid data

- Apply robustness testing to the full set of functions as supported by DCRTT without provision of context information and with activation of fault injection wrappers in the application software.
- Activate the fault injection wrappers in context of testing with the entry-point function.

For tests extended to invalid data

- Apply robustness testing to the full set of functions as supported by DCRTT without provision of context information and with activation of fault injection wrappers in the application software.
- Activate the fault injection wrappers in context of testing with the entry-point function.

6. ACKNOWLEDGEMENT

The authors want to thank the tool suppliers AbsInt, The Mathworks and QA Systems for providing an evaluation license for their tools and giving an excellent and very helpful support in context of tool execution and report evaluation.

For export of report line numbers, The Mathworks provided a modified version of BugFinder, and QA Systems provided a functional extension of the license. Thanks to that, consolidation and off-line evaluation were made possible.