Gerlich System and Software Engineering

Overview on the FAST Process and the DCRTT Tool

Table of Contents

1. INTRODUCTION

1.1 THE SCOPE OF AUTOMATION IN CONTEXT OF FAST AND DCRTT

The FAST process (**F**low-optimised Automated, **S**ource-code-based **T**est) extends the scope of test automation to an earlier step in the test process compared to the current understanding of the term "*test automation*", thereby supporting massive stimulation, and in consequence collection of plenty of information on the properties of the source code which is automatically exposed to a high number of test conditions.

The test process already starts from provided source code and ends up with generated reports, covering

- preparation of the test environment for function and integration testing,
- generation and injection of test data,
- injection of faults at the given interfaces,
- support of stress and robustness testing by massive stimulation,
- collection of information on anomalies occurring during execution,
- collection of information on data ranges and execution times,
- generation of test drivers for execution on host and target platform and of regression tests,
- requirements-based testing and generation of oracles for automated test evaluation and bottom-up propagation of pass/fail information from low-level to high-level requirements,
- an open interface to other test management tools,
- generation of a number of code metrics, amongst which are code coverage and cyclomatic complexity, and
- provision of reports documenting the test results and properties of the source code.

The tool DCRTT (Dynamic C Random Test Tool) was the starting point for definition of the FAST process in context of the European Space Agency (ESA) project "Automated, Source-code based Testing".

1.2 HISTORY

The starting point for development of DCRTT was to replace the time-consuming manual task of test preparation for function tests by an automated approach together with random generation of test data – as the "R" in its name is saying. The first tool based on this approach was DARTT (Dynamic Ada Random Test Tool), from which the name was created by replacing the "A" for "Ada" by "C" for C language.

The feedback obtained from real-life projects executed for ESA or German Space Administration stimulated a number of improvements, and helped to prove the scalability of the implementation.

Over the years, the tool evolved and a number of additional features were added. Grid-based and constraint-based test data generation were added. Black-box testing was complemented by white-box testing. The focus of reporting was extended from pure detection of exceptions to a number of further anomalies and properties of the source code. Test vectors were converted into test drivers. The support for target platforms was extended.

1.3 CONTENTS OF THIS DOCUMENT

This document highlights the features of the process and the tool DCRTT implementing the FAST process.

In Ch. 2 an overview is given on the FAST process and principal features of DCRTT as the tool supporting the process.

Then, the following chapters provide an overview on the DCRTT features for the test process regarding

- test preparation (Ch. 3),
- test monitoring and control (Ch. 4),
- test evaluation (Ch. 5),
- reporting (Ch. 6),
- the open tool interface (Ch. 7),
- exploring an integrated version (Ch. 8), and
- using wrapper functions (Ch. 10).

Ch. 0 provides information on the platforms supported by DCRTT.

For further information please contact GSSE at dcrtt@gsse.biz.

1.4 DEFINITIONS

1.4.1 The Test Process

Software testing aims to find faults in existing software by exposing the software – or parts thereof – to pre-fabricated stimuli, observing the reaction of the software and ascertaining whether the reaction conforms to the expected behaviour of the software – as expressed by requirements – given the stimuli.

The *process of software testing* consists of

- the selection of stimuli,
- the determination of the expected behaviour of the software under test,
- the injection of stimuli,
- execution of the software-under-test,
- the extraction of the actual reaction of the software under test, and
- the determination of whether the actual behaviour of the software under test conforms to the expected behaviour.

Software unit testing is software testing applied to the so-called unit level. The term *unit* usually refers to the smallest non-separable functional elements of a software product. Often these are the individual functions or procedures defined at code level, but a unit may also be composed from a group of such functions which are intended to be used in combination in order to provide atomic functionality. The latter is often the case when object-oriented methods are used in the design and implementation of the software.

Testing of the integrated software system exposes its top-level function (main function) and its external data interfaces to stimuli similarly to unit testing.

Automated software testing refers to the process of performing the task of software testing in an automated manner. In this context, the term shall be used specifically to mean the full automation of the process, consisting of all steps of the software testing process.

Automated software unit testing as a special case is the application of automated software testing at the unit level.

A combination of stimuli and expected reactions/outputs is usually referred to as a *test case*. If the actual behaviour of the software matches the expected behaviour, the test case is said to be *passed*, otherwise it is said to *fail*. This result is called the *verdict*.

Note that while the expected behaviour is often expressed in terms of requirements, testing may also be used for validation, challenging the validity of the requirements in the first place. The more general term "expected behaviour" encompasses both cases.

1.4.2 Terms

In the context of the project and the respective test process the following terms are of relevance.

Term	Definition
CBTDG Constraint-based Test Data Generation	An approach applying constraint-programming methods to identify values of function parameters and global data by which a certain location (checkpoint) in the code can be reached or a certain condition or decision can be made to evaluate to TRUE or FALSE.
Context	In case of FAST context is related to the environment in which functions are executed. It is represented by a set of possible values for (global) variables and parameters used by a function. This set may be constrained to an actual subset of the values allowed by the respective variable and parameter types when a function is called. The constraints may be imposed due to initial conditions (e.g. values of global variables and parameters), conditional execution or operations yielding only a specific set of outcomes, including the history of execution preceding the actual call of a function.
Defect	A *defect* commonly refers to troubles with a software product, with its external behaviour or its internal features (e.g., its maintainability). This includes consideration of the risk of faults by potential changes of the context
Error	An *error* is a bad or undesired state in a software system
Failure	A *failure* is a non-compliance regarding external behaviour being recognized between expected and observed properties of the software product as a consequence of an error.
FAST	Flow-optimised Automated Source-code-based testing Process A test process which starts with automation of test preparation, auto-generation of test stimuli, covers automation of test execution and documentation, generation of test drivers according to coverage criteria. Previously the “F” stood for “fully” but was replaced in the context of the project by “flow-optimised” to avoid the impression that also result evaluation would be automated in any case.

Term	Definition
Input vector	The set of data provided in the test input relevant for a test.
Fault	A *fault* is the cause of an error having its origin in the code which may be called a mistake
Oracle	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.
Output vector	The set of data related to the test output at the end of a test step.
Requirements-based Testing	A testing approach aiming to prove that requirements are fulfilled. The executed test cases and the obtained results are correlated with requirements. In case of auto-stimulation of functions the issue is to perform the correlation with requirements automatically.
Robustness testing	In context of the process described in this document, robustness testing systematically explores the domain of all inputs or conditions to which the software may be exposed under nominal and non-nominal conditions. Robustness testing in terms of performance is out-of-scope in context of this process.
RQBT	see Requirements-based Testing
SOW	Statement Of Work
Test step	The execution of a function-under-test for a given test vector.
Test vector	The combination of input and out vector representing the test data

1.4.3 Acronyms and Abbreviations

ATG	Automatic Test Generation
CG	Code Generator
CBTDG	Constraint-Based Test Data Generation
DCRTT	Dynamic C Random Test Tool
ESA	European Space Agency
FAST	Flow-optimised Automated, Source-code-based Test process
FI	Fault Injection
ISVV	Independent Software Verification and Validation
LC	Lifecycle
LOC	Lines Of Code
MMU	Memory Management Unit
n/a	not applicable
OS	Operating System
OSS	Open Source Software
RQBT	Requirements-based Testing
RTF	Rich Text Format
TOC	Table of Contents
V&V	Verification and Validation

2. THE FAST PROCESS AND DCRTT

2.1 THE LOGIC FLOW

Fig. 2-1 shows the principal logic flow of the FAST process from test preparation to result evaluation. DCRTT is the tool implementing the FAST process.

The process is driven by

- the source files,
- requirements or oracles
- 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 in case of unit testing.

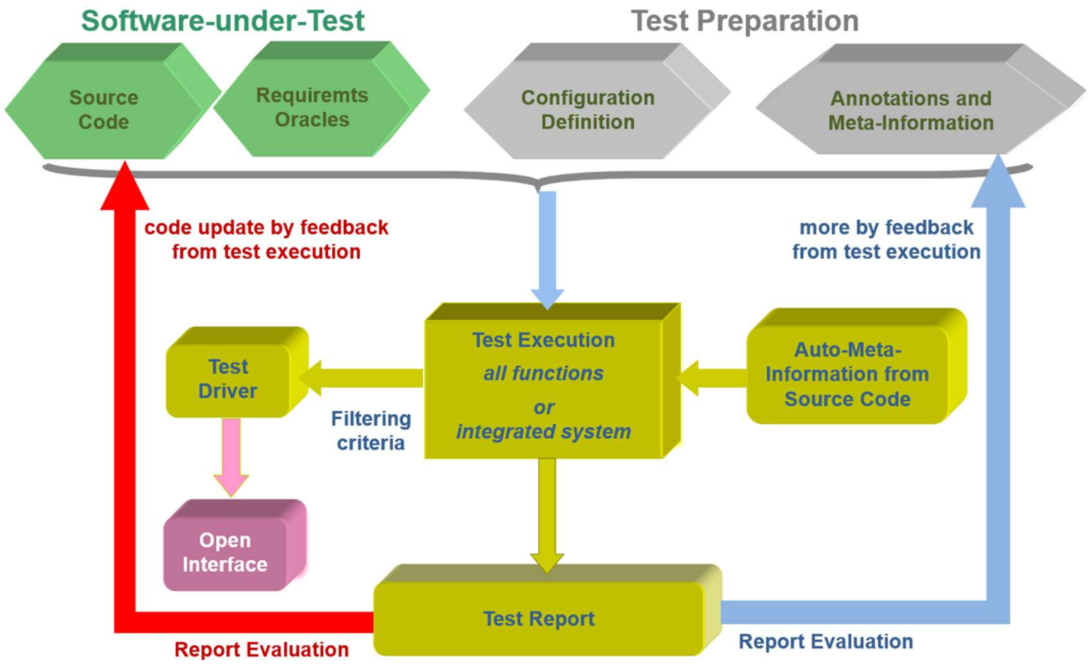
Software-under-Test
Test Preparation
Source Code
Requiremts Oracles
Configuration Definition
Annotations and Meta-Information
code update by feedback from test execution
more by feedback from test execution
Test Driver
Filtering criteria
Test Execution all functions or integrated system
Auto-Meta-Information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

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

Multiple runs may be executed under different test configurations.

The process starts with analysing the source code and ends with provision of test reports and test drivers with recorded input-output vectors of functions-under-test, proposing test cases for regression testing with DCRTT or other test tools. Every step 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 conforming compliance with requirements. The test drivers can be used for regression testing on host and target.

The manual compliance check replaced by automated checks when oracles are derived automatically from – suitable – requirements. Then the passed/failed result can be propagated bottom-up in the requirements hierarchy – if the tracking information is available. An oracle represents a requirement and may be derived automatically from a machine-interpretable requirement.

Annotations and meta-information complement a possibly insufficient context, e.g., to provide constraints and correlations. Constraints represent information not available in the source code, but required for certain test configurations, e.g., to constrain test data to nominal data – thereby excluding invalid, non-nominal data from stimulation – and to ensure proper initialisation of the test environment. Other correlations on data items instruct DCRTT to consider dependencies between function parameters, e.g., between a pointer and the length of the area it points to.

DCRTT tries to derive such information automatically from the source code to the degree possible. A suitable programming style does help to do so.

Stimulation of the functions and test driver generation is performed in a host environment. The input vectors are automatically generated.

2.2 PROCESS DETAILS

As shown in Fig. 2-2, the individual functions-under-test – in the sense of program subroutines or functions – are automatically stimulated in the test environment based on information extracted from the source code, possibly complemented by user-provided meta-information on pre-conditions or from oracles.

For stimulation, information on the prototypes of the functions is automatically extracted from the body of the function, providing the types of the parameters to be passed, as well as information on type definitions and control flow information. Similar information is extracted for global data which are used in the function-under-test.

Target-specific features in the source code may be automatically adapted to the host environment by auto-porting. Such support is already available for a number of platforms.

Feedback to the user (test engineer) is given immediately after automatic building of the test environment and after stimulation and execution of the test drivers on the host or target system.

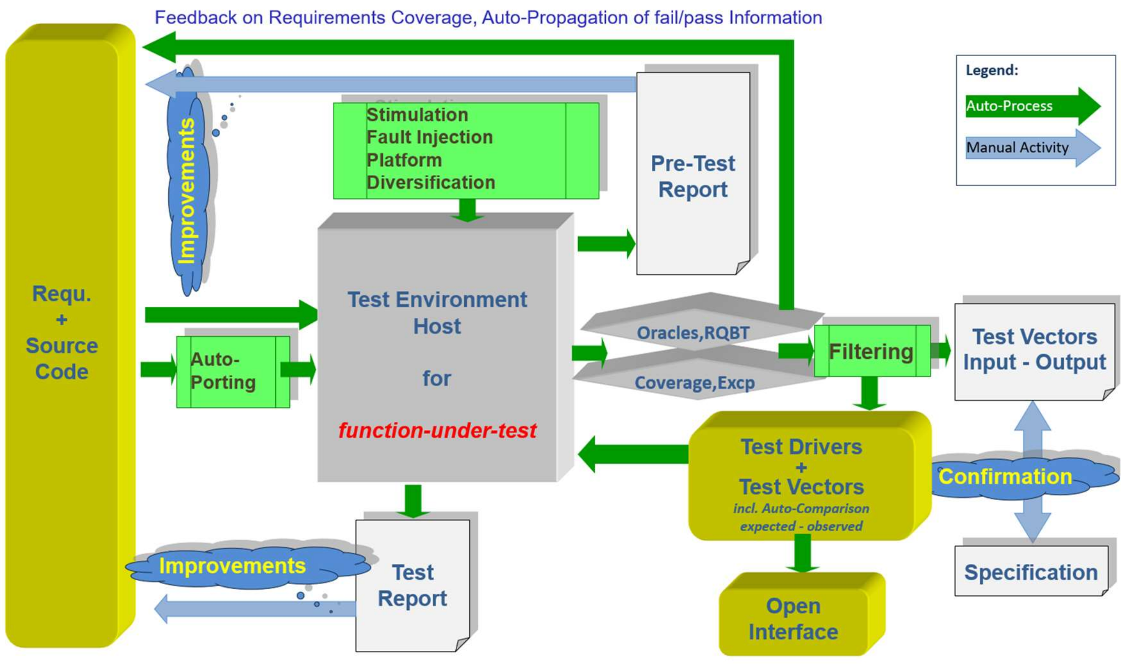

Fig. 2-2: The FAST Process – Detailed Flow

Information on resource consumption by the function-under-test, coverage, anomalies and more properties are recorded. The test drivers form the base of the Open Interface to other tools (see Chapter 7 for more details).

From the large set of stimuli, a subset of test case candidates is automatically selected based on criteria of interest such as coverage or occurrence of anomalies, where each candidate is defined by the input provided to the FUT and the observed output from the FUT. Using these data, test drivers are automatically generated for (re-)execution on the target, followed by automated comparisons of results achieved on target and host, documented in detailed and summarised reports.

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.

When the stimulation phase is completed, the reported anomalies are manually analysed. The data in the report point to the location of the defect – except when no post-mortem information is available – line in case of a heavy crash. In latter case the closest location for which recording was possible is reported.

The source code of the application or the generated test environment may be modified for further analyses – e.g., by inserting test printouts for debugging purposes – and tests can be easily and quickly rerun.

The anomaly report – a csv-file – provides information on line numbers in the pre-processed c-files and the original files.

Explicit fault injection can be activated. Then constraints placed on the allowed values for input parameters and global variables in stimulation are ignored – without any need for manual intervention. Modification of global variables declared as constant, blanking of initializers for global variables and modification of output and return values from called functions are supported.

Together with massive stimulation, fault injection creates a harsh environment for the functions subject of testing in order to increase the probability of occurrence of anomalies.

2.3 INTERFACES

Fig. 2-3 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.

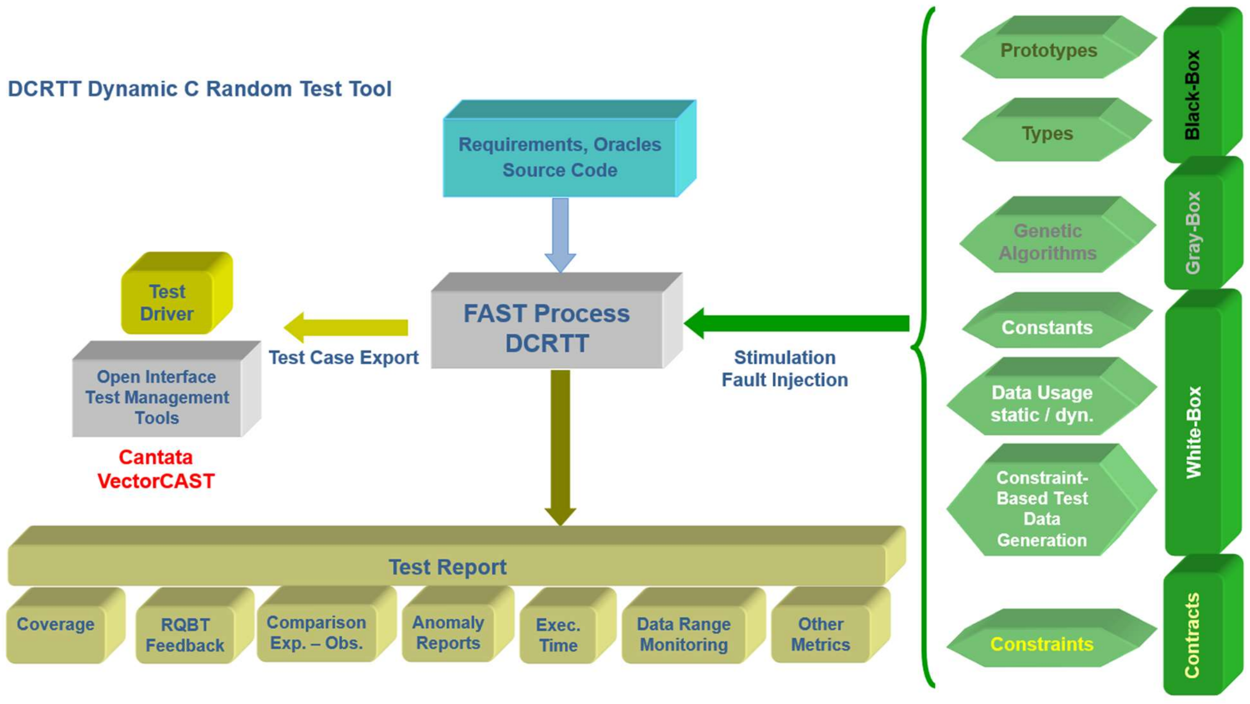

Fig. 2-3: Interfaces of the FAST Process

Four principal stimulation mechanisms drive test data generation using different sources of information:

- black-box stimulation
  based on a function's prototype
- grey-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 such mechanism may be applied to global data used in the function-under-test, too.

For robustness testing the information on type ranges is sufficient to derive stimuli. Then the contracts are not required. However, if the range of the stimuli shall be limited to the valid range, i.e., the range for which a function shall work correctly, more information on the context in the intended operational environment is required.

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 heuristic rules applied by DCRTT.

Missing functions or data are generated automatically. For out- and return-parameter values are generated randomly according to the type. Composite structures are supported.

A number of reports is provided as text or graphics.

3. TEST PREPARATION

The task of test preparation is to support test execution under – an intentionally huge – variety of conditions. The following steps are part of test preparation;

- provision of an environment for execution of the tests,
- generation of test data.

3.1 PROVISION OF A TEST ENVIRONMENT

Provision of a test environment comprises the following steps for DCRTT:

- checking of the provided source code for feasibility of compilation,
- generation of batch-files driving the tests,
- completion of the source code regarding missing symbols (functions and data),
- identification of the parameters to be stimulated,
- support of initialisation, and
- provision of means for check on run-time errors, exception handling, test monitoring and test evaluation.

Checking of the source code is performed by compiling the provided files on the host environment. If the source code is not developed for MS-Windows, the files are adapted automatically based on additional files provided by DCRTT for already supported platforms. In case a platform is not supported, GSSE needs to define the support. If compilation errors occur the process is terminated.

Generation of batch-files for execution and control of the tests is done automatically. Batch-files related to unit / module tests are collected in a single batch file for execution. It is started automatically when test preparation is completed.

Completion of the source code is required when functions or data are not part of the provided source files. In this case the declaration of the missing symbols must be visible, so that DCRTT can generate stubs for the functions and define the missing data. Random values are assigned to return and output-parameters of stubs.

Identification of parameters of a function is required for calling and execution of the function-under-test, which also could be the main-function. For every parameter – including global data used in a function – a copy is created which takes the values before the function call, so that a comparison before-after can be done – even for every element of a composed structure.

To **support initialisation** patterns for functions and files can be provided to select initialisation functions which shall be called prior to test execution to ensure that the tests are executed in a properly initialised application environment. The functions may be part of the set of functions-under-test or provided in addition.

Provision of means for test execution and result evaluation is required to ensure that a huge number of test steps can be executed and the result be documented. In principle, the function-under-test is a non-cooperative partner which may cause a crash of the test sequence – including e.g., an endless loop – and prevent any further test progress. Therefore, means have to be implemented which ensure a continuity of the – possibly thousands of – functions-under-test, and to continuously record the execution results.

DCRTT does all these tasks automatically.

3.2 TEST DATA GENERATION

Test data generation has to consider a number of aspects:

- the test purpose,
- the test mode, and
- the generation of stimuli for a function-under-test.

3.2.1 Test Purpose

A **test purpose** supported by DCRTT may be

- unit testing (module testing),
- robustness testing, or
- testing of the integrated software.

In all cases the test process is automated by DCRTT to allow injection of a huge number of test steps.

Unit testing aims to provide test data for the function-under-test to demonstrate compliance with requirements by a check whether the output vector complies with the input vector. In most cases, this implies that the input data must be valid data w.r.t. to the code of a function, i.e., the values must lie in the nominal range. Depending on the test requirements invalid data, i.e., non-nominal data, may be fed in as well. In the latter case unit testing does cover robustness testing, too.

Consideration of the nominal range in case of auto-generation of test data is a challenge, as the required information on the valid data range might not be provided explicitly nor implicitly in the source code. In this case, additional information needs to be provided in files or by annotations. However, DCRTT tries to identify the missing information from the source code – to the extent possible – by applying heuristic rules.

In case of automated provision of test data – as supported by DCRTT – the function-under-test is called in a loop for a given number of test steps.

In case of **robustness testing** a function-under-test is exposed to nominal and non-nominal data. Then test data generation is not limited to the nominal range. A function is considered as robust if it can tolerate non-nominal conditions and does behave as defined in the requirements. The check on compliance of input-output vector is also part of robustness testing.

Testing of the integrated software means that the main function and – possibly – external data interfaces receive the test data – nominal or non-nominal data, similarly as for a function-under-test.

If more than one main-function does exist – e.g., in case of multitasking – a single entry-point function should be provided or constructed. The entry-point function is called in a loop like for the function-under-test in case of unit testing.

All the means (see Ch. 4 and Ch. 6) provided for unit or robustness testing can and should be used for integration testing, too. Therefore, testing of the integrated system requires generation of the test environments for all the functions-under-test in a previous step, i.e., execution of unit or robustness testing.

In contrast to automated unit testing, auto-testing of the integrated testing brings the advantage that the internally passed data are in the valid range for all functions below the main-function, except when a function behaves erroneous and provides invalid data.

In case of tasking a challenging issue is the automatic construction of the entry-point functions and the implementation of task synchronisation in the entry-point function, because of a missing standardised interface for access of the required information.

Currently, DCRTT supports a specific interface, which however can be reused for an application applying the format or adapted to another customer-specific format.

3.2.2 Test Mode

By the test mode fault injection (FI) may be activated, i.e., exposing a function to non-nominal conditions. By enabling or disabling this DCRTT feature either unit or robustness testing can be performed.

Faults may be enforced for the following cases:

- fault injection after a function call[1]
    - return value of a function
      e.g., to enforce that malloc returns NULL
    - output-parameter of a function
- fault injection for input vectors.

The first case is supported for scalar return and pointer parameters. The second case applies to all data types subject of auto-test data generation.

If the general FI feature is disabled it is still possible to inject NULL pointers by activation of a configuration option.

The rate of fault injection after a function call w.r.t. to a sequence of calls can be configured – independently for the scalar and pointer case. As the number of calls during a test step is not known, a fault is injected always for the first call.

3.2.3 Generation of Test Stimuli

Several approaches are applied for generation of test stimuli:

- type-based generation
- constrained-based generation, and
- generation with genetic algorithms.

Type-Based Stimulation is supported for every type including complex user-defined types, except for function pointers (development of support for the latter is in progress). In this case the value of a scalar type is derived

- randomly, or
- grid-based / incrementally

from the related range in C. In non-nominal case this is the full range of C types, in nominal case the range is limited by provided constraints, if any.

Grid-based / incremental generation means that a grid – not necessarily with equidistant intervals – of a given number of points is put over the type range / input domain, starting with the minimum available value for the type and then incrementing stepwise until the maximum value for a user-defined number of steps. In this case for every parameter of the function-under-test – including global data used by it – a loop is generated for stepping through the grid.

Random generation means use of a pseudo-random generator for which a user can give a seed value. In this case the function-under-test is called from one loop only, and the test data for the items of a parameter are generated simultaneously.

Constraint-based generation and genetic algorithms complement type-based generation when activated. They need significantly more time. However, as by random or grid-based date generation not all branches may be reached which results in a coverage figure less than 100%, constraint-based data generation may be needed.

[1] Functions declared as static are currently not supported by this feature.

In case of **constraint-based generation,** a coverage target is identified which has not yet been met. For missing block/statement coverage, a location in the code is given which was not reached, yet. Similarly, for decision or condition coverage, an edge may be specified which has not been traversed yet. In both cases a path through the code is identified – starting with the related parameters at the top – by which the desired coverage can be achieved. The constraints – in terms of conditions – along the path yield a system of inequations which needs to be solved. The result is a set of values for the parameters which are fed into the function-under-test during additional test steps.

Genetic Algorithms[2] apply the principle of natural evolution to generate test data for achievement of coverage. They are still a research topic. Typically, they are applied to untyped byte streams – like occurring for telecommands or telemetry data – of which the structure is dynamically varying and unknown when being received. In this case the decoding or encoding is defined in the code.

When knowing the correlation of such a stream with a parameter and its rules for constructing the contents, the construction rules could be applied. Such a typical case is the provision of contents of telecommands. For example, when getting a composed type where the contents can vary dynamically based on included information, genetic algorithms may be the right choice.

Genetic algorithms correlate the chance of survival and procreation of individuals with their level of adaptation to their environment. The formalised concept describes the progression of a population of candidate solutions over a number of generations.

Each generation arises from the previous by application of a set of evolutionary operators to the individuals of the previous generation, including recombination in pairs, mutation, survival of elite individuals into the next generation and immigration of new individuals. An individual's progression into the next generation or preservation of part of its features depends on its fitness to the purpose of optimisation.

A cost function decides on the fitness of a generation, by considering the relevant properties required for survival.

[2] Genetic algorithms are currently a research topic in context of DCRTT. This feature is currently evaluated in context of generation of telecommands.

4. TEST MONITORING AND CONTROL

Test monitoring aims to collect information about

- detection of anomalies,
- coverage,
- data ranges for data used inside a function, and
- resource consumption.

Test Control modifies the control flow, e.g., for fault injection after a function call.

Instrumentation of the source code is required and implemented by adding code, expanding existing code or substituting the code, e.g., to modify the control flow.

4.1 ANOMALY DETECTION

DCRTT support for anomaly detection includes

- index checking in the provided source code,
- index checking for a subset of C library functions,
- address verification in case of pointers, and
- checking for corrupted malloc-memory.

For support of **index checking** index expressions are instrumented. In case of (constrained) arrays the size of the array is used to detect out-of-bounds conditions. In case of pointers or unconstrained arrays the size is detected by bookkeeping of DCRTT at run-time. This allows to identify the size of statically or dynamically allocated memory blocks pointed to by the array base pointers used.

Parameters are allocated in the test environment. In case of pointers and unconstrained arrays the actually required size is not part of the prototype of type information. It may be deeply hidden in the code and/or even defined at run-time. To avoid false positives due to allocation of the wrong size, DCRTT supports dynamic and silent resizing of such objects, and does not issue a message, if the index is beyond the current size. Instead, it logs the required size so that a later check against requirements can be done in order not mask a true positive.

C library functions are substituted by DCRTT functions which check the parameters for valid address and out-of-bound conditions, and perform resizing of the passed parameters if required. The following functions are currently supported:

strchr
strcmp, strncmp, memcmp
bzero, memset
memmove, memcpy
strcpy, strcat, strncpy, strncat
sprintf

The feature of **address verification** checks addresses for whether they are writable, readable or none of both. For objects the start and end addresses are checked. Although this is done in the host environment where the memory layout may be different from the one on the target system, invalid addresses detected this way on the host environment may occur in both environments. It even may happen that the chance to detect it on the host may be higher than for the target because the host uses an MMU, but the target does not.

DCRTT supports **detection of corrupted malloc-memory**. The previous checks aim to localize an anomaly in the source code and have information on the file and line. However, if such checks are not applied at the respective location, memory can be corrupted because the invalid condition cannot be prevented.

DCRTT applies its own housekeeping for malloc, for detection of memory usage in general and memory leaks in particular, and for detection of corrupted memory. DCRTT allocates malloc-memory for declaration of data in the test environment, to the degree possible. To increase the chance to detect corrupted memory DCRTT adds buffer zones in front and behind the allocated memory area and initialises it with a certain pattern. A check on the integrity of these two additional areas for all allocated data is performed after each call of the function-under-test per default. If integrity of a data is violated, the whole allocated area is printed in hex-format into the log-file together with more information and the corrupted bytes are marked. The location of corrupted bytes may already give an idea on the source of the defect

When a violation is detected, the test run is terminated and a re-run is started with a debugging option allowing generation of a detailed pre-mortem trace. To localise the location at a higher granularity the coverage instrumentation does checks when a block is entered and left, to identify the earliest location of violation. If the information on the block level is not sufficient, the user may insert own checks into the code. This procedure is described in the User Manual.

Tab. 4-1 provides the list of messages issued by DCRTT on anomalies in the third column. In the first column of the table standardised terms are given onto which all DCRTT specific messages are mapped. These so-called “standard defect types” were defined to support comparison of messages coming from tools which usually differ in the message text, although referring to the same symptom. The contents of the second column “Criticality” shall indicate a measure for the probability that the defect type will impact the quality of service. In fact, “Criticality” here represents experience from evaluation of tool messages and following assessment on the impact weighted with the probability that the defect really will impact quality of service.

The classification on criticality shall allow prioritisation of reports based on experience at presence of the challenge of evaluation of a large set of reports.

Standard Defect Type	Criticality	DCRTT Messages	Description
Array Index Out-of-Bounds	Critical	CorrMem	Corrupted memory detected
Array Index Out-of-Bounds	Critical	OutOfRangeLow	Index <0
Array Index Out-of-Bounds	Critical	OutOfRangeHigh	Index > maximum value for constrained arrays
Assert failure	Critical	AssertFailed	Assertion failed
Dereference of Invalid or NULL Pointer	Critical	*Excp	A number of different messages on exceptions depending on the location in code (application or test environment)
Dereference of Invalid or NULL Pointer	Critical	ExcpMissFunc	Exception in a generated stub
Dereference of Invalid or NULL Pointer	Critical	ExcpBasicFunc	Exception in a support function of a stub
Dereference of Invalid or NULL Pointer	Critical	ExcpDataProcess	Exception in data range monitoring function
Dereference of Invalid or NULL Pointer	Critical	ExcpNULLInj	Exception after injection of a NULL pointer
Dereference of Invalid or NULL Pointer	Critical	StdExcpC++	Standard exception from C++
Dereference of Invalid or NULL Pointer	Critical	TermExcpC++	Termination exception from C++
Dereference of Invalid or NULL Pointer	Critical	InvalidAddr	Access of an invalid address, general message if source cannot be exactly determined
Dereference of Invalid or NULL Pointer	Critical	AddrIsReadOnly	Address is not writable
Dereference of Invalid or NULL Pointer	Critical	AddrIsNotRW	Address is not readable and not writable
Dereference of Invalid or NULL Pointer	Critical	AddrIsNULL	Address is NULL, e.g. passed to index checking
Dereference of Invalid or NULL Pointer	Critical	NULLptrDeref	Dereference of a NULL pointer
Dereference of Invalid or NULL Pointer	Critical	Uexit	Unexpected termination of a test, condition could not covered by any of the implemented exception handlers, probably due to an invalid address
File Access Error	Critical	FileHandleErr	File handling error (open, close, file access, not opened, not closed)
Non-terminating Loop	Critical	FileTooBig	Log-file too large, possibly an indication of an infinite loop condition
Possible Recursion	Critical	RecursExcp	Exception during exception handling
Possible Recursion	Critical	Recursion	Stack overflow possibly due to recursion
Resource leak	Critical	Resource leak file	File not closed
Resource leak	Critical	Resource leak malloc	malloc-memory not freed

Standard Defect Type	Criticality	DCRTT Messages	Description
Arithmetic Overflow	Warning	FpNan	Contents of floating-point data is not a number
Arithmetic Overflow	Warning	FpInf	Contents of floating-point data represents infinite
Arithmetic Overflow	Warning	intOverflow	Integer overflow occurred
Concurrency Issues	*Warning*	*tbd*	*The support will be given soon.*
Invariant Condition	Warning	WasAlwaysTrue	The condition was always true, possibly invariant condition
Invariant Condition	Warning	WasAlwaysFalse	The condition was always false, possibly invariant condition
Timeout during Execution	Warning	TimeoutIntMonitor	The test run was terminated due to reaching the time limit, possibly a deadlock or the system hangs
Unreachable Code	Warning	WasNotReachedBlk	The block was never reached[3]
Unreachable Code	Warning	WasNotReachedCnd	The condition was never reached[4]

Tab. 4-1: List of Reported Defect Types

4.2 COVERAGE

Block, condition and path coverage is supported. In case of **block coverage,** a checkpoint is inserted at the entry and the end of each block. The end of a block is reached if no exception occurs in the block. In latter case exceptions of a block are flagged.

Block coverage is recorded for

- each FUT separately,

  i.e., the block coverage when executing the function-under-test,
- the sum over all FUT, collected over all function tests, and
- the cumulated coverage,

  considering contributions to coverage of a certain FUT when execution another FUT as function-under-test.

Differential block coverage is supported for test cases in both directions, i.e., correlation of a block with test cases, and correlation of a test case with blocks covered when it is executed.

Path coverage is integrated into the block coverage recording – when the related configuration option is on.

In case of **condition coverage,** the cumulated condition coverage is recorded for true and false. From this information MC/DC can be derived. Also, **differential condition coverage** is recorded for test cases as for blocks.

The coverage of a *case* of a *switch* is handled by block coverage. Discrete values occurring for a switch are currently not collected (as the set could result in a huge number of cases), but could be.

As an option, coverage obtained during initialisation can be counted, too.

Graphics are provided for the structure of a function based on decision coverage and for differential coverage w.r.t. test cases.

As an option coverage information of blocks of a function-under-test can be printed into the log-file together with the number of accesses of a block and consumed time starting measurement at the function entry. This may help to detect the block of a deadlock or an infinite loop.

---

[3] Such a report should only considered as valid if the block coverage is about 90% or higher. Otherwise the probability is high that it is just a matter of insufficient test data.

[4] Similarly, such a report should only considered as valid if the condition coverage is about 90% or higher. Otherwise the probability is high that it is just a matter of insufficient test data.

4.3 DATA RANGE MONITORING

For every location in a function where an assignment happens – including initialisation in a data definition – the data contents is recorded before and after the assignment, and for both cases the minimum and maximum values are determined and reported. This also applies to complex and nested data structures.

The intention of supporting tracking of ranges is that this way unexpected or invalid values can be identified, or a feedback can be given on the ranges.

4.4 RESOURCE MONITORING

Resource monitoring is supported for

- malloc usage,
- heap and stack usage,
- file usage
- execution time

Malloc usage is tracked with size and location (function, file, line). This applies to allocation and free. In case data can be subject of resizing, the size of the related type is tracked, too. Due to tracking resource leaks can be detected. Optionally, information can be printed into the log-file on which data items were allocated by the application, but not freed.

In case of resizing, all data items which depend on the same literal initially used when allocating memory, are subject of update.

Heap and stack usage is tracked with total size, element/type size, data name, type name, object properties such as the array represented, and location (file, line). In case of stack objects and dynamic heap objects, allocation and free is considered.

By taking the address, information can be obtained at run-time regarding malloc, heap and stack objects. Especially, this feature is used by DCRTT itself to minimise false positives.

As an option time consumption of blocks of a function-under-test starting measurement at the function entry can be printed into the log-file together with the number of accesses of a block. This may help to detect the block of a deadlock or an infinite loop (see also 4.2).

5. TEST EVALUATION

Apart from the provided reports as described in Ch. 6 below, evaluation support is given by

- Test drivers, and
- Requirements-Based Testing (RQBT)[5].

Test drivers are generated according to selection criteria (coverage, exceptions, requirements coverage). They represent test cases when the test input can be correctly linked to a requirement, otherwise they are test case candidates. A test driver allows to (re-)execute a certain test step which is considered as interesting according to the selection criteria.

In case of test case candidates, the check on compliance of input- and output vector regarding requirements has to be done manually. Once done, the output-vector is a reference for regression testing.

Test drivers **support regression testing** and comparison of output-vectors between host and target system, e.g., to detect **numerical instabilities**.

A test driver

- establishes the environmental conditions observed before the call of a function-under-test,
- calls the function-under-test, and
- records the output-vector when executing a test step, and compares this vector with the vector obtained by execution of the function-under-test,
- compares input- and output vector for in-out parameters in order to identify which element did change.

Input- and output vector generation and comparison apply to the full set of elements of a possibly complex and nested data structure (currently except function pointers).

In case of DCRTT and Cantata all test drivers generated during a test of a function are collected in a test environment for re-execution. In case of VectorCAST re-execution is organised according to the tool interface (see Ch. 7).

Support of **RQBT** aims to close the gap between test inputs independently generated of requirements and requirements coverage and verification, and takes benefit of massive stimulation to check the requirements.

Requirements coverage means that a requirement was covered by a certain test step executed for a certain function-under-test, *requirements verification* to assess whether the test yielded pass/fail for a requirement. The **information** on **coverage and verification** can be **propagated bottom-up** provided that the information for propagation is made available in machine-readable format. DCRTT expects a certain format, but other formats may be converted if needed.

A requirement should contain information on the related subject, the initial property (pre-condition) and the expected, final property (post-condition). For example, a subject may be a state transition (subject) from an initial state (pre-condition) to the final state (post-condition).

Support of RQBT requires information on a requirement in machine-interpretable format. Currently, such requirements do not exist in space projects – according to our current knowledge. Therefore, a machine-interpretable format has been defined for expressing suitable test requirements.

In DCRTT such a suitable requirement is formalised by an oracle (Fig. 5-1) with pre- and post-condition.

[5] For details see Ch. 8

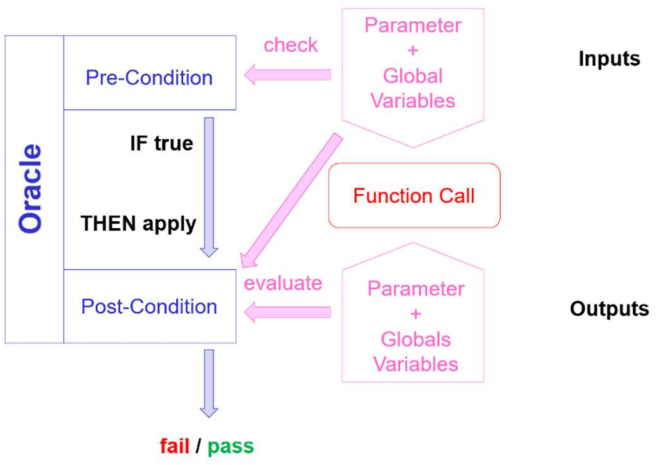

Fig. 5-1: Principle Format of an Oracle

For each oracle a set of functions is constructed automatically and included into the test environment. Whenever the pre-condition of an oracle is true for a test vector, the post-condition is checked. The result of the check (pass/fail) is recorded and later propagated bottom-up through the requirements hierarchy. Usually, the pre-condition check only applies to the output-vector, but it also may apply to the input-vector when the value of an item before execution of the function-under-test is referred to. In this case the conservation of the input vector by DCRTT is required for evaluation after execution of the function.

The notation of an oracle is

```
<short name> ; <pre-condition> ?6 <post-condition>; [<function pattern>[,<file pattern>]];
```

Pre- and post-conditions must be valid C / C++ logical expressions evaluating to a boolean value.

A short name refers to a requirement and correlates a requirement with an oracle. More than one requirement may be correlated with an oracle. Vice versa, more than one oracle may be correlated with a requirement.

The function and file patterns are optional. They may constrain the set of functions to be considered for matching of oracles. Any names supported by the respective file system are allowed. Wild cards (*, ?) are allowed, with “*” matching any sequence of characters, and “?” matching a single character. If a function or file pattern is missing, “*” is taken instead by default. *Tab. 5-1* provides some examples of (simple) oracles and *Tab. 5-2* shows the results for mathematical oracles and the information provided by DCRTT on the oracle checks.

Function	Oracle	Description
x *x (square)	FORALL(x)?(sqrt(ret)-fabs(x))<eps	An inverse function is applied in the post-condition
abs(x)	FORALL(x)?ret>=0	The abs-function should always return a positive result or 0
invariant	p==c? pInput==p	In this case p should not change, and pInput is the value of p before execution, preserved by DCRTT

Tab. 5-1: Examples for Oracles

[6] The question mark used here is part of the oracle syntax. It should not be interpreted as “?” used in the syntax of the C language in case of a conditional expression. It has a similar meaning as it follows a condition expressed in C code, but it is not inserted in the code context of the FUT. Only the pre-condition and the post-condition must be provided in valid C-syntax.

Function	Requirement	Oracle		Number of Tests	Oracle Output			RQ fully covered	RQ verified
		Pre-Condition	Post-Condition		Coverage	true	false		
x*x	∀x∈{double} $\sqrt{x^2}$ shall not differ from x more than ε	`x≤-1.0 || x≥1.0`	`(fabs(fabs(x)-sqrt(retVal))/x)<eps`	302	299	225	74	yes	no
		`x>-1.0 || x<1.0`	`fabs(fabs(x)-sqrt(retVal))<eps`		3	3	0		
abs(x)	∀x∈{sint} abs(x) shall be ≥0	`RQBT_FORALL(x)`	retVal>=0	302	302	301	1	yes	no

Tab. 5-2: Results Obtained for Mathematical Oracles

6. REPORTING

Reporting about test results can be grouped in

- the log-file for a test (Ch. 6.1),
- a set of files providing information during the test process for a number of topics (Ch. 6.2),
- the anomaly report file provided in csv-format (Ch. 6.3), and
- the test document (Ch. 6.4).

6.1 THE TEST LOG-FILE

For every test log files are created:

- lg<testId> for the stimulation step,
- lgExecDCRTT<testId> for execution of the corresponding test driver in DCRTT mode,
- lgExecCantpp<testId> for execution of the corresponding test driver in Cantata mode,

The log-file provides information on test mode, test progress, files left open, malloc-memory not freed (optionally), and anomalies.

For every anomaly the execution trace over checkpoints is printed together with information on the context. Also, summary information on anomalies is provided and optionally checkpoint counts and execution time related to checkpoints (see Ch.4.4, Execution Time).

6.2 GENERAL INFORMATION

Tab. 6-1 provides a list of files with information recorded during execution of DCRTT referring to different features supported by DCRTT.

File	Description
recordCBTDGerror.file	Errors related to constraint-based testing (CBTDG)
recordCBTDGunreachableLoc.file	Information about unreachable locations as reported by CBTDG
recordCompletionStatus.file	Information about completion of a test of a function-under-test (FUT)
recordCompMsgs.file	Messages from compilation and linking of test environments
recordCompMsgs_filtered.file	Messages from compilation and linking of test environments, but filtered for critical messages
recordConstraintDef.file	Statistics on applied constraints
recordCriticalIndex.file	Messages on out-of-bound conditions with full information on the context and for every location supporting identification of value==upLim conditions
recordCritIdxCompr.file	Filtered messages on out-of-bound conditions
recordDefDupl.file	Information of duplicated symbols found in more than one file, but not declared as static
recordEnforcedTestModeSwitch.file	Information whether the configuration for type-based stimulation had to be modified
recordFileOpen.file	Information on file open and close per location
recordFileSummary.file	Summary on file open/close
recordMallocAllocated.file	Details on malloc-usage
recordMallocSummary.file	Summary on malloc-usage
recordMemUsedInfo.file	Information on malloc-memory
recordMinMax1.file	Information on automatically defined constraints (min/max) for arrays derived from index information in the source code
recordParaInitTime.file	Information on (max.) time consumed for generation of test data for a parameter
recordParaPrintByte.file	Information on the number of bytes printed into the report for parameters
recordResizeReports.file	Information on resizing of pointer arrays and unconstrained arrays
recordRunTimeMsgs.file	List of observed run-time messages extracted from log-file of a test regarding error messages from DCRTT library functions
recordSfmfLeftUndefSymbols.file	List of missing symbols for which an instance could not be provided (excluding symbols form compiler library)
recordSfmfUndefSymbols.file	List of missing symbols for which an instance could not be provided (including symbols form compiler library)
recordTCcnt_vs_testId.file	Information on number of test cases generated for a certain function-under-test
recordTCfilePreemption.file	File pre-emptied while reading the information on which block has been covered by a test case
recordTCinherit.file	Information on test cases already generated in previous unit tests to minimise the number of test cases
recordTCmatrix.file	Information due to which event (block/condition coverage, exception etc.) a test case was generated
recordTCmatrixBlockNotPath.file	Similar ot recordTCmatrix.file, but only blocks are considered
recordTCmatrixMCDConly.file	Similar ot recordTCmatrix.file, but only conditions are considered
recordTCmatrixPathNotBlock.file	Similar ot recordTCmatrix.file, but considers contributions from path coverage, only
recordTCsccCover.file	Information on coverage of short circuit conditions for decisions
recordTCselection.file	Information on TC and related blocks and condition for further evaluation, for DCRTT internal use
recordTCselMissTestId.file	Information on unit tests for which basic path did not contribute
recordTestAnomalies.file	Information on anomalies occurred during test execution
recordTestdriverStmtLimit.file	Information on test cases where the test driver could not be completely generated as the limit of source code lines was exceeded
recordTestModeTimeAnalysis.file	Information on number of test cases, covered blocks and conditions, execution time for each of the generation modes random, incremental, CBTDG, genetic (cumulated information over the generation modes)
recordTestSteps.file	Statistics on number of generated stimuli
recordWrongParaList.file	Information on functions for which an inconsistent parameter list was detected by DCRTT

Tab. 6-1: Overview on Files providing Information on Test Results

6.3 THE ANOMALY REPORT FILE

The found anomalies are reported by DCRTT in compressed manner in a csv-file called

PerToolReportList_DCRTT.csv

The format of the file can be processed as text file or imported by a spreadsheet program.

This format may also be used to collect information from other test tools for purpose of comparison of results from other tools.

The file format comprises 34 elements in total of which the most relevant for defect report are shown in Tab. 6-2. The other elements address links to evaluation, more information on the location of the defect provided by other tools, assessment results and justifications of the assessments.

The file provides information on the defect type and its location. The defect type is provided as “standard defect type” (see Ch. 4.1 and Tab. 4-1) together with the DCRTT specific message on the defect type.

Fields 33 and 34 provide additional information on the context (Tab. 6-3) and the affected (instrumented) code (Tab. 6-4). The contents in Tab. 6-3 and Tab. 6-4 are not related to the same defects. Tab. 6-3 shows two examples for out-of-bound conditions and Tab. 6-4 examples for out-of-bound in a strcpy, a failed index check and a NULL pointer dereference (buf).

#	Item	Description
1	Issue Id Tool List, unique	Unique number in the list, starting at 1
5	File	Filename of file-under-test
6	Function	Qualified name of the function-under-test
7	Mangled Name	Unique name of the function in case of C++, for C identical with function
8	Line, actual	Line number in the file instrumented by the tool or original lime number
9	Line, original	Line number in the original source file
10	Column	Column in the line pointing to the originator of the anomaly
11	Stmt-No	Statement number in the line, if more than one statements are present in the line
12	Top-Level Function	Y / yes if it is not called by any other function in the set of functions-under-test, else n / no
13	Standard Defect Type	An identifier describing the anomaly used for classification of anomaly messages from different tools
14	Tool Primary Message	A message from the tool considered as short message describing or classifying the anomaly
15	Tool Secondary Message	Another message used to map it onto the standard defect type
16	Tool Report Text	Another message from the tool – if any- describing the anomaly
33	Additional information1,	Information on context of the anomaly, for other tools optional
34	Additional information2	Statement or block, for other tools optional

Tab. 6-2: Description of the Anomaly Report Format (Subset)

additonal information 1: Information on condition, arrays, indices and limits
testId=0, func=00000, block=13 condId=n/a arrId=555 idx=1 Value=33, upLim=25 lcnt=590
testId=0, func=00000, block=1 condId=n/a arrId=666 idx=1 Value=13, upLim=13 lcnt=600

Tab. 6-3: Examples for “Additional Information 1”

additonal information 2: related statement or block (instrumented)
stmt=strcpy_BSSE(buf,"12345678901234567890123456789012")!
stmt=buf[DCRTT_INDEX_CHECK(32, buf, -1, 0, 1, ptrTypePtr, DCRTTscopeGlobal, sizeof(char), 0)]=0!
stmt=*(buf+32)=0!

Tab. 6-4: Examples for “Additional Information 2”

Issue-ID	File	FuncName	Mangled Name	Line, actual	Line, original	Col	Stmt-No	Top-Level	FSVW Standard Defect Type	Tool Primary	Tool Sec.	Tool Report Text
1	test_assert.c	testAssert	testAssert	6897	96			y	Array Index Out-of-Bounds	CorrMem	2	CorrMem
2	test_assert.c	testAssert	testAssert	47	47			y	Assert failed	AssertFailed	32	AssertFailed
3	test_assert.c	testAssert	testAssert	6917	109			y	Array Index Out-of-Bounds	OutOfRangeHigh	21	OutOfRangeHigh
4	test_assert.c	testAssert	testAssert	6879	83			y	Dereference of NULL-Pointer	ExcpNULLInj	12	ExcpNULLInj
5	test_assert.c	testAssert	testAssert	6904	100			y	Dereference of NULL-Pointer	ExcpNULLInj	12	ExcpNULLInj
6	test_assert.c	testAssert	testAssert	6929	117			y	Dereference of NULL-Pointer	ExcpNULLInj	12	ExcpNULLInj
7	test_assert.c	testAssert	testAssert	6838	54			y	Array Index Out-of-Bounds	OutOfRangeHigh	21	OutOfRangeHigh
8	test_assert.c	testAssert	testAssert	6867	75			y	Dereference of NULL-Pointer	ExcpNULLInj	12	ExcpNULLInj
9	test_assert.c	testAssert	testAssert	6867	75			y	Dereference of Invalid Pointer	InvalidAddr	11	InvalidAddr
10	test_assert.c	testAssert	testAssert	6892	92			y	Dereference of NULL-Pointer	ExcpNULLInj	12	ExcpNULLInj
11	test_assert.c	testAssert	testAssert	6917	109			y	Dereference of NULL-Pointer	ExcpNULLInj	12	ExcpNULLInj

Tab. 6-5: Example Anomaly Report Issued by DCRTT

6.4 THE TEST DOCUMENT

The information in the test document can be grouped into

- general test information,
- exception information,
- coverage information, and
- information on test cases.

The contents of the document may be customised. The chapters and sections may be all included in one document or spread over several documents, and a subset of supported documentation features may be chosen.

A number of so-called “documentation operators” are provided each one generating the documentation for a certain report feature. The layout can be customised and free text can be inserted in additional sections or a section into which an operator inserts the desired results.

The document is provided in rtf-format. If required, other formats may be supported as well.

6.4.1 General Test Information

For the following topics information is provided

- Statistics and metrics

  Statistics on the whole test run, summary report
  Execution time statistics for host and target

  Cyclomatic Complexity (CC) per function together with information on coverage.

- Status information

  Anomalous termination per file and function
  Completion status of a test
  Block and check point information per function (coverage, exceptions) with coverage<100%
  Exceptions per file and function

- Graphics

  Anomalous termination
  Exceptions per file and functions
  Execution time on test run
  Number of blocks per function for all functions

6.4.2 Resource Consumption

The following information is provided:

- Summary Malloc Information
- Malloc Events Information
- Malloc allocation Information
- Detailed Malloc Information

- Summary File Information
- File Events Information
- File open Information
- Detailed File Information

- Heap and Stack Usage (calculated from the code and measured during execution)

6.4.3 Information on Code Anomalies

Following code anomalies detected during parsing of the source code are reported:

- Implicit record definition
- Incomplete types
- Multiple symbols
- Recursive types
- Type and variable name identical
- Wrong parameter list
- Nonsense types (types which do not make any sense)

6.4.4 Information on Run-Time Anomalies

The following information is provided:

- Test driver statement limit exceed
- Too many bytes to print per test step
- Test anomalies
- Enforced Test mode switch due to a too high number of total test steps
- Parameter initialisation takes too long

6.4.5 Anomaly Analysis

The following information is provided:

- Anomaly type vs. function, sorted
- Function and file vs. anomaly type, sorted
- Anomaly type vs. function for every anomaly type, detailed

6.4.6 Coverage

The following information is provided:

- Summary on block, checkpoint and condition coverage over all tests
- Summary on block, checkpoint and condition coverage per function
- Checkpoint coverage per function for all functions
- List on non-covered checkpoints
- Graphics on covered blocks over all functions
- Graphics on coverage showing the structure of a function and covered blocks in green, non-covered blocks in red, blocks with exceptions in blue, and non-existing *else*-blocks in orange.
- Histograms on block coverage

6.4.7 Decisions and Conditions

Information on conditions is provided as summary per function (occurrences of *false*, *true* and *false AND true*) and as truth table for every decision.

For every line in a truth table the number of occurrences is provided as observed during stimulation. The observed branching ratio may be compared with the expected one. A difference will indicate a logical error in the implemented decision.

6.4.8 Exceptions

Information on exceptions sorted by different criteria

- filename

- exception type,
- location
- user-defined category

6.4.9 Input-Output Vectors

The contents of input- and output-parameters and their first (numerical) derivative are represented by graphics for every test step during stimulation. Composite types are mapped onto a scalar by a metric considering all elements of the type. The derivative is calculated by the difference between two consecutive steps. For an input parameter the denominator is 1, for an output parameter the denominator is the difference of the input parameter.

The intention of the 1st derivative is to support identification of areas where the output does not change significantly or does change significantly, so that a user can decide whether the observed behaviour is compliant with the expected behaviour. This figure is only calculated for the incremental / grid-based mode.

Changes are recorded during execution of test drivers on host and target for

- in-out-parameters before and after the call separately for host and target,
- out-parameters for expected / confirmed and observed values on host and target, and
- out-parameters between host and target for the observed values.

In case of composite types all elements are considered.

Expected / confirmed relates to the value observed during the stimulation step which is confirmed either manually or by RQBT. *Observed* addresses the value actually obtained during execution of a test driver.

6.4.10 Execution Time

Figures and graphics on execution times are provided for

- the total time consumed for a test of a function, and
- the average, minimum and maximum execution time consumed by a function during execution on the host during a stimulation step.

6.4.11 Data Range Monitoring

For every location where a value is assigned the minimum and maximum value before and after assignment are recorded. In case of composite structures, the figures for every element are reported.

6.4.12 Test Case Visualisation

In order to provide information on the distribution of test cases over the input domain, graphics are provided. An example is shown in Fig. 6-1. For every parameter the position of the relevant value w.r.t. the full range is shown. In case of composite parameters, a metric is applied to represent the values of all elements.

For integer, floating point and composite types the range is divided into several categories like *low*, *high*, *very high* etc. For pointers *NULL* and *not NULL* is displayed, and for strings *NULL*, *empty* or *not Null not empty*.

For enumeration types all literals should be considered to the degree possible. In case of too many literals the range is displayed as for integer, floating point and composite types.

In addition, invalid ranges are indicated by “invalid low” and “invalid high”.

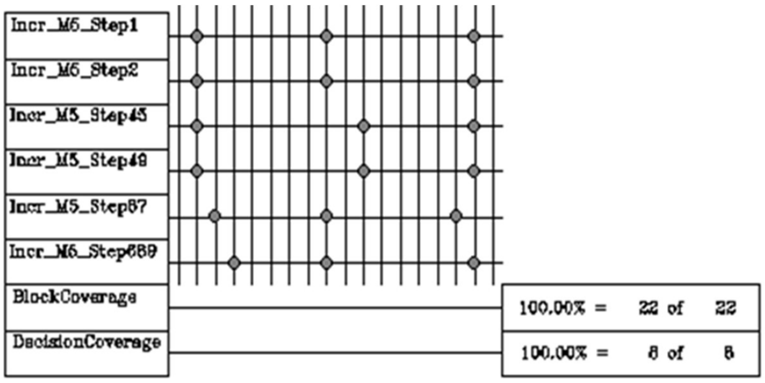

Fig. 6-1: Example for Parameter Value Distribution

6.4.13 Test Cases and Coverage

The relationship between coverage (block, conditions) as well as enforcing exceptions and test cases are shown as graphics and in tables:

- test case vs. block id

  For every function the affected blocks are listed and displayed as a structural graphic similar to the one in Ch. 6.4.6, but showing differential coverage.
- test case vs. exception

  Test cases of a function are listed for which generation was enforced by an exception.
- test case vs. block id

  the list of blocks is shown covered by the test case
- block id vs. test case

  For every block of a function the test case is shown which was the first one covering this block.
- true in condition vs. test case

  For every condition the test case is shown by which the condition evaluated to true the first time.
- test case vs. true in condition

  For every test case the condition affected for true in a function is shown.
- false in condition vs. test case

  For every condition the test case is shown by which the condition evaluated to false the first time.
- test case vs. false in condition

  For every test case the condition affected for false in a function is shown

6.4.14 Code Analysis

When the option is activated, graphics are generated for

- function hierarchies

  caller-callee and callee-caller
- type hierarchies

  fully and filtered according to a number of pre-defined criteria.

7. OPEN TOOL INTERFACE

DCRTT supports an open interface (DOIF, DCRTT Open tool InterFace) to other tools on the basis of the generated test drivers. It provides a set of files which shall allow other test tools to use the information on derived test cases or test case candidates (TC) and to execute them. Currently,

- the DOIF is implemented for Cantata and VectorCAST,
- only one tool may be attached during test execution.

The principal approach for the DOIF is shown in Fig. 7-1.

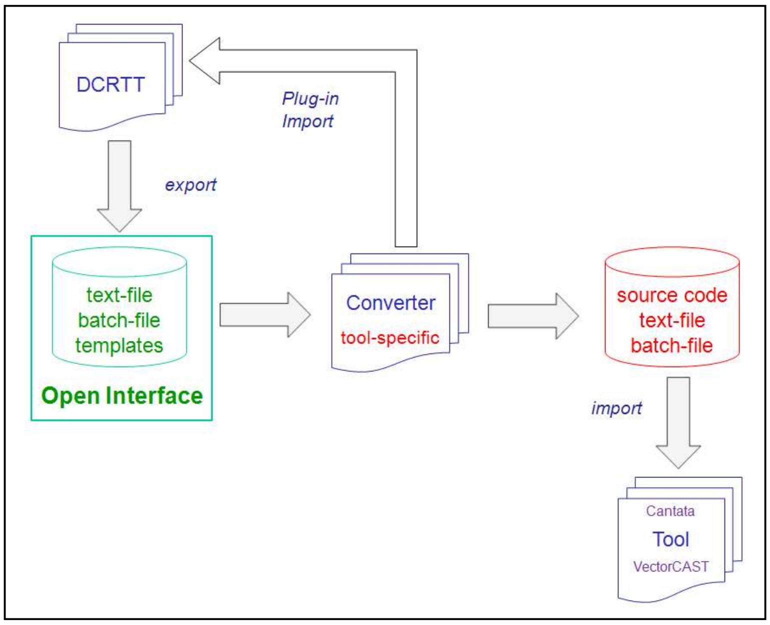

Fig. 7-1: Principal Approach for Exchange of Information via the Open Interface

DCRTT exports information as needed for re-execution of the test cases / regression testing via text-files, batch-files and templates and imports information required to establish the environment for the external tool and to start and control its execution.

This way a tool can be attached without any need to change DCRTT. However, for every tool to be attached such a converter must be established because the tools require rather different representations of test information.

The interface is established by a *tool-specific* ***converter.*** It consists of a set of executables, batch-files and support files. Its main purpose is to convert the provided information on the TC (input-output vectors) in a format suitable for the tool. However, it also provides batch-files needed for control of the link between DCRTT and the external tool.

Converters exist for Cantata and VectorCAST. Also, DCRTT needs a converter to make the information expressed in generic notation executable under C or C++.

The converter generates the inputs in a format understood by the external tool. Based on this information and controlled by the associated batch-file(s) the tool executes the TC in its own environment.

The converter provides plug-in information to DCRTT by batch files, which allow DCRTT to build the environment of the attached tool.

The execution of test drivers by the attached tool is controlled by DCRTT. There are two cases of execution:

1. The attached tool is immediately called from DCRTT after generation of the test drivers.
2. The attached tool is called from a batch-file generated by DCRTT during test execution, including all functions-under-test. A user may edit this file to select certain tests for re-execution.

   This batch-file only exists after completion of all tests when having activated the option for the attached tool during test execution according to case (1).

8. REQUIREMENTS-BASED TESTING

Currently requirements are only available as free text. Therefore, the transition from requirements to oracles cannot be automated.

The support provided by DCRTT is based on an open interface integration of oracles into the DCRTT test process. They are automatically inserted into the relevant functions by DCRTT by applying mapping rules. The concept allows to manually correlate (text-based) requirements with oracles and to auto-correlate oracles with functions. It supports propagation of the requirement status bottom-up based on provided dependencies between requirements.

Oracles, i.e., a formalized notation, need to be provided for requirements on lowest level only, while higher level requirements may remain in text form, if their respective contents are completely represented by associated formalized low-level requirements.

The implementation of the oracle concept allows to demonstrate the potential of automation (and related cost saving potential) which is possible once the low-level requirements are formalized.

8.1 THE CONCEPT

The intention of the concept is to automatically correlate requirements with functions to the degree possible based on oracles representing formalized requirements. Use of the concept does not imply that all requirements must be provided in a formalized form.

Oracles in DCRTT need to be provided only for requirements which directly address concepts testable on the software itself, specifically such concepts that can be expressed in terms of function inputs and outputs. Typically, such requirements occur on the lowest level of a hierarchy of requirements. An n:m relationship between requirements and functions as well as between oracles and functions may exist.

Given that machine-readable information for correlation of higher-level requirements with lower-level requirements is available, conclusions on the fulfilment of higher-level requirements can be automatically drawn from the results of testing of lower-level requirements.

The basic elements of the concept are (Fig. 8-1)

- requirements, represented by
  - a short identifier, which may be a reference and point to a requirement in a document or an arbitrary, but unique number or word, and
  - full text (optionally)

    The full text is the verbose representation of a requirement. It is the representation usually found in requirements documents today. For the automated approach this text is optional because the automated approach can only use the correlated oracle(s) as reference. For pure text requirements, the text is mandatory, given that for these types of requirements it is the only source of information. Note that this full text is in no way processed by DCRTT but is rather given for user information and reference.
- dependencies between
  - several requirements, establishing a hierarchy of requirements
  - a short name of a requirement and oracles

  Provision of dependencies between requirements are demanded usually by standards like ECSS between two consecutive requirement levels for requirements traceability. They may be provided by a requirements management tool – if applied – or manually. If a requirement directly addresses a function or a set of functions, then an oracle or a set of oracles can be derived from it and a dependency can be established between both – fully automatically.

- oracles consisting of
  - a pre-condition and
  - a post-condition

  both represented by valid C code, which is automatically integrated into the test environment of a FUT by DCRTT.
- functions to be tested

  with parameters, used global variables and (possibly) a return value, and the functions affected by requirements and oracles may be distributed over a number of files.

A requirement may depend on n other requirements.

The identifier of a requirement is correlated with an oracle or many oracles on next lower level by an 1:n dependency, i.e. one requirement may refer to n oracles, representing multiple, not necessarily mutually exclusive usage situations. Note that oracles may also be function-independent, e.g., if they represents invariants over the global state which always have to be fulfilled and thus do not depend on individual function invocations or associated function parameters or return values.

An oracle may thus apply to multiple functions, and multiple oracles may apply to one function, which results in an m:n relationship between oracles and functions, where the functions may be distributed across several files.

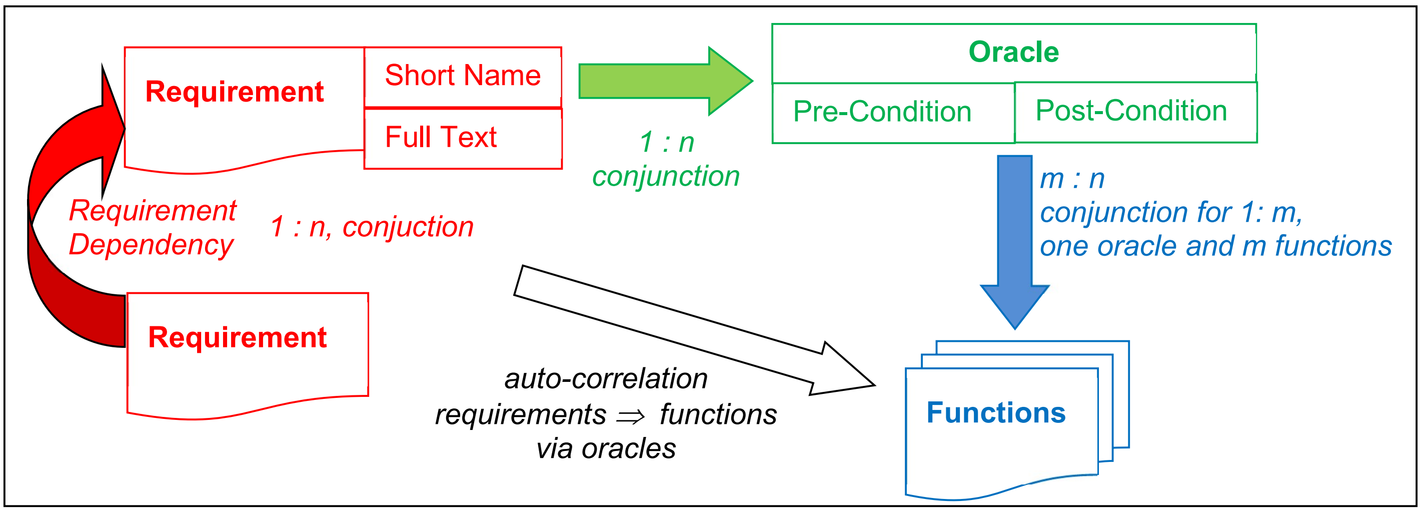

Fig. 8-1: Overview on the Concept

All relationships between the elements are considered as conjunctions, i.e., the parent element is considered as fulfilled only if all children fulfil the respective conditions.

Pre- and post-conditions must be valid C / C++ logical expressions evaluating to a boolean value.

The correlation between identifiers of requirements, oracles, functions and files is defined by the following syntax, called RQBT oracle syntax (all syntax definitions are given in EBNF-notation):

```
<short name> ';' <pre-condition> '?' <post-condition> ';' [ <function pattern> [ ',' <file
                                    pattern> ] ];
```

The question-mark notation can be seen as a reference to the conditional operator in C (a representation of if-then-else expressions) in that the expression on the right side of the question-mark is only considered when the left side evaluates to true.

The function and file patterns are optional. They may constrain the set of functions to be considered for matching of oracles. Wild cards (*, ?) are allowed, with “*” matching any sequence of characters, and “?” matching a single character. If a function or file pattern is missing, “*” is taken instead by default.

Correlation of an oracle with a function depends on data matching. A function or file pattern does not necessarily imply that the oracle will be applied to the addressed set of functions. They just constrain the possible choices out of the full set of functions of an application.

The data occurring in the pre- and post-conditions must be found as variables used in the code of the functions to be considered. Otherwise, compilation errors would be issued, as the pre- and post-conditions are embedded in the code context of the FUT.

In addition to the variables declared in the context of a function, a copy of an *original* may be referred to as

<*original*>Input

which takes the value of the *original* seen before the call of the FUT.

Any function visible in the context of pre- and post-conditions may be used in the code of pre- and post-conditions. For example, functions from the C library such as *sqrt* may be used in a post-condition. Similarly, any data and literal known in the context may be used. Globally visible symbols will avoid syntax errors.

The list of correlations between requirements, oracles, functions, and files may be complemented by additional information on

- constants
  which are used in oracles, but not known in context of a function, e.g., an ε used as accuracy limit for floating point comparisons,
- replacements,
  to achieve a mapping between identifiers used in oracles and their corresponding counterparts in the code. However, it is recommended to use the same identifiers in requirements and oracles for the code.

All this information may be provided in a set of files

Having executed all tests for the chosen set of FUT, the summary file on test results with raw data is evaluated, and contributions from different tests / FUT to the same oracle are harmonized and printed.

If an oracle failed for at least one FUT, it is considered as being failed in total, even if in another FUT its post-conditions was fulfilled. The results from the oracles are correlated with the respective requirements, and finally, the status of requirements is propagated bottom-up based on the provided dependencies on requirements.

The resulting information is provided in a number of files and reported from different perspectives, e.g. showing the full status (successful / failed) of all oracles related to a requirement (collected over all hierarchy levels), or limited to failed oracles and FUT only.

8.2 EXAMPLE ORACLES

Fig. 8-2 provides some examples for different types of oracles aiming to check whether

- a variable remains unchanged in the context of a function call.
- a function (e.g. *abs*) does really what it ought to do.
- a function always returns a valid result using an inverse function.
- the status of data is compliant with the requirement.

Oracle for unchanged check
```
p1==c? pInput==p
```

Oracle for int abs(int xAbs)
```
FORALL(xAbs)? ret>=0
```
Will fail! Why?

Oracle for inverse function of f(xSq)=xSq^2
```
FORALL(xSq)? (sqrt(ret)-fabs(xSq))<eps
```
Check on relative deviation needed! Wrong! Why?
Two checks needed due to division by xSq
Even then check will fail, when and why?

Oracle for Status Monitoring
```
status==active && mode==mode1 ? moniFlag==true
status==active && mode==mode2 ? moniFlag==true
status==active && mode==mode3 ? moniFlag==false
```

Fig. 8-2: Examples for Oracles

These are the explanations related to *Fig. 8-2*:

- abs-function

  There is an edge case for $x=-2^{31}-1$ for which the converted positive value cannot be represented in signed int with 32 bits. Therefore, the original value is returned violating the requirement on the abs-function.

- inverse function *sqrt* for x^2

  The square of a double cannot always be represented in the 64bit-representation, then yielding the result INF as a specific binary representation of double. Taking this value as argument of sqrt will not yield the original value x – apart of a relative deviation *epsilon*.

8.3 INTERFACES TO DCRTT

Fig. 8-3 shows the interfaces between the RQBT concept and the current version of DCRTT. Starting with the requirements – formalised in terms of oracles – the additional code is included in the test environment, which then generates test cases or finds counter examples instead of pure test vectors.

The counter examples may be found due to stimuli generation over the given input domain, only limited by the discrete set of stimuli, also this number may be huge, but may not hit the test vector of a counter example.

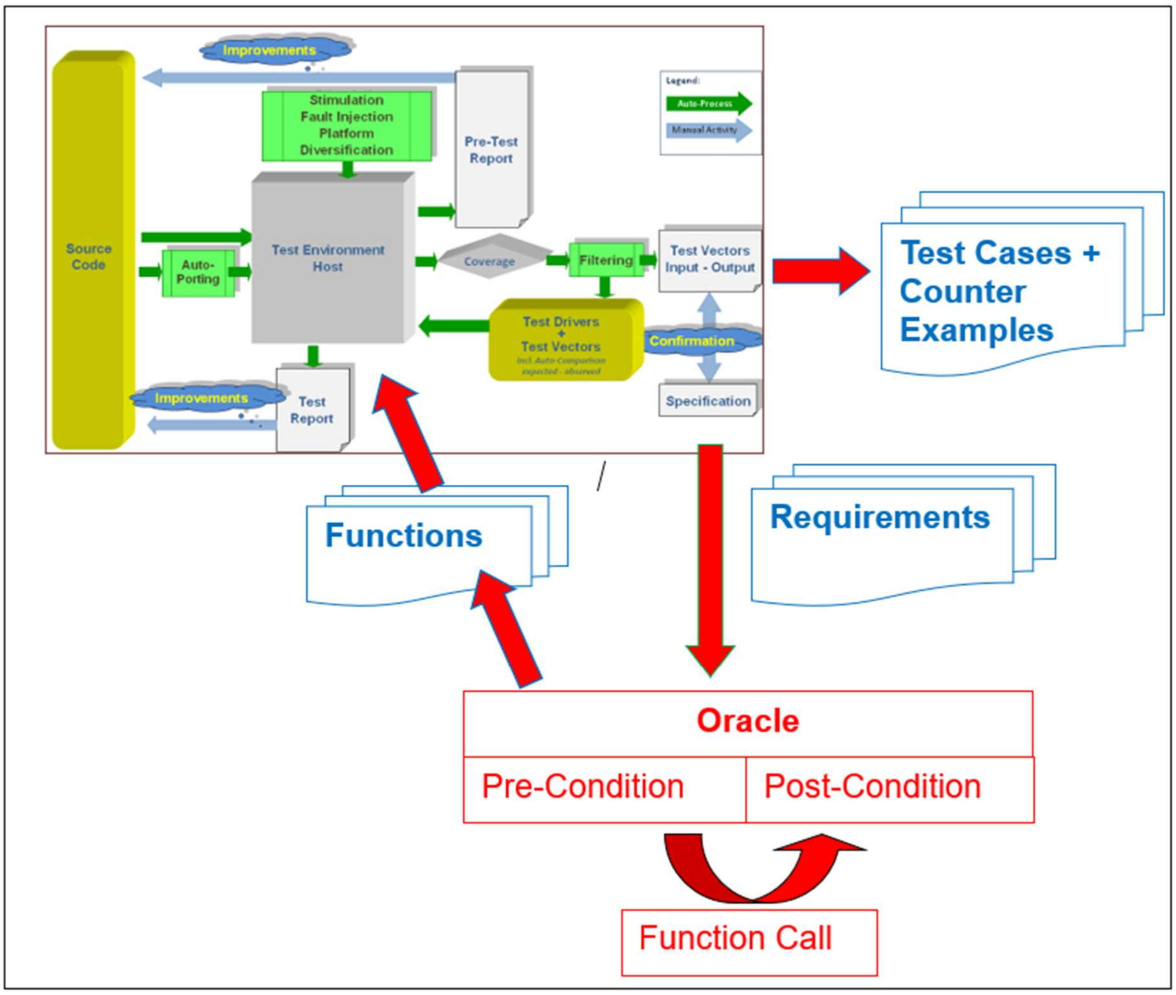

Fig. 8-3: RQBT Interfaces to DCRTT

9. EXPLORING AN INTEGRATED SOFTWARE

In case of an integrated software the full set of functions of the software-under-test is executed like under operational conditions, but the source code is instrumented as used for unit testing. Therefore, the same features as during unit testing can be monitored and evaluated.

As several top-level functions like task bodies may exist, DCRTT generates an environment, called “entry-point-function”, for calling all such functions according to several execution policies which may be

- the sequential call of a set of top-level functions in a sequence defined by the user,
- the execution of tasks based on a scheduling scheme defined by the user,
- a combination of both, where the sequential calls are executed at the end when the task execution has completed.

Task execution is non-pre-emptive, but scheduling could follow the intended scheme of the application. Periodic, synchronous, and sporadic tasks as well as major and minor cycles are supported based on information available in the source code.

If required, stimulation of the integrated system can be performed at two interfaces:

- the command interface like for telecommands, and
- the interface to external data.

For automated stimulation of these interfaces, specifications are required. Based on these specifications, generators can be established using DCRTT features which already exist for stimulation during unit testing.

For injection of telecommands (TC) a generator already exists which takes an XML specification and provides functions for stimulation with TC of random contents, either valid or invalid.

For stimulation of the data interface information of the data types is required from a user and – possibly – also on protocols.

10. USING WRAPPER FUNCTIONS

DCRTT supports generation of wrapper functions for application software. By a wrapper function an application function (original function) is embedded into an automatically generated function, with the intent to avoid fault propagation. DCRTT currently supports the generation of code for the following principal options:

- checking the return value for any scalar types and pointers
- catching exceptions
- managing reporting of errors identified by checks
- interrupting the control flow in case of an error to return to a function on a higher level of the call hierarchy.

A detailed list on the options is provided in Tab. 6 below.

In case of scalar types, the check is based on a comparison of the return value with a user-defined expression (valid in C or C++). In case of a pointer either a check on NULL is performed or on a valid address (regarding the memory area of the platform). All options of Tab. 6 apply to both kind of checks.

To provide expressions for the check of a return value following options are supported:

- a type-specific expression as defined by a user, to be applied to all relevant functions with the same return type,
- expressions manually defined for dedicated functions, or
- a default expression, e.g. “<0”, valid for all functions, except those for which specific expressions are defined.

The original function is renamed, and the wrapper function keeps the original name. This implies that all other DCRTT features like fault injection are performed in the context of the wrapper function, so that, e.g., an exception caused by an injected valid input, can be handled by the wrapper.

In addition,

- execution time of the original function can be measured,
- exceeding of a given deadline can be monitored.

Minimum, average and maximum execution time are reported.

If a deadline is exceeded, this event is handled and reported like an error for the return value.

A user may insert additional code into the wrapper at following locations (see options for bits 11, 12 and 13):

- before the call of the original function,
- after the call of the original function, and
- in the error handling part.

If an error occurred for the return value check, isErrRet is set to 1. If a deadline was exceeded isErrTime is set to 1. Both variables are set to 0 before a call.

Following options exist for generation of a wrapper body:

Option	
only call of associated function	
establish exception handler around a call	
check return code	
if return check fails	raise exception
	print message to stdout and stderr
	record message to file
	store return value on stack
if an exception occurred pass the exception upward to the next upper function	
call a user-provided C-function for error handling with a DCRTT-define standard interface.	
disable exception reporting per level if exceptions are passed upward	
enable monitoring of execution time	
enable deadline check	
insert user code before function call	
insert user code after function call	
insert user code in the error handling part	

Tab. 6: Options for Bodies of Wrapper Functions

Errors which are stored on a stack can be accessed by two DCRTT functions, which either return the next message on the stack or print all recorded errors which are on the stack.

11. PLATFORMS

11.1 LANGUAGES

DCRTT has been tested for source code written in C99, C11 and C++ (2011, support for 2020 is under development).

For C++, use of templates requires specific support for the templates used. A small subset of the STL is already supported. Development activities for the removal of this restriction are currently under way.

11.2 OPERATING SYSTEMS

DCRTT runs on Windows® XP, 7 and 10 as host platform.

For execution of the generated test drivers on target systems support is given for RTMES, VxWorks®, Linux and RODOS.

11.3 COMPILER

Currently, the gcc is supported from version 3 to 8 on host and target.

On request, Visual C++ may be supported.

For remote test execution on the target system the compiler and the target link can be configured.