Assisted Coverage Closure*

Adam Nellis[1], Pascal Kesseli[2], Philippa Ryan Conmy[1], Daniel Kroening[2], Peter Schrammel[2,4], Michael Tautschnig[3]

[1] Rapita Systems Ltd, UK
[2] University of Oxford, UK
[3] Queen Mary University of London, UK
[4] University of Sussex, UK

Abstract. Malfunction of safety-critical systems may cause damage to people and the environment. Software within those systems is rigorously designed and verified according to domain specific guidance, such as ISO26262 for automotive safety. This paper describes academic and industrial co-operation in tool development to support one of the most stringent of the requirements — achieving full code coverage in requirements-driven testing. We present a verification workflow supported by a tool that integrates the coverage measurement tool RapiCover with the test-vector generator FShell. The tool assists closing the coverage gap by providing the engineer with test vectors that help in debugging coverage-related code quality issues and creating new test cases, as well as justifying the presence of unreachable parts of the code in order to finally achieve full *effective* coverage according to the required criteria. We illustrate the tool's practical utility on automotive industry benchmarks. It generates 8× more MC/DC coverage than random search.

1 Introduction

Software within safety-critical systems must undergo strict design and verification procedures prior to their deployment. The ISO26262 standard [1] describes the safety life cycle for electrical, electronic and software components in the automotive domain. Different activities are required at different stages of the life cycle, helping ensure that system safety requirements are met by the implemented design. The rigor to which these are carried out depends on the severity of consequences of failure of the various components. Components with automotive safety integrity level (ASIL) D have the most stringent requirements, and ASIL A the least strict. One of the key required activities for software is to demonstrate the extent to which testing has exercised source code, also known as code coverage. This can be a challenging and expensive task [4], with much manual input required to achieve adequate coverage results.

This paper presents work undertaken within the **Ve**rification and **Te**sting to Support Functional **S**afety **S**tandards (VeTeSS) project, which develops new tools and processes to meet ISO26262. The paper contains three contributions:

* The research leading to these results has received funding from the ARTEMIS Joint Undertaking under grant agreement number 295311 "VeTeSS".

Type	Description	ASIL
Function	Each function in the code is exercised at least once	A, B (R); C, D (HR)
Statement	Each statement in the code is exercised at least once	A, B (HR); C, D (R)
Branch	Each branch in the code has been exercised for every outcome at least once.	A (R); B, C, D (HR)
MC/DC	Each possible condition must be shown to independently affect a decision's outcome.	A, B, C (R); D (HR)

Table 1. ISO26262 Coverage Requirements (HR = highly recommended, R = recommended)

1. We integrated the FShell tool [7] with an industrial code coverage tool (RapiCover) in order to generate extra test cases and increase code coverage results. This work represents an effort in the integration of formal-methods based tools with industrial testing software. In the safety-critical domain these two areas are generally separated from one another, with formal methodology used only for small and critical sections of software to prove correctness and viewed as an expensive procedure. The tool is at an evaluation stage of development, assessing future improvements to prepare its commercialisation.
2. We present a discussion as to how this technology is most appropriately used within the safety life cycle. Achieving 100% code coverage can be a complex and difficult task, so tools to assist the process are desirable, however there is a need to ensure that any additional automatically generated tests still address system safety requirements.
3. Finally, we apply the technology to three sizeable automotive benchmarks to demonstrate the utility and the limitations in practice.

Safety standards require different depths of coverage depending on the ASIL of the software. The requirements of ISO26262 are summarized in Tab. 1. The aim of requirements-based software testing is to ensure the different types of coverage are achieved to 100% for each of the categories required. In practice this can be extremely difficult, e.g. defensive coding can be hard to provide test vectors for. Another example is code that may be deactivated in particular modes of operation. Sometimes there is not an obvious cause for lack of coverage after manual review. In this situation, generating test vectors automatically can be beneficial to the user providing faster turnaround and improved coverage results.

This paper is laid out as follows. In Sec. 2 we provide background to the coverage problem being tackled, and criteria for success. In Sec. 3 we describe the specific tool integration. Sec. 4 describes an industrial automotive case study. Sec. 5 looks at both previous work and some of the lessons learnt from the implementation experience

2 Assisted Coverage Closure

Testing has to satisfy two objectives: it has to be effective, and it has to be cost-effective. Testing is effective if it can distinguish a correct product from one that is incorrect. Testing is cost-effective if it can achieve all it needs to do at

the lowest cost (which usually means the fewest tests, least amount of effort and shortest amount of time).

Safety standards like ISO26262 and DO-178B/C demand requirements-driven testing to increase confidence in correct behavior of the software implemented. Correct behavior means that the software implements the behavior specified in the requirements *and* that it does not implement any unspecified behaviors. As a quality metrics they demand the measurement of *coverage* according to certain criteria as listed in Tab. 1, for instance. The rationale behind using code coverage as a quality metrics for assessing the achieved requirements coverage of a test suite is the following: Suppose we have a test suite that presumably covers each case in the requirements specification, then, obviously, missing or erroneously implemented features may be observed by failing test cases, whereas the *lack of coverage*, e.g. according to the MC/DC criterion, indicates that there is behavior in the software which is not exercised by a test case. This may hint at the following software and test quality problems:

(A) Some cases in the requirements specification have been forgotten. These requirements have to be covered by additional test cases.
(B) Features have been implemented that are not needed. Unspecified features are not allowed in safety-critical software and have to be removed.
(C) The requirements specification is too vague or ambiguous to describe a feature completely. The specification must be disambiguated and refined.
(D) Parts of the code are unreachable. The reasons may be:
    (1) A programming error that has to be fixed.
    (2) Code generated from high-level models often contains unreachable code if the code generator is unable to eliminate infeasible conditionals.
    (3) It may actually be intended in case of defensive programming and error handling.

    In the latter case, fault injection testing is required to exercise these features [8]. Dependent on the policy regarding unreachable code, case (2) can be handled through justification of non-coverability, tuning the model or the code generator, or post-processing of generated code.

The difficulty for the software developer consists in distinguishing above cases. This is an extremely time consuming and, hence, expensive task that calls for tool assistance.

2.1 Coverage Closure Problem

Given
- an implementation under test (e.g. C code generated from a Simulink model),
- an initial test suite (crafted manually or generated by some other test suite generation techniques), and
- a coverage criterion (e.g. MC/DC),

we aim at increasing *effective* test coverage by automatically
- generating test vectors that help the developer debug the software in order to distinguish above reasons (A)–(D) for missing coverage;

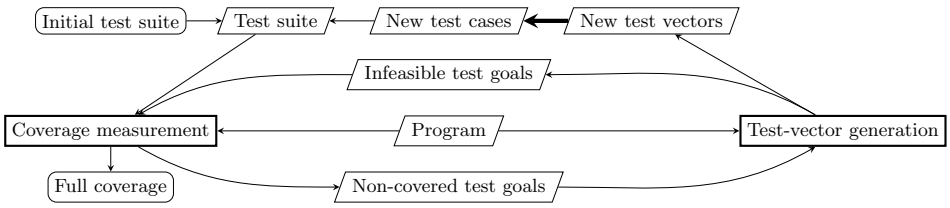

Fig. 1. The Coverage Closure Process

- in particular, suggesting additional test vectors that help the developer create test cases to complete requirements coverage in case (A);
- proving infeasibility of non-covered code, thus giving evidence for arguing non-coverability.

Note that safety standards like to DO-178C [11] allow only requirements-driven test-case generation and explicitly *forbid* to achieve full structural code coverage by blindly applying automated test-vector generation. This can easily lead to confusion if the distinction between test-*case* generation and test-*vector* generation is not clearly made. Test-*vector* generation can be applied blindly to achieve full coverage, but it is without use by itself. A test vector is only a *part* of a test case because it lacks the element that provides information about the correctness of the software, i.e. the expected test result. Only the requirements can tell the test engineer what the expected test result has to be. Test-*case* generation is thus *always* based on the requirements (or a formalized model thereof if available). Our objective is to provide assistance for test-case generation to bridge the coverage gap.

2.2 Coverage Measurement

Combining a test-case generator with a coverage tool provides immediate access to test vectors needed to obtain the level of coverage required for your qualification level. Coverage tools determine which parts of the code have been executed by using instrumentation. Instrumentation points are automatically inserted at specific points in the code. If an instrumentation point is executed, this is recorded in its execution data. After test completion, the coverage tool analyzes the execution data to determine which parts of the source code have been executed. The tool then computes the level of coverage achieved by the tests. We use the coverage tool RapiCover, which is part of the RVS tool suite developed by Rapita Systems Ltd.

2.3 Test Vector Generation by Bounded Model Checking

We use the test vector generator, FShell [7] (see Sec. 3.2 for details), which is based on the Software Bounded Model Checker for C programs, CBMC [3].

Viewing a program as a transition system with initial states described by the propositional formula *Init*, and the transition relation *Trans*, Bounded Model Checking (BMC) [2] can be used to check the existence of a path π of length k from *Init* to another set of states described by the formula ψ. This check is

performed by deciding satisfiability of the following formula using a SAT or SMT solver:

$$Init(s_0) \wedge \bigwedge_{0 \leq j < k} Trans(s_j, i_j, s_{j+1}) \wedge \psi(s_k) \tag{1}$$

If the solver returns the answer "satisfiable", it also provides a satisfying assignment to the variables $(s_0, i_0, s_1, i_1, \ldots, s_{k-1}, i_{k-1}, s_k)$. The satisfying assignment represents one possible path $\pi = \langle s_0, s_1, \ldots, s_k \rangle$ from *Init* to ψ and identifies the corresponding input sequence $\langle i_0, \ldots, i_{k-1} \rangle$.

Besides being useful for refuting safety properties (where ψ defines the error states), BMC can be used for generating test vectors (where ψ defines the test goal to be covered).

The analysis performed by CBMC is bit-exact w.r.t. the machine semantics of the execution target and CBMC provides full bit-exact support for floating point arithmetic. Architecture-specific settings can be configured via command line in FShell and RapiCover supports on-target coverage measurement. We are hence guaranteed that the generated test vectors are going to cover the test goals. In addition, using BMC in a test-vector generator permits generating the shortest test vectors possible to cover a certain test goal or even a whole group of test goals, which helps keeping test suites concise and test execution fast [12].

An advantage of using a model checker is also its ability to find test vectors for corner cases ("Under which conditions can this floating point variable take the value NaN?"). Moreover, in our experience, due to the high precision of the analysis, it is even very likely to discover inconsistencies and holes in the requirements specification during test-vector generation.

BMC can give a proof of unreachability of a test goal in certain conditions, e.g., if loops can be unrolled completely or using k-induction [13], which is a BMC-based technique for unbounded model checking.

2.4 The Coverage Closure Process

The algorithm that we implement to assist the coverage closure process is given in Fig. 1. It proceeds as follows:

1. We start with an *initial test suite* that has been crafted manually or has been generated using other test-case generation techniques like directed random testing. The initial test suite may be empty, but many test goals can be easily covered using test-case generation methods that are cheaper than Bounded Model Checking. It is thus recommended to start with such a base test suite.
2. In the next step, this *test suite* is run using the *coverage measurement* tool in order to obtain a list of *non-covered test goals*. Coverage measurement can be performed on a developer machine to obtain approximate coverage, but final certification data has to be obtained by running the test suite on the actual target platform.
3. The *test-vector generator* takes the list of non-covered test goals and tries to compute input values to cover them. Ideally, the test-vector generator is parametrized with the architectural parameters of the target platform in

order to obtain guarantees that the goals are indeed going to be covered. As our test-vector generator is a Bounded Model Checker, there will be three possible outcomes of an attempt to cover test goals:

(a) A test goal has been covered. In this case this *new test vector* is presented to the user who has to turn it into a *new test case* to be added to the *test suite*. Note that building the new test case is the only part of the process (bold edge) that is not fully automatic since human judgment is required to identify why the corresponding test goal has not been covered in the first place, i.e. distinguishing reasons (A)–(D) in Sec. 2.

(b) It is *infeasible to cover a test goal*. This happens when the test-vector generator comes up with a proof of unreachability of the test goal. As mentioned above, a Bounded Model Checker can provide such proofs if the loops have been unwound completely, for instance. In this case, the corresponding test goal can be annotated in the coverage report as *proven infeasible* to justify its non-coverability. This increases *effective* coverage by reducing the number of genuinely coverable test goals.

(c) The goal has not been covered and we were unable to prove infeasibility of the test goal. With a Bounded Model Checker this can happen if the chosen bound k has been too low. In this case the test goal will remain uncovered and it can be tried to cover it with a higher value for k in the next iteration of the process.

4. Coverage of the enhanced test suite is then measured again to identify test goals that remain uncovered, and the process is repeated. Generated tests typically will cover more test goals than intended. Measuring coverage between generating tests increases cost-effectiveness of the process by eliminating unnecessary test-case generations.
5. If there are no more non-covered test goals we have achieved *full coverage* and the process terminates.

Note that the process depicted in Fig. 1 is not specific to our tool but applies in general. In particular, it does not rely on the test-vector generator to guarantee that a generated test vector covers the test goal it has been generated for, because the coverage measurement tool will check all generated test cases anyway for increasing the coverage. However, the generation of useless test cases can be avoided by using a tool such as FShell that can provide such guarantees.

Then, in theory, termination of the process achieving full coverage can be guaranteed, because embedded software is finite state. In practice, however, this depends on the reachability diameter of the system [10] and the capacity of the test-vector generator to cope with the system's size and complexity.

3 FShell plugin for RVS Implementation

The input to the tool[5] is a C program with an initial test suite. The output of the tool is twofold. The first output is a set of generated test vectors that

[5] RVS is licensed software. An evaluation version can be requested from `http://www.rapitasystems.com`. The licensing policy disallows anonymous licenses. To

augment the initial test suite to increase its coverage. The second output is a coverage report detailing the level of coverage achieved by the initial test suite, and the extra coverage added by the generated test cases.

FShell has been integrated into RapiCover as context menu option, illustrated in Fig. 3. RapiCover can be used to select a single function, call, statement, decision or branch. The tool then uses FShell to generate a test vector for this element. Alternatively, the tool has a button to generate as much coverage as possible. When this option is chosen, the tool goes around the loop described in Fig. 1, using FShell to repeatedly generate test cases to increase the coverage as much as possible, verifying the obtained coverage with RapiCover.

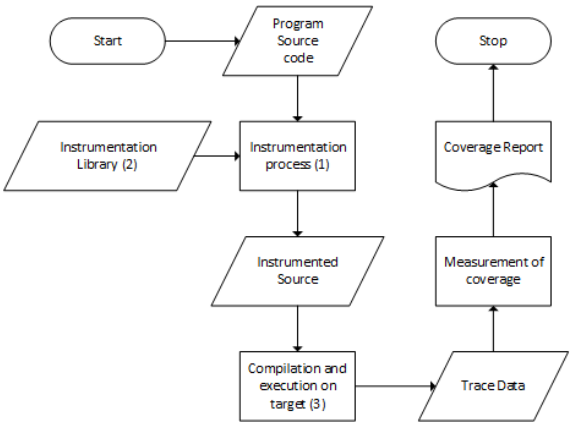

Fig. 2. RVS Process

There is tension between the need to demonstrate that the activities prescribed by ISO26262 have been met in spirit as well as with quantifiable criteria. Recall that achieving 100% code coverage during testing does not ensure the code meets its intent. Consequently the FShell plug-in would be provided as advisory service, generating candidate test vectors, which a user can examine to help them identify why their planned testing was inadequate. Values generated need to be assessed for being valid for the system under test, i.e. reflect real world values that could be input to a function, e.g. from a sensor.

3.1 Introduction to RapiCover

RapiCover[6] uses instrumentation to determine which program parts have been executed. Instrumentation points are automatically inserted at specific points in the code. Execution of an instrumentation point is recorded in its execution data. Upon test completion, RapiCover analyzes the execution data to determine which instrumentation points have been hit.

The first step in the RapiCover analysis process is to create an instrumented build of the application ((1) in Fig. 2). RapiCover automatically adds instrumentation points ((2) in Fig. 2) to the source code. The instrumentation code itself takes the form of very lightweight measurement code that is written for each target to ensure minimal impact on the performance of the software, and

compensate for this, we provide a video showing the plug-in here: `http://www.cprover.org/coverage-closure/rvs-fshell-demo.mp4`.

[6] `http://www.rapitasystems.com/products/rapicover`

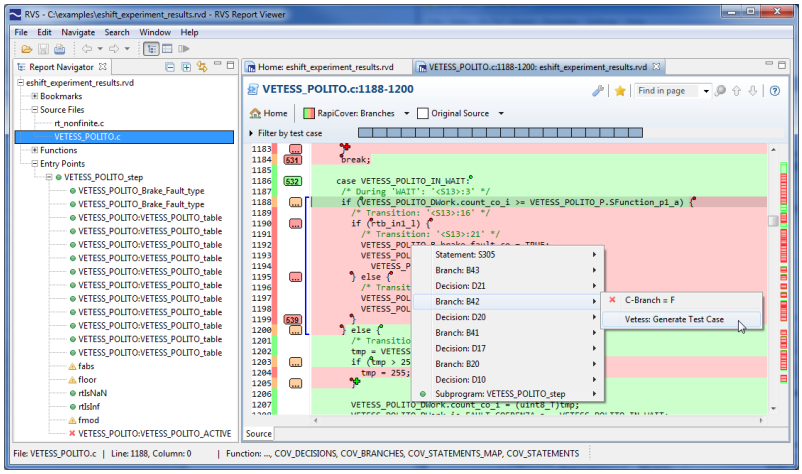

Fig. 3. Screenshot of RapiCover with the FShell Plug-in

to support on target testing for environments with limited resources. The instrumented software and possibly an instrumentation library are compiled and linked using the standard compiler tool chain. The executable produced is then downloaded onto the target hardware. The executable is exercised and instrumentation data ((3) in Fig. 2) is generated and retrieved. This data is used to generate coverage metrics.

3.2 Introduction to FShell

FShell[7] is an extended testing environment for C programs supporting a rich scripting language interface. FShell's interface is designed as a database engine, dispatching queries about the program to various program analysis tools. These queries are expressed in the FShell Query Language (FQL). Users formulate test specifications and coverage criteria, challenging FShell to produce test suites and input assignments covering the requested patterns. The program supports a rich and extensive interface. The expressions used for the FShell plugin for RVS implementation are listed in Tab. 2 with syntax and examples.

@CALL(X) requires generated test cases to call function *X*. This is the only primitive expression used in the module. The concatenation operator . joins two expressions, requiring them to be satisfied subsequently. As an example, a test case generated by *@CALL(X).@CALL(Y)* covers

Expression Name	Syntax	Example
Function Call	@CALL(...)	@CALL(X)
Concatenation	.	@CALL(X).@CALL(Y)
Sequence	->	@CALL(X)->@CALL(Y)
Negation	"NOT(...)"	"NOT(@CALL(X))"
Repetition	*	@CALL(X)*
Alternative	+	(@CALL(X) + @CALL(Y))

Table 2. FShell expressions

[7] Available from: `http://forsyte.at/software/fshell`

```
int main() {
    // ...
    if(a == b || b != c) {
        printf("%d %d\n", a, b);
    }
    return 0;
}
```

```
int main() {
    // ...
    Ipoint(1);
    if(Ipoint(4, Ipoint(2, a == b) ||
       Ipoint(3, b != c))) {
      Ipoint(5);
      printf("%d %d\n", a, b);
    }
    Ipoint(6);
    return 0;
}
```

Fig. 5. Code example before and after after RapiCover instrumentation

a call to *X* immediately followed by *Y*. This is similar to the sequence operator ->, which requires the second call to occur eventually. *@CALL(X)->@CALL(Y)* is thus fulfilled if a call to *X* is eventually followed by a call to *Y*. The negation *"NOT(@CALL(X))"* is satisfied by every statement except a call to function *X*. The repetition operator is implemented along the lines of its regular expression pendant, such that *@CALL(X)** is satisfied by a series of calls to *X*. Finally, the alternative operator implements logical disjunction, such that *(@CALL(X) + @CALL(Y))* will be satisfied if either a call to *X* or *Y* occurs.

The expressions and operators above are all that is used by the FShell plugin to generate the test vectors requested by RapiCover. Sec. 3.3 illustrates how these expressions are used to convert test goals to equivalent FQL queries.

3.3 Use of FShell within RapiCover

The FShell plugin for RVS translates test goals requested by RapiCover into FQL queries covering these goals in FShell, as illustrated in Fig. 4. Test goals are specified using marker elements from the RapiCover instrumentation, which can identify arbitrary statements in the source code by assigning them an *instrumentation point id*. In accordance with MC/DC criteria, decisions and their constituting conditions are further identified using unique *decision and condition point ids*.

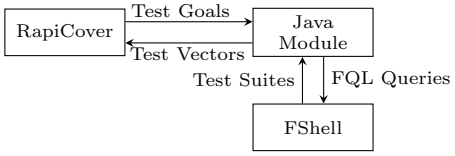

Fig. 4. Plugin Architecture

Fig. 5 shows an example program before and after instrumentation. The module supports two categories of test goals: *Instrumentation Point Path Test Goals* and *Condition Test Goals*. The former specifies a simple series of points to be covered by FShell. The system also permits *inclusive or* and *negation* operators in instrumentation point paths, allowing to specify a choice of instrumentation points to be covered or to make sure that a requested instrumentation point is not covered by the provided test vector. As an example, the instrumentation point path *1->5->6* in Fig. 5 is only covered if the decision in the *if* statement evaluates to *true*. Conversely, the path *1->NOT(5)->6* is only covered

Category	Goal	FQL
Instrumentation Point Path Goal	Simple	@CALL(Ipoint5) ->@CALL(Ipoint6) ->...
	Disjunction	(@CALL(Ipoint5) + @CALL(Ipoint6) + ...)
	Complement	@CALL(Ipoint1)."NOT(@CALL(Ipoint5))*".@CALL(Ipoint6)->...
Condition Goal	Condition	@CALL(Ipoint2f)."NOT(@CALL(Ipoint1))*". @CALL(Ipoint2t)."NOT(@CALL(Ipoint1))*".+...
	Decision	@CALL(Ipoint4t)

Table 3. Test Goal Types and FShell Queries

if it evaluates to *false*. The former can be achieved with inputs a=1, b=1, c=2, whereas the latter could be covered using the input vector a=1, b=2, c=2. *Condition Test Goals* on the other hand are specified by a single *decision point* and multiple *condition points*, as well as the desired truth value for each decision and condition. This allows us to cover branch conditions with precise values for its sub-conditions. As an example, the condition test goal *(4,true) -> (2,false) -> (3,true)* would be covered by the input vector a=1, b=2, c=3.

The instrumentation elements introduced by RapiCover need to be mapped to an equivalent FQL query using the features presented in Tab. 2. For this purpose, we replace their default implementation in RapiCover by synthesized substitutions which are optimized for efficient tracking by FShell. These mock implementations are synthesized for each query and injected into the program on-the-fly at analysis time. Standard FQL queries are then enough to examine these augmented models for the specified coverage goals. Tab. 3 shows explicitly how these goals can described using the FShell query syntax.

4 Evaluation

The FShell plugin for RVS has been tested on three industrial automotive use cases: an airbag control unit ("airbag"), a park control unit ("eshift"), a break-by-wire controller ("vtec") and a smaller message handler benchmark ("msg").[8]

4.1 Case Study: e-Shift Park Control Unit

To illustrate the characteristics of these benchmarks we describe the e-Shift Park Control Unit.[9] This system is in charge of the management of the mechanical park lock that blocks or unblocks the transmission to avoid unwanted movement of the vehicle when stopped. The park mode is enabled either by command of the driver via the gear lever (PRND: park/rear/neutral/drive) or automatically.

[8] The code for these benchmarks was provided by the respective companies under a GPL-like license and can be downloaded here: `http://www.cprover.org/coverage-closure/nfm-package.zip`

[9] Provided by Centro Ricerche Fiat.

Fig. 6 shows the architectural elements the e-Park system is communicating with. The vehicle control unit monitors the status of the vehicle via sensors and informs the driver, in particular, about the speed of the vehicle and the status of the gears via the dashboard. The e-Park Control Unit is responsible for taking control decisions when to actuate the mechanical park lock system.

Among many others, the following requirements have to be fulfilled:

1. Parking mode is engaged if vehicle speed is below 6 km/h and the driver presses parking button (P) and brake pedal.
2. If vehicle speed is above 6 km/h and the driver presses the parking button (P) and brake pedal then commands from the accelerator pedal are ignored; parking mode is activated as soon as speed decreases below 6 km/h.
3. If vehicle speed is below 6 km/h and the driver presses the driving button (D) and brake pedal, then forward driving mode is enabled.
4. If vehicle speed is above 6 km/h then backward driving mode (R) is inhibited.

As is typical for embedded software, the e-Park Control Unit software consists of tasks that — after initialization of the system on start-up — execute periodically in the control loop until system shutdown. A test vector hence consists of a sequence of input values (sensor values and messages received via the communication system) that may change in each control loop iteration. We call the number of iterations the *length* of the test vector.

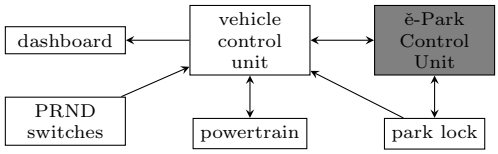

Fig. 6. e-Shift Park Control Unit

To generate valid test vectors, a model of the vehicle is required. Otherwise, the test vector generator may produce results that are known not to occur in the running system, such as infinite vehicle velocity. For the case study this model consisted of assumptions about the input value ranges, such as "The speed of the car will not exceed 1000 km/h, or reduce below 0 km/h." These assumptions are part of the admissible operating conditions as stated in the requirements specification.

4.2 Experimental Setup

In order to evaluate the FShell plugin for RVS, we used four different industrial C source code case studies with a cumulative ∼6700 LOC. We started out with an initial test suite consisting of 100 random test vectors of length 5 uniformly distributed over the admissible input ranges[10]. Then we incrementally extended this test suite by test vectors generated by the following two approaches:

[10] We chose length 5 because it seemed a good compromise between increasing coverage and keeping test execution times short for these case studies: on the e-Shift case study, adding 100 test vectors of length 5 increased coverage by 1.1%; 100 test vectors of length 10 increased it by only 1.3% while test execution times would double and only half as many test vectors could be explored.

1. FShell plugin for RVS following the process illustrated in Fig. 1.
2. A combination of test vector generation based on random search and greedy test suite reduction.

We compared the achieved coverage gain and resulting test suite sizes after running both approaches for 8 days, with the exception of the message handler, which we only ran for 3 hours due to its smaller code size. [11] Tab. 4 describes our experimental setup.

The runtime of FShell is worst-case exponential in the loop bound of this main loop. Choosing a too high loop bound results in FShell taking prohibitively long to run, yet setting the loop bound too low results in some branches not being coverable. As mitigation, we started the experiment with a loop bound of 1, then we gradually increased the loop bound to cover those branches that we were not able cover in previous iterations. As explained in Section 2.1, step 6 in Tab. 4 is not automatic since it needs information from the requirements specification. For the sake of our comparison that does not care about the pass/fail status of the test, we skipped the manual addition of the expected test outcome.

4.3 Results

The results of our experiment are detailed in Tab. 5. They indicate that more than 99.99% of the generated test vectors added by the random search are redundant and do not increase coverage. This confirms that these case studies

[11] The msg benchmark achieved 100 loop unwindings in 3 hours, compared to 37, 6 and 58 unwindings for airbag, eshift and vtec in 8 days.

	FShell plugin for RVS	random search + reduction
1.	Start with the initial test suite.	
2.	Compile and run the C source code with the current test suite, using RapiCover to generate a coverage report.	
3.	RapiCover provides FShell with a list of non-covered test goals.	
4.	FShell generates a test vector for these non-covered test goals.	Generate a random test vector, uniformly distributed over the admissible input ranges.
5.	FShell feeds back information about infeasible test goals and test vectors for feasible test goals.	
6.	Automatically create C test cases based on these test vectors.	
7.	Re-compile and re-run the C code with this new test case, using RapiCover to verify that the generated test case does indeed cover the test goal.	
8.		If the coverage has increased then keep the test case; otherwise discard it.
9.	Repeat from step 3.	

Table 4. Experimental setup of the two approaches that we compare.

	airbag			eshift			vtec			msg		
Test Cases	Init	Rnd	FS	Init	Rnd	FS	Init	Rnd	FS	Init	Rnd	FS
Generated	100	35k	6	100	35k	6	100	16k	4	-	9k	1
New	-	0	6	-	13	6	-	2	4	-	0	1
Coverage (%)	Init	Rnd	**FS**	Init	Rnd	**FS**	Init	Rnd	**FS**	Init	Rnd	**FS**
Statement	41.6	41.6	**83.8**	52.2	53.0	**53.2**	76.3	77.3	**79.3**	87.9	87.9	**89.6**
Increase	-	0.0	**42.2**	-	0.8	**1.0**	-	1.0	**3.0**	-	0.0	**1.7**
MC/DC	16.0	16.0	**68.0**	31.2	34.5	**36.8**	40.0	48.0	**64.0**	53.8	53.8	**61.5**
Increase	-	0.0	**52.0**	-	3.3	**5.6**	-	8.0	**24.0**	-	0.0	**7.7**

Table 5. Evaluation results: Comparing FShell plugin for RVS against test vectors generated by random search.

represent particularly challenging cases for black-box test vector generation and that only very few test vectors in the input range lead to actual coverage increase.

The FShell plugin for RVS outperforms the random search strategy in all tested benchmarks. The difference between the two approaches becomes more pronounced for more complex benchmarks, which is expected. As an example, the random search is unable to generate any coverage for the complicated, multi-threaded airbag example, whereas the FShell plugin for RVS more than triples the initial coverage. On average our approach increased MC/DC coverage by 22.3% and statement coverage by 11.9%. By comparison, the random search only achieved an average 2.8% and 0.5% increase. The average test vector length generated by FShell plugin for RVS is 7.4.

This evaluation thus underlines the benefit from our tool integration to support the coverage closure process on industrial case studies. The expected reduction in manual work needs to be investigated in a broader industrial evaluation involving verification engineers performing the entire coverage closure process.

5 Background and Applicability

There is much work existing for test case generation using Model Checking techniques [5], but a smaller amount targeted directly at the high criticality safety domain where the criterion and frameworks for test case generation are restricted. A useful survey relating to MC/DC can be found in [15]. In [6] Ghani and Clark present a search-based approach to test generation for Java—a language which is rarely used for safety-critical software, and particularly not for the most critical software. Their goal is to generate tests to ensure that the minimal set of truth tables for MC/DC were exercised, but without consideration of the validity of any of the test data by on-target coverage measurement. Additionally, we emphasize that our approach takes into account existing coverage that has already been achieved and complements the requirements based testing, rather than completely replacing it. Other work such as [9] looks at modification of the original source through mutation testing in order to assess effectiveness of the tests. This could be considered a useful adjunct to our methodology.

Lessons Learnt. In order to encourage wider adoption of this integrated tool, we need to consider where it would fit in users' workflow and verification processes, as well as meeting the practical requirements of the standard. As noted earlier, fully automated code coverage testing is not desirable as it misses the intent of the requirements based testing process. However, achieving full code coverage often requires a large amount of manual inspection of coverage results to examine what was missing. Hence providing the user with suggested test data is potentially very valuable and could improve productivity in one of the most time consuming and expensive parts of the safety certification process.

Another benefit of integrating test case generation and coverage measurement is test suite reduction. The coverage measurement tool returns for each test case a list of covered goals. Test suite reduction is hence the computation of a minimal set cover (an *NP*-complete problem). Approximate algorithms [14] may be used to achieve this in reasonable runtimes.

FShell uses a class of semantically exact, but computationally expensive, *NP*-complete algorithms relying on SAT solvers. Depending on the programs or problems posed to the solver the analysis may take long time to complete. Initial feedback on the tool showed that the concept was very well received by automotive engineers. Speed was considered an issue, however, keeping in mind that today's practice for full coverage testing may take several person months with an estimated cost of $100 per LOC,[12] there is great potential for cutting down time and cost spent in verification by running an automated tool in the background for a couple of days.

Initially, we sometimes failed to validate that a test vector that was generated to cover a test goal actually covers that test goal. E.g., one reason were imprecise decimal number representations in the test vector output. Using the exact hexadecimal representation for floating point constants fixed the problem. This highlights the value of bit-exact analysis as well as the importance of re-validating coverage using RapiCover in the process (Fig. 1).

Note also that this process itself is independent of the tools used which offers a high degree of flexibility. On the one hand, it is planned that in future RVS will support alternative backends in place of FShell. On the other hand, FShell can be combined – without changing the picture in Fig. 1 – with a mutation testing tool (in place of RapiCover) to generate test vectors to improve mutation coverage.

6 Conclusion

This paper has demonstrated the successful integration of the FShell tool with an industrial code coverage tool. Using the integrated tools we were able to increase MC/DC code coverage of four industrial automotive case studies by 22.3% on average. When compared to a random black-box test vector generation strategy, our approach was on average able to generate 796% more MC/DC coverage within the same amount of time. Our tool achieves this coverage gain with

[12] Atego. "ARINC 653 & Virtualization Solutions Architectures and Partitioning", Safety-Critical Tools Seminar, April 2012.

half as many test vectors, and these test vectors are much shorter than those generated by random search, leading to more compact test suites and faster test execution cycles. Moreover, the integration of the two tools simplifies test case generation and coverage measurement work flows into a unified process.

Future work will consider better integration with the debugging environment to inspect test vectors, and warning the user about potentially unrealistic environment assumptions such as ∞ for vehicle speed. In addition, better support should be provided for exporting the test vectors into the users' existing test suite and testing framework. Moreover, we would like to compare with other tools and further evaluate the coverage benefit from the exact floating point reasoning that we use in comparison to, e.g., rational approximations.

References

1. ISO26262 road vehicles – functional safety, Part 6: Product development at the software level, Annex B: Model-based development (2011)
2. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: TACAS. pp. 193–207 (1999)
3. Clarke, E.M., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: TACAS. pp. 168–176 (2004)
4. Dupuy, A., Leveson, N.: An empirical evaluation of the MC/DC coverage criterion on the HETE-2 satellite software. In: Digital Avionics Systems Conference. vol. 1, pp. 1B6/1–1B6/7 (2000)
5. Fraser, G., Wotawa, F., Ammann, P.: Testing with model checkers: a survey. Software Testing, Verification & Reliability 19(3), 215–261 (2009)
6. Ghani, K., Clark, J.A.: Automatic test data generation for multiple condition and MCDC coverage. In: ICSEA. pp. 152–157 (2009)
7. Holzer, A., Schallhart, C., Tautschnig, M., Veith, H.: FShell: Systematic test case generation for dynamic analysis and measurement. In: CAV. pp. 209–213 (2008)
8. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. Transactions on Software Engineering 37(5), 649–678 (2011)
9. Kandl, S., Kirner, R.: Error detection rate of MC/DC for a case study from the automotive domain. In: Software Technologies for Embedded and Ubiquitous Systems. pp. 131–142 (2010)
10. Kroening, D., Strichman, O.: Efficient computation of recurrence diameters. In: VMCAI. pp. 298–309 (2003)
11. Rierson, L.: Developing Safety-Critical Software: A Practical Guide for Aviation Software and DO-178C Compliance, Chapter 14.3 Potential Risks of Model-Based Development and Verification. CRC Press (2013)
12. Schrammel, P., Melham, T., Kroening, D.: Chaining test cases for reactive system testing. In: ICTSS. pp. 133–148 (2013)
13. Sheeran, M., Singh, S., Stålmarck, G.: Checking safety properties using induction and a SAT-solver. In: FMCAD. pp. 108–125 (2000)
14. Tallam, S., Gupta, N.: A concept analysis inspired greedy algorithm for test suite minimization. In: PASTE. pp. 35–42 (2005)
15. Zamli, K.Z., Al-Sewari, A.A., Hassin, M.H.M.: On test case generation satisfying the MC/DC criterion. International Journal of Advances in Soft Computing & Its Applications 5(3) (2013)