Testing Diagnostics Components Supervising Functional Safety Requirements

Mihai Nica[1], Ingo Pill[2], and Franz Wotawa[3]
authors are listed in alphabetical order

[1] *AVL GmbH, 8020 Graz, Austria*
mihai.nica@avl.com

[2,3] *Institute for Software Technology, TU Graz, Inffeldgasse 16b, 8010 Graz, Austria*
{ipill,wotawa}@ist.tugraz.at

ABSTRACT

For safety-critical applications, safety diagnostics components are an attractive safeguard for meeting some specified safety requirements under operation. Like a monitor, such a software artifact shall supervise a system under operation, and furthermore, if needed, it overrides the system's control software in order to maintain safety. In this paper we contribute to testing such a component, suggesting an approach that draws on fault injection and, in order to enhance deployability, accommodates also needs in respect of business issues like intellectual property disclosure and ressource efficiency. The required testing oracle we directly obtain from the defined and formalized functional safety requirements, for the purpose of assessing that the safety diagnostic component indeed maintains safety also under faulty conditions.

1. INTRODUCTION

Scientific evolution has been allowing us to continuously step ahead and develop solutions tackling problems that priorly seemed intractable. Consequently, the technology to assess and manage risks involved with our designs and their possibly faulty behavior has been facing constantly rising demands. For instance, in respect of effectiveness in highly dynamic environments, efficiency at handling complex designs, and robustness in order to name just a few challenges. For many a project, we seek to pro-actively address related safety concerns, as is demanded by public regulations via standards like IEC 61508[1] and its adaptation ISO 26262[2] (Automotive Safety Integrity Level (ASIL) as used in the automotive industry) on one side, and customers on the other one. For instance, in electronic design automation industry, logics like the Linear Temporal Logic LTL (Pnueli, 1977) or the Property Specification Language PSL (Eisner & Fisman, 2006) have been used to describe desired system requirements (properties) to be used for automated verification like model-checking (Clarke, Grumberg, & Peled, 1999). Recent work assists designers in formalizing these requirements (Bloem, Cavada, Pill, Roveri, & Tchaltsev, 2007). Also an AI-based diagnosis approach for diagnosing faults in such formalized requirements has been proposed (Pill & Quaritsch, 2013).

For the design and operation of nuclear power plants, space applications, or avionics, the importance of designing and satisfying safety requirements is obvious to each and everyone of us. However, also for more mundane systems like private cars, such concerns and corresponding requirements are, and have been, of utmost importance in their design and operation. Imagine, for instance, an automated parking brake. Certainly we would like it to be released only on a driver's command (manual release, tipping the gas pedal, ...). The assistance systems available in today's car would take many variables into account for deciding about this (e.g. driver present, doors closed, seatbelts fastened, ...), which makes it important to clearly specify safety targets like the one above that have to be met under all circumstancess. Such specific requirements to be verified during the design stages and monitored under operation shall help that safety (or related functionality aspects) are ensured, no matter the system's complexity or unexpected issues in ”live” real-world situations.

Diagnostic reasoning (de Kleer & Williams, 1987; Reiter, 1987) as explored by the Artificial Intelligence community (i.e. the DX community) is a powerful asset for tackling related issues (Weber & Wotawa, 2010). That is, finding a fault and identifying its root cause(s) certainly is an important and essential step towards keeping a system operating as optimal

[1]e.g., http://www.iec.ch/functionalsafety/
[2]e.g., http://www.iso.org/iso/catalogue_detail?csnumber=43464

as possible. Complementing the theoretical and application-specific research on solutions to the diagnosis problem itself, in order to enhance deployability of corresponding solutions, we also have to address the question of how to accommodate diagnostic reasoning components in the system design process.

For our current work, we consider this very question and focus on the specific issue of testing a diagnostic component's effectiveness. That is, we assume the scenario of a safety diagnostic component *SDC* supervising some control software *CS* (like a controller for the automated parking brake of above) and that can even override it in case of the software violating specified functional safety requirements. The task we face is testing that, indeed, *SDC* maintains safety as defined by the requirements, even if the control software would show some potentially hazardous behavior.

The testing concept that we propose easily integrates into an overall system's development process. Aiming to minimize needed resources, we reuse the input parts of the test cases available for the supervised component *CS*, since they were already designed to exercise the system thoroughly (also covering a wide selection of interesting scenarios). Since our test purpose is to assess whether *SDC* indeed lives up to our expectations of maintaining compliance to some safety requirements *REQ*, we use the concept of mutation testing in order to inject faults and simulate control software faults to be considered and dealt with by the diagnostic engine. The product of the reused input sequences and designed fault scenarios (mutants) shall be used to evaluate *SDC*'s capabilities. In this context, we judge safety conformance with a test oracle directly derived from the safety requirements *REQ* that are specified (or need to be translated) in a concise logic in order to support automated reasoning.

Before discussing our concept in Section 4 and related work in Section 3, we introduce a motivating example in Section 2 where we elaborate also on our problem description. A discussion of our approach, as well as corresponding conclusions and directions for future work are provided in Section 5.

2. A MOTIVATING SCENARIO AND TASK ANALYSIS

The basic task that the considered diagnostic component *SDC* has to tackle is to ensure that some defined safety requirements are met even if the control software *CS* would show some hazardous behavior. The latter could result either from some fault(s) in the software design like forgotten corner cases, faults in the implementation, some memory corruption (e.g. due to (Kim et al., 2014)) or other hardware faults, compiler errors, and many other issues. Certainly software designers add assertions and other sanity checks to their control software, possibly complemented with some elaborate code that aims at keeping the control software in a safe state (e.g. fault tolerant control concepts).

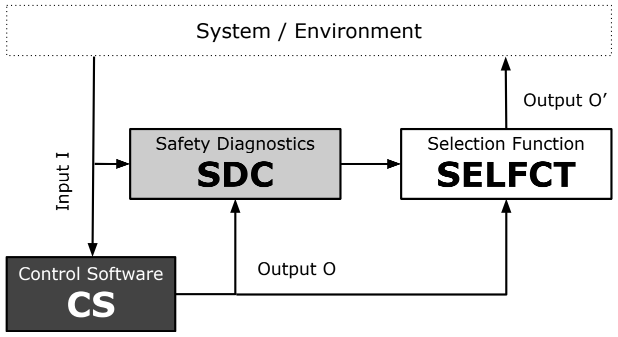

Figure 1. Integrating safety diagnostics

Now let us assume that we do have also some "external" *SDC* that sort of acts as last defense. As is illustrated in Figure 1, it supervises the control software *CS* in that it collects all of *CS*'s relevant inputs I, its outputs O, and then decides whether the I/O scenario complies with defined safety goals *REQ*, using diagnostic reasoning to investigate encountered issues. If necessary, it can override *CS* by altering the output from O to O' in order to maintain safety. The latter (depicted by an abstract selection function *SELFCT* in Fig. 1), obviously, can be implemented in many ways. One option would be a dedicated component on the main bus, another one the use of individual priority signal lines to the final actuators controlled by *CS* (like a gear box or a clutch). In practice, the choice of implementation might also be made individually for each isolated signal or component controlled by *CS*.

Definition 1 *A* Safety Diagnostics Component (SDC) *(see also Figure 1) observes the inputs I and outputs O of the supervised system CS, monitors for given requirements REQ the property of $I \cup O \cup$ REQ being consistent (satisfiable), and for a violation designs a new output O' delivered via a given component SELFCT such that $I \cup O' \cup$ REQ is consistent (satisfiable)*

Let us discuss this concept in the context of a simple WUMPUS-like scenario. The main character in this sce-

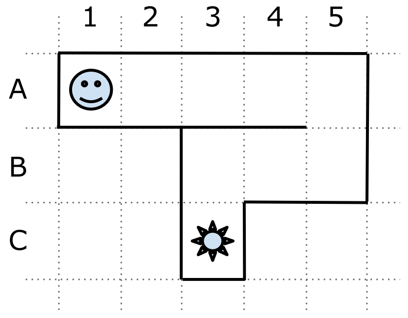

Figure 2. Move from $A1$ to $C3$ in a small world with walls

nario is a small robot that explores a grid-based world with walls. It may rotate 90-degree-wise in both directions, shift gears to forward, backward, or neutral, can move step-wise by one grid-element (see Figure 2 for an example world), and the drive unit can be shut off. The corresponding commands at control software level would be *RL* / *RR* to rotate left or right, *SGF* / *SGB* / *SGN* to shift the gearbox, M to move, and *CP* for cutting the drive unit's power. Let us assume that the robot can sense its position, and that the current task is to move from $A1$ to $C3$ in the world depicted in Figure 2. Assuming the robot is oriented towards the east, a possible command sequence to achieve this is *SGF, M, M, M, M, SGF, RR, SGF, M, SGN, RR, SGF, M, M, SGN, RL, SGF, M (,CP)*. Now let us assume that for some reason, like a bit flip or memory corruption, for the first command *SGF*, the gearbox changes to backward instead. Depending on where the exact fault occurred, besides assertions that check position, orientation, and other available internal knowledge, the robot might not be aware of the fact that when set in motion it will not proceed to $A2$, but hit the western wall in $A1$. Thus, when the robot executes the next activity *M*, without some interference it would actually hit a wall.

However, a safety requirement any designer would definitely derive for this application scenario is that the robot should never hit a wall. The safety diagnostics component implementing *REQ* then most likely would monitor *continuously* the robot's proximity to obstacles, and, e.g., in two violation levels could first shift the gearbox to neutral and if this does not suffice (gearbox malfunction) could even cut the power from the drive. An elaborate reasoning engine might try to determine (diagnose) the detailed reason for the violation (e.g. a second *moving* robot vs. a wall) in order to derive the best course (like triggering an emergency protocol in *CS* for fleeing from another swarm robot that seems out of control) and degrade task performance in a sensible way.

Even the simple WUMPUS-like scenario shows why having an external component supervising the main software could be of advantage. For instance,

- it can consider information at a different level of abstraction or at different frequencies.
- while the main software's complexity could ask for state-of-the-art components, the diagnostic engine could run on special, e.g. radiation-hardened, hardware that has been showing reliability in the designated environment in the past (concerning special demands in respect of heat, humidity, radiation, electric/magnetic fields,...).
- for *SDC′s* development, conceptional details from the main software (data handling, operation frequencies, real-time issues) represent no restrictions, so that one can concentrate on a requirements-based development (focusing on *REQ*) vs. a system-design oriented adaption of relevant functionality in *CS*.
- flexibility is offered also in the direction of collecting I and O. That is, we could possibly grab signal values from the software itself, or also at some location in the hardware lines so that a malfunction in the lines would also be covered.

Obviously, an *SDC* offers many advantages, best combined with afore-mentioned functionalities in the control software itself. For actual deployment, confidence in *SDC*'s capabilities is however of utmost importance. The specific challenge we consider in this paper is related to this very issue, in that we propose a testing concept for the purpose of assessing whether *SDC* actually lives up to the task of avoiding violations of specified safety requirements *REQ*, even if the control software would malfunction. More formally, we aim to test whether *SDC* conforms to *REQ* as of Def. 2.

Definition 2 *Let SDC be a safety diagnostics component as of Definition 1 which supervises some system CS. Then for some given requirements REQ and possible fault modes FAULTS of CS, SDC conforms to REQ if and only if for all inputs I, we have that REQ* $\cup I \cup O'$ *is consistent (is satisfiable).*

Obviously, a formal proof of such a conformance would be optimal, but is not a likely scenario since (a) the system is most likely too complex to apply techniques like model-checking (Clarke et al., 1999), and (b) we cannot assume the system to be a white box for us. Furthermore, considering a model only, would not support us in finally checking the very implementation "live" on actual hardware and in the actual environment (relevant e.g. in respect of radiation or other straining effects that could possibly affect timings, memory contents, and other electronics). Thus, we propose to implement a *testing* concept for evaluating the desired conformance. This concept then can be implemented at various abstraction levels (from specification level to the final product).

Aside academic questions like those regarding test design and coverage, for deployment in industry, e.g., for an automotive application, we have to consider also issues and restrictions originating from business issues to be faced in the development process. For instance, the availability of detailed system information (white vs. grey vs. black box) certainly is of an issue for such an automotive scenario. Thus it is also unlikely that we can inject faults for purposefully exercising *SDC* at will (see Sec. 3 for a discussion of mutation testing) or have access to all the internal models for imagining and deriving test cases. However, we can assume that a function enabling specific faults in the software can be made available, since this suits also testing the individual components themselves. Also the safety requirements to be enforced should be available (although not necessarily in a formal syntax), since they are essential for the design of the system.

To this end, in Section 4 we outline an approach for testing *SDC* in the exemplary context of an automotive application, which takes also business-related issues into account and addresses the conformance problem as of Definition 2. The underlying concept is to execute the system under test (SUT) using the test inputs originally designed for testing *CS* (or the overall system). We inject faults $f \in FAULTS$ into *CS* using provided fault injection functions, and then record the resulting outputs O'. A test oracle derived from the given (safety) requirements *REQ* then classifies the input and recorded output in respect of conformance to *REQ* as of Definition 2.

Please note that our testing concept is independent from the concepts used for developing and implementing the *SDC*. Depending on the actual system and requirements (and their structure), one might be able to implement individual monitors for the requirements that then trigger some signal that per construction results in a safe system state. More sophisticated solutions that perform some abstract model-based diagnosis (de Kleer & Williams, 1987; Reiter, 1987) reasoning and/or try to alter the system to degrade in a graceful way (maintaining the best possible functional performance) (Weber & Wotawa, 2010) can also be implemented, which we account for in using the term *SDC*.

3. RELATED RESEARCH

In automated software testing, we can distinguish between two different kinds of methods: active testing and passive testing. Active testing (see (Broy, Jonsson, Katoen, Leucker, & Pretschner, 2005)) is a method that makes use of a SUT's model for deriving test suites and a corresponding oracle. The model represents a formalization of the SUT's essential behavior, from which tests, i.e., the required input and the expected output, can be obtained more or less directly. This usually results in abstract test cases, requiring a transformation into executable ones where parameters and actions are mapped to concrete values and actions.

Passive testing (Arnedo, Cavalli, & Nunez, 2003), or monitoring, is a testing methodology that mainly applies in situations where a SUT is at least not supposed to be fully controlled. The input then solely comes from users or the environment, and from real scenarios at that. Corresponding traces for these interactions are monitored. The passive tester then takes a system specification as *reference* model, and evaluates the collected traces with respect to this model in order to qualify a certain trace as correct or incorrect. Since we do not derive new test cases but reuse existing ones (possibly extended by a full live operation where we do not provide any inputs at all) and classify the results according to an abstract (safety) requirements specification, rather than checking the detailed functionality, our work could be classified as passive testing approach with an active "touch". Obviously, also the concepts for the *SDC* component have to tackle many challenges of passive testing.

A method central to our approach is fault injection. *Software fault injection* (Voas & McGraw, 1999) is a common technique used in software testing as a modality to verify the application's robustness and also tolerance of selected faults. This technique assumes the availability of a selection of operators that inject faults into the application. Among fault injection techniques commonly used for software applications, *mutation testing* is the oldest one, being introduced for the first time in 1971 (DeMillo, Lipton, & Sayward, 1978). Different types of software systems may use the benefits of mutation testing, since it can be successfully applied to different levels of testing and for different programming environments. In the context of mutation testing, we evaluate the quality of a test suite, i.e., a set of test cases, by injecting small software changes, i.e., *mutations*, at source code or byte code level, and then we verify whether there is some anomalous response. That is, the generated test suite is run against all the mutants generated (the altered software versions), and we investigate whether there is some test case in the suite s.t. the output differs for the original program and a mutant. The *mutation score* (i.e., the percentage of mutants for which the test suite offers such a test case) serves as metric for assessing a test suite's quality.

There are two major drawbacks with mutation testing. For one, there are *time complexity* issues in respect of the ressources needed to run all the tests for all the mutants, specifically if we include many mutation operators. The other issue is related to interpreting the mutation score. That is, since, most likely, the functional equivalence between a mutant and its original program is not investigated beforehand, we run into the problem of identifying those mutants for which the test suite does not offer a killing test case, but which are functionally (semantically) equivalent to the original program (and thus represent an alternative correct implementation). If this cumbersome process is not executed, such *equivalent mutants* result in a lower mutation score, which has to be taken into account. There are many tools available for mutation testing, e.g.: **FIAT** (Fault Injection-based Automated Testing) (Segall et al., 1988), **PROTEUM** (Delamaro, 1993; Agrawal et al., 1989; Ghosh, 2000) tools for C source code mutation testing, **MUJAVA** (Ma, Offutt, & Kwon, 2006) a Java based mutation tool, and **SQLMutation** (Tuya, Suárez-Cabal, & la Riva, 2007) and **JDAMA** (Zhou & Frankl, 2009) for SQL. In this paper we assume that there is such a mutation testing tool for the desired development environment, that in turn allows us to inject faults into the control software.

4. TESTING THE SAFETY DIAGNOSTICS COMPONENT

In Sec. 2, we formalized the purpose of a safety diagnostics component in Def. 1 (see also Fig. 1). Informally, its task is to monitor all the supervised system *CS*'s activities, and, if it detects hazardous behavior (such that the safety requirements would not be met), actions shall be taken in order to maintain safety. To this end, the *SDC* collects all the inputs I as considered by the control software, as well as *CS*'s output O, and, if needed, issues special signals overriding O and resulting in output O' s.t. the scenario I/O' actually implements *REQ*. The obvious question then is whether *SDC* indeed lives up to this task and actually conforms to *REQ* as of Def. 2.

For practical purposes, rather than considering a completely unrestrained fault model, we included in Def. 2 also a set of faults *FAULTS* that should be considered for our conformance tests. For those faults $f \in \mathit{FAULTS}$, we can derive faulty mutants as of Definition 3 in order to exercise *SDC*. A relevant fault mode for the WUMPUS example could be that a gear switch would result in the wrong gear, and for the parking brake a bit-flip in the derived wheel blocking signal such that the wheels would be released without the driver requesting it.

Definition 3 *For some program CS, a mutant CS′ is an altered version of the original program. A mutant is equivalent to CS if and only if they do not differ in their behavior.*

An advantage of our setting is that the identification of equivalence in the traditional sense is not of importance in respect of achieved scores. Since *SDC* shall remedy the effects of a mutation (at the least in respect of the requirements *REQ*, but possibly also concerning graceful degradation), we have the situation that the original program and all the mutants are even expected to be equivalent in respect of conformance to *REQ*. Verifying whether all the mutants' behavior, as described by I/O' scenarios, implements *REQ* thus directly translates to addressing our question.

For judging whether some I/O' scenario actually implements *REQ*, we can directly take the requirements' formalization and use a SAT or constraint solver to implement an oracle and check the satisfiability of $I \cup O' \cup \mathit{REQ}$ as of Def. 2.

This leaves us with the question of which inputs to use. Obviously, an exhaustive solution is impractical. The motivation to effectively and efficiently exercise *CS* with our input scenarios is however shared with those test cases that were designed for functionally evaluating *CS* itself. Thus we propose to actively exploit this and reuse those inputs in our context, minimizing test design efforts.

Now that we have established the basic concepts for our approach, let us briefly consider possible deployment issues to be faced in industrial applications. An automotive application, for instance, certainly qualifies for implementing some *SDC* due to the complexity of related products (like cars) and designated operating environments, and it is a domain sensitive to safety concerns. There, for business and complexity reasons, we cannot assume, for instance, the system to be available to us as a white box, but rather a box with sub-boxes, interfaces, and intellectual property cores in varying shades of grey. Furthermore, an easy integration into existing and proven development concepts is essential in order for an approach to be attractive enough for actual deployment. In order to accommodate such concerns, we make the following assumptions in respect of the data available to us.

- First, the requirements *REQ* to be monitored and enforced by *SDC* have to be available. If we have to convert their informal characterization into a formal syntax (as accessible by automated tools) first, research like (Pill & Quaritsch, 2013) can identify mistakes in this process and provides the means to investigate unexpected results - but is out of the scope of this paper. The formalized requirements *REQ* will be used to asses whether an individual test scenario passes or fails our expectations in that it complies with *REQ* or not. Please remember that the requirements, most likely, implement abstract safety requirements and do not encode the system's detailed functionality (if the latter is taken to the extreme, *SDC* could become redundant to *CS*). That is, some safety requirement like that a robot should never hit a wall, or that an automated parking brake should never block the wheels when the car is still moving (i.e. at high velocity).
- Second, we assume that the control software *CS* offers us controls to inject / simulate / enable faults $f \in \mathit{FAULTS}$ via a function $\mu(CS, f)$. Thus secrecy about detailed intellectual property can be maintained, and the developers working on the very components and system aggregation can integrate the best fault injection (mutation) operators for the individual context. What is needed, however, is (a) some guidelines in order to assist these designers in providing meaningful faults (b) a full list *FAULTS* of available fault models, and (c) a guideline for interpreting the impact of each fault $f \in \mathit{FAULTS}$ so that we, on one hand, gain confidence in the results, and on the other hand can act on encountered issues (e.g., when a test case produces unexpected results - see also the discussion of Algorithm 2).
- Third, we assume the availability of the test cases designed for *CS* (or the overall system). This test suite *TS* was derived in order to extensively exercise the system and ensure its correctness, so that there is no need to come up with entirely new input scenarios. We, however, are interested only in the input part of these test cases, since the evaluation of the system's response is done with respect to the abstract system safety requirements *REQ* instead of the original functional or performance concerns. To this end, the formalized and thus executable requirements are used as testing oracle in order to classify the test output.

Now let us propose an easily integrable testing approach.

The underlying concept of the algorithm as depicted in Algorithm 1 is as follows. For any fault $f \in \mathit{FAULTS}$, we create a corresponding mutant *SUT′* using the injection function μ. Then, for any test case $t \in \mathit{TS}$ we execute the mutant for the corresponding inputs of t, and record the trace of this execution. This trace is checked for compliance with the requirements *REQ* and classified accordingly. Naturally, this raises the question of whether to include also the unmodified program in the tests, which we allow the test engineers to answer for Algorithm 1 such that they can include a corresponding fault ϵ in *FAULTS* that results in an unaltered program.

Note that our concept is orthogonal to the decision of whether the system/environment (or possibly a mock-up) is part of a *SUT*'s model or not - which might depend on the test setup and design stage. Thus, while the *SUT* might contain it, we will omit it in our algorithmic presentations.

Algorithm 1 TEST-SDC$(\mathit{SUT}, \mathit{FAULTS}, \mu, \mathit{REQ}, \mathit{TS})$

Input: The system under test *SUT* $(\mathit{CS} + \mathit{SDC} + \mathit{SELFCT})$, the set *FAULTS* of faults to be considered, the fault injection function μ, the safety requirements *REQ*, and the original test suite *TS* for *CS* (or S).
Output: *PASS* if the system under test behaves as demanded by the given requirements *REQ*, and *FAIL* otherwise.

```
for all $f \in \mathit{FAULTS}$ do
    $\mathit{SUT}' = \mu(\mathit{CS}, f) + \mathit{SDC} + \mathit{SELFCT}$
    for all $t \in \mathit{TS}$ do
        res := EXECUTE($\mathit{SUT}'$, $\mathit{input}(t)$)
        if $\mathit{res} \cup \mathit{input}(t) \cup \mathit{REQ} \models \bot$ then
            return FAIL
        end if
    end for
end for
return PASS
```

Algorithm 1 answers the most basic question of whether there was a scenario (a combination of a test case t and an injected fault f) that violated the safety requirements, or if we have that all the scenarios complied with *REQ*. A viable alteration to the algorithm would be to return for an encountered failed scenario, both the test case t and the injected fault f. Another variant could run all the tests (archiving the results) instead of stopping on the first unveiled violating scenario.

Reusing the test cases that were designed to extensively exercise *CS*, for one, allows us to compare and connect test results for *CS* (or the overall system $S = \mathit{system/environment} + \mathit{CS}$) with our special purpose tests of $S^+ = \mathit{CS} + \mathit{SDC}$ (or $S' = S^+ + \mathit{system/environment}$) for evaluating the effectiveness of *SDC*, and furthermore we do not require additional ressources for test design (not counting the mutation function μ). Comparing the evaluation of testing *CS* and our special tests for S^+, however, is not as simple as it might sound, since the testing purposes (and thus the oracles for classifying the results) differ significantly. That is, when we test the safety diagnostics component *SDC*, the focus is solely on the safety requirements, while for testing *CS* such concerns are mingled with others like functionality and performance. In other words, regardless of whether a test scenario in *TS* was originally intended to meet or fail some other goal, *REQ* is to be met anyhow. Nevertheless, reusing the test inputs helps us in making the connections when interpreting the results in more detail.

While Algorithm 1 focuses on verifying whether *SDC* lives up to the expectations encoded in *REQ*, it shall be noted that it does not aim at verifying whether adding *SDC* to the system would result in degraded functionality or performance (in some respect other than conformance to the safety requirements *REQ*). To this end, a system integration test could be of interest also for S^+ (S'). Also there it makes sense to reuse the test cases generated for S, and even to reuse the same evaluation principles (oracle). That the runs for these system integration tests and our testing purpose could be executed in unison (with multiple or varying evaluations/oracles) is most evident and implemented by Algorithm 2.

Algorithm 2 TEST-SDC-EXT$(\mathit{SUT}, \mathit{FAULTS}, \mu, \mathit{REQ}, \mathit{TS})$

Input: The system under test *SUT* $(\mathit{CS} + \mathit{SDC} + \mathit{SELFCT})$, the set of faults to be considered *FAULTS*, the fault injection function μ, the safety requirements *REQ*, and the original test suite *TS* for *CS* (or S).
Output: The test results *TR*, which is a list of tuples $\{t, f, res, r\}$ such that $t \in \mathit{TS}$, $f \in \{0 \cup \mathit{FAULTS}\}$ tells us the injected fault f (0 indicates that we did not inject a fault), *res* is the output obtained when executing the scenario (the combination of t and f), and $r \in \{\mathit{REQPASS}, \mathit{REQFAIL}\}$ indicates whether for the described scenario the requirements *REQ* are violated or not.

```
$\mathit{TR} \leftarrow \emptyset$
for all $t \in \mathit{TS}$ do
    res := EXECUTE($\mathit{SUT}$, $\mathit{input}(t)$)
    if $\mathit{res} \cup \mathit{input}(t) \cup \mathit{REQ} \models \bot$ then
        $\mathit{TR} = \mathit{TR} \cup \{t, 0, res, \mathit{REQFAIL}\}$
    else
        $\mathit{TR} = \mathit{TR} \cup \{t, 0, res, \mathit{REQPASS}\}$
    end if
    for all $f \in \mathit{FAULTS}$ do
        $\mathit{SUT}' = \mu(\mathit{CS}, f) + \mathit{SDC} + \mathit{SELFCT}$
        res := EXECUTE($\mathit{SUT}'$, $\mathit{input}(t)$)
        if $\mathit{res} \cup \mathit{input}(t) \cup \mathit{REQ} \models \bot$ then
            $\mathit{TR} = \mathit{TR} \cup \{t, f, res, \mathit{REQFAIL}\}$
        else
            $\mathit{TR} = \mathit{TR} \cup \{t, f, res, \mathit{REQPASS}\}$
        end if
    end for
end for
return TR
```

Each test case is executed for the correct *SUT* as well as all the mutated versions *SUT′* obtained by injecting the individual faults $f \in \mathit{FAULTS}$. Then the obtained outputs as well as the verdict whether the outputs satisfy or contradict the given requirements *REQ* are archived. Please note that one could easily add to the archived tuple also any further functional/non-functional evaluation results. With Algorithm 2, we thus can directly compare the results for the unmodified *SUT* S^+ (S') (including the *SDC*) with those for the original system (excluding the *SDC*) in order to assess the impact of *SDC* on the performance. Furthermore, we do store also the outputs for the test scenarios with injected faults, to the end of supporting later inspections of further aspects.

While the two algorithms offer varying details to the user, the main question addressed is whether the *SDC* component is indeed able to keep the overall system for the test scenarios within the boundaries defined by *REQ*. If this is not the case, there is the obvious question of how to debug this situation. While answering this question is not in the scope of this paper, dynamic slicing techniques (Korel & Rilling, 1998; Zhang, He, Gupta, & Gupta, 2005) could certainly be of help if we start from the signals involved in the violated requirement and reason backwards to the inputs. An urgent question for every failed scenario will be whether the *SDC* did not catch the issue at all, s.t. its monitoring capabilities are insufficient, or whether its response to the situation was inadequate. For an enhanced automated support of debugging which individual parts of the requirements were involved in their violation, adopting the work on requirements/specification diagnosis presented in (Pill & Quaritsch, 2013) could be of interest.

5. DISCUSSION AND CONCLUSIONS

In this paper, we propose a testing concept tailored towards gaining confidence that a functional safety diagnostic component indeed lives up to our expectations of monitoring a system at an abstract level and maintaining safety for malfunctions in the more detailed and highly complex main control system. We depict an initial attempt at such an approach that does neither demand for the design of additional tests, nor requires us to entirely restructure our testing efforts in the given development cycle. Instead, we reuse the input part of those test cases designed for the monitored system, and the oracle is implemented directly from the given formalized requirements. Thus we minimize needed resources and efforts.

The only thing we have to rely on is a fault injection function for the main control software that allows us to inject faults so that we can effectively trigger actions from the safety diagnostics component. This accommodates also business concerns related to intellectual property rights and the availability of detailed system models, such that designers only have to add corresponding functionality that they can also use for their testing of the individual components. In the worst case of having no such function, we could revert to simple mutations on the software's output signals. We outline two algorithms that (1) offer a quick search for a scenario violating the requirements, or (2) run the complete test suite for the unmodified SUT as well as all the mutated versions achievable via the given fault injection function, storing all the obtained results. While the first offers us some quick check whether everything is fine in respect of compliance to the safety requirements, the second variant is more tailored towards a full inspection where the obtained data are also stored for future evaluation in respect of other aspects.

Future work will have to show the practical viability of our concept. While the reuse of tests does have its advantages as discussed, we will also explore ideas for (additional) tests solely derived for testing an SDC and their impact on general fault detection performance. A specific aspect of this research will be to isolate means that can help us in selecting test scenarios (combinations of faults and test inputs). That is, while we showed with Algorithm 2 that our work can be integrated easily into the system integration tests that we certainly would like to run, the available testing resources might not suffice to run all the combinations of mutated SUT variants and input sequences in the test suite. Identifying an effective prioritization scheme would then certainly be of interest.

Also the process of identifying an effective selection of mutation operators/functions to be used for some project will be a target of our future research. That is, mutation testing relies on the Competent Programer Hypothesis and the Coupling Effect, which assume that (a) a faulty program is close to the correct one and suggest that (b) a test suite that catches simple mutations is also effective at catching more complex faults (see e.g. the discussion in (Offutt, 1992)). Newer findings like the discussion in (Gopinath, Jensen, & Groce, 2014) investigate the impact of the selection of mutation operators on the performance for a specific project, and suggest that for a specific project, further empirical data should provide a solid empirical footing for the underlying hypotheses' validity. In this context, we limit our algorithms currently to injecting single faults, where the accommodation of multiple faults for more complex scenarios is subject to future work. In the latter respect, combinatorial testing like it was used in (Wotawa & Pill, 2014) for configuration testing in an automotive context will be of interest.

Enhancing the support for a user debugging those scenarios that violated the requirements, adopting the work on specification/requirements diagnosis in (Pill & Quaritsch, 2013) could provide interesting stimuli, as could the work for diagnosing failed test cases for service-oriented software architectures (Hofer, Jehan, Pill, & Wotawa, 2014).

Last but not least, we currently consider functional safety requirements only. In the future, also *SDC* solutions for non-functional requirements should profit from an improved concept accommodating also non-functional requirements.

ACKNOWLEDGMENT

Our research was conducted as part of the eDAS project funded by the European Commission under FP-7 grant agreement number: 608770.

REFERENCES

Agrawal, H., DeMillo, R. A., Hathaway, B., Hsu, W., Hsu, W., Krauser, E. W., ... Spafford, E. (1989, March). *Design of Mutant Operators for the C Programming Language* (techreport No. SERC-TR-41-P). West Lafayette, Indiana: Purdue University.

Arnedo, J. A., Cavalli, A., & Nunez, M. (2003). Fast testing of critical properties through passive testing. In *Lecture Notes in Computer Science* (Vol. 2644, pp. 295–310). Springer.

Bloem, R., Cavada, R., Pill, I., Roveri, M., & Tchaltsev, A. (2007). RAT: A tool for the formal analysis of requirements. In *Computer aided verification* (p. 263-267).

Broy, M., Jonsson, B., Katoen, J.-P., Leucker, M., & Pretschner, A. (2005). Model-based testing of reactive systems. In *Lecture Notes in Computer Science* (Vol. 3472). Springer.

Clarke, E. M., Jr., Grumberg, O., & Peled, D. A. (1999). *Model checking*. Cambridge, MA, USA: MIT Press.

de Kleer, J., & Williams, B. C. (1987). Diagnosing multiple faults. *Artificial Intelligence*, *32*(1), 97–130.

Delamaro, M. E. (1993). *Proteum - a mutation analysis based testing environment* (phdthesis). University of São Paulo, Sao Paulo, Brazil.

DeMillo, R. A., Lipton, R. J., & Sayward, F. G. (1978). Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, *11*, 34–41.

Eisner, C., & Fisman, D. (2006). *A practical introduction to PSL (series on integrated circuits and systems)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc.

Ghosh, S. (2000). *Testing Component-Based Distributed Applications* (phdthesis). Purdue University, West Lafayette, Indiana.

Gopinath, R., Jensen, C., & Groce, A. (2014, Nov). Mutations: How close are they to real faults? In *2014 IEEE 25th International Symposium on Software Reliability Engineering (ISSRE)* (p. 189-200). doi: 10.1109/ISSRE.2014.40

Hofer, B., Jehan, S., Pill, I., & Wotawa, F. (2014). Functional diagnosis of a SOA's BPEL processes. In *25th International Workshop on Principles of Diagnosis (DX).*

Kim, Y., Daly, R., Kim, J., Fallin, C., Lee, J. H., Lee, D., ... Mutlu, O. (2014, June). Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)* (p. 361-372). doi: 10.1109/ISCA.2014.6853210

Korel, B., & Rilling, J. (1998). Dynamic program slicing methods. *Information & Software Technology*, *40*(11-12), 647-659.

Ma, Y., Offutt, A. J., & Kwon, Y. (2006). MuJava: a Mutation System for Java. In *Proceedings of the 28th international Conference on Software Engineering (ICSE '06)* (pp. 827–830). Shanghai, China.

Offutt, A. J. (1992). Investigations of the software testing coupling effect. *ACM Trans. Softw. Eng. Methodol.*, *1*(1), 5–20. doi: 10.1145/125489.125473

Pill, I., & Quaritsch, T. (2013). Behavioral diagnosis of LTL specifications at operator level. In *Twenty-Third International Joint Conference on Artificial Intelligence (IJCAI'13)* (pp. 1053–1059).

Pnueli, A. (1977). The Temporal Logic of Programs. In *Annual Symposium on Foundations of Computer Sc.* (p. 46-57).

Reiter, R. (1987). A theory of diagnosis from first principles. *Artif. Intelligence*, *32*(1), 57–95.

Segall, Z., Vrsalovic, D., Siewiorek, D., Yaskin, D., Kownacki, J., Barton, J., ... Lin, T. (1988). FIAT-fault injection based automated testing environment. In *Eighteenth International Symposium on Fault-Tolerant Computing, 1988. FTCS-18, Digest of Papers* (p. 102-107). doi: 10.1109/FTCS.1988.5306

Tuya, J., Suárez-Cabal, M. J., & la Riva, C. d. (2007, April). Mutating Database Queries. *Inf. Softw. Technol.*, *49*(4), 398–417. doi: 10.1016/j.infsof.2006.06.009

Voas, J., & McGraw, G. (1999). Software fault injection: inoculating programs against errors. *Software Testing, Verification and Reliability*, *9*(1), 75–76.

Weber, J., & Wotawa, F. (2010). Combining runtime diagnosis and ai-planning in a mobile autonomous robot to achieve a graceful degradation after software failures. In *ICAART 2010 - Proc. of the Int. Conf. on Agents and Artificial Intelligence, Volume 1 - Artificial Intelligence* (pp. 127–134).

Wotawa, F., & Pill, I. (2014). Testing configuration knowledge-bases. In *16th International Configuration Workshop* (pp. 39–46).

Zhang, X., He, H., Gupta, N., & Gupta, R. (2005). Experimental evaluation of using dynamic slices for fault localization. In *Sixth International Symposium on Automated & Analysis-Driven Debugging (AADEBUG)* (pp. 33–42).

Zhou, C., & Frankl, P. (2009). Mutation Testing for Java Database Applications. In *Proc. of the 2009 Int. Conf. on Software Testing Verification and Validation* (pp. 396–405). doi: 10.1109/ICST.2009.43