Combining Fault-Injection with Property-Based Testing

Benjamin Vedder
SP Electronics
SP Technical Research
Institute of Sweden
benjamin.vedder@sp.se

Thomas Arts
Quviq AB
thomas.arts@quviq.com

Jonny Vinter
SP Electronics
SP Technical Research
Institute of Sweden
jonny.vinter@sp.se

Magnus Jonsson
School of Information Science,
Computer and Electrical
Engineering
Halmstad University
Magnus.Jonsson@hh.se

ABSTRACT
In this paper we present a methodology and a platform using Fault Injection (FI) and Property-Based Testing (PBT). PBT is a technique in which test cases are automatically generated from a specification of a system property. The generated test cases vary input stimuli as well as the sequence in which commands are executed. FI is used to accelerate the occurrences of faults in a system to exercise and evaluate fault handling mechanisms and e.g. calculate error detection coverage. By combining the two we have achieved a way of randomly injecting different faults at arbitrary moments in the execution sequence while checking whether certain properties still hold. We use the commercially available tool QuickCheck for generating the test cases and developed FaultCheck for FI. FaultCheck enables the user to utilize fault models, commonly used during FI, from PBT tools like QuickCheck. We demonstrate our method and tools on a simplified example of two Airbag systems that should meet safety requirements. We can easily find a safety violation in one of the examples, whereas by using the AUTOSAR E2E-library implementation, exhaustive testing cannot reveal any such safety violation. This demonstrates that our approach on testing can reveal certain safety violations in a cost-effective way.

Keywords
Fault model, QuickCheck, FaultCheck, Fault Injection, Property-Based Testing

1. INTRODUCTION
Testing cannot reveal the absence of software defects, but it is one of the most cost effective ways of demonstrating that certain requirements are not fulfilled. Property-Based Testing (PBT) with QuickCheck [1] is a demonstrated way to get more effective testing in a cost effective way [2]. Among others, it has been used for large scale testing of AUTOSAR software [3]. When using PBT for complex software, a model is created that describes certain properties of the software specified by so called functional requirements. The PBT tool automatically generates and runs many tests in an attempt to falsify these properties. When the properties are falsified a counterexample in a minimalistic form is shown that can be used to either fix a bug in the implementation or to revise the specification. The case where the PBT tool is not able to falsify a property is not a formal proof that the property holds, but it shows that it holds for numerous randomly generated inputs.

In particular when it comes to costly procedures of certifying software where additional arguments have to be provided concerning the correctness of the implementation, one would better be sure it is very well tested. Not only for the common case, but also in cases in which faults occur and the software should deal with those faults.

Under normal circumstances it is unlikely that errors occur that the software should react upon. Therefore the purpose of Fault Injection (FI) is to introduce errors while testing. Thus, the goal of FI is to inject faults into software and/or the hardware connected to the software in order to ensure that the system still fulfills certain requirements while these faults are present. For example, in safety-critical systems, it has to be made sure that the system is not dangerous when certain faults are present. This means that FI deals with non-functional requirements.

In general, the models used for PBT are describing the behaviour of the system in case no faults occur. Injecting faults might very well change the functional behaviour and make tests fail w.r.t. the functional behaviour, even though this behaviour would still be correct from the safety point of view. The challenge we address in this paper is to enhance the model with specifying its behaviour in case of occurring faults, but still be able to detect functional defects in case no faults occur. For that, we need to make the models aware of the injection of faults and the generated test cases should also be controlling the fault injection.
In this paper, we present a method for combining PBT and FI in order to test safety-critical systems in an effective way. According to our knowledge, this has not been reported before. We demonstrate the gained possibilities through experiments with an example utilizing the AUTOSAR End-to-End (E2E)-library.

The rest of the paper is organized as follows. In Section 2 we introduce FI with related work, and in Section 3 we introduce PBT with related work. In Section 4 we introduce the FaultCheck tool that is developed within this study. Section 5 shows a use case where this platform is used and in Section 6 we present our conclusions from this study.

2. FAULT INJECTION

FI is used to accelerate the occurrences of faults in a system [4]. The aim is to exercise and evaluate fault handling mechanisms and calculate fault-tolerance measures such as error detection coverage. Traditionally, targets for FI have been hardware (microprocessors and memories), simulation models of hardware and software running on hardware. Examples on different approaches for hardware-implemented FI includes heavy-ion FI [5], pin-level FI [6, 7], Scan-chain implemented FI [8] and Nexus-based FI [9]. Examples on approaches for FI in models of hardware include simulation-based FI [10–12]. Software-implemented FI can be used pre-runtime [13] or during runtime [14]. Other approaches have been presented in the literature dealing with FI directly in source code [15] and in models of, e.g., software (denoted here as model-based FI) [16, 17] from which source code may be generated. Such techniques can be used early in the software development process and dependability flaws can be corrected less costly compared to flaws found during later phases. Model-based FI is particularly useful during model-based development of systems. This paper presents ideas on how to combine knowledge from FI (fault tolerance against hardware faults) with knowledge on PBT (fault removal of software faults) to enhance PBT testing tools such as QuickCheck. The ideas are implemented in a tool called FaultCheck, which is presented in detail in Section 4.

One simple example of model-based fault injection is illustrated in Figure 1. In this example, a model is fed with the same inputs over and over again while different faults are injected. The output is compared to the output from a golden run where everything was run without faults present.

In summary, fault injection deals with showing how fault tolerant a system is, evaluating fault handling mechanisms and determining error detection coverage. One opportunity for improvement lies in checking how numerous auto-generated input sequences affect the impact of faults.

3. PROPERTY-BASED TESTING

Property-based testing [18] is a technique in which test cases are automatically generated from a specified property of a system. For example, if we would have a function computing one trusted value from three possibly different sensor values, then one could express a property of the output value in relation to the input values. With PBT one would automatically generate test sequences that vary the sensor input data and compare the output data with the “modelled” output data in the property.

One example of PBT is shown in Figure 2 where the QuickCheck tool [1, 19] is used. In this example, a very simple application is tested and the model contains all the details of the implementation. In more sophisticated applications, the model might contain only a few details of the implementation relevant for evaluating the interesting properties.

3.1 Combing Property-Based Testing and Fault Injection

In this paper we present how techniques from PBT and FI are used together. Advantages are that we can automatically generate test cases and check if the state of the System Under Test (SUT) is as expected while injecting faults at the same time.

In order to combine FI with PBT, a set-up as illustrated in Figure 3 is used. We show that it is possible to define properties that are supposed to hold with certain faults present in the system and run tests with randomly generated Application Programming Interface (API) calls.

In this study we have implemented one way to perform FI on C code while using techniques from PBT and QuickCheck to generate thousands of “golden runs” automatically. The
approach presented in this paper shows the concept of the idea.

4. FAULTCHECK

FaultCheck is a tool under early development with the aim to support FI into C and C++-code. It consists of a library written in C++ with a wrapper around C, so it can easily be included and linked against in existing applications. FaultCheck is designed to be used by other tools that perform property-based testing with QuickCheck or other PBT tools, such as ScalaCheck [20]. A block diagram of a typical use case of FaultCheck is shown in Figure 4.

Currently, there are two parts of FaultCheck under development:

Probing, which is done by modifying existing C and C++ applications. This way, faults can be injected anywhere in the application, to simulate hardware faults that manifests as errors at the software level, at the cost of some overhead in execution time and code space.

Communication channel emulation, which is an interface that provides an emulated communication channel into which a number of communication faults can be injected to evaluate to which extent the application can handle them.

The motivation for making the communication channel accessible from C is that many programming languages such as Python [21], Java and Scala [22] have the ability to interface with C libraries. In this way, FaultCheck can easily be used from other tools and languages.

4.1 Fault Models

A hardware fault model can be defined as the number of faults, the duration and the type of the fault. An example is a single transient bit-flip fault. Another example is multiple permanent stuck-at-zero faults. In this study we have implemented several hardware fault models as well as communication fault models. Several of these fault models are handled by the AUTOSAR E2E library [23]. We have tested that injecting these faults when using that library did not violate our safety requirements.

4.2 Supported Fault Models

FaultCheck currently supports the following fault models that can be injected into C and C++-code via probing:

- **BITFLIP**
  - Flips a specific bit in a variable.
- **BITFLIP_RANDOM**
  - Flips a randomly selected bit in a variable.
- **STUCK_TO_CURRENT**
  - Freezes a variable to the last value it had.
- **SET_TO**
  - Sets a variable to a pre-set value.
- **AMPLIFICATION**
  - Scales a variable with a factor.
- **OFFSET**
  - Adds an offset to a variable.

The communication channel emulation currently supports these fault models:

- **REPEAT**
  - Repeats a packet a number of times.
- **DROP**
  - Drops a number of packets (loss of information).
- **CORRUPTION**
  - Alters the data of a packet with any of the fault models specified in the probing part for variables. Since the same code as for the probing interface is used, features added to the probing interface can be made available for the corruption fault easily.
4.3 Probing C-Code

The probing interface of FaultCheck can be used by including the FaultCheck headers and linking against its dynamic libraries. This way of probing C code has been inspired by a tool called PROPANE [24] which supports fault injection probes and monitoring probes.

The following sample code shows how a C application can be probed by using the FaultCheck tool:

```c
#include "faultcheck_wrapper.h"

// C code...

SensorValue sensor_evaluate(int S1val, int S2val, int S3val) {
    SensorValue sv;
    // some code...
    faultcheck_injectFaultInt("S1val", &S1val);
    faultcheck_injectFaultInt("S2val", &S2val);
    faultcheck_injectFaultInt("S3val", &S3val);
    // Some more code...
    return sv;
}
```

Here, pointers to the integers S1val, S2val and S3val are sent to FaultCheck with string identifiers, and based on the configuration and previous events they may be modified.

Probing combined with the triggering functionality described in Section 4.6 can, for example, be used to precisely affect certain iterations of loops in C programs in a way that is not possible by only using the external interface of the program.

4.4 Communication Channel Emulation

A packet-based communication channel can be emulated by FaultCheck and used from C programs. The following example shows how one packet is encapsulated by the AUTOSAR E2E-library [23] and passed to the communication channel through the AUTOSAR FaultCheck and used from C programs. The following example illustrates how one packet is encapsulated by the AUTOSAR E2E-library [23] and passed to the communication channel through the AUTOSAR FaultCheck.

```c
void sensor(unsigned char *data) {
    unsigned char buffer[config.DataLength + 2];
    memcpy(buffer + 1, data, config.DataLength);
    E2E_P01Protect(xconfig, ksender_state, buffer);
    faultcheck_packet_addPacket("airbag",
        (char*)buffer, config.DataLength + 2);
}
```

This packet will be added to a buffer in FaultCheck and can later be read by using `faultcheck_packet_getPacket` in a similar manner from the application. When the packet is read, the communication channel faults that are enabled will be applied to the packet.

4.5 Integration with other Tools

FaultCheck is not intended to be used as a stand-alone testing framework. It should be used together with other tools that preferably use PBT. Many tools that do PBT on C code already have access to the interface of that code, therefore FaultCheck extends the interface of the C applications with functions to control the FI. This means that the tool that performs PBT can use the functions provided by FaultCheck in the same way it uses the other functions of that C application.

The following sample code shows how to activate a bit-flip fault for one identifier:

```c
faultcheck_addFaultBitFlip("S1val", 3, 1);
```

This would flip bit number three (zero is the least significant bit) in the variable S1val in the same manner as shown in Section 4.3. Every time the function sensor_evaluate would be called after this point, the third bit of S1val would be flipped.

Multiple faults can also be added to the same identifier at the same time. They will be applied to the data in the order they were added. For example, in order to first flip bit number two, then add 23 and then flip bit number 11, the following calls would be used:

```c
faultcheck_addFaultBitFlip("S1val", 2);
faultcheck_addFaultOffset("S1val", 23);
faultcheck_addFaultBitFlip("S1val", 11);
```

Note that this combination of faults might not be a realistic fault model, but it shows the flexibility of FaultCheck to design complex fault models.

4.6 Temporal Triggers for Faults

In addition to information about the type and parameters of the faults, FaultCheck also keeps track of when faults should be activated. This mechanism is called temporal triggering and essentially means that fault activations can be delayed for a number of iterations and then be active for a number of iterations. The notion of iteration in this context means that every time the probing function is called, one iteration has occurred. As multiple faults can be added to each identifier, each of these faults can also have an individual trigger.

The following is an example where multiple faults are added for the same identifier (S1val) with different triggers:

```c
faultcheck_addFaultBitFlip("S1val", 2);
faultcheck_setTriggerAfterIterations("S1val", 120);
faultcheck_setDurationAfterTrigger("S1val", 45);

faultcheck_addFaultOffset("S1val", 23);
faultcheck_setTriggerAfterIterations("S1val", 45);
faultcheck_setDurationAfterTrigger("S1val", 200);

faultcheck_addFaultBitFlip("S1val", 11);
faultcheck_setTriggerAfterIterations("S1val", 130);
faultcheck_setDurationAfterTrigger("S1val", 10);
```

The trigger will be applied to the latest fault that was added. So, the way to add multiple faults with triggers is to add one
fault, set the trigger for it, add another fault, set the trigger for it and so on. The code snippet above will cause the following to happen:

- A bit flip on bit number two that is active from iteration 120 and for 45 iterations after that.
- An offset of 23 that is active from iteration 45 and for 200 iterations after that.
- A bit flip on bit number 11 that is active from iteration 130 and for 10 iterations after that.

Note that the offset will be triggered before the first bit flip, but when the first bit flip is also triggered it will be applied before the offset. This scenario will occur during iteration 120 to iteration 165. When several faults are triggered at the same time, they will be applied to the variable in the same order as they were added in the code.

The previous examples illustrate how to use triggers with the probing functionality, but triggers can also be used in the same way with the communication channel of FaultCheck. For the communication channel, one iteration is defined as every time `faultcheck_packet_getPacket` is called.

5. AN EXAMPLE: AUTOSAR E2E

When designing safety-critical control systems, one needs to ensure that safety requirements are met. In different domain areas corresponding safety standards help to guide safety engineers through the process of formulating requirements and designing for safety.

In the automotive domain the standard that covers functional safety aspects of the entire development process is called ISO 26262 [25], which is an adaptation of the IEC 61508 [26] standard. The AUTOSAR E2E-library takes the ISO 26262-standard into consideration and is supposed to work with all Automotive Safety Integrity Levels (ASILs) regarding communication when the implementation recommendation is followed. This can reduce the development effort and make the implementation compatible with other AUTOSAR components.

In short, the AUTOSAR E2E library supports the detection of corruption and loss of data when transporting data from one end to the other. This is a building block for safety solutions, since typical fault models include that data on a bus may occasionally be corrupt or even be totally lost.

On top of this library we developed an example of an airbag system with three sensors to detect a collision. The sensors are continuously sampled and their combined data is constantly sent to the airbag ignition system. It would be unsafe if that data could be corrupted in such a way that the airbag would spontaneously fire. Hence, the airbag ignition system needs to know that the data received from the sensors can be trusted. Thus, safety requirements state that the airbag should explode when the car crashes, but even more important that it does not explode at low speed or without a crash.

In order to test to whether the E2E-library indeed offers good protection against some of the fault models it is designed to handle, a system as shown in Figure 5 has been developed. It works in the following manner:

1. A series of calls is generated and fed into a C application that uses the AUTOSAR E2E-library. In this case, a series of calls with a certain command, [85, 170], should update a variable in the state. When these commands are not sent, the state is not allowed to change regardless of what other commands are sent.
2. The commands from the calls are encapsulated by the E2E protection wrapper that uses the E2E library. The encapsulated packets are then passed to the FaultCheck tool.
3. If a fault is activated, FaultCheck will alter the packet based on the chosen fault model before it is fetched by the application.
4. The E2E protection wrapper fetches packets from FaultCheck and checks them by using the E2E library. Then it performs state updates based on the results.
5. The QuickCheck tool analyses the state of the application and determines whether the state follows the specification or not.

The reason that multiple commands are sent to update the state is that the AUTOSAR E2E-library recommends that the application is able to handle one faulty packet by itself.

In order to validate our framework, we also tested the application without using the AUTOSAR E2E-library to see which failures we can detect. Later we confirmed that the E2E-library protects against the faults that cause these failures.

The C-code is tested with four different QuickCheck-commands: `sensor`, `bit_flip`, `repetition`, and `explosion`. The `sensor` command is run 10 times more frequently than the other commands and looks as follows:
sensor(Data) ->
  DataPtr = eqc_c_create_array(unsigned_char,Data),
  c:call:airbag_iteration_e2e();
  no ->
c:call:sensor(DataPtr),
c:call:airbag_iteration();
end,
eqc_c:value_of(airbag_active).

% Always send 170 as the second byte, otherwise it is very unlikely that an explosion will occur since a double fault would be required most of the time.
sensor_args(S) ->
  [byte(), 170].
sensor_pre(S, [Data]) ->
  not is_explode(Data).
sensor_post(S, [], Data, Res) ->
  Res = 0.

1. sensor(Data) ->
2. DataPtr = eqc_c_create_array(unsigned_char,Data),
3. case ?USE_E2E of
4.   yes ->
5.     c:call:sensor_e2e(DataPtr),
6.     c:call:airbag_iteration_e2e();
7.   no ->
8.     c:call:sensor(DataPtr),
9.     c:call:airbag_iteration();
end,
10. eqc_c:value_of(airbag_active).
11. length(S#state.faults) < 4 andalso
12. not lists:member(\{repetition, Num\}, S#state.faults).
13. repetition_next(S, [], [Num]) ->
14.   S#state\{faults = S#state.faults ++ \{repetition, Num\}\}.)

The explosion command is used to test that the airbag actually will explode when there are no faults. It looks as follows:

1. explosion(Data) ->
2. DataPtr = eqc_c_create_array(unsigned_char,Data),
3. c:call:faultcheck_packet_addFaultRepeat(\"airbag\", Num).
4. case ?USE_E2E of
5.   yes ->
6.     % Call the sensor function often enough for the E2E library
7.     % to recover (15 times)
8.     [ c:call:sensor_e2e(DataPtr) || _<--lists:seq(1,15) ],
9.     c:call:airbag_iteration_e2e();
10. no ->
11.     c:call:sensor(DataPtr),
12.     c:call:airbag_iteration();
end,
13. Res = eqc_c:value_of(airbag_active),
14. c:call:application_init(),
15. Res.
16. length(\{S#state.faults\}) < 4 andalso
17. not lists:member(\{bitflip, Byte, Bit\}, S#state.faults).
18. bitflip_pre(S, [Byte, Bit]) ->
19.   S#state\{faults = S#state.faults ++ \{bitflip, Byte, Bit\}\}.

The arguments are the byte to affect in the packet (0 to 1 without the E2E-library or 0 to 3 with the E2E-library) and which bit to flip in that byte (0 to 7). The precondition to run this command is that there is not already another bit flip on the same bit, as this would be meaningless because they take each other out. Another part of the precondition is that there are less than 4 faults active simultaneously.

The repetition command will make FaultCheck repeat a packet a certain number of times. It looks as follows:

1. repetition(Num) ->
2. c:call:faultcheck_packet_addFaultRepeat(\"airbag\", Num).
3. repetition_args(S) ->
4.   [ choose(1, 3) ].
5. repetition_pre(S, [Num]) ->
6.   length(S#state.faults) < 4 andalso
7.   not lists:member(\{repetition, Num\}, S#state.faults).
8. repetition_next(S, [], [Num]) ->
9.   S#state\{faults = S#state.faults ++ \{repetition, Num\}\}.)

The reason that the explosion command calls the sensor function several times is to give the E2E-library a chance to recover after many possible injected faults. The reason to use this command is to make sure that the E2E-library actually will pass data to the application when there is no fault present.

5.2 Experiment Results
First we tested the application without the E2E-library to see how it behaves. The following commands constitute a typical sequence that makes the airbag explode when it should not:
This sequence shows one command that flips bit number 1 in byte number 0. The next command will repeat whatever is sent once. The third command, sensor, will send [87, 170] to the airbag. 87 will be changed to 85 when the first bit is flipped and then [85, 170] will be sent twice because of the repetition.

Note that this short sequence is easy to understand and exactly points to the problem. The small sequence is obtained from a much longer sequence of calls that failed. QuickCheck automatically searches for smaller failing test cases when a failure is detected. Thus, all commands unnecessary for this unintended explosion to occur are removed by QuickCheck’s shrinking technique. As two consecutive commands are required for the airbag to explode, the repetition fault combined with the bit flip were necessary.

It took around 1000 auto-generated tests before this unintended explosion occurred, even when one out of 10 commands was an injected fault, so the mechanism in the application to require two consecutive commands was useful. Without that mechanism, when only one [85, 170] command for an explosion was required, it usually took less than 50 tests with the otherwise same set-up for the failure to occur.

When we run the same QuickCheck model against the Airbag implementation based upon the AUTOSAR E2E-library, the airbag never exploded unintentionally. Not even after running more than 100 000 tests.

Since the possible combinations of injected faults and state of the system is huge, one cannot draw much conclusion from a lot of passing test cases. Failing test cases reveal a problem, but until you find that, you cannot say much about the implementation. The distribution of the test data, collected during testing, is the only hint we have to see what we tested and whether we think this is a good test distribution.

While analysing the data, we realized that it is hard to jump from an arbitrary number to 85 by flipping a bit. We did reduce the search space by instead of sending two arbitrary integers, always send 170 as the second integer and choose the first integer in the set of values {84, 87, 117, 213, 21}. i.e., values that easily mutate to 85. The data generator to express this is written in QuickCheck as follows:

```haskell
1| sensor_args(SI) -> ->
2| [elements([84, 87, 117, 213, 21]), 170]]
```

Here only one bit differs an innocent command from a command that causes an explosion. This made the application fail after less than 50 tests most of the time without the AUTOSAR E2E-library (as opposed to 1000 tests). The test output typically looked like the following after shrinking:

```
airbag_eqc:bit_flip(0, 0) -> ok
airbag_eqc:repetition(1) -> ok
airbag_eqc:sensor([84, 170]) -> 0
airbag_eqc:sensor([84, 170]) -> 1
```

The first command flips bit number 0 in byte number 0. This will cause [84, 170] to be changed to [85, 170] and two such commands will make the airbag explode. In this case, two sensor commands with the same data were more likely to occur than a repetition command because of the limited amount of data for the sensor command to be chosen from.

However, with the E2E-library included, even 100 000 tests with the modified generator would not make the airbag explode when it should not, while the explosion command that sends [85, 170] still could make the airbag explode when the faults where disabled for several iterations.

In this experiment, the Protection Wrapper communicates directly with FaultCheck and is not the one provided by the real application that uses one of the communication interfaces provided by AUTOSAR. This does however not imply that the Protection Wrapper of the application to be tested has to be modified to support the FaultCheck interface. One might as well use an existing Protection Wrapper and mock the interface that AUTOSAR provides with an additional C component. This way, the same experiment can be carried out without intruding on the original Protection Wrapper of the application. The only reason that we connected the Protection Wrapper directly to FaultCheck in the example here is that we do not have any different protection wrapper to begin with; and therefore we could as well connect the one we create directly to FaultCheck.

6. CONCLUSIONS

We have presented a platform and methodology that uses FI and PBT to test safety-critical systems. Advantages include that faults can be injected while inputs are auto-generated based on properties and property-based checks on the software under investigation are performed.

An Airbag example based on the AUTOSAR E2E-library is presented where we run thousands of auto-generated tests. In this experiment, we have discovered faults that will cause unexpected behaviour with certain inputs when the E2E-library is disabled. We have also confirmed that enabling the E2E library will protect against these types of faults. This way, we have shown how non-functional requirements can be tested by using FI combined with PBT.

This methodology presents how to combine FI with PBT for evaluation of realistic use cases with one or more software components in order to exercise and evaluate fault handling mechanisms. This will indicate whether the evaluated fault handling mechanism is enough to cope with the expected faults or if something additional is needed.

Although the AUTOSAR example is from the automotive industry, which is one of the areas where fault injection is used today, the same techniques can be applied to other areas. Wherever it makes sense to test one or several parts of a system in a realistic use case, this platform can be used to evaluate fault handling capabilities.

7. ACKNOWLEDGEMENT

This research has been funded by the PROWESS EU project (Grant agreement no: 317820) [27]. We would like to thank Hans Svensson for useful feedback and comments.