Property-Based Testing with External Test-Case Generators

Bernhard K. Aichernig, Silvio Marcovic and Richard Schumi
Institute of Software Technology, Graz University of Technology, Austria
{aichernig,rschumi}@ist.tugraz.at
marcovic@student.tugraz.at

***Abstract*—Previous work has demonstrated that property-based testing (PBT) is a flexible random testing technique that facilitates the generation of complex form data. For example, it has been shown that PBT can be applied to web-service applications that require various inputs for web-forms. We want to exploit this data generation feature of PBT and combine it with an external test-case generator that can generate test cases via model-based mutation testing. PBT already supports the generation of test cases from stateful models, but it is limited, because it normally only considers the current state during exploration of the model. We want to give the tester more control on how to produce meaningful operation sequences for test cases. By integrating an external test-case generator into a PBT tool, we produce a smaller set of test cases that meet certain coverage criteria. This also reduces the test execution time. We demonstrate our approach with a simple example of an external generator for regular expressions and perform an industrial case study, where we integrate an existing model-based mutation testing generator.**

I. Introduction

Property-based testing (PBT) is a testing technique that tries to falsify a given property by generating random input data and verifying the expected behaviour [8]. It enables testing of algebraic equations as well as complex state machine models. For input generation it has a number of generators for common data-types and also for creating random walks on models. The major strength of PBT is its flexibility. New generators for complex data objects can be easily created by combining existing generators and also properties can be composed out of sub-properties.

In this work we show how a PBT tool can be extended in order to support other sources for the test-case generation instead of the default random walks on the model. This gives the tester more control on how to produce meaningful operation sequences for the test cases. For example, it can be applied to combine random testing with a mutation-based test-case generation method, which results in a powerful testing strategy [1]. By integrating an external test-case generator into a PBT tool we can combine the dynamic test-data generation feature of PBT with the ability to generate operation sequences, respectively test cases, according to various coverage criteria. Although we only show our approach for a specific external generator, it is also possible to reuse the structure of our integration for all kinds of testing tools. Especially model-based testing tools are particularly suitable, because they generate operation sequences based on models that can be combined with generated test data from a PBT tool.

We apply the PBT tool FsCheck in order to demonstrate our approach. Figure 1 shows its process. In the first step we translate XML business-rule files to input models for FsCheck, that are in the form of Extended Finite State Machines (EFSMs) [2]. As an alternative to directly using our EFSMs within FsCheck, we can also further transform the models in order to provide them to an external test-case generator. For example, we applied the model-based mutation testing tool MoMuT::UML [17] in our case study in order to generate abstract test cases based on mutation coverage. This allows us to generate smaller test suites that still cover important model parts. An externally generated abstract test case can serve as input for a state machine property within FsCheck, where it is executed instead of performing a random walk on the model. Additionally, we can enrich the operation sequences from the external generator with test data generated from FsCheck. The reason why we do not also apply the external generators for the test data generation is that they are often limited regarding the supported data types. Moreover, FsCheck is more suitable for this task, because it has powerful generators that can be combined very flexibly and thereby facilitate the generation of complex test data like attributes for web-forms. In order to execute externally generated abstract test cases within FsCheck, we extend its state machine specification with new functionality for external data sources.

A. Related Work and Contribution

PBT has gained a lot of attention over the last years and a variety of approaches combine model-based testing in combination with PBT. The most similar approaches to our work are described in the following.

Hughes et al. [13] presented an approach that utilizes QuickCheck to adapt random test-case generation in order to avoid rediscovering the same type of bugs. In order to avoid the generation of the same sequences multiple times, the minimum counterexamples of already found bugs were stored. The approach is similar in a way that it uses feedback from the test sequences in order to optimize the test-case generation process. The main difference to our approach is their focus on a specific technique to adapt the test-case generation.

A technique to generate test sequences that cover business rules was presented by Jensen et al. [14]. Business rules are

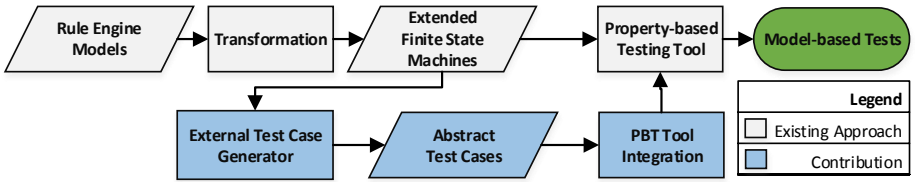

Fig. 1. Overview of the steps for the integration of an external test-case generator.

translated into logical formulae and a constraint solver is used to generate the test sequences. Their work is similar in a way that test sequences are generated from business rules with the help of a constraint solver. Our external test-case generator MoMuT::UML also applies a constraint solver for test-case generation. While their work can be interpreted as a variant of model-based testing, they present their work in a business rule language. Our technique focuses more on translating the business rules into a model and providing more options for the test sequence generation process.

The work of Vedder, Vinter, and Jonsson [23] combines QuickCheck with a fault injection tool. The created testing platform is used to run a quad-copter simulator to improve the collision-avoidance mechanism. They inject faults into the simulator and verify the property that copters do not collide. Similar to our work, they inject faults in order to acquire test sequences. In their work the model is not created automatically and they focus more on how to use the knowledge of found bugs in order to improve the collision-avoidance mechanism. In comparison, this paper focuses more on automating the approach and acquiring test sequences in a different manner, rather than improving the SUT based on the injected faults.

An alternative to control the generation of operation sequences is to specify the frequency distribution for generators, as, e.g., supported by Quviq QuickCheck [3]. Weights can be assigned to generators in order to select certain operations with a higher or lower probability. This method allows more control over the distributions of operation sequences. However, with our external generators we have even more control, e.g., we can generate test cases that meet a certain coverage criterion.

A framework to test web-services with the PBT tool PropEr for Erlang was presented by Lampropoulos and Sagonas [18]. They automatically read the WSDL specification of a web service to invoke the operations of the web service with random input. Similar to this work they used data types, but only supported a few constraints for the data. They implemented automated properties that are not satisfied in case the parsing of the SOAP response encounters an error. In comparison to our work they do not use state machines to build their models and other properties have to be implemented by the user.

Francisco et al. [11] presented another similar approach. The framework tests web services by automatically deriving QuickCheck models from the web-service's WSDL description and OCL semantic constraints. They show how to test stateless and stateful web services by deriving respective models. In comparison to this paper their generators do not consider constraints for the data and they added the OCL semantic constraints manually. They applied QuickCheck to generate the test sequences but unlike in this paper, they did not use other test sequence generation strategies.

To the best of our knowledge, we could not find any other work that derives PBT models to automatically test a system and decouples the test sequence generation process from the PBT tool giving more control over the generation process.

In our previous work we showed how PBT can be applied in a test-case generation process that uses XML business-rule models in the form of EFSMs as input for PBT [2].

Building upon our previous work, we present the following novel contributions. We demonstrate how an external test-case generator can be integrated within a PBT tool by extending state machine specifications so that they can execute operation sequences from external sources together with generated test-data. Another contribution is the comparison of the random testing approach of a PBT tool with model-based mutation testing. Furthermore, we present an industrial case study of a web-service application in order to evaluate our approach.

B. Structure

The rest of the paper is structured as follows. First, Section II explains the basics of PBT, FsCheck and the external test-case generation tool that we applied in our case study. Next, in Section III we describe details about the structure and implementation of our approach. Then, in Section IV we show a small example of external sequence generation with regular expressions. Section V presents an industrial case study. Finally, we draw our conclusions in Section VI.

II. Background

A. Property-based Testing

Property-based testing (PBT) is a random test-case generation technique that tries to evaluate a given property. In its original form, the test-case generation is driven by algebraic properties that specify the expected behaviour of functions under test. The test-case generator produces a high number of random input values for each function parameter and checks if the properties are satisfied. A simple example of an algebraic property is that the reverse of the reverse of a list must equal the original list:

$$\forall xs \in Lists[T] : reverse(reverse(xs)) = xs$$

A PBT tool will generate a series of random lists xs, execute the reverse function and evaluate the property. If a property fails, the responsible test case is returned as a counterexample. In order to facilitate debugging PBT searches for a simpler counterexample. This process is known as shrinking. The search for simpler counterexamples can be specified individually for different data types [2], [12], [20], [21].

Today, most PBT tools also support model-based testing with models in the form of extended finite state machines (EFSMs) [16]. An EFSM can formally be described as a 6-tuple (S, s_0, V, I, O, T): S is a finite set of states, $s_0 \in S$ is an initial state, V is a finite set of variables, I is a finite set of inputs, O is a finite set of outputs, T is a finite set of transitions, $t \in T$ can be described as a 5-tuple (s_s, i, g, op, s_t), s_s is the source state, i is an input, g is a guard, op is a sequence of output and assignment operations, s_t is the target state [16].

In order to use such an EFSM for PBT the permitted transition sequences have to be defined with preconditions and also the effect of each transition has to be defined via post-conditions. Preconditions, postconditions and the execution semantics of transition are encapsulated in so-called commands $Cmds$, also called operations. The property of an EFSM is that for each permitted path on the model, the postcondition of each transition respectively command of the path must hold. In order to verify this property a PBT tool produces random transition sequences and checks the postconditions after each transition. Formally a property for an EFSM can be defined as follows.

$$\forall s \in S, i \in I, cmd \in Cmds :$$
$$cmd.pre(i, s) \implies cmd.post(cmd.runActual(i, s), cmd.runModel(i, s))$$
$$cmd.runActual : S \times I \to S \times O$$
$$cmd.runModel : S \times I \to S \times O$$

The Boolean function $cmd.pre$ is the precondition. It defines the valid inputs and states of a command. The postcondition $cmd.post$ relates the new states and the outputs of the SUT and the model after the execution of the command on both the SUT $cmd.runActual$ and the model $cmd.runModel$. Note that sometimes the postcondition and the $runActual$ function can be fused together. For example, in this work we used an experimental version of a PBT tool, where this was the case. Also commands were called operations in this version. Hence, we use this term in the following.

PBT constitutes a flexible and scalable model-based testing technique, because it is random testing. It has been shown that it generates a large number of tests in reasonable time [24].

The first PBT tool was QuickCheck [8] for Haskell. There are many other tools that are based on QuickCheck, e.g., ScalaCheck [19] or Hypothesis[1] for Python. In our approach we work with FsCheck, because the testing framework of our industrial partner is based on C#.

B. FsCheck

FsCheck is a PBT tool for .NET based on QuickCheck and influenced by ScalaCheck. Like ScalaCheck it extends the basic QuickCheck functionality with support for state-based models. A small limitation of the current stable version is that it does not consider preconditions when shrinking command sequences. This feature is included in an experimental release.[2] Therefore, we worked with this experimental release in this paper. With FsCheck, properties can be defined both in a functional programming style with F# and object-oriented with C#. Similar to QuickCheck it has default generators for basic data types and more complex ones can be defined via composition. It has an Arbitrary instance that groups together a shrinker and a generator for a custom data type. This enables the specification of properties that include variables of custom data types. New Arbitrary instances can be dynamically registered at run time and then the new data type can be directly used for input data generation [2].

C. MoMuT

MoMuT (MOdel-based MUtation Testing) is a tool family based on mutation testing. Instead of programs, abstract models of the SUT are mutated. Mutations are performed via mutation operators which inject small faults into abstract models. The objective of MoMuT is to generate a small number of strong test cases that cover these faults. MoMuT::UML [17] is a mutation testing tool which uses UML state machines as input. The models are transformed into object oriented action systems (OOAS) and then converted into labelled transition systems (LTS). The test case generator tries to find differences between the original and mutated model by performing a refinement check and if non-refinement is reached, then a check for input output conformance ($ioco$) is executed [15]. If non-conformance is detected, a test case is generated that shows the difference and we can denote the mutant as killed. A trace showing non-conformance represents such a test case. Model-based mutation testing is a computationally expensive strategy. Numerous mutants have to be analysed and generating test cases involves a conformance check between two models [1].

For the test case generation, two different back ends can be used. The "enumerative back end", developed in the MOGENTES[3] project and the "symbolic back end", developed within the TRUFAL[4] project. The latter showed better performance for our models. Therefore, the symbolic back end was used. Both back ends support the following three test case generation strategies: random testing with random walks, mutation testing and a combination of both. The symbolic back end uses Microsoft's SMT Solver Z3 [9] and is written in SICStus Prolog [7]. MoMuT::UML produces abstract test cases in the Aldebaran format, which is a file format for representing labelled transition systems [10].

In this paper, business rule models are translated to OOAS models instead of UML models. Therefore, the UML mutator and the translation step from UML to OOAS of the tool are not used. Action systems (AS) were first introduced by Back and Kurki-Suonio [4] and used as a modelling formalism for distributed systems. AS start at an initial state. The state is changed by executing actions on the system. One action is chosen at each step in a non-deterministic manner. Actions

[1]https://pypi.python.org/pypi/hypothesis

[2]https://fscheck.github.io/FsCheck/StatefulTestingNew.html

[3]http://www.mogentes.eu

[4]https://trufal.wordpress.com

can only be chosen if they are enabled. They are enabled if their guard is satisfied in the current state. If there is no action in the set of enabled ones, the execution terminates. The object oriented action systems (OOAS) language is an object-oriented extension which is based on the work of Bonsangue, Kok, and Sere [6]. Details of the concrete syntax of these AS are omitted due to space limitations. For a more detailed syntax definition of the OOAS language we refer to the work of Tiran [22].

III. Architecture and Implementation

In this section we present the architecture of our integration of an external test-case generator into FsCheck. The idea behind the presented approach is to use an external generator to create a sequence of operations to form a test case. A PBT tool like FsCheck uses a function $Next$ to generate the next operation. This function focuses on one operation at a time and it only relies on the current state of the model. However, there is other data that can be utilized to generate a sequence of operations, e.g., information about covered model parts. Before executing any operation on the model, we can generate sequences from an external source. This allows us to generate operations that do not only rely on the current state. It is possible to include the path up to the operation and even after the operation itself, which enables a better control of the generation of operation sequences since this process is decoupled from the PBT tool. For example, when the test cost of execution is high, then an external generator can generate a smaller test suite that still checks all relevant model parts, e.g., by ensuring transition coverage.

The simplest use of an external generator could be to use a sequence from a PBT tool and save it externally. This sequence can then be reused as an external source and supplied to the PBT tool via the interface. This means we are effectively replaying the previously run test case. This use case was discovered as a by-product of this work since it was very useful to try to reproduce errors or to create very specific test cases. FsCheck already supplies some sort of replay functionality. It is possible to set the seed of the random generator and make the test sequence generation deterministic. That way it is possible to replay a certain test case. However, as soon as the model changes or multiple test cases need to be replayed, or just some parts of the sequence should be replayed, this approach does not work any more. That is were the replay functionality via an external generator showed its usefulness. It proved to be effective for testing if any changes on the test framework were implemented to show if it still identifies the SUT correctly. Short test cases were written in order to target mentioned changes. This usually resulted in finding mistakes earlier as compared to waiting until the generators produce a sequence that operates in the area that was changed.

In order to use an external sequence in FsCheck, an interface to the tool had to be built. Note that here, interface does not refer to the programming structure used in object-oriented languages. Here, interface refers to a shared boundary between two components, namely the PBT tool and the external generator, to enable an information exchange between both components. The PBT tool can pass model information to the external generator and in return it receives generated test cases. A test case consists of a sequence of operations. An operation will need access to a model and an SUT. It is executed on the SUT with the state prior to the operation. The resulting SUT state is then compared to the expected post state from the model. This is the property of the operation. Data is needed in order to perform an operation. This data is supplied by using generators for the required data types to generate values. In order to not rely solely on the generators, data will also be included from external generators. That means that external generators have to supply a set of data for each operation.

In traditional PBT, the length of a test case is decided by the PBT tool. Then, based on the current state an operation is generated by choosing the type of operation and the operation data (attributes) with generators. In our integration approach the external generator is responsible for the length of the sequence. That means that in order to use our approach it must be possible to control the length of an operation sequence. The operation type is also defined by the external generator. The data needed for an operation can be supplied by either the external generator, by a regular PBT generator or by a combination of both. Some external generator might not be suitable to generate complex operation data, which is needed for the execution of the operation. In this case it is useful that PBT generators can be used for the operation data and the external generator only needs to generate the operation types of a test case. In our case study in Section V this was the case.

The interface for the external generator was implemented by extending the abstract class $Machine$ of FsCheck with new functionality. We derived from the $Machine$ class in order to create an $ExternalMachine$ class, which implements the new functionality. Figure 2 shows the new components and how they are wired together with the regular FsCheck parts. In FsCheck the $Machine$ uses the model and the SUT and executes the operations on them. FsCheck uses a $Runner$, which is responsible for checking the machine and reporting the test results. The machine has to be transformed into a property to be checked. This functionality is already provided by FsCheck and most parts are hidden from the user. The $ExternalMachine$ is responsible for transforming the external test arguments into operations and forwarding the SUT and model to the machine. An external generator has to be linked to the $ExternalMachine$ to provide the test case arguments. There are two types of arguments to create test sequences, $SetupArg$ and $OperationArg$. They will be explained later in this section. Usually the external generator will also require information from the SUT or the model. This is not shown in the diagram since it is not necessarily required. Note that the $ExternalMachine$ is abstract, this means the user will have to implement how the arguments are used to generate test sequences. In the following the implementation of the $ExternalMachine$ and how the arguments can be used is explained. The implementation is shown in Listing 1.

The first notable extension are the two new generic types: $SetupArg$ and $OperationArg$ (Line 1). They define the ar-

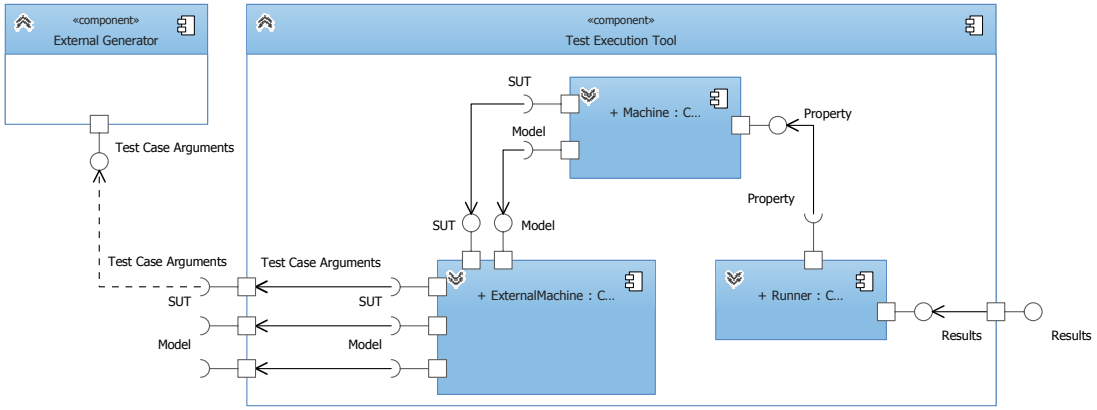

Fig. 2. Component diagram showing how the FsCheck test case generation process is extended to support external generators.

gument types that are used in the *Setup* and *Next* functions to generate the initial state and the operations (Lines 14 & 21). A queue (Line 2) holds the external data that is used to generate its operation sequences. The data in the queues is supplied by external generators. Each element in this queue is an object of the *MachineRunArguments* class and represents a single test case. A *MachineRunArguments* object contains a setup argument *SetupArg* and a queue *OperationArg*, which contains the operation sequence. Simplified, the interface uses a queue of queues, where the outer queue represents the set of externally generated test cases and the inner queue represents the sequence of operations of each test case.

During a regular FsCheck test-case generation/run, setup is called in the beginning to create the initial state of the model and then a number of operations is generated by calling *Next* each time. The *ExternalMachine* uses *MachineRunArguments* to supply external information to the *Setup* and *Next* functions. These arguments can be used to supply the whole data required for an operation or just some information for generation process of operations.

In Line 7–13 the *Setup* function is implemented. This function is sealed in order to enforce the use of the new *SetupArbitrary* function in Line 14, which uses a setup argument. The *Setup* simply dequeues a test case and assigns it as the current test. The new *SetupArbitrary* function is then called with the setup argument. The user now has to implement the *SetupArbitrary* instead of the *Setup* function. In Line 15–20 the *Next* function is shown. Similar to *Setup*, this function dequeues an argument which is used in our introduced *Next* function (Line 21). The original *Next* function is sealed as well to enforce the use of our new function, which has to be supplied by the user in a derived class. In Line 17 a stop operation is generated if the test case is finished. The stop operation is a functionality that was implemented during our work into FsCheck in order to enable a break-off during the generation of an operation sequence. It is now part of the FsCheck library. When a stop operation is returned by the *Next* function, FsCheck identifies this as the end of the current test case and stops generating new operations. This can be useful if a model is in a final state or as in our case if the test case needs to be terminated manually.

IV. Example

The example of a bank account will illustrate how model-based testing (MBT) can be performed with FsCheck. Furthermore, we will show how a simple generator based on regular expressions (regex) can be integrated as an external generator for the generation of operation sequences. Listing 2 shows

```
1  public abstract class ExternalMachine<Actual, Model, SetupArg, OperationArg> : Machine<Actual, Model> {
2    public Queue<MachineRunArguments<SetupArg, OperationArg>> TestCases { get; protected set; }
3    MachineRunArguments<SetupArg, OperationArg> currentTest;
4    public ExternalMachine(Queue<MachineRunArguments<SetupArg, OperationArg>> testcases): base(int.MaxValue) {
5      TestCases = testcases;
6    }
7    public sealed override Arbitrary<Setup<Actual, Model>> Setup {
8      get {
9        if (TestCases.Count != 0 && (currentTest == null || currentTest.Count == 0))
10         currentTest = TestCases.Dequeue();
11       return SetupArbitrary(currentTest.SetupArgument);
12     }
13   }
14   public abstract Arbitrary<Setup<Actual, Model>> SetupArbitrary(SetupArg arg);
15   public sealed override Gen<Operation<Actual, Model>> Next(Model m) {
16     if (currentTest.Count == 0)
17       return Gen.Constant((Operation<Actual, Model>) new StopOperation<Actual, Model>());
18     var arg = currentTest.Dequeue();
19     return Next(m, arg);
20   }
21   public abstract Gen<Operation<Actual, Model>> Next(Model m, OperationArg arg);  }
```

Listing 1. External machine which represents the interface for external generators.

```
1  public class BankAccount {
2    public int Money { get; private set; }
3    public BankAccount(int initMoney){Money = initMoney;}
4    public void Deposit(int inc) { Money += inc; }
5    public void Withdraw(int dec) { Money -= dec; }
6    public override string ToString()
7    { return String.Format("BankAccount = {0}", Money); }
8  }
```

Listing 2. Simple bank account implementation as system under test.

the SUT of this example. In a real world example we have to imagine complex operations behind the logic of a bank account. However, this example attempts to be minimal.

Listing 3 shows an FsCheck interface implementation for MBT. FsCheck uses the interface *Machine* which has the SUT and the Model as template parameters (Line 1). We model the *BankAccount* as an integer number representing the money on the account. Moreover, the *Machine* interface contains method signatures for *Setup* and *Next*. The function *Setup* returns an *Arbitrary* of the type *Setup*. A *Setup* is a class that contains two functions. One that returns the initial state of the SUT (*Actual*) and another one that returns the initial state of the model. The *Setup*, as well as the *Arbitrary*, are shown in Listing 4. The function *Next* uses a generator to generate an operation that will be performed on the SUT and the model.

The last parts we need are the operations. Listing 5 shows a *DepositOperation*. We want our bank account balance to remain between 0 and 100. We ensure that the bank account is not tested outside these boundaries via preconditions. The *Run* function adds the amount that we want to deposit to our model. It is executed during the generation of a test case and the model state is stored for a comparison with the SUT. After the generation phase, *Check* exercises the SUT and compares the stored model state with the value on the bank account. This represents our postcondition and is returned as a property. The *WithdrawOperation* is implemented in a similar manner. An *Arbitrary* class that includes a shrinker can be registered to shrink such operations, e.g., by minimizing their attributes.

In order to execute our specification we transform our machine into a property with the provided FsCheck functionality.

```
new BankAccountMachine().ToProperty().QuickCheck();
```

By converting the machine into a property, we can apply the *QuickCheck* method, which produces 100 test cases with increasing length. If we execute the above line, we will get output for each test case. Test-case outputs will have the form as shown in the following lines.

```
(92, Setup BankAccount)
Deposit:6 -> 98
Withdraw:2 -> 96
Withdraw:3 -> 93
Withdraw:1 -> 92
Withdraw:5 -> 87
```

In the first line we see the initial state of our model, which is 92. The following lines contain the output of the *ToString* format of the operation and the model value after execution of the operation. In this example the first operation deposits an amount of six. With the initial balance of 92 the balance should be 98 after execution of the operation. Note how defined limits are adhered to. Almost no *Deposit* operations were executed since the balance is close to the upper limit of 100.

To show how an external generator can be included into a PBT tool, a sequence generator based on regex was created and integrated into FsCheck. The generator shall serve as an example to demonstrate our approach. It uses a regex to create a sequence of identifiers encoded as a string. These identifiers are then used to create operation types of the sequence. For example, by using the regex $(Operation1)^{+}.(Operation2)^{*}$ the generator will create test cases which always start with at least one operation of the first type and then may add operations of the second type.

In Listing 6, an implementation based on the *ExternalMachine* for the bank account is presented. The *SetupArgument* is of type integer and is ignored. The initial state of the bank account is generated using the PBT generators. The *OperationArgument* is of type string and contains the name of the operation to be created. In Lines 6–11 the *Next* function is shown. It implements the corresponding abstract function of Listing 1. In the previous case (Listing 3), one of the two generators for the operations was chosen randomly with the FsCheck function *Gen.OneOf*. In this case the type of operation is decided by the name of the operation (*opName*) which was generated by the external regex-generator. The value that is withdrawn or deposited is generated as previously, using FsCheck generators. In order to test the property of an external machine the maximum number of test cases has to be set in the configuration and a queue of *MachineRunArguments*, which contains the sequence of operation names, has to be passed to the constructor.

The property can then be checked with the following lines of code where *tests* is the queue of *MachineRunArguments* which represent the test cases.

```
Configuration config = Configuration.VerboseThrowOnFailure;
config.MaxNbOfTest = tests.Count;
new BankRegexBasedMachine(tests).ToProperty().Check(config);
```

This queue is generated by a generator function which uses a regex to generate test cases. Fare,[5] a .NET port for Google's Java library Xeger[6] was used to generate strings from the

[5]https://github.com/moodmosaic/Fare

[6]https://code.google.com/archive/p/xeger

```
1  public class BankAccountMachine : Machine<BankAccount, int> {
2    public override Arbitrary<Setup<BankAccount, int>> Setup { get { return new BankAccountSetupArb(); } }
3    public override Gen<Operation<BankAccount, int>> Next(int value){
4      var incGen = Gen.Choose(1, 10).Select(i => (Operation<BankAccount, int>) new DepositOperation(i));
5      var decGen = Gen.Choose(1, 10).Select(i => (Operation<BankAccount, int> )new WithdrawOperation(i));
6      return Gen.OneOf(incGen, decGen);
7  } }
```

Listing 3. Bank account machine interface implemantation.

```
public class BankAccountSetup : Setup<BankAccount, int> {
  public int Initial { get; }
  public BankAccountSetup(int inital) { Initial = inital; }
  public override BankAccount Actual() { return new BankAccount(Initial); }
  public override int Model() { return Initial; }
}
public class BankAccountSetupArb : Arbitrary<Setup<BankAccount, int>> {
  public override Gen<Setup<BankAccount, int>> Generator {
    get { return Gen.Choose(0, 100).Select(i => (Setup<BankAccount, int>)new BankAccountSetup(i)); }
  }
  public override IEnumerable<Setup<BankAccount, int>> Shrinker(Setup<BankAccount, int> _arg1) {
    foreach (var i in Arb.Default.Int32().Shrinker(((BankAccountSetup)_arg1).Initial)) {
      yield return new BankAccountSetup(i);
} } }
```

Listing 4. Bank account setup and arbitrary.

```
public class DepositOperation:Operation<BankAccount,int>
{
  public int Amount { get; private set; }
  public DepositOperation(int amount){Amount = amount;}
  public override bool Pre(int m) {
    if (m + Amount > 100)
      return false;
    return true;
  }
  public override int Run(int m) { return m + Amount; }
  public override Property Check(BankAccount a, int m){
    a.Deposit(Amount);
    return (a.Money == m).ToProperty();
  }
  public override string ToString() {
    return String.Format("Deposit:{0}", Amount);
} }
```

Listing 5. Implementation of the deposit operation.

supplied regex. A string based on a given regex can be easily generated with the following lines.

```
var gen = new Xeger(regex);
string sequence = gen.Generate();
```

The generated string is a sequence of operation types which represents a test case. It has to be parsed and stored in a queue in order to use it within our $ExternalMachine$.

A test case executed with the regular FsCheck $Machine$ interface will create sequences which will use both operations approximately as often. Of course, the likelihood of the operations can be changed to favour one operation over the other. However, the flexibility is limited. With a regex we can focus better on certain aspects of the system by adjusting the regex. A specification that focuses heavily on withdrawing can utilize the following regex $Withdraw.(Withdraw)^{+}.(Withdraw|Deposit)^{+}$. In a specification as simple as a bank account, modelling the sequences based on regex might not be very useful since the random generator of a PBT tool will most likely cover enough useful scenarios. With more complex systems this approach can be used to target certain critical scenarios that are more likely to produce errors. It can also help to target rare operations that can only be triggered in certain corner cases.

V. Industrial Case Study

The approach was developed for testing a web-service application provided by our industrial project partner, AVL.[7] The tested system is called Testfactory Management Suite (TFMS) and is used in the automotive industry.[8] It enables the management of data, resources, activities, work flows and information of test fields. The test field may test, e.g., car engines or power trains. Various activities are supported, like test preparation, planning, execution, data management and analysis. The application comprises a number of modules, each providing different functionalities, like managing test orders.

As explained in Section II-C, we apply the tool MoMuT and we compare its mutation-based test case generation method with the random testing of FsCheck. Our aim for the integration of MoMuT is to minimize the test suite and still have certain guaranteed coverages on the model. This reduces the test execution time without keeping important model aspects untested. The process of our PBT integration with this tool is shown in Figure 3. The first step is to parse Rule Engine Models (REMs) to input models for FsCheck that are in the form of EFSMs. Then, we further transform these EFSMs for MoMuT. In order to run this tool, we need a model in the form of an object-oriented action system. Furthermore, we add observer automata to our models in order to specify the model components that should be covered. An observer automata is

[7]https://www.avl.com

[8]https://www.avl.com/-/avl-testfactory-management-suite-tfms

```
public class BankRegexBasedMachine : ExternalMachine<BankAccount, int, int, string> {
  public BankRegexBasedMachine(Queue<MachineRunArguments<int, string>> tcs) : base(tcs) { }
  public override Arbitrary<Setup<BankAccount,int>> SetupArbitrary(int ignored){
    return Arb.From(new BankAccountSetupArb());
  }
  public override Gen<Operation<BankAccount, int>> Next(int m,string opName){
    if (opName.Equals("Deposit"))
      return Arb.Default.PositiveInt().Generator.Select(i => (Operation<BankAccount,int>)new DepositOperation(i.Get));
    else if (opName.Equals("Withdraw"))
      return Arb.Default.PositiveInt().Generator.Select(i => (Operation<BankAccount,int>)new WithdrawOperation(i.Get));
    else throw new NotImplementedException("Operation name " + opName + " is not implemented.");
} }
```

Listing 6. Regex-based external machine for testing a bank account.

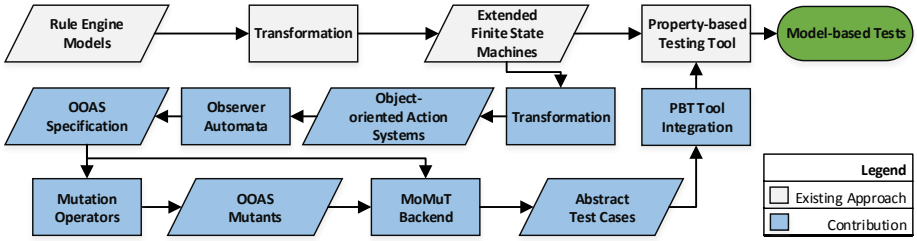

Fig. 3. Overview of the steps for the integration of MoMuT as an external test-case generator.

a state machine that can monitor and accept traces or sets of traces, i.e. test suites of a given EFSM. Coverage criteria usually consist of a list of model aspects that needs to be covered. The observer monitors the traces that are produced during the execution of the EFSM. It reports acceptance when all required aspects of the EFSM are observed [5]. Combining observer automata with mutation testing allows us to generate test cases that can find a difference in the coverage of a model and a mutant. We declare the accepting state of the observer as observable output. When non-conformance is found, then we know that the coverage is different, because either the observer of the model or of the mutant are in the accepting state, but not both. For generating mutations of the model we need to specify a mutation operator that injects faults into the model. MoMuT performs a conformance check between the model and a mutated version in order to produce an abstract test case in the case that there is a difference. This abstract test case represents a command sequence and is fed into FsCheck. The commands are then completed with test data that is generated within FsCheck.

The case study for this paper was applied to one module of the TFMS, which is called the Test Equipment Manager (TEM). The main function of the TEM module is the administration of test equipment and it comprises two REMs, one for test equipment (TE) and one for test equipment types (TET). Equipment is grouped into base equipment types such as dynamometers, sensors, test beds, measurement devices, input/output modules and many others. These pieces of equipment are created, configured, edited, calibrated and maintained in this module. A state machine of an example test equipment, a dynamometer, is shown in Figure 4. We skip the required test data associated to transitions. It can be seen that the model has a number of tasks to manage/edit the test equipment and also that these tasks lead to different states. In our case a task represents the submission of a web form. It can be possible in various states and also lead to different next states. Note that the models for this case study were more complex than the figure might suggest, because we had to consider several instances of the equipment. Hence, we also added operations to switch between instances of this REM. The model of a TET is similar to this state machine and is therefore omitted.

Table I shows how many states, tasks, transitions and attributes were tested of the two REMs. The model was only partly tested since some transitions and states were not fully supported or not implemented, because they were special cases. The numbers in the parentheses represent the total numbers including these untested items. To evaluate the approach of implementing external test case generators, we compared the test cases generated with MoMuT to the regular FsCheck generation approach. The experiments investigate how adequate the two techniques cover the model.

We first discuss the experiments with MoMuT. In order to generate test cases, certain parameters have to be set. The number of instances per REM was fixed to five. Although these instances were dynamically created with $Duplicate$ or $Create$ tasks, the state space needs to be built initially, because MoMuT does not support dynamic data structures. We set the number of mutants that are created to ten. It could be observed that most mutant operators resulted in similar abstract test cases. Therefore, we only considered one operator that changes the destination state of a transition. We used different depths for the $ioco$ check and a depth of 20 for the refinement check. If no error is found within the maximum depths, no test case will be generated. It was observed that the runtime of the test generation process is at least exponential to the $ioco$ depth. The depth was increased step by step until the process took too long or the depth of 10 was reached.

Table II shows the exploration times and the number of test cases tc that were generated for different models and observer strategies (automata). We used an observer that requires that all states have to be visited at least once and also one for all tasks. Both count a state or tasks as visited if it was observed at

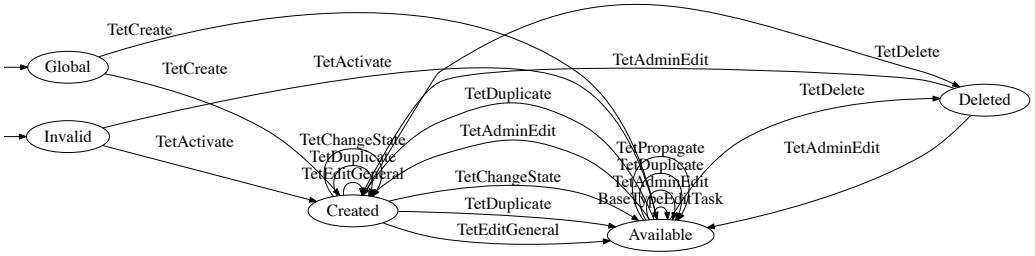

Fig. 4. Test Equipment Manager model of a test equipment. Global and Invalid are the initial states. Tasks of the Global state are always possible. Invalid is a special state for equipment that is created via a copy operation and it has to be adapted to be valid again.

TABLE I
NUMBER OF STATES, TASKS AND TRANSITIONS IN REMS OF THE TEM.

Model	States	Tasks	Transitions	Attributes
TestEquipmentType	4 (5)	8 (10)	16 (20)	35 (43)
TestEquipment	5 (7)	7 (13)	13 (38)	18 (23)

any instance. The results were obtained with a Lenovo T450s notebook running Windows 8.1 with a 2.2 GHz Intel i5 with 4 cores and 8 GB RAM. d_n stands for the $ioco$ depth n, e.g., d_{10} shows the time it took for the exploration for a $ioco$ depth of 10. If the computation took too long for a depth, an estimation is given and the time value is marked with an asterisk (*). The estimation is calculated using exponential regression. We can observe that the task strategies take longer than the state strategies. A deeper depth is needed to find mutants, since it needs longer test sequences to cover all tasks. For the TET model no test case was found for the task strategy, because a deeper $ioco$ depth would be required.

The model coverage is analysed as follows. The abstract test cases from the MoMuT generation process are used as a reference. For a comparison of the coverage items the amount and length of test cases is controlled. The FsCheck test run is set to the number and length of the MoMuT test cases. The following figures show the coverage of the different strategies and models. For a strategy a test suite is generated composed of x test cases of a fixed length. The length is four for the two state coverage observers and eight for the task coverage observer. The number of test cases per suite is plotted on the x-axis. On the y-axis the coverage of the model is plotted in percent. Each graph contains two data series, one for the test cases generated with MoMuT and one for FsCheck.

The analysed coverage criteria are state, task and transition coverage. The experiment is repeated 50 times for the FsCheck sequences and the average value is plotted. Since the MoMuT generation process is costly, it is only performed once. Repeating the experiment multiple times would help approximate the expected coverage values of the MoMuT sequences.

In Figure 5 and Figure 6 it can be seen that the state coverage is 100% with one test case for the MoMuT approach. This is expected and a validation that the state observer works as intended. The transition coverage of MoMuT and FsCheck is similar, because MoMuT does not always find new test sequences for all mutants. Multiple test cases with the same sub-sequences are produced, which deteriorates the transition coverage. However, MoMuT has an option to reuse existing test cases, which could resolve this issue. Figure 7 shows that state as well as task coverage is 100% for the MoMuT approach. In our models no unreachable states are included. This means if all tasks are covered all states are covered as well. This is not evident in all models. In all strategies it can be observed that MoMuT covers the model better for smaller test suites. For larger test suites both methods are able to cover most of the model. The MoMuT sequences are also guaranteed to cover a certain criteria with only one test case based on the observer used. Since FsCheck relies on random testing no matter how many test cases are generated, it cannot be guaranteed that full coverage is achieved with a test suite.

TABLE II
EXPLORATION TIMES OF TEM REMS USING MOMUT WITH OBSERVERS.

Scope	Strategy	d_2		d_4		d_6		d_8		d_{10}	
		time	tc	time	tc	time	tc	time	tc	time	tc
TET	States	4.34s	0	38.10s	9	1.65m	8	11.54m	9	53.51m	9
TET	Tasks	6.41s	0	2.58m	0	31.10m	0	8.56h*	-	140.63h*	-
TE	States	3.60s	0	25.76s	4	50.47s	10	5.02m	9	1.56m	10
TE	Tasks	9.01s	0	3.06m	0	54.06m	2	13.45h*	-	213.63h*	-

VI. CONCLUSION

We have presented a testing technique that integrates an external test-case generator into a PBT tool in order to combine the features of two test-case generation strategies. We discussed the architecture of our implementation and illustrated the integration method with a bank account example and an external generator based on regular expressions.

A web-service application from AVL called testfactory management suite was tested for the evaluation of our approach. The model-based mutation testing tool MoMuT was our external generator for this evaluation. The experiments conducted in this case study have shown that it can be useful to integrate an external test case generator into a PBT tool. The most notable differences between the test generation with MoMuT and the random approach with FsCheck is the difference in computation times. The test suite generation time for MoMuT is very high. It can take several minutes to hours to generate useful sequences, hence it becomes infeasible for bigger models. This is a known limitation of the approach. FsCheck can generate a large number of test cases within one second. Since the MoMuT approach uses observer automata we can guarantee that a single test case is able to fulfil a coverage criterion like state coverage. The test suits generated with MoMuT covers more of the model with fewer test cases as compared to plain FsCheck. This advantage becomes negligible as soon as the amount and length of test cases is increased. It is evident that a smaller test suite will need less execution time. Therefore, it makes sense to optimize for a small test suite if the test-execution time is expensive. The short MoMuT sequences cover most parts of the model and are therefore well suited for regression testing.

In the future we plan to evaluate a combined approach of mutation testing and random testing with FsCheck. It has been shown that combination of these two test strategies provides benefits for killing mutants [1]. An evaluation with further external generators is also an option, as our integration is applicable to all kinds of test case generators that can produce more meaningful test cases than random testing.

ACKNOWLEDGEMENT

This research was funded by the Austrian Research Promotion Agency (FFG), project number 845582, Trust via cost function driven model based test case generation for non-functional properties of systems of systems (TRUCONF). The authors are grateful to Martin Tappler and the anonymous reviewers for their valuable comments. Finally, we thank Kurt Schelfthout for providing FsCheck and supporting the necessary adaptations to the tool.

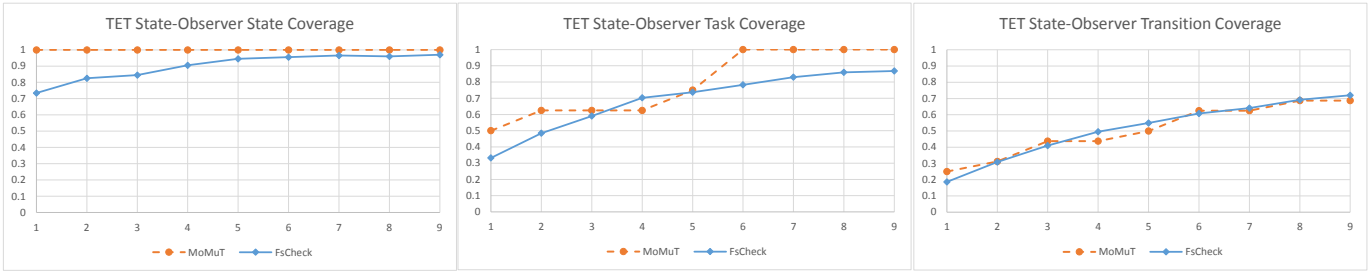

Fig. 5. TET state-observer coverage per number of test cases.

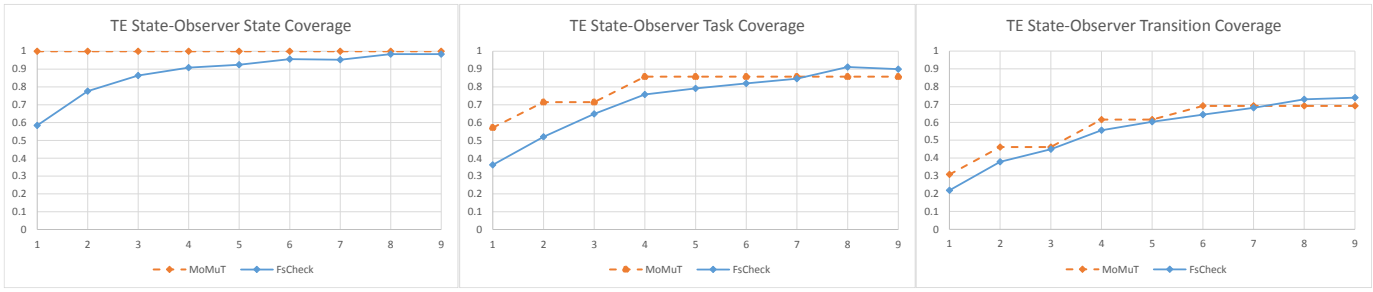

Fig. 6. TE state-observer coverage per number of test cases.

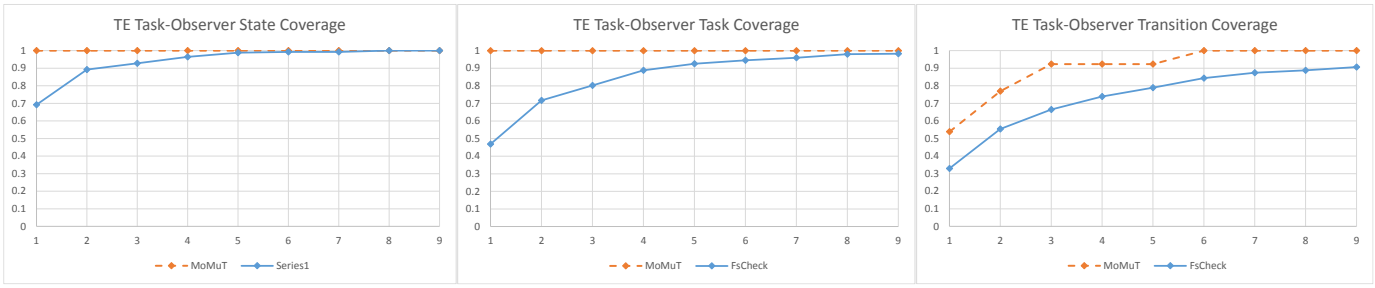

Fig. 7. TE task-observer coverage per number of test cases.

References

[1] B. K. Aichernig, H. Brandl, E. Jöbstl, W. Krenn, R. Schlick, and S. Tiran, "Killing strategies for model-based mutation testing," *Softw. Test., Verif. Reliab.*, vol. 25, no. 8, pp. 716–748, 2015.

[2] B. K. Aichernig and R. Schumi, "Property-based testing with FsCheck by deriving properties from business rule models," in *ICSTW 2016*. IEEE, 2016, pp. 219–228.

[3] T. Arts, J. Hughes, J. Johansson, and U. T. Wiger, "Testing telecoms software with Quviq QuickCheck," in *Erlang'06*. ACM, 2006, pp. 2–10.

[4] R. Back and R. Kurki-Suonio, "Decentralization of process nets with centralized control," *Distributed Computing*, vol. 3, no. 2, pp. 73–87, 1989.

[5] J. Blom, A. Hessel, B. Jonsson, and P. Pettersson, "Specifying and generating test cases using observer automata," in *FATES 2004*, ser. LNCS, vol. 3395. Springer, 2004, pp. 125–139.

[6] M. M. Bonsangue, J. N. Kok, and K. Sere, "An approach to object-orientation in action systems," in *MPC'98*, ser. LNCS, vol. 1422. Springer, 1998, pp. 68–95.

[7] M. Carlsson and P. Mildner, "SICStus Prolog - the first 25 years," *Theory and Practice of Logic Programming*, vol. 12, no. 1-2, pp. 35–66, 2012.

[8] K. Claessen and J. Hughes, "QuickCheck: A lightweight tool for random testing of Haskell programs," in *ICFP'00*. ACM, 2000, pp. 268–279.

[9] L. M. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *TACAS 2008*, ser. LNCS, vol. 4963. Springer, 2008, pp. 337–340.

[10] J. Fernandez, H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu, and M. Sighireanu, "CADP - A protocol validation and verification toolbox," in *CAV'96*, ser. LNCS, vol. 1102. Springer, 1996, pp. 437–440.

[11] M. A. Francisco, M. López, H. Ferreiro, and L. M. Castro, "Turning web services descriptions into QuickCheck models for automatic testing," in *Erlang'13*. ACM, 2013, pp. 79–86.

[12] J. Hughes, "QuickCheck testing for fun and profit," in *PADL 2007*, ser. LNCS, vol. 4354. Springer, 2007, pp. 1–32.

[13] J. Hughes, U. Norell, N. Smallbone, and T. Arts, "Find more bugs with QuickCheck!" in *AST 2016*. ACM, 2016, pp. 71–77.

[14] S. H. Jensen, S. Thummalapenta, S. Sinha, and S. Chandra, "Test generation from business rules," in *ICST 2015*. IEEE, 2015, pp. 1–10.

[15] E. Jöbstl, "Model-based mutation testing with constraint and SMT solvers," Ph.D. dissertation, Institute for Software Technology, Graz University of Technology, Austria, 2014.

[16] A. S. Kalaji, R. M. Hierons, and S. Swift, "Generating feasible transition paths for testing from an extended finite state machine (EFSM)." in *ICST 2009*. IEEE, 2009, pp. 230–239.

[17] W. Krenn, R. Schlick, S. Tiran, B. K. Aichernig, E. Jöbstl, and H. Brandl, "MoMut::UML model-based mutation testing for UML," in *ICST 2015*. IEEE, 2015, pp. 1–8.

[18] L. Lampropoulos and K. F. Sagonas, "Automatic WSDL-guided test case generation for PropEr testing of web services," in *WWV 2012*, ser. EPTCS, vol. 98. Open Publishing Association, 2012, pp. 3–16.

[19] R. Nilsson, *ScalaCheck: The Definitive Guide*, ser. IT Pro. Artima Incorporated, 2014.

[20] M. Papadakis and K. Sagonas, "A PropEr integration of types and function specifications with property-based testing," in *Erlang'11*. ACM, 2011, pp. 39–50.

[21] C. Runciman, M. Naylor, and F. Lindblad, "SmallCheck and lazy Small-Check: Automatic exhaustive testing for small values," in *Haskell'08*. ACM, 2008, pp. 37–48.

[22] S. Tiran, "The Argos manual," Institute for Software Technology, Tech. Rep. IST-MBT-2012-01, 2012. [Online]. Available: https://pure.tugraz.at/portal/files/1139896/manual.pdf

[23] B. Vedder, J. Vinter, and M. Jonsson, "Using simulation, fault injection and property-based testing to evaluate collision avoidance of a quad-copter system," in *DSN Workshops 2015*. IEEE, 2015, pp. 104–111.

[24] Y. Wada and S. Kusakabe, "Performance evaluation of a testing framework using QuickCheck and Hadoop," *Journal of Information Processing*, vol. 20, no. 2, pp. 340–346, 2012.