Towards the Property-Based Testing of an L4 Microkernel API

Cosmin Dragomir
Faculty of Automatic Control and Computers
University POLITEHNICA of Bucharest
Splaiul Independentei nr. 313
Sector 6, Bucuresti, Romania
cosmin.dragomir@cti.pub.ro

Lucian Mogosanu
Faculty of Automatic Control and Computers
University POLITEHNICA of Bucharest
Splaiul Independentei nr. 313
Sector 6, Bucuresti, Romania
lucian.mogosanu@cs.pub.ro

Mihai Carabas
Faculty of Automatic Control and Computers
University POLITEHNICA of Bucharest
Splaiul Independentei nr. 313
Sector 6, Bucuresti, Romania
mihai.carabas@cs.pub.ro

Razvan Deaconescu
Faculty of Automatic Control and Computers
University POLITEHNICA of Bucharest
Splaiul Independentei nr. 313
Sector 6, Bucuresti, Romania
razvan.deaconescu@cs.pub.ro

Nicolae Tapus
Faculty of Automatic Control and Computers
University POLITEHNICA of Bucharest
Splaiul Independentei nr. 313
Sector 6, Bucuresti, Romania
nicolae.tapus@cs.pub.ro

Software testing has been a significant part of the software development process for the last 30 years and is gaining even more importance with the increasing complexity of software products. As each application has its own requirements, multiple software testing methodologies exist. It is the decision of the developers to choose the best suited types of testing methodologies for their product. This paper presents the design and implementation of a property-based testing framework. Unlike traditional testing methods this methodology uses the formal specification of the API to automatically generate the input and validate the output. The framework will be used to test the API of an L4 microkernel (called VMXL4); VMXL4 possesses the constraints of an embedded environment and of an ongoing development of a stateful system.

Property-Based Testing, L4 Microkernel, API

1. INTRODUCTION

The software industry has been constantly growing in the last decades and the liability and robustness of the software products must match their requirements in order to remain competitive. To obtain a stable product, the entire software stack must be reliable. Therefore software testing must be done at each layer of the software stack, starting with the lowest level: the operating system.

Multiple software testing methodologies are in use nowadays, each of them targeting a degree of test cases coverage and test writing complexity. Alongside the well known unit testing method, another functional methodology named property-based testing has gained ground among software developers. It uses the concept of "tests as specification", in which tests are written to cover most of the specification.

Writing a large number of tests for the same specification implies a sizable effort from the developers. Property-based testing mitigates this by automaticallly generating the input and creating general and abstract tests known as *properties*. Those can be similar to unit tests, except for the way input is generated and output is validated.

This paper presents an user space framework named QC that is based on an open source basic implementation of a property-based testing framework implemented by Pennebaker (2012). Although the well-known related released frameworks are written in functional programming languages, QC is written in C due to the VMXL4 native environment support.

Porting a new language environment would have meant a sizable and unnecessary effort.

The QC framework solves issues commonly encountered in unit testing by using properties and testing those properties on a large number of randomly generated inputs and by maintaining the internal state of the system. As a downside, QC introduces the problem of formalizing the specification.

The paper is organised as follows: Section 2 is introducing the concepts necessary to understand the paper. Section 3 briefly presents the existing similar frameworks. Section 4 discusses the design and implementation of QC and Section 5 its evaluation in comparison with the existing testing methods for the API of an L4 microkernel, VMXL4. Lastly, in Section 6 we present the overview of the paper and future work.

2. BACKGROUND

This section presents the basic concepts required to understand the next sections of the paper. It starts with the overview and importance of software testing, followed by the essential concepts of unit testing and property-based testing. We show the power of property-based testing in contrast with unit testing. We also do a short introduction of the testing environment, describing the VMXL4 microkernel.

2.1. Software Testing

Nowadays the number of different programming languages, hardware platforms and software libraries is increasing. Requirements, for both specifications and performance, are also more rigorous as time passes by. As a consequence, software applications are becoming more complex and bugs are introduced in new software at all levels. As a countermeasure software testing has gained more importance and attention from programmers.

It is important that applications and services can be stateful or stateless. In a stateful system, an internal state is being held and some of the actions have side effects which might change the state. The design of a stateful software system can be modeled using a finite-state machine and a formal specification of the inputs for every possible state. The summarizing difference between stateful and stateless systems is that for the first one the output depends on the input and possibly on the internal state, whereas for the second one the output always depends only on the input. A well known example of stateful versus stateless are the TCP and UDP transport layer protocols from the TCP/IP stack, where TCP creates and maintains a connection between the client and the server and UDP does not.

At a higher level, software testing can be defined as the process of executing a part or the entire application in order to find errors or to evaluate the quality of the user experience. Any moderately sized application has flaws, but finding them is a complex activity. It is usually unfeasible to do an exhaustive testing on stateful systems, since the number of different possibilities for the input values associated with the existing internal system state is too large. Even for some stateless applications this testing would imply a sizable effort. We conclude that it is more resource and time demanding to test a stateful system instead of a stateless system.

There are multiple methodologies in software testing which must be used in various steps of the development process. In this paper we insist only on those that can be split in two major categories: functional and nonfunctional testing.

Functional testing verifies the client or design specifications by testing the system functionality: checking if the program operations and features behave as they should. In summary it is used to ensure that the application does not have bugs. There are two categories of functional testing: positive and negative. Positive testing is done using valid inputs and comparing the actual output with the expected output, whereas negative testing is done by supplying the system functionalities with invalid or unexpected inputs and operations. In the case of negative functional testing, usually the system must not behave nondeterministically and rather inform the user of the input error.

On the other side, nonfunctional testing is concerned with the user experience, including tests for performance, security, availability, usability. Using this type of testing, one can measure and compare the results in different situations and cover the blanks left by functional testing. For a competitive software product, developers must test the program using both functional and nonfunctional methodologies.

2.2. Unit Testing

One of the most used and successful software testing methodologies is unit testing (Binder (2000); Hunt et al. (2004); Osherove (2010)). It is centered on the concept of unit of work, meaning a single, invokable, logical and functional use case of the system.

Unit testing is composed of a suite of tests that can be run anytime during the development cycle to test certain functions, logic and capabilities of the code. Each test uses a predefined set of inputs, runs a functionality of the system and compares the output with the desired output. If the outputs

differ, the tested functionality has at least one bug. Although unit tests usually verify only one small feature, sometimes it is not easy to find the bug, due to the black-box nature of the methodology. This means that unit testing does not use the internal structure of the functionality, but only the higher level invoked part, and therefore the bug may be in the internal logic and further debugging must be done. One big advantage of unit testing is its reusable nature. Even if the internal logic changes, usually the requirements of the function remains the same and the old test can still be used.

Although unit testing is very useful, it has a very important flaw, which may leave hard detectable bugs in the system. Using unit tests a programmer only verifies a small finite set of inputs and for every different input set he must write another test. Therefore, using unit test programmers cannot even get close to an exhaustive testing. In addition to not finding possible subtle bugs, the work of the programmer is hardened by thinking of all the corner cases and writing more code for them. In the end, these drawbacks are less important than the benefits of unit testing: because it gives good results in practice, it is very used and every major language has frameworks for this testing methodology.

2.3. Property-Based Testing

A software testing methodology which addresses the problems left by unit testing is property-based testing (Fink and Bishop (1997); Fernandez et al. (2004); Machado et al. (2007)). Its main advantage is that it covers substantially more test cases than unit testing; moreover, it can do exhaustive testing though the time required to do this is not directly proportional to the benefits. This is done using only one generic test, named *property*, and some functions to generate the inputs, named *generators*.

A property is the replacement of a unit test and is run multiple times with different automated generated inputs. Every running of the property generates a different test. The input of the property depends on the type and domain specified by the programmer; because the input is generic, the validation conditions of the test must be also generic, following a formal specification. The name "property" comes from the type of test validation, where each test result must pass a general formal property. In layman's terms a property validation condition specifies in a generic way how the tested functionality should behave. A drawback of property-based testing is that the formal specification does not usually exist and the programmer must infer it from the business and logic specifications.

A generator is a callback that does random data generation at each running of the property. Because complicated non-basic data types may be needed, a property-based testing tool must allow user defined generators. In order to achieve a good test coverage, the generated data must be uniformly distributed across its domain.

To show the differences between unit testing and property-based testing, let's assume one would want to test a function named `getMax`, a function which returns the maximum of two integer values. In a unit test he would hard-code two values and test if the output equals to the maximum value. If he wants to test multiple cases, possibly corner cases, then multiple tests need to be written. A pseudocode implementation of the unit test is shown in Listing 1.

```
max = getMax(2, 5)
assert(max == 5)
```
Listing 1: Pseudocode for getMax unit testing

Using property-based testing, a pseudocode implementation would be the one from Listing 2.

```
a = generator()
b = generator()
max = getMax(a, b)
assert((max == a || max == b) &&
       (max >= a && max >= b))
```
Listing 2: Pseudocode for getMax property-based testing

As shown above, the property is generic and more powerful than the unit test. However, the validation condition is bigger and must be correctly determined by the test writer, otherwise the property may give false positives or, even worse, false acceptances.

The quality of the automatic testing tool may be improved by reducing the number of failing test case inputs in order to obtain the minimum set of inputs determining a given failure, a method known as shrinking. This has the advantage of improving the debugging process by providing the programmer with minimal necessary information for debugging; another advantage is that the overhead of this method is not significant.

All in all, property-based testing has the benefits of unit testing and some advantages over it: bigger test coverage, improved specification completeness and it is easier to maintain because of the reduced code size, as illustrated by Nilsson (2014).

2.4. VMXL4

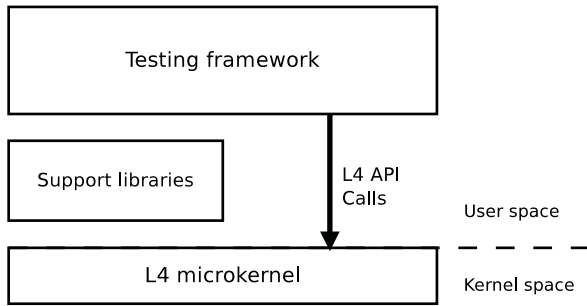VMXL4 is a general purpose, high performance L4 microkernel (Liedtke (1995)), developed in

Figure 1: VMXL4 testing infrastructure

partnership with VirtualMetrix, Inc[1]. It provides mechanisms for performance management and a minimal layer of hardware abstraction on which virtualized operating systems personalities can be built. Using the VMXL4 API, trust and security models can be implemented. Examples of systems built using VMXL4 are given in Carabas et al. (2014), Manea et al. (2015) and Mogosanu et al. (2015).

An L4 microkernel was chosen due to the fact that the L4 API's formalization was proven to be feasible by Kolanski and Klein (2006). The seL4 microkernel is the first operating system kernel to be fully formally specified and verified, as shown by Elkaduwe et al. (2008) and Klein et al. (2009). Furthermore, other implementations have been proposed for formally verifiable L4 microkernels (Kauer and Völp (2005)). The property-based testing approach proposed by this paper is in some respects similar to previous work, as it also relies on a formal specification. The most important difference between the two approaches is that property-based testing is more efficient in terms of development resources, as opposed a full mathematical refinement proof, which may require a significant number of man-months to be implemented.

Figure 1 shows the architecture of the current VMXL4 testing infrastructure. The L4 microkernel runs in the privileged CPU mode commonly known as kernel space, while the tests run as user space applications. The testing infrastructure is implemented using support libraries, but the test themselves call the L4 API directly in order to validate it functionally.

Currently most API tests are following unit testing principles, so test coverage is not nearly as extensive. However, microkernels are stateful systems, some of their core mechanisms being strongly coupled. As a result, the current testing framework does not employ true unit tests and only partially validates the interaction between components.

We propose that the QC testing framework presented in Section 4 use the same testing design, with the addendum that additional support libraries may be needed, e.g. in order to generate random numbers. This converges with our goal to provide a POSIX compliant native environment based on VMXL4.

3. FRAMEWORKS FOR PROPERTY-BASED TESTING

The idea of a property-based testing framework is not new. Previous frameworks have been developed, but the most successful are for functional languages, due to some of their distinctive features: higher order programming, which is very useful for properties and data generators, lack of side effects, time of development. Moreover, functional programming fits better for random testing than imperative programming because it uses fine-grained properties. This section presents an overview of three of the most influential existing frameworks and of some open source projects.

Haskell QuickCheck
Haskell QuickCheck[2] is the first well known framework for property-based testing and future frameworks were inspired by it. QuickCheck is a tool which automates testing for Haskell programs. As shown in Claessen and Hughes (2002, 2011), it does this by defining a formal specification language, which is powerful enough to represent common forms of specifications: algebraic, model-based and preconditional or postconditional. QuickCheck uses combinators to define properties and test data generators and obtain the test generated data distribution. An important feature of the framework is the shrinking of the generated data when a test fails, to give the minimum input which still fails the property.

Erlang QuickCheck
The programmers who developed Haskell QuickCheck saw the bigger commercial opportunity offered by Erlang and developed a new version of the framework[3], which has its specifications in Erlang. Linking specification in Erlang to code under test in other languages is easier than in Haskell. Two very important distinctive features of the Erlang version are the ability to test stateful systems by using state machine testing and the ability to generate and run parallel test cases in order to find race conditions (Arts and Hughes (2003)).

---

[1] http://www.virtualmetrix.com/

[2] https://hackage.haskell.org/package/QuickCheck
[3] http://www.quviq.com/products/erlang-quickcheck/

ScalaCheck

ScalaCheck[4] is the third main framework used for property-based testing and is used for automated randomized property-based testing of programs developed in Scala or Java (Odersky (2010)). It was inspired by Haskell QuickCheck and implements most of its features, but also some in addition to its predecessor, such as stateful testing. Nilsson (2014) provides a comprehensive guide to ScalaCheck.

Open Source Initiatives

Due to the success of Haskell QuickCheck, open source implementations in most major programming languages were started, such as C, C++, Java, Python, but with less features and success. QC was inspired by one of those open source initiatives, employed by Pennebaker (2012).

4. QC DESIGN AND IMPLEMENTATION

This section presents the design and implementation of the QC framework and the motivation behind it. QC intends to test the L4 microkernel API in a functional manner, following the property-based testing methodology. It may be used alongside unit tests, because it tries to generalize them, but on a long term it may strive to replace unit testing for the VMXL4 microkernel API.

4.1. Implementation Starting Point

The implementation is based on the open source project developed by Pennebaker (2012). It is a basic framework, supporting only two features: random data generation and one property per test, which was run for a predefined number of times. A part of the implementation is not really usable, because the programmer who uses the framework must know the size of the generated types and create tests accordingly, which is error-prone. The only part which we partially used is the test data generation component.

The framework is implemented in the C programming language because it was the most convenient option taking into consideration the testing environment. Porting a new language environment can be very complicated, because the native environment offered by a microkernel is very low-level. Moreover, implementing a POSIX environment is equivalent with the implementation of an entire operating system. Also, because the API had already been written in C, there is no need for further linking between different languages.

---

[4]`https://www.scalacheck.org/`

4.2. General Design

Because a kernel is a stateful system and QC is developed to test the L4 microkernel API, being evaluated on VMXL4, two more concepts used by QC have to be introduced. Preconditions are a set of predicates that must be true prior to the execution of a property and postconditions are a set of predicates that must be true after the execution of an action in the property. If all the preconditions of a property are true, then the property is applicable, otherwise it is not. If all the postconditions of a property are true, than the property has passed.

Due to the fact that QC is designed for a stateful system, it uses tests containing multiple properties that are used as actions with side effects in the stateful system. Therefore QC borrows some elements from integration testing, a methodology in which individual software modules are tested together. Each test consists of at least one property, randomly generated from the available properties. Each property has a finite number of arguments with known data types at compile time, a fact that provides the opportunity to use property-based testing. When at least one of the postconditions of a property has failed, then the test fails, the entire generic test completes and the actions taken during the test are printed alongside their input. The second situation in which a test fails is when its state becomes inconsistent, meaning that no property has all of its preconditions passing and therefore no future action can be taken. When a test fails, the programmer sees all the randomly generated data used by the test and this facilitates easier debugging.

Properties are divided in two categories: normal properties and cleanup properties. Normal properties are placeholders for actions that the system may take anytime, provided the preconditions are satisfied. Cleanup properties are used to end tests and to free allocated resources. Every test must have at least one cleanup property. If a test does not have allocated resources, QC provides the `empty_clean_property` macro for an empty property, whose only purpose is to end the test. Although most of the time only one cleanup property will be used for a generic test, having multiple cleanup properties may be useful in some situations. One can use fine-grained cleanup properties if the system can have many internal states. This makes the code cleaner and in a system with many generic tests, the probability to reuse cleanup properties is bigger if they are fine-grained.

A higher level design of QC is shown in Figure 2. The programmer must call QC with an array of normal properties and an array of cleanup properties, as previously discussed. To generate random input and
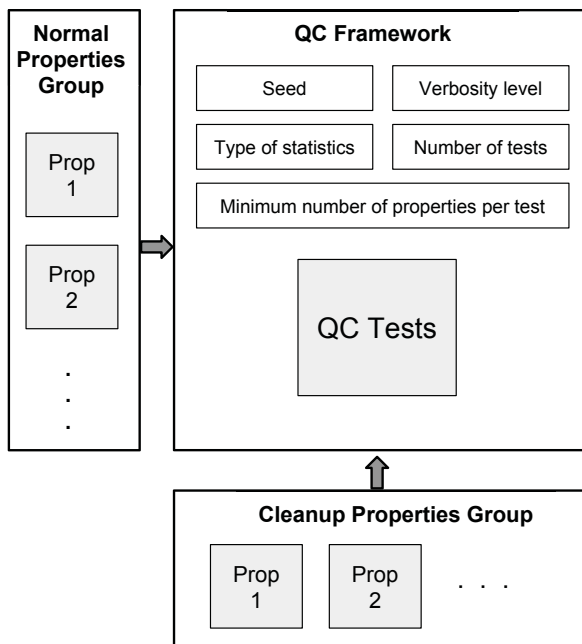
Figure 2: *QC design*

randomly pick properties, QC uses a seed. When a test fails, the programmer will want to reproduce the exact same sequence of properties and inputs to test the fix for the bug. Because the random generation is deterministic given the seed, QC shows it for every generic test so the programmer can use that seed if he wants to reproduce the tests. Otherwise, QC generates a random seed to assure random tests and a good test coverage.

QC will generate a fixed number of tests, previously given by the user. Another available option is the verbosity level for generic tests. The user can see the sequence used by every test or only by failing tests. Viewing the sequence used by every test can be useful to improve the generic test and its test coverage.

There may be cases when tests will end prematurely, after using only a few properties, because QC may choose and use a cleanup property whenever its preconditions are satisfied. To mitigate this, the user chooses the minimum number of properties that must be used during a test.

The last parameter from Figure 2 represents the type of statistics shown at the end of the generic test. QC supports two types of statistics for every generic test: user defined and automatically generated. The user can choose to see both categories, only one or none.

4.3. Generators

As previously mentioned, generators are the callbacks that randomly produce the input data.

A QC generator consists of 2 callbacks: one to generate the data and the other to print the data, as the C `printf` function needs different formats depending on the printed type. Because of that, the QC equivalent of a generator is `struct generator_printer`.

QC offers a set of predefined generators for C basic data types, such as `int` and `char`, and also for `bool`, `string` (stored as a `char` dynamic array) and generic arrays. Moreover, a user can write his own generators or printers and use them for his properties.

In order to generate arrays, only the basic type generator is needed, because QC offers a wrapper which automatically generates new array types. The array type can have fixed or random size in a given range, depending on the user needs. All generated arrays are dynamically allocated and their memory is freed after their associated property ends. This avoids out of memory errors for big tests with many generated arrays, but can introduce subtle bugs if the user forgets to copy the content of the array in case he needs it after the property ends.

Sometimes basic types may need additional features such as a maximum or minimum value. QC offers two solutions for this. The first one is that miscellaneous parameters can be added to the generators, in order to modify the generated value to match the requirements. The second solution is to change and update the generated values from the property code. Both solutions are acceptable for code readability, but in general cases the first option should be used, because it's reusable and only the parameters will be changed.

All generated data for a property is stored in a dynamically allocated array with the size in bytes equal to the maximum number of arguments of a property multiplied by the maximum size in bytes of an argument type. This approach solves the problem with the variable number of property arguments and their different types. The value of the generated data is obtained in the property and the user must only know the data type and the index of the argument, something that he had already defined in the state machine test specification.

4.4. Statistics

In order to measure various metrics, QC offers the possibility to attach user defined statistics to a property. After a property ends successfully, each of its statistics callbacks is called and the metrics are updated. This can help the user to investigate bugs and also measure the test coverage. An example of this logging category is shown in Listing 7.

By default, QC logs statistics regarding properties and their sequence. For every property, QC logs the number of total calls, the number of starting test property calls and for every property how many times it followed the current property. An example of this type of logging is shown in Listing 8. The default logging done by QC can be very useful to detect preconditions bugs and see if the tests are surfacing the desired states.

The user decides if he wants to see any of the two statistics category when the QC framework is called to generate and validate tests.

4.5. Properties, Preconditions and Postconditions

Since QC supports stateful system testing, using a property requires the following steps: testing preconditions, getting the values for the randomly generated data, applying actions and testing postconditions.

The preconditions are optional but if they are missing, the property can be always chosen by the framework as the next part of the test. Preconditions are implemented as a callback, differently from the property callback. Because the preconditions depend only on the internal state, not on the generated data, it is better to obtain the new data only if the property can be applied, avoiding generation of useless data, which will be replaced afterwards. Therefore, preconditions must be tested before the data generation step and this can be achieved by having another callback, associated with a property. This approach has another benefit: some preconditions are used by multiple properties and having them as functions gives better reusability. If a property does not have any preconditions, their callback will be NULL. To address any possible usage, preconditions can be used from inside the property too, but this is not a good practice, as explained above.

Actions are the main content of properties, because they change the state of the system and their side effects are verified by the postconditions. Actions can be interleaved with their postconditions, which are obtained from the formal specification of the API. As opposed to preconditions, postconditions are located in their corresponding property callback, because they depend on the generated data and we do not obtain a performance improvement if we have them in separate callbacks. Moreover, if the property contains multiple actions, then it is recommended to check the postconditions for an action or a group of actions as soon as possible, in order to have good code readability.
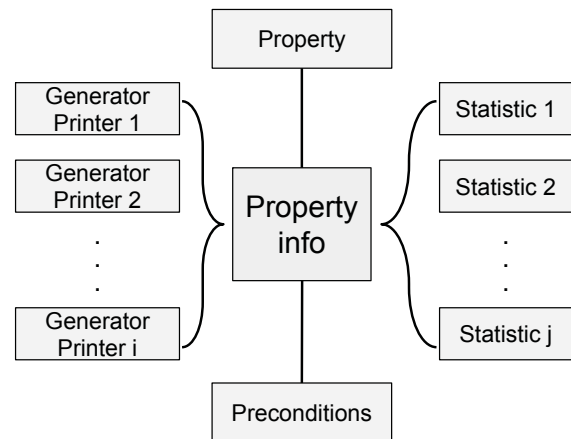

Figure 3: *Property info callbacks*

As can be seen in Figure 3, QC properties are composed of multiple callbacks, stored in a structure named property info. We need an array of generators for the property input and another array of user defined statistics to gather various data. On the other side, we need a callback for preconditions and a callback for the property itself, to make the API calls and test the postconditions. Having all of those callbacks, different components of generic tests become easier to integrate with each other.

```
struct property_info {
    /* callback for the property */
    prop function;

    /* displaying name */
    char const * const name;

    /* array of generators */
    struct generator_printer *gp;

    /* number of generators */
    int gp_size;

    /* preconditions callback */
    precondition prec;

    /* array of statistics callbacks */
    struct user_statistic *stats;

    /* number of statistics callbacks */
    int stats_size;
};
```
Listing 3: Struct property_info

In QC's implementation, property descriptions and callbacks are contained by the `property_info` structure, as shown in Listing 3. In addition to what was previously discussed, the `name` field assigns a

descriptive name to the property and is used for the verbosity option `QC_INFO` or for failing tests.

4.6. QC test logic

Having all the previously discussed elements, QC can generate and run tests. Figure 4 shows a state machine with the actions taken by QC during a test. Until a test fails or the required number of tests have been run, the framework tries to falsify the generic test by finding a failing test.

Starting a test, QC chooses a property, checks its preconditions (should they exist) and, based on the result, picks another property or continues with the current one. If the preconditions have passed, QC generates test data, executes the actions and the postconditions are checked. If any of the postconditions fails, the testing is over, because QC falsified a sequence of properties. Otherwise, the statistics are updated and if the property is from the cleanup group, the current test completes and another test starts; if the property is from the normal group, the test continues and another property is chosen.

4.7. Pseudorandom number generator

QC has a random module which currently supports two implementations: the POSIX `rand` function and the Mersenne Twister PRNG. The default implementation is Mersenne Twister (presented in Matsumoto and Nishimura (1998)), because it provides better data distribution than `rand` and always has the same output for a given seed, on 32 bits, in contrast with `rand`, whose result may vary depending on the architecture.

4.8. VMXL4 Influence over QC

Internally QC uses a seed for randomizing the test data generation and the chosen property at every step of a test. Because the VMXL4 native environment is under ongoing development and some POSIX functions (e.g. `rand`) are not yet implemented, inside the testing environment the seed is actually a numerical value obtained from a hardware timer provided by the development platform. However, the framework does not depend on a specific platform and is portable, requiring only POSIX functionality.

The VMXL4 API is currently being tested using the Check Unit Testing Framework for C. In order to be as easy as possible to use and because unit tests usually need little changes to become properties, the QC interface has been designed to have some similarities with the Check framework. For that reason, postconditions can be tested using

`prop_fail_if` and `prop_fail_unless`, wrappers similar to Check's `fail_if` and `fail_unless`.

5. QC EVALUATION AND TESTING

This section describes evaluation, results and implications, while the framework is still under development. To evaluate the performance of the QC framework, its impact on the test coverage and code size will be detailed.

```c
/* normal properties */
struct property_info p[] = {
  {init_property, "init",
    (gp_array){qc_u_int}, 1,
    q_is_not_initialized,
    (stats_array){queue_size_stat}, 1},
  {dequeue_property, "dequeue",
    (gp_array){}, 0,
    q_is_initialized,
    NULL, 0},
  {enqueue_property, "enqueue",
    (gp_array){qc_int}, 1,
    q_is_initialized,
    (stats_array){element_sign_stat}, 1}
};

struct property_group normal_group = {
  .prop = p, .size = 3
};

/* cleanup properties */
struct property_info clean_p[] = {
  {clear_property, "clear",
    (gp_array){}, 0,
    q_is_initialized,
    NULL, 0}
};

struct property_group clean_group = {
  .prop = clean_p, .size = 1
};

qc_for_all(
  /* property groups */
  normal_group, clean_group,
  /* minimum properties */
  5,
  /* verbosity level */
  QC_ERROR,
  /* number of tests */
  1000,
  /* statistics level */
  QC_SHOW_STATS
);
```
Listing 4: QC initialization and calling to test a circular queue library API
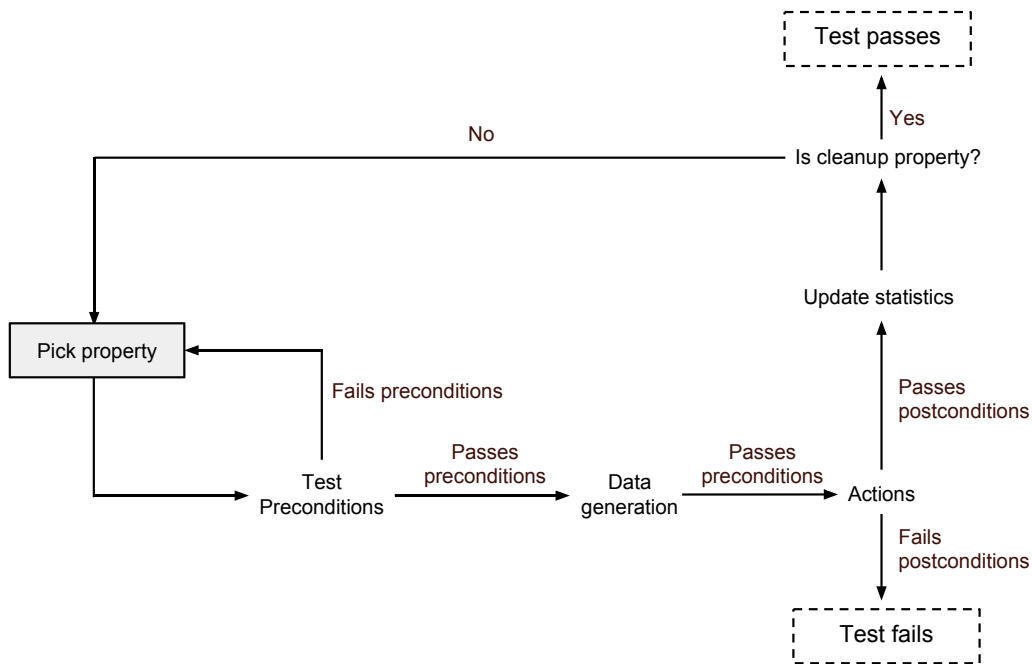
Figure 4: *Test state machine*

QC has been tested so far on two modules. The first module is an implementation of a circular queue, which has been chosen for the following three reasons: it is easier to validate new features of the framework with a simpler module, it is a stateful system, with an internal representation for the queue, and it is a portable module which can be used to validate QC against Haskell QuickCheck. The second module is the thread scheduling module of VMXL4, currently being tested with unit tests using the Check framework.

For the circular queue, the code from Listing 4 was used to initialize QC in order to test the queue library public API. It can be observed that the code uses normal properties and cleanup properties, as mentioned in Section 4. With just four properties, similar to unit tests, the framework automatically generates 1000 test cases with a different number of properties and different sequences of properties, each with automatically generated different inputs. As one can see, there is not much difference in the code logic complexity for unit testing and property-based testing, but the benefits of property-based are significant. When different bugs were introduced on purpose in the queuing logic, QC detected all of them, using only the code from Listing 4 and its four properties.

An example of QC finding a bug for the circular queue is Listing 5. It displays, in order, all the properties taken during the test and their generated input. Judging by the output, it is most likely that

there are problems with the enqueue operation when the queue gets full. This is one of the corner cases which the programmer should have taken care of personally if he were to use unit testing.

```
*** Test Failed! ***
Test number 43
————————————————
init: 2
dequeue:
enqueue: −392470180
enqueue: −692402
```
Listing 5: QC failing test

After solving the bug, QC validates the implementation in Listing 6.

```
+++ Success: passed 1000 tests. +++
```
Listing 6: QC tests passing

For the generic test from Listing 4, the generated user defined statistics are shown in Listing 7. QC displays the total number of statistics for every property. For the `init` property, we wanted to see in what range is the queue size. For the `enqueue` property, we wanted to see the sign of the enqueued number. It can be observed that the numbers are showing a balance for the generated data.

```
——TESTS USER DEFINED STATISTICS——

"init" INFO
total statistics: 1
```

```
name: "queue_size"
    <20: 201
    <40: 201
    <60: 202
    <80: 203
    <100: 193
───────────────────────────────────────
"enqueue" INFO
total statistics: 1
    name: "element_sign"
        negative: 1457
        positive: 1423
```
Listing 7: User defined statistics

Last but not least, for the generic test from Listing 4, the QC default statistics are shown in Listing 8. For every property, QC displays the total number of calls, how many times it was the first property of the test and afterwards for every property how many times it followed the current property. Note that for cleanup properties QC doesn't show the following properties, because the cleanup property will be the last from the test. Generally, QC default statistics are useful not only to balance the tests, but also to debug preconditions.

```
───TESTS PROPERTY SEQUENCE STATISTICS───

"init" INFO
total calls: 1000
starting calls: 1000
    init: 0
    dequeue: 527
    enqueue: 473
    clear: 0
───────────────────────────────────────
"dequeue" INFO
total calls: 3003
starting calls: 0
    init: 0
    dequeue: 1264
    enqueue: 1239
    clear: 500
───────────────────────────────────────
"enqueue" INFO
total calls: 2880
starting calls: 0
    init: 0
    dequeue: 1212
    enqueue: 1168
    clear: 500
+++++++++++++++++++++++++++++++++++++++++
"clear" INFO
total calls: 1000
starting calls: 0
```
Listing 8: QC statistics

When porting some of the unit tests for the VMXL4 thread scheduling module to QC properties, the VMXL4 API Reference Manual was needed to understand the behavior of tested functions and to infer the formal specification, which, in the end, is not a sizable effort for a programmer who is accustomed to the design of the module. For some of the ported unit tests the specification was very simple and their content remained almost the same.

Although only a few unit tests were ported to QC properties, the framework already found one inconsistency in the unit test. The faulty unit test was verifying if two threads with different priorities are scheduled accordingly; however on symmetric multiprocessing (SMP) the validation condition was always true. The test would have passed even if the system had a bug.

The inconsistency was found after transforming the unit test into a property and using the same wrong specification. The property was failing, therefore only two causes could have been possible: the property was wrong or the module had a bug. Fortunately, the first case was true and the unit test was the cause. QC found the inconsistency using its random generation feature. This emphasizes that unit tests are not very reliable compared to properties, because usually they do not take into consideration many test cases, therefore they may hide system bugs or even test design bugs.

6. CONCLUSIONS AND FURTHER WORK

Every software system needs testing in order to fulfill its business requirements and, as a consequence, be reliable and successful. This paper concentrates on property-based testing, because although it is more powerful than unit testing, due to its bigger input coverage, it is used less frequently than unit testing. In order to emphasize the property-based testing applicability and importance, the paper gives an overview of the QC framework.

QC is an automated testing tool written in C which runs in the native environment of an L4 microkernel and whose purpose is to test the microkernel API in a functional manner. Because the microkernel is a stateful system, the framework allows the testing of multiple controlled series of operations, besides the usage of random generated input. In order to obtain a thorough testing, QC offers support for generating any data type, using predefined generators which can be combined to obtain new test data generators. To test and evaluate the framework, the native environment of the VMXL4 microkernel is used.

As future work, QC failing tests will be shrinked to a more suggestive failing test, to ease the work of the debugging programmer. Additionally, we aim to analyze QC's code coverage, compare it to that of other L4 testing infrastructures and find ways to improve it.

REFERENCES

T. Arts and J. Hughes. Erlang/quickcheck. In *Ninth International Erlang/OTP User Conference*, 2003.

R. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Professional, 2000.

M. Carabas, L. Mogosanu, R. Deaconescu, L. Gheorghe, and N. Tapus. Lightweight display virtualization for mobile devices. In *Secure Internet of Things (SIoT), 2014 International Workshop on*, pages 18–25. IEEE, 2014.

K. Claessen and J. Hughes. Testing Monadic Code with QuickCheck. `http://www.cs.tufts.edu/~nr/cs257/archive/john-hughes/quick.pdf`, 2002.

K. Claessen and J. Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. *Acm sigplan notices*, 46(4):53–64, 2011.

D. Elkaduwe, G. Klein, and K. Elphinstone. Verified protection model of the sel4 microkernel. In *Verified Software: Theories, Tools, Experiments*, pages 99–114. Springer, 2008.

J.-C. Fernandez, L. Mounier, and C. Pachon. Property oriented test case generation. In *Formal Approaches to Software Testing*, pages 147–163. Springer, 2004.

G. Fink and M. Bishop. Property-Based Testing; A New Approach to Testing for Assurance. `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.93.5559&rep=rep1&type=pdf`, 1997.

A. Hunt, D. Thomas, and P. Programmers. *Pragmatic unit testing in Java with JUnit*. Pragmatic Bookshelf, 2004.

B. Kauer and M. Völp. L4. sec preliminary microkernel reference manual. 2005.

G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, et al. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220. ACM, 2009.

R. Kolanski and G. Klein. Formalising the l4 microkernel api. In *Proceedings of the Twelfth Computing: The Australasian Theory Symposium-Volume 51*, pages 53–68. Australian Computer Society, Inc., 2006.

J. Liedtke. *On micro-kernel construction*, volume 29. ACM, 1995.

P. D. Machado, D. A. Silva, and A. C. Mota. Towards property oriented testing. *Electronic Notes in Theoretical Computer Science*, 184:3–19, 2007.

V. Manea, M. Carabas, L. Mogosanu, and L. Gheorghe. Native runtime environment for internet of things. In *Advanced Computational Methods for Knowledge Engineering*, pages 381–390. Springer, 2015.

M. Matsumoto and T. Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 8(1):3–30, 1998.

L. Mogosanu, M. Carabas, R. Deaconescu, L. Gheorghe, and V. G. Voiculescu. VMXHAL: A Versatile Virtualization Framework for Embedded Systems, 2015.

R. Nilsson. ScalaCheck: The Definitive Guide, 2014.

M. Odersky. Contracts for scala. In *Runtime Verification*, pages 51–57. Springer, 2010.

R. Osherove. *The art of unit testing*. mitp, 2010.

A. Pennebaker. A C port of the QuickCheck unit test framework. `https://github.com/mcandre/qc`, 2012.