μQC: a property-based testing framework for L4 microkernels

Cosmin Dragomir, Lucian Mogosanu*, Mihai Carabas, Razvan Deaconescu and Nicolae Tapus

Faculty of Automatic Control and Computers,
University POLITEHNICA of Bucharest,
Splaiul Independentei nr. 313, Sector 6,
Bucuresti, Romania
Email: cosmin.dragomir@cti.pub.ro
Email: lucian.mogosanu@cs.pub.ro
Email: mihai.carabas@cs.pub.ro
Email: razvan.deaconescu@cs.pub.ro
Email: nicolae.tapus@cs.pub.ro
*Corresponding author

Abstract: As the complexity of software is increasing, traditional testing methods are unable to provide a high level of assurance for critical software systems, particularly operating system kernels, which represent the root of trust in computing systems. At the same time, formal verification is costly and difficult even for small system software such as microkernels, which are relied on for applications with high security and/or safety requirements, e.g., automotive and cellular radio. Our work closes a trade-off between the strong guarantees of formal verification and the flexibility of traditional software testing methods such as unit testing. We argue that this trade-off can be readily closed by property-based testing. In this paper we present μQC, a property-based testing framework for L4 microkernels. We illustrate our prototype by evaluating a significant subset of the L4 API (threading and scheduling) starting from its specification.

Keywords: software testing; property-based testing; L4 microkernel; application programming interface; API.

Reference to this paper should be made as follows: Dragomir, C., Mogosanu, L., Carabas, M., Deaconescu, R. and Tapus, N. (2018) 'μQC: a property-based testing framework for L4 microkernels', *Int. J. Critical Computer-Based Systems*, Vol. 8, No. 1, pp.1–24.

Biographical notes: Cosmin Dragomir obtained his Master's degree in Artificial Intelligence at the University POLITEHNICA of Bucharest and was a Teaching Assistant at the same university. His main interests are related to machine learning, with a particular focus on deep learning. His other interests include natural language processing, information retrieval, alongside software testing and security.

Lucian Mogosanu is a PhD student and Teaching Assistant at the University POLITEHNICA of Bucharest. His main research focus is on devising new techniques for verification and enforcement of safety and security properties in systems software. His other interests include hardware architectures, programming languages and compilers.

Mihai Carabas is an Assistant Professor at the University POLITEHNICA of Bucharest. His main research interest is on studying and developing mechanisms to improve virtualisation in operating systems. His other interests include system administration, high performance computing and advanced network protocols.

Razvan Deaconescu is an Assistant Professor at the University POLITEHNICA of Bucharest, Romania. He has always been fond of operating systems and low-level programming with recent interest in runtime application security and reverse engineering. He enjoys working with binaries, executables and assembly language. For the past years he has been involved in local practical security related activities such as CTF contests and summer schools. He has recently taken interest in iOS security, focusing on the lower-layers of the iOS software stack.

Nicolae Tapus is a Professor at the University POLITEHNICA of Bucharest. He is fond of teaching and doing research in parallel and distributed architectures, operating systems-related and networking topics, with interests also in systems security.

This paper is a revised and expanded version of a paper entitled ‘Towards the property-based testing of an L4 Microkernel API’ presented at 9th Workshop on Verification and Evaluation of Computer and Communication Systems (VECoS 2015), Bucharest, Romania, 10–11 September 2015.

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.

While traditional testing methodologies such as unit testing are necessary for software projects, they are usually unable to exhaustively cover the input space and the state space of complex software systems. As such, they cannot guarantee the absence of bugs. This holds true even for operating system software that is designed to be minimal, i.e., microkernels, that implement otherwise complex functionality such as scheduling, message passing and memory management mechanisms

A breakthrough in the area of formal verification of systems software was achieved by Klein et al. (2009), who devised a formal specification of the seL4 microkernel (Elkaduwe et al., 2008) and verified its implementation with respect to the mathematical model. The project has shown that formal methods are feasible for real-world software verification; however, this achievement required more than 20 person-years of effort (Klein et al., 2014), which makes the approach highly impractical.

We set the goal of testing the implementation of an L4 microkernel using a method called *property-based testing* (Fink and Bishop, 1997), which provides a trade-off between the strength of formal verification and the flexibility of traditional functional testing approaches. Property-based testing uses the concept of ‘tests as specification’,

taking the formal specification of a software component as a set of *properties*, and using them to automatically generate test cases to cover as much of the input space as possible.

We argue that this approach is suitable for testing an L4 microkernel at the application programming interface (API) level, as it allows us to build upon previous efforts to formalise its API (Kolanski and Klein, 2006; Elkaduwe et al., 2008). At the same time we think that writing new properties requires similar engineering effort to write unit tests, as the former are often generalisations of the latter.

In this paper we describe the design, implementation and validation of μQC, a property-based testing framework for an L4 microkernel named VMXL4. We develop this framework as a native application running in the microkernel user space, i.e., that accesses microkernel functionality via system calls. Unlike other property-based testing approaches (see Section 3), we develop the framework using C, as porting the run-time of higher-level functional languages requires a sizable effort.

We bring the following contributions:

- We design and implement a generic, minimal, practical property-based testing framework named μQC, and we port it on an embedded platform running an L4 microkernel.
- We develop a minimal set of correct and erroneous programs used to validate μQC and provide a set of usage examples.
- We evaluate the effectiveness of μQC, including the development of μQC properties pertaining to a subset of the L4 API.

The rest of the paper is organised as follows: Section 2 introduces the concepts necessary to understand the paper. Section 3 briefly presents existing similar frameworks, used as a starting point for μQC. Section 4 discusses the design and implementation of μQC. Section 5 provides a proof-of-concept evaluation and a discussion of the trade-offs involved in implementing tests using μQC. Lastly, in Section 6 we present an overview of the paper and provide perspectives for future work.

2 Background and related work

This section places the subject of property-based testing of L4 microkernels in context. We first give an overview of microkernels, with a particular emphasis on the L4 family of microkernels. Then we outline related approaches for verifying and validating operating system kernels. Finally, we describe property-based testing and we propose it as an alternative to existing approaches for microkernel testing and verification.

2.1 L4 microkernels

L4 microkernels (Liedtke, 1995) are a family of operating system kernels designed to meet two basic requirements: minimal performance overhead and a minimal set of abstractions exposed to the programmer, leaving the implementation of most system functionality as unprivileged user space services. Examples of L4 microkernels include Lietdke's original L4 (Liedtke, 1995), L4Ka: Pistachio (Ruocco, 2008), Fiasco (Lackorzynski and Warg, 2009) and seL4 (Elphinstone and Heiser, 2013).

The L4 API exposes four fundamental abstractions: threads, scheduling, address spaces and communication endpoints; they are used to represent execution, processor allocation, virtual memory management and inter-process communication (IPC) respectively. Hardware interrupts and other events such as software exceptions and page faults are abstracted as IPC calls to a handler thread, while communication with peripheral devices is made directly using the functionality provided by the hardware architecture (e.g., I/O ports, memory mapping). Moreover, modern L4 interfaces are implemented using capabilities (Levy, 1984; Mullender and Tanenbaum, 1986; Shapiro et al., 1999; Lackorzynski and Warg, 2009), making them amenable to formal specification and verification, as is the case with seL4 (Klein et al., 2009).

For our work we use VMXL4, a general purpose, high performance L4 microkernel, developed by VirtualMetrix, Inc. VMXL4 provides mechanisms for performance management and a minimal layer of hardware abstraction on which virtualised 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. (2016).

2.2 Related approaches for verifying low-level software

Generally, operating system kernel components such as device drivers are tested individually as they are integrated with the kernel. Modern commodity operating system kernels are also tested using system-specific test suites such as Linux test project (Larson, 2002), the FreeBSD Test Suite (Merino, 2013) or stress tests such as the FreeBSD Kernel Stress Test Suite (Holm, 2016). There also exist system-independent test tools, e.g., for POSIX systems, one such example being Ballista (Koopman and DeVale, 1999).

In particular, microkernels are often used for high-assurance applications such as real-time systems, which require more rigorous approaches: user space components, as well as kernel subsystems are extensively tested and/or verified (Revuelta et al., 2006; Schierboom, 2007; Brucker et al., 2015). The L4 microkernel API has been formally specified using the B method (Kolanski and Klein, 2006; Hoffmann et al., 2007), leading to the development of the seL4 protection model (Elkaduwe et al., 2008) and seL4's full formal verification (Klein et al., 2009). Additionally, Verbeek et al. (2015) specify the API of PikeOS, a separation kernel based on L4; Baumann et al. (2009) use the verifying C compiler (Cohen et al., 2009) to verify safety properties of the PikeOS kernel code; Kuznetsov et al. (2010) use selective symbolic execution to test several closed-source kernel device drivers.

The VMXL4 microkernel is tested using a traditional test suite comprising unit tests, as well as more complex tests addressing the relationship between different subsystems such as scheduling and IPC, which are strongly coupled. Figure 1 shows the architecture of the legacy VMXL4 testing infrastructure. The L4 microkernel runs in the privileged processor mode commonly known as supervisor mode, while the tests run as user space applications. The testing infrastructure is implemented using support libraries, but the tests themselves call the L4 API directly in order to verify its functionality.

We propose that the μQC testing framework presented in Section 4 uses the same system-level testing design, with the addendum that additional support libraries may be needed, e.g., in order to generate random numbers.

Each of these verification approaches has its own limitations. Traditional unit testing is a standard requirement, but it lacks coverage in terms of the functionality it verifies.

Formal methods are a comprehensive approach to system verification, but they incur high economic costs (Klein et al., 2009). Model checking and similar methods (e.g., symbolic execution) mitigate these costs but they are computationally expensive, being vulnerable to the state explosion problem (Valmari, 1998).

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 a *property*, and some functions to generate the inputs, named *generators*.

Figure 1 Testing infrastructure running on top of an L4 microkernel

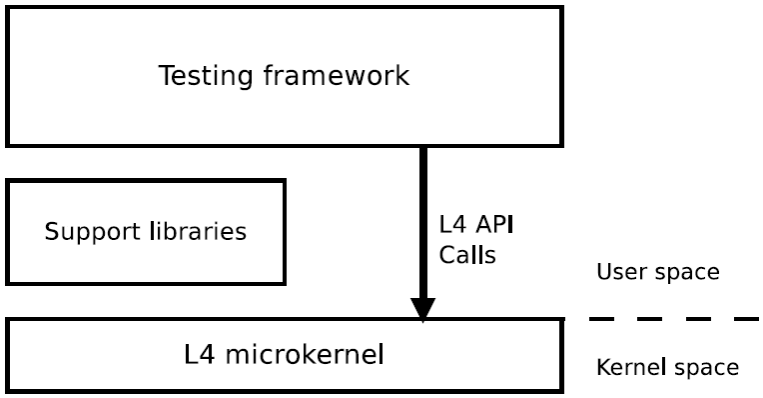


Figure 2 Pseudocode for getMax unit testing

```
max = getMax (2, 5)
assert (max == 5)
```

A *property* is the replacement of a unit test and is run multiple times with different automatically-generated inputs. Every run 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 also be 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. 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 illustrate the differences between unit testing and property-based testing, we propose the following scenario. We want to test a function named getMax, a function

which returns the maximum of two integer values. In a unit test we would hard-code two values and test if the output equals to the maximum value. If we want to test multiple cases, possibly corner cases, then multiple tests need to be written. A Pseudocode implementation of the unit test is shown in Figure 2.

Using property-based testing, a Pseudocode implementation would be the one from Figure 3. As shown, the property is generic and more powerful than the unit test, and it is closer to the formal definition of getMax. However, the validation condition is more complex and must be correctly determined by the test writer, otherwise the property may give false positives or, even worse, false negatives.

Figure 3 Pseudocode for getMax property-based testing

```
a = generator ()
b = generator ()
max = getMax (a, b)
assert ((max == a || max == b) &&
        (max >= a && max >=b))
```

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 this purpose.

We posit that property-based testing provides a powerful trade-off between traditional testing and formal methods. Property-based testing provides superior test coverage to traditional testing; it relies on a formal model, but it does not require a full refinement proof of implementation correctness; unlike state space search methods such as model checking and symbolic execution, it explores the system state space at run-time, and thus it does not suffer from state explosion. We evaluate this claim in Section 5.

3 Choosing a framework for property-based testing

The idea of a property-based testing framework is not new. Previous frameworks have been developed, with the most successful ones designed for functional programming 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 as it uses fine-grained properties. This section presents an overview of some of the most influential existing frameworks and of some open source projects.

Haskell QuickCheck[1] is a popular framework for property-based testing. 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 pre-conditional or post-conditional. QuickCheck uses combinators to define properties and test data generators, and to 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.

QuickCheck's success led to the development of property-based frameworks for other languages, the most notable being Erlang QuickCheck, ScalaCheck, and QC.

The programmers who developed Haskell QuickCheck saw the bigger commercial opportunity offered by Erlang and developed a new version of the framework[2], which has its specifications in Erlang. Linking Erlang specifications to code written 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).

ScalaCheck[3] is a property-based testing framework used for 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.

System-level programs such as operating system kernels are easier to test using lower level programming languages such as C, which implement lightweight run-time libraries and are easier to be run even in the absence of an operating system. Thus to develop our μQC framework, we started from one of the most popular property-based testing frameworks written in C, namely QC.

Although we have found other property-based testing frameworks written in C, such as quickcheck4c[4] and Theft[5], we chose to make an L4 port of QC (Pennebaker, 2012). Our decision was based on the fact that it is the oldest and most stable project implementing property-based testing in C.

4 μQC design and implementation

μQC intends to test the L4 microkernel API in a functional manner, following the property-based testing methodology. It may be used alongside unit tests, as it tries to generalise them, but on the long term it may strive to replace unit testing for the VMXL4 microkernel API. In this section we present the design and implementation decisions for the μQC framework and the motivation behind them. We begin by describing the implementation starting point and by giving a general design overview. Afterwards we provide details regarding specific elements of the framework, ending with the port of μQC on an L4 microkernel.

4.1 Design overview

Before describing μQC, we will give a short description of the implementation starting point: the open source project developed by Pennebaker (2012), at its source code snapshot in March 2015. From now on, we will mention it as QC. QC is a basic framework for property-based testing of stateless systems. Therefore, it can be viewed as a unit test generalisation. In a single test, QC runs the same property for a fixed number of times. QC elements include generators, properties and post-conditions.

Preconditions are a set of predicates that must be true prior to the execution of a property and post-conditions 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 post-conditions of a property are true,

then the property has passed. Every property used in a property-based test needs post-conditions in order to validate the result of its actions. On the other hand, preconditions are required only in stateful testing, in order to verify if the actions of the property have to be tested in the current state of the system. Therefore, QC does not use preconditions, in contrast with μQC, which is designed for stateful testing of an L4 microkernel API. In μQC, the only part from QC that is partially used is the test data generation component, namely generators, although μQC enhances their functionality by adding further options, such as parameters for data generation. Those features will be described in Section 4.2. The properties were entirely redefined, in order to make μQC usable for stateful testing and as generic as possible. Post-conditions are mentioned just at the theoretical level, because in the case of an L4 microkernel those are calls to the L4 API, therefore there is no mandatory callback for them.

The μQC framework is implemented in the C programming language, as QC is, because it was the most convenient option taking into consideration the testing environment. Porting a new language environment can be difficult, as the native environment offered by a microkernel is very low-level. Moreover, implementing a POSIX environment can be equivalent with the implementation of an entire operating system. Also, because the L4 API had already been written in C, there is no need for further linking between different languages.

Due to the fact that μQC 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 μQC 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 post-conditions 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, making debugging easier.

In μQC, 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. Because resources may remain allocated in the system after running several normal properties and because we want to allow tests with a different number of run properties, we introduced the concept of cleanup properties; cleanup properties are used to end tests and to free allocated resources. Every property-based test in μQC must have at least one cleanup property. If a test did not allocate resources, μQC 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 is 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.

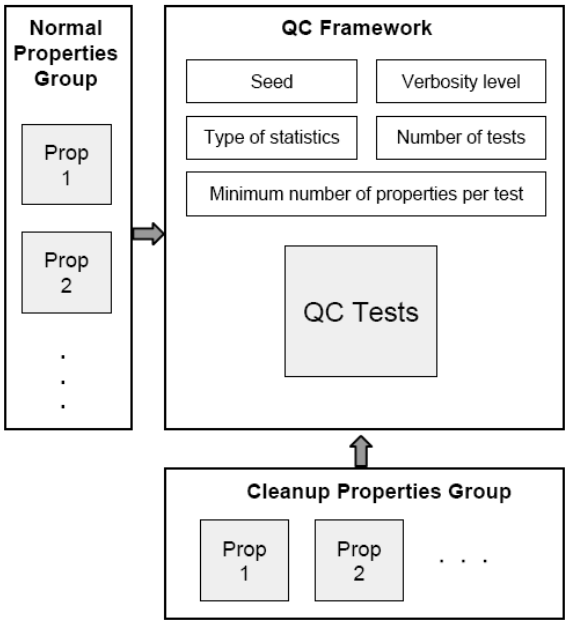
A higher level design of μQC is shown in Figure 4. The programmer must call μQC with an array of normal properties and an array of cleanup properties, as previously discussed. To generalise random input and randomly pick properties, μQC uses a seed.

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, μQC shows it for every generic test so the programmer can use that seed if he wants to reproduce the tests. Otherwise, μQC generates a random seed to assure random tests and a good test coverage.

μQC 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 coverage.

There may be cases where tests will end prematurely, after using only a few properties, because μQC 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.

Figure 4 μQC architecture overview



The last parameter from Figure 4 is the type of statistics shown at the end of the generic test. μQC 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.

4.2 Generator callbacks

Generators are callbacks that randomly produce input data for properties. A μQC generator consists of two callbacks: one to generate the data and the other one to print the data, as the C print f function needs different formats depending on the printed type. With this approach, the programmer can use complex data structures, such as trees. Because of that, the μQC equivalent of a generator is struct generator_printer.

μQC 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 μQC offers a wrapper which automatically generates new array types. The array type can have fixed or random size in a given range, depending on 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; at the same time, it can introduce subtle bugs if the user forgets to copy the array contents in case he needs it after the property ends.

Sometimes basic types may need additional features, such as a maximum or minimum value. μQC offers two solutions for this. The first one is that miscellaneous parameters can be added to 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 are 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.

4.3 Properties, preconditions and post-conditions

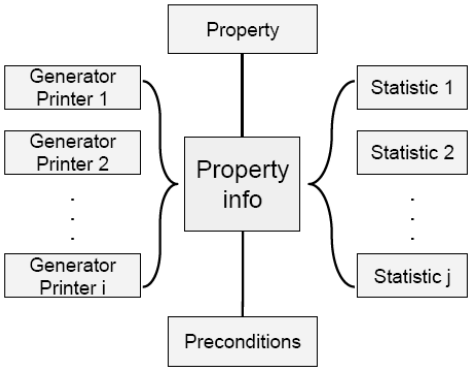
Since μQC supports stateful system testing, using a property requires the following steps: testing its preconditions, getting the values for the randomly generated data, applying its actions and testing its post-conditions.

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 callbacks, differently from the property callback. Because 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 post-conditions. Actions can be interleaved with their post-conditions, which are obtained from the formal specification of the API. As opposed to preconditions, post-conditions 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 post-conditions for an action or a group of actions as soon as possible, to reduce the debugging time necessary for finding the API call that made the system inconsistent.

Figure 5 Property info callbacks



In order to measure various metrics, *μ*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 investigate bugs and also measure the test coverage of elements that can be accessed during testing, either from the generated data or from the internal state of the system, if this data is accessible through API calls. By default, *μ*QC logs statistics regarding properties and their sequence. For every property, *μ*QC logs the number of total calls, the number of starting test property calls and again, for every other property, how many times it followed the current property. The default logging done by *μ*QC can be very useful to detect preconditions bugs and see if tests are surfacing the desired states. The user decides if he wants to see any of the two statistics category when the *μ*QC framework is called to generate and validate tests.

As can be seen in Figure 5, *μ*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 post-conditions. Having all of those callbacks, different components of generic tests become easier to integrate with each other. 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.

4.4 μQC test logic

Having all the previously discussed elements, *μ*QC can generate and run tests. Figure 6 shows a state machine with the actions taken by *μ*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.

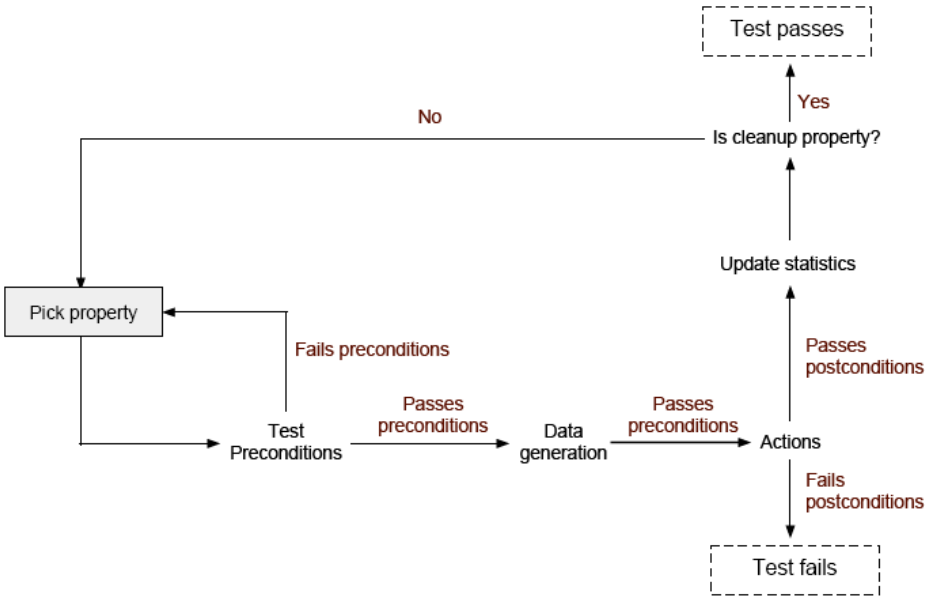
Starting a test, *μ*QC chooses a property, checks its preconditions (should they exist) and, based on the result, picks another property or continues with the current one. The

properties are randomly picked from the list of properties which were not tried in the current state of the system. Therefore, if in a given state there is no eligible property, either the preconditions specifications were too strict or the system is in an inconsistent state, meaning a bug was found. If the preconditions have passed, μQC generates test data, executes the actions and the post-conditions are checked. If any of the post-conditions fail, the testing is over, because μQC 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.

4.5 Implementation: porting μQC on an L4 microkernel

As previously mentioned, μQC runs as an user space application on top of an L4 microkernel. This means that actions made inside the properties are actually system calls. Other than that, porting the framework is easy as long as the environment supports the C POSIX library. Of course, specific properties, generators, preconditions, post-conditions and statistics need to be written for the system under test.

Figure 6 μQC test state machine (see online version for colours)



In order to automatically generate test cases, μQC uses the same approach as other property-based testing frameworks (see Section 3): it samples values from an input space using a random-number generator as a source. We assume that μQC may run in environments where high entropy is not always available for random number generation, and thus we rely on a pseudo-random generator seeded from any sources of randomness that the system programmer may be able to provide. μQC has a random module which

currently supports two implementations: the POSIX rand function and the Mersenne Twister PRNG. The default implementation is Mersenne Twister (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.

Internally *μ*QC uses a seed for randomising 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. We designed the *μ*QC interface to be similar the check framework. This makes it easier to use and requires little changes in unit tests to convert them to properties. 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 *μ*QC proof of concept evaluation and discussion

In order to validate *μ*QC as a practical solution for testing the API implementation of L4 microkernels, we analyse the following aspects:

- Its *effectiveness* at finding bugs in generic stateful systems – to show that property-based testing can be effectively used to find potential bugs, we devised an implementation of a simple circular queue and we specified its behaviour using *μ*QC. We artificially introduced bugs in its C code to determine whether the test generator will find implementation errors.
- The e*xpressiveness* of property-based testing with respect to L4's formal model – we implemented *μ*QC properties for a subset of the operations provided by VMXL4: thread creation, deletion and changing the thread scheduler priority. We found that general properties of these operations can be easily expressed as *μ*QC tests.
- The *engineering* effort involved in implementing a property-based tester for an L4 microkernel – we provide an empirical account of our experience with *μ*QC on top of VMXL4 and we discuss the potential challenges of providing a full property-based test suite.

The remainder of this section discusses these aspects in more detail. We begin by introducing a formal framework that describes the general approach to specifying system properties in *μ*QC. We then use this formalisation to illustrate two test scenarios: the simple circular queue initially devised to validate *μ*QC, and a proof-of-concept scenario to verify a subset of the L4 microkernel API.

At the end of the section we evaluate the engineering effort of developing *μ*QC and adding the new test scenarios. Additionally we discuss the trade-offs of property-based tests with respect to other software verification and validation methods.

5.1 μQC property formalisation

As described in Section 4, μQC takes as input a set of properties P and an arbitrary system state $\sigma \in \sum$, and it checks whether properties $p \in P$ hold in that state. A property is only checked if certain preconditions hold true in the given state, which may be interpreted as a form of existential quantification. However, in contrast to existential quantification, when a precondition does not hold, the property is vacuously considered to be true. This makes it possible to avoid states where the property is not applicable, e.g., popping an element from an invalid queue object.

Each μQC property quantifies universally over a set of states $\sum$ and a set of programmer defined inputs denoted I. In practice μQC randomly generates inputs from I and passes them as parameters to the property. Given that μQC is a stateful property-based testing framework, checking properties may have side effects, i.e., they may cause a system state transition from σ to a state σ'. For example if we want to verify how the size of a queue object changes when we add a new element to it, the enqueue operation will alter the queue.

$$(\text{pre}, \delta, \text{post})$$

where δ is a transition function from a state σ to a state σ', and pre and post are predicates quantified over $\sum$ and, optionally, I.

We write pre and post as functions:

$$\begin{aligned} &\text{pre} : \Sigma \times I \rightarrow \{0, 1\} \\ &\quad (\sigma, i) \mapsto \text{1logical expression}(\wedge, \vee, =) \\ &\text{post} : \Sigma \rightarrow \{0, 1\} \\ &\quad (\sigma') \mapsto \text{logical expression} \end{aligned}$$

where 0 is returned if the logical expression is false and 1 otherwise. Similarly, δ is a function:

$$\begin{aligned} &\delta : \Sigma \times I \rightarrow \Sigma \\ &\quad (\sigma, i) \mapsto \text{state transition expression} \end{aligned}$$

Generally, we say that a property P described by (pre, δ, post) holds $\Leftrightarrow \forall \sigma \in \sum, i \in I$ in which pre(σ, i) holds, if $\delta(\sigma', \text{i}) = \sigma'$, then post($\sigma'$) holds.

We note that in our design and implementation (Section 4) preconditions are handled such that generated inputs cannot invalidate them, i.e., if a precondition pre is parameterised over a set of inputs I, then inputs $i \in I$ will be generated so that the precondition will hold. This is necessary in order to eliminate the need to search for a valid input i in a large set I.

5.2 Circular queue proof of concept

The first program we used to validate μQC's effectiveness 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, the circular queue is a stateful system, with an internal representation for the queue, and it is a portable module

which can be used to validate *μ*QC against other property-based testing frameworks such as Haskell's QuickCheck (as described in Section 3).

The queue library implements an API which exposes the following four operations:

- init: $\mathbb{N} \to \sum$; $n \mapsto$ create a new queue of maximum sixe n
- enqueue: $E \times \sum \to \emptyset$; $(e, Q) \mapsto$ add an element e at the end of Q
- dequeue: $\sum \to \emptyset$; $Q \mapsto$ remove an element fro the beginning of Q
- front: $\sum \to E$; $Q \mapsto$ return the element from the beginning of Q.

where E is an arbitrary set of potential queue elements.

We implemented a test case for each of the operations. Note that in our specification, a queue Q and a state $\sigma \in \sum$ are equivalent, as the queue is the only stateful entity in the system. This yields the following four properties.

Property 1: Initialisation

$$\text{pre}_{\text{init}} : \sum \to \{0, 1\}$$
$$Q \mapsto \neg\text{valid}(Q)$$

$$\delta_{\text{init}} : \sum \times \mathbb{N} \to \sum$$
$$(Q, n) \mapsto Q', \text{ where } Q' = \text{init}(n)$$

$$\text{post}_{\text{init}} : \sum \to \{0, 1\}$$
$$Q' \mapsto, \text{valid } (Q') \wedge \text{size}(Q') = 0 \wedge \max \text{Size}(Q') = n$$

where Q, Q' $\in \sum$ are queue states, $n \in N$ is the maximum size of the queue, valid is a predicate that tests whether a given queue was previously initialised, size is a function that returns the number of elements in a queue and maxSize is a function that returns the maximum number of elements in a queue.

We make the observation that the definition of valid and the notion of 'valid queue' relate to the language semantics of C, in which we expect the result of init to be a non-NULL pointer. We also observe that n is a parameter of δ, so in practice *μ*QC verifies the property by generating arbitrary values of n and using them to create new queues.

We specify enqueue, dequeue and front similarly. Note that Properties 2, 3 and 4 are enough to cover the basic specification of the entire circular queue interface.

Property 2: Enqueuing.

$$\text{pre}_{\text{enqueue}} : \sum \to \{0, 1\}$$
$$Q \to \text{valid}(Q)$$

$$\delta_{\text{enqueue}} : \sum \times E \to \sum$$
$$(Q, e) \mapsto Q', \text{resulting from enqueue}(e, Q)$$

$$\text{post}_{\text{enqueue}} : \sum \to \{0, 1\}$$
$$Q' \mapsto \text{valid}(Q') \wedge n = n' \wedge \big((s = n \wedge s = s') \vee (s < n \wedge s' = s + 1)\big)$$

where s = size(Q), s' = size(Q'), n = maxSize(Q) and n' = maxSize(Q').

Property 3: Dequeuing

$$\text{pre}_{\text{enqueue}} : \Sigma \to \{0, 1\}$$
$$Q \to \text{valid}(Q)$$

$$\delta_{\text{enqueue}} : \Sigma \to \Sigma$$
$$Q \mapsto Q', \text{ resulting from dequeue}(Q)$$

$$\text{post}_{\text{enqueue}} : \Sigma \to \{0, 1\}$$
$$Q' \mapsto \text{valid}(Q') \wedge n = n' \wedge \big((s = 0 \wedge s = s') \vee (s < 0 \wedge s' = s - 1)\big)$$

where s = size(Q), s' = size(Q'), n = maxSize(Q') and n' = maxSize(Q').

Note that each of Properties 2 and 3 capture in fact multiple aspects of the circular queue:

1 maxSize is invariant under enqueue and dequeue

2 size varies only between 0 and maxSize

3 enqueue and dequeue will fail when $s = n$, and respectively when $s = 0$.

Property 4: Peeking the front of the queue

$$\text{pre}_{\text{front}} : \Sigma \to \{0, 1\}$$
$$Q \to \text{valid}(Q) \wedge s > 0$$

$$\delta_{\text{front}} : \Sigma \to \Sigma$$
$$Q \mapsto Q$$

$$\text{post}_{\text{front}} : \Sigma \to \{0, 1\}$$
$$Q' \mapsto e = \text{front}(Q) \wedge e \in E \wedge \text{exprected}(e)$$

where s = size(Q), e is the return value of front(Q), E is an arbitrary set of potential elements from Q and expected is a predicate that tests the value of e against the expected value at the front of the queue.

We implemented expected by using a reference queue in the implementation of each of Properties 1 to 4: we initialise a simple array of random size and randomly generated elements to be enqueued/dequeued in Q. We hold two pointers to the array, representing the back and the front of the queue respectively. We update the two pointers on each application of Properties 2 and 3 and we check the value of front(Q) against the front value in the array.

To test the circular queue implementation we instructed the framework to automatically generate a large number of test cases with a different number of properties and different sequences of properties, each with automatically generated different inputs. As can be observed, the difference in the code logic complexity is not significantly different for unit testing and property-based testing, but the benefits of the latter are significant. When different bugs were introduced on purpose in the queuing logic, μQC detected all of them, using only the four properties.

For example we artificially introduced a bug in the circular queue implementation and tested it using μQC. The bug consists of a corner case where on an enqueue operation, the check for maxSize is omitted. The test fails when checking $post_{enqueue}$; more specifically, both the assertions size(Q) = maxSize(Q) and size(Q) < maxSize(Q) fail. After fixing the bug, μQC validates the implementation and reports that all the tests are passed.

5.3 Tests for L4 thread operations and scheduling

As shown in related work (Kolanski and Klein, 2006; Elkaduwe et al., 2008), the L4 microkernel API is small enough to be described using formal modelling. Unlike monolithic kernels, L4 microkernels provide a minimal set of abstractions for users: threads, scheduling, inter-process communication and memory management. The existing testing infrastructure already provides unit tests for each of these mechanisms.

Modern L4 APIs are based on capabilities (Lackorzynski and Warg, 2009). From the user's point of view, kernel objects (e.g., threads, communication end-points) can be addressed via L4 system calls using tokens named *capability pointers*, which are mapped at run-time to the objects themselves by the microkernel. Although we did not develop any properties related to capabilities, we had to use the latter to keep track of some of the kernel objects that were created or deleted at run-time.

As an exercise in testing the L4 API using μQC, we chose to look at the following aspects regarding threads and scheduling:

- Thread creation should succeed when invoked on a valid 'empty' capability (a slot in a capability space that does not contain a capability). Conversely, thread deletion should succeed when invoked on a valid capability to a thread.
- Thread creation should fail when invoked on a 'non-empty' capability (a slot that already addresses a kernel object). Conversely, thread deletion should fail when it is given an 'empty' capability.
- Thread creation and deletion should fail when invoked on a non-'valid' capability pointer, i.e., a pointer to a non-existing slot in a capability space. For the sake of simplicity, we only stated this property for thread creation, and additionally we referred to capability pointers as indices in lists (which could possibly go out-of-bounds), although microkernels such as seL4 implement hierarchical addressing schemes using guarded page tables (Liedtke and Elphinstone, 1996; Elkaduwe et al., 2008).
- Since L4 does strict priority scheduling (Ruocco, 2008), higher-priority threads should always run before lower-priority threads.

We implemented μQC properties for these particular mechanisms because they are straightforward and easy to understand and implement by system programmers with a small degree of familiarity with L4 microkernels. However, we think that the formalisation below can be easily extended to other L4 microkernel abstractions; or full formalisations such as the one in Kolanski and Klein (2006) can be potentially reused. We discuss this aspect in more detail at the end of this section.

We denote threads as $\tau_1, \tau_2, \ldots, \tau_i \in T$, and capability pointers as $c_1, c_2, \ldots, c_i \in C$; we refer to non-existing threads using the special thread $\phi \in T$ (a so-called 'nil thread'). We refer to thread creation and deletion using the operations $e = C(c_i)$ and $e = D(c_i)$

respectively, and setting a thread's scheduler priority as $e = S(c_i, p)$. $e \in \varepsilon$ is an error condition and p is a positive number denoting a thread's priority.[6]

Additionally we denote the validity of a capability c (as per the previous definition of 'valid') by considering a subset $C_v \subseteq C$, such that $\forall_c \in C_v$, a predicate $v(c)$ holds. Given a valid capability c that points to a thread τ, we denote this as $p(c, \tau)$. In this paper we assume that any valid capability pointer points either to an existing thread τ or to the 'nil thread', i.e., an empty capability slot, denoted as $\tau = \phi$.

In Properties 5 to 8 we use $\sigma \in \sum$ to refer to the state of the L4 microkernel. All the previously defined functions implicitly depend on the microkernel state, e.g., $v(c)$ is in fact a function $v(c, \sigma)$, as a capability may be valid in a given state, but invalid in another. Since we do not have access to the internal state of the microkernel, we implement $v(\cdot)$ and $p(\cdot, \cdot)$ as part of the test suite and we keep track of each thread creation/deletion operation. To avoid ambiguities in the formal language, we explicitly quantify σ in pre, post and δ, but we leave functions implicitly parameterised by the state quantified in the predicate/function.

Property 5: Thread creation/deletion (1).

$$\begin{aligned}
&\text{pre}_{\text{thread1}} : \sum \times c \to \{0, 1\} \\
&\qquad (\sigma, c) \mapsto v(c) \\
&\delta_{\text{thread1}} : \sum \times c \to \sum \\
&\qquad (\sigma, c) \mapsto \sigma', \text{derived from}: \\
&\qquad\qquad \text{if } p(c, \phi) \text{ then } e = C(c) \text{ else } e = D(c) \\
&\text{post}_{\text{thread1}} : \sum \to \{0, 1\} \\
&\qquad Q' \mapsto e = \text{OK}
\end{aligned}$$

That is, given a valid capability pointer c and a thread τ, and $p(c, \tau)$, then either $\tau = \phi$ and $C(c)$ succeeds, or $\tau \neq \phi$ and $D(c)$ succeeds. In both cases, e = OK.

Property 6: Thread creation/deletion (2).

$$\begin{aligned}
&\text{pre}_{\text{thread2}} : \sum \times c \to \{0, 1\} \\
&\qquad (\sigma, c) \mapsto v(c) \\
&\delta_{\text{thread2}} : \sum \times c \to \sum \\
&\qquad (\sigma, c) \mapsto \sigma', \text{derived from}: \\
&\qquad\qquad \text{if } p(c, \phi) \text{ then } e = C(c) \text{ else } e = D(c) \\
&\text{post}_{\text{thread2}} : \sum \to \{0, 1\} \\
&\qquad Q' \mapsto e = \text{InvalidThread}
\end{aligned}$$

That is, given a valid capability pointer c and a thread τ, and $p(c, \tau)$, either $\tau = \phi$ and $D(c)$ fails, or $\tau \neq \phi$ and $C(c)$ fails. In both cases, e = InvalidThread.

In the context of Properties 5 and 6 we make the observation that InvalidThread is a generic error, and it may be returned when $v(c)$ holds. This assumes that at run-time the system designer knows for example the structure of the capability space, and can distinguish between invalid and valid but improperly used capability pointers.

We also remark that Properties 5 and 6 could have been decomposed into four properties, or defined altogether as a single property. We chose to group them by the expected error condition for convenience reasons.

Property 7 specifies the expected behaviour for invalid threads.

Property 7: Thread creation/deletion (3).

$$\text{pre}_{\text{thread3}} : \sum \times c \to \{0, 1\}$$
$$(\sigma, c) \mapsto v(c)$$

$$\delta_{\text{thread3}} : \sum \times c \times \{0, 1\} \to \sum$$
$$(\sigma, c, i) \mapsto \sigma', \text{ derived from :}$$
$$\text{if } i = 0 \text{ then } e = C(c) \text{ else } e = D(c)$$

$$\text{post}_{\text{thread3}} : \sum \to \{0, 1\}$$
$$Q' \mapsto e = \text{Invalid Thread}$$

That is, given a capability pointer c such that $v(c)$, both $C(c)$ and $D(c)$ fail with e = InvalidThread. $C(\cdot)$ and $D(\cdot)$ are selected based on an input $i \in \{0, 1\}$.

Although Properties 5, 6 and 7 were initially implemented as generalisations of existing unit tests, running them together helped us find a bug in the user space L4 thread utility library, more specifically in the thread creation function. Creating a thread using the utility library from within μQC automatically helped us detect that the library function was not checking the return value of the thread creation kernel system call, and so it was always returning OK. Thus in a μQC run in which $\text{post}_{\text{thread1}}$ was followed by $\text{post}_{\text{thread2}}$, the latter failed because it was expecting the utility function to return InvalidThread. Given that in L4 a thread's stack is allocated by another user space thread (in our case, the tester's thread), this also helped us discover that multiple stacks would be created by the utility function even when thread creation failed. This is a potentially dangerous bug, as such behaviour can lead to the situation where the system runs out of memory and ultimately crashes.

Finally, in Property 8weillustrate a simple case in strict priority scheduling, consisting of two threads of potentially different priorities which are simultaneously runnable on a single- CPU system. For this purpose, we introduce a predicate runsBefore: $T \times T \to \{0, 1\}$; runsBefore (τ_1, τ_2) implicitly performs a temporal check of the order in which τ_1 and τ_2 run and holds whenever τ_1 has at a given moment 'run before' τ_2. As any thread τ is accessible from user space only through its capability c, we quantify it using $p(c, \tau)$.

Property 8: Ordering in strict priority scheduling

$$\text{pre}_{\text{sched}} : \sum \times c \times c \to \{0, 1\}$$
$$(\sigma, c_1, c_2) \mapsto v(c_1) \wedge p(c_1, \tau_1) \wedge p(c_2, \tau_2) \wedge$$
$$\tau_1 \neq \phi \wedge \tau_2 \neq \phi \wedge \tau_1 \neq \tau_2$$

$$\delta_{\text{sched}} : \sum \times c \times c \times \mathbb{N} \times \mathbb{N} \to \sum$$
$$(\sigma, c_1, c_2, p_1, p_2) \mapsto \sigma', \text{ derived from :}$$
$$e_1 = S(c_1, p_1); e_2 = S(c_2, p_2)$$

$$\text{post}_{\text{sched}} : \sum \times c \times c \times \mathbb{N} \times \mathbb{N} \to \{0, 1\}$$
$$(\sigma', c_1, c_2, p_1, p_2) \mapsto e_1 = \text{OK} \wedge p(c_1, \tau_1)\, p(c_2, \tau_2) \wedge$$
$$\Big(\big(p_1 \geq p_2 \wedge \text{runsBefore}(\tau_1, \tau_2) \big) \vee$$
$$\big(p_1 \leq p_2 \wedge \text{runsBefore}(\tau_2, \tau_1) \big) \Big)$$

That is, given two distinct threads τ_1 and τ_2, $\tau_1 \neq \phi$ and $\tau_2 \neq \phi$, and two capabilities c_1 and c_2 such that $p(c_1, \tau_1)$ and $p(c_2, \tau_2)$, then for any p_1, p_2, $S(c_1, p_1)$ and $S(c_2, p_2)$ will succeed (e = OK) and one of these events will occur:

- if $p_1 > p_2$ then τ_1 will *run before* τ_2
- if $p_1 < p_2$ then τ_2 will *run before* τ_1
- otherwise τ_1 and τ_2 can run in any order, depending on the order they have been placed in the kernel's scheduling queue.

Note that this property might not hold on multi-processor systems where τ_1 and τ_2 could be scheduled on different processors; we take this scenario into account in μQC by explicitly pinning threads to a given CPU. We implement runsBefore by simply setting a variable that is shared between τ_1, τ_2 and the tester's thread: both threads synchronise with the tester, after which the tester checks the variable's value to determine which thread set it last.

5.4 Test engineering effort and limitations

First we detail the effort invested into developing μQC itself. The work in Section 4 required about six man-weeks of full-time involvement. This includes the initial design, porting the original QC on top of VMXL4, integrating various components such as a random number generator and enhancing μQC to fully support stateful testing and statistics collection. As VMXL4 is a proprietary microkernel and requires extra time to study, we consider the development of the μQC framework to be a success.

Writing the circular queue implementation and its properties required approximately one man-day. Writing the L4 API proof of concept properties was done in approximately seven man-days. A large part of this time was spent fixing bugs found by μQC either in user space libraries or in the test cases themselves; the latter was needed due to differences in assumptions about the system state in unit testing and property-based testing. Because the properties described in this section were fairly simple, no bugs were found in the microkernel itself.

We argue that this work can be extended to encompass a significant portion of L4's formal model with a moderate amount of effort. Further enhancements could be brought to μQC to support more elaborate preconditions and thus combine properties from different test cases (e.g., scheduling, threading and inter-process communication, memory mapping, address space management and thread management), resulting in complex run-time scenarios to stress-test the microkernel.

We show that μQC allows a flexible approach to verification, by allowing us to test a partial specification of the L4 microkernel. This flexibility extends to the properties themselves: other automated methods such as model checking and symbolic execution greatly depend on how system properties are specified, e.g., certain approaches to

specification may cause state explosion; in contrast, property-based testing does not rely on a particular approach for specifying properties, although we acknowledge that some approaches could cause properties to never be applied, e.g., if a precondition is never satisfied. In this case the trade-off is that a subset of the system states may never be reached at run-time, which would potentially prevent rare bugs from being found.

Another trade-off of μQC is the granularity of tests: while other approaches may for example permit reasoning about local variables in functions, property-based testing is a so-called 'black-box' approach, and thus we cannot explicitly verify the internal state of a function that we put under test. In the case of the L4 microkernel API, and particularly for the properties specified in this section, we mitigated this issue by maintaining 'shadow state', e.g., the testing framework maintained its own list of capabilities and their state of being valid or empty. We think this is a potential challenge in scaling property-based testing to the entire L4 API, and we will address it in future work.

Finally, since property-based testing is a dynamic analysis method and the system under test is a black box, we currently have no way of rigorously determining the level of quality assurance offered by our approach, e.g., which parts of the microkernel state space were reached by μQC. In the future we plan to devise metrics that allow us to visualise the impact of a test run on finding bugs and/or how many runs are needed for the testing to be statistically significant in terms of the assurance it provides.

6 Conclusions and further work

Software quality assurance is a challenging and often costly task that scales poorly with system complexity. This aspect carries more weight in critical systems, where the absence of errors needs to be guaranteed by the system designer. Moreover, even non-critical software containing bugs such as security vulnerabilities can incur major financial costs for companies.

This paper is built on the premise that the core component of any complex software system, its kernel, requires a testing method that relies on sound design principles, but on the other hand can be applied in a flexible manner. We adhere to the idea that property-based testing is such a solution and thus we set to use it to test the API of an L4 microkernel.

We designed and implemented μQC, a property-based testing framework for L4 microkernels. Unlike existing test suites for L4, which verify the most common use cases, μQC uses properties, a powerful abstraction that allows tests to cover most of the input space.

μQC 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, μQC offers support for generating any data type, using predefined generators which can be combined to obtain new test data generators.

We used μQC to provide a proof of concept on the top of the VMXL4 microkernel. We implemented a minimal test scenario to validate μQC itself, then we built a number of properties that cover a subset of the L4 API. We showed that μQC is an effective tool for

finding bugs, and that formally defined properties can be easily implemented as μQC tests.

In the future we aim to enhance μQC's bug-finding capabilities, by providing shrinking for failing tests. We also believe μQC should be able to handle more complex scenarios that test the interworking between the various L4 microkernel mechanisms. Finally, we wish to be able to use μQC to reason about the level of assurance provided by the test scenarios generated using property-based testing, and, ideally, automatically provide guarantees regarding the absence of bugs in L4 microkernels.

References

Arts, T. and Hughes, J. (2003) 'Erlang/quickcheck', in *Ninth International Erlang/OTP User Conference*.

Baumann, C., Beckert, B., Blasum, H. and Bormer, T. (2009) 'Formal verification of a microkernel used in dependable software systems', in *International Conference on Computer Safety, Reliability, and Security*, Springer, pp.187–200.

Brucker, A.D., Havle, O., Nemouchi, Y. and Wolff, B. (2015) 'Testing the IPC protocol for a realtime operating system', in *Working Conference on Verified Software: Theories, Tools, and Experiments*, Springer, pp.40–60.

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

Claessen, K. and Hughes, J. (2002) *Testing Monadic Code with QuickCheck* [online] http://www.cs. tufts.edu/~nr/cs257/archive/john-hughes/quick.pdf (accessed 29 August 2017).

Claessen, K. and Hughes, J. (2011) 'Quickcheck: a lightweight tool for random testing of haskell programs', *ACM Sigplan Notices*, Vol. 46, No. 4, pp.53–64.

Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W. and Tobies, S. (2009) 'VCC: a practical system for verifying concurrent c', in *International Conference on Theorem Proving in Higher Order Logics*, Springer, pp.23–42.

Elkaduwe, D., Klein, G. and Elphinstone, K. (2008) 'Verified protection model of the sel4 microkernel', in *Verified Software: Theories, Tools, Experiments*, Springer, pp.99–114.

Elphinstone, K. and Heiser, G. (2013) 'From l3 to sel4 what have we learnt in 20 years of l4 microkernels?', in *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, ACM, pp.133–150.

Fernandez, J-C., Mounier, L. and Pachon, C. (2004) 'Property oriented test case generation', in *Formal Approaches to Software Testing*, Springer, pp.147–163.

Fink, G. and Bishop, M. (1997) *Property-Based Testing; A New Approach to Testing for Assurance* [online] http://nob.cs.ucdavis.edu/bishop/papers/1997-sen/ pbt.pdf (accessed 29 August 2017).

Hoffmann, S., Haugou, G., Gabriele, S. and Burdy, L. (2007) 'The b-method for the construction of microkernel-based systems', in *International Conference of B Users*, Springer, pp.257–259.

Holm, P. (2016) *The FreeBSD Kernel Stress Test Suite* [online] https://people.freebsd.org/ ~pho/stress/ (accessed 29 August 2017).

Klein, G., Andronick, J., Elphinstone, K., Murray, T., Sewell, T., Kolanski, R. and Heiser, G. (2014) 'Comprehensive formal verification of an OS microkernel', *ACM Transactions on Computer Systems* (*TOCS*), Vol. 32, No. 1, p.2.

Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M. et al. (2009) 'sel4: formal verification of an OS kernel', in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, ACM, pp.207–220.

Kolanski, R. and Klein, G. (2006) 'Formalising the l4 microkernel api', in *Proceedings of the 12th Computing: The Australasian Theory Symposium*, *Australian Computer Society, Inc.*, Vol. 51, pp.53–68.

Koopman, P. and DeVale, J. (1999) 'Comparing the robustness of posix operating systems', in *29th Annual International Symposium on Fault-Tolerant Computing*, *Digest of Papers*, IEEE, pp.30–37.

Kuznetsov, V., Chipounov, V. and Candea, G. (2010) 'Testing closed-source binary device drivers with ddt', in *USENIX Annual Technical Conference*, EPFL-CONF-147243.

Lackorzynski, A. and Warg, A. (2009) 'Taming subsystems: capabilities as universal resource access control in l4', in *Proceedings of the second Workshop on Isolation and Integration in Embedded Systems*, ACM, pp.25–30.

Larson, P. (2002) 'Testing linux® with the linux test project', in *Ottawa Linux Symposium*, p.265.

Levy, H.M. (1984) *Capability-Based Computer Systems*, Butterworth-Heinemann, Newton, MA, USA.

Liedtke, J. (1995) 'On micro-kernel construction', *ACM*, Vol. 29, No. 5, pp.237–250.

Liedtke, J. and Elphinstone, K. (1996) 'Guarded page tables on mips r4600 or an exercise in architecture-dependent micro optimization', *ACM SIGOPS Operating Systems Review*, Vol. 30, No. 1, pp.4–15.

Machado, P.D., Silva, D.A. and Mota, A.C. (2007) 'Towards property oriented testing', *Electronic Notes in Theoretical Computer Science*, Vol. 184, No. 1, pp.3–19.

Manea, V., Carabas, M., Mogosanu, L. and Gheorghe, L. (2015) 'Native runtime environment for internet of things', in *Advanced Computational Methods for Knowledge Engineering*, 11–13 May, pp.381–390, Springer, Metz, France.

Matsumoto, M. and Nishimura, T. (1998) 'Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator', *ACM Transactions on Modeling and Computer Simulation* (*TOMACS*), Vol. 8, No. 1, pp.3–30.

Merino, J. (2013) *Introducing the FreeBSD Test Suite* [online] http://julio.meroh.net/2013/12/introducing-freebsd-test-suite.html (accessed 29 August 2017).

Mogosanu, L., Carabas, M., Deaconescu, R., Gheorghe, L. and Voiculescu, V.G. (2016) 'VMXHAL: a versatile virtualization framework for embedded systems', *Journal of Control Engineering and Applied Informatics*, Vol. 18, No. 1, pp.68–77.

Mullender, S.J. and Tanenbaum, A.S. (1986) 'The design of a capability-based distributed operating system', *The Computer Journal*, Vol. 29, No. 4, pp.289–299.

Nilsson, R. (2014) *ScalaCheck: The Definitive Guide*, 1st ed., Artima Press.

Odersky, M. (2010) 'Contracts for scala', in *Runtime Verification*, 1–4 November, pp.51–57, Springer, St. Julians, Malta.

Pennebaker, A. (2012) *AC Port of the QuickCheck Unit Test Framework* [online] https://github.com/mcandre/qc (accessed 29 August 2017).

Revuelta, M., Espada, P., Tejedor, I. and Sánchez, A. (2006) 'Simulation environment for testing and verifying the l4 microkernel mapping database', in *Proceedings of the 5th WSEAS International Conference on Software Engineering, Parallel and Distributed Systems*, World Scientific and Engineering Academy and Society (WSEAS), pp.19–23.

Ruocco, S. (2008) 'A real-time programmer's tour of general-purpose l4 microkernels', *EURASIP Journal on Embedded Systems*, Vol. 11, No. 1, Article ID 234710, 14pp.

Schierboom, E. (2007) *Verification of FiascoâŁ™s IPC implementation*, PhD thesis, Masteral thesis, Radboud University, Computing Science Department.

Shapiro, J.S., Smith, J.M., and Farber, D.J. (1999) 'EROS: a fast capability system', ACM, Vol. 33.

Valmari, A. (1998) 'The state explosion problem', in *Lectures on Petri nets I: Basic Models*, pp.429–528, Springer.

Verbeek, F., Havle, O., Schmaltz, J., Tverdyshev, S., Blasum, H., Langenstein, B., Stephan, W., Wolff, B. and Nemouchi, Y. (2015) 'Formal api specification of the pikeos separation kernel', in *NASA Formal Methods Symposium*, 27–29 April, pp.375–389, Springer, Pasadena, CA, USA.

Notes

1 https://hackage.haskell.org/package/QuickCheck

2 http://www.quviq.com/products/erlang-quickcheck/

3 https://www.scalacheck.org/

4 https://github.com/nivox/quickcheck4c

5 https://github.com/silentbicycle/theft

6 In L4 priorities are numbers between 0 and 255, with 0 representing the lowest priority (that of the Kernel's idle thread) and 255 the highest. In our representation of the properties we assign priorities between 100 and 200, due to other priority ranges being used for system services such as the console, interrupt threads, etc.