Experiences from testing a radiotherapy support system with QuickCheck

Aiko Fallas Yamashita[1], Andreas Bergqvist[2], and Thomas Arts[2]

[1] Simula Research Laboratory, Box 134, 1325 Lysaker, Norway
aiko@simula.no
[2] IT University of Göteborg, Box 8718, 402 75 Göteborg, Sweden
{bergqvia, thomas.arts}@ituniv.se

Abstract. We present a case study on the use of lightweight formal methods for testing part of a real-time organ position tracking system used in radiotherapy. Several properties were modeled and verified through automated test cases generated by QuickCheck. QuickCheck was found useful in reducing the complexity inherent to testing medical devices by detecting faults at system level, and assisting in the exploration of atypical errors that could later be analyzed and fixed in the system. We suggest that a combination of lightweight formal methods and random test generation, supported by automated simplification of test cases may represent a feasible option in the medical domain; particularly for those projects with high-pace development, a need for proof-based techniques/tools for certification processes, and when the non-deterministic nature of real-time devices demands the exploration/identification of heterogeneous fault sources.

Keywords: Lightweight formal methods, model-based testing, medical software, software verification, software testing.

1 Introduction

With the increased use of software in medical devices, high demands on software verification and analysis in the medical domain are inevitable. In many cases formal methods are used to model medical devices [1], medical protocols [2], or even entire systems [3]. Proofs are becoming an important aspect in medical device certifications as organizations like the Food and Drugs Administration (FDA) [4] and its European counterpart, Medical Device Directive (MDD) [5] are moving from process-centered towards proof-based certification [6].

Nevertheless, full formalization of systems implies potentially high costs [7] and, in some industrial contexts, it may constitute an unrealistic task. Yet after the correctness of a model has been formally proven, its implementation still needs to be tested. A combination comprising of lightweight formal methods and testing has been proposed as a means for connecting the actual implementation and the formal properties of the system in a feasible and more direct way, avoiding errors in implementation details [8]. Studies addressing the usage of lightweight formal methods within different industrial contexts can be found in current literature [9–12].

In this article we present a case study on the use of lightweight formal methods for testing an implementation of a medical device. The device (constructed by Micropos Medical [13]) is a real-time organ position tracking system used in radiotherapy, i.e., a tumor is positioned in real-time in order to be able to accurately deliver a dose of radiation. Part of this system was tested with QuickCheck [14] as a master thesis project.

The software to determine the position of the organ is critical, since that area receives radiation during therapy; healthy tissue should not be destroyed unnecessarily. Proving correctness of the software is extremely hard; radio signals are combined in order to determine a position. Imagine a radio antenna of which the top determines the position. Even if the top is fixed, the antenna can be in many different positions, all resulting in different radio signals. Even worst, the same antenna position not necessesarily results in exactly the same signals each time one measures. Thus, we have deterministic software that provides a lot of computations on radio signals and returns a measured position. However, the software reacts very non-deterministic in a test setting, since placing the antenna in a certian position, may result in different signals and therefore slightly different measured positions. The algorithms for computing the measured position are under constant impovement, still we want to be able to ensure accuracy in any product that is released.

QuickCheck is a testing tool that randomly generates test cases from a given model and supports automated simplification of failed test cases. Thus, in theory we are able to generate test cases from the same specification that is used in to formulate and prove correctness of the system. The study aims to explore the potential contributions as well as the challenges of this specific set of techniques (i.e. lightweight formal methods and random testing supported by automated test case simplification) in testing a safety critical medical device, in order to assess the viability of lightweight methods within the medical domain.

The paper is subsequently organized as follows: Sect. 2 introduces briefly the context for medical device verification, proposing the necessity of integrating test tools and formal methods in the medical industry. Sect. 3 details related work to the approach used in the case study. Sect. 4 provides the motivations for our approach, and a description of the case study; indicating the testing setup, and the properties tested in the SUT. Sect. 5 describes the results and analysis derived from this study. Finally, Sect. 6 specifies the conclusions reached based on this study.

2 Verifying medical devices

Two primary approaches to the process of medical devices delivery are utilized: process centered verification and artifact-centered verification [15]. Process centered verification is often described by standards that suggest a series of practices for the medical practitioners to base the development of the safety-critical software products on [16, 17]. However, there is a need for more artifact-centered approaches as medical devices turn more and more sophisticated, complex and wide-ranging; and general guidelines are proving to be insufficient for delivering a safe product [18, 15].

The application of formal methods in medical devices supports this line of action and could add significant confidence in the system by revealing errors in both the system's model and its implementation [19]. There are many success stories regarding the use of formal methods in the medical domain, which range from medical protocols [2, 20] to medical equipment controllers [21, 3] and medical devices [1]. Studies elucidate the need of well-established processes that include formal methods and ensure safe systems [22]. Despite the numerous advantages of formal methods, the actual implementation still needs to be tested given the differences that could exist between the model and the implementation. Furthermore, a testing process is a "must" in current certification processes regulated by medical authorities [23–25].

In our case study we had a mathematical formula to compute the difference between a position and a measured position. Given such a clear formal model, we liked to investigate how we could automatically generate tests from that model. What is needed is a connection between mathematics and the software in the implementation. We looked for a general approach instead of a specific solution for this case, in order to be able to apply the method to other parts of the system later on.

3 Related work

Our approach is based on a tool called QuickCheck, which is a property-based testing tool that automatically generates tests from a specification and has the ability to simplify failing test cases (or counter examples) automatically. Properties are modeled and used as input in QuickCheck in order to generate random test cases. Although we use the term property-based, it is clear that QuickCheck can be seen as a *model-based testing* tool, since it relies on a formal abstraction of a system property in order to generate the test cases. Another definition that may be applicable is *specification-based testing*, as we specify several aspects of the system in the form of properties. In the subsequent text, we describe related work in the area of testing (such as model-based testing, specification-based testing, boundary-based testing, on-the-fly testing) as well as other associated approaches in the field of formal methods (e.g. model checking).

Model-based testing (MBT) [27] is an approach that bases common testing tasks such as test case generation and test result evaluation on a model of the system. Examples of MBT can be found in [28–30]. Specification-based testing on the other hand, tries to demonstrate that an implementation conforms to a certain specification of a system. Some examples are [31–38]. Both approaches can be considered orthogonal and most of the time well complemented with formal methods. An attempt to establish a taxonomy for MTB can be found in [39]. Boundary-driven testing as well as coverage-oriented testing are approaches that can be found together with MBT and Formal Methods. Boundary-driven testing selects values that are directly on, above, and beneath the edges of the legal input and output values. In contrast to random testing, boundary testing may require some expertise in order to select effective boundary cases [40]. Examples of boundary-driven approaches can be found in [41, 42]. A study addressing a combination of MBT and the coverage-oriented approach can be found in [43].

Model-based testing and Specification-based testing differ as white-box testing and black-box testing. Given a model, one has insight in border cases that occur during runtime (or simulation-time), whereas specifications only give borders that can be specified and do not consider borders that occur during runtime. For example, a specification could state that input values should be in a certain range, a model of the computation of could reveal that a buffer is overflow for specific input values, not necessarily the border cases of the given range. We used QuickCheck in a Specification-based approach, trusting that random generation of sufficiently many values would reveal errors. We are not actively looking for border cases, since many of these cases are caused by how well the radio signals can be interpreted, which cannot be read from the source code of an algorithm or its model. Moreover, the algorithm for computing the values is constantly improving and changing, therefore, the model would have to be re-designed after each such change.

The properties we checked with QuickCheck were somehow side-effect free. We send an antenna to a certain position, measure the position and compare with the actual position. In that sense, we differ from a model checking approach, where one may check that certain properties hold in a state, no matter which path one chooses to get to that state. In theory, no matter from which direction we reach a point in space, the computation should always return the same value for the same position. This is, though, not true, but that is not because of non-deterministic software, but because of radio signals not being deterministic.

Some may classify QuickCheck's approach under the rubric on-the-fly test generation, since it generates random test cases and verifies properties on the fly. On-the-fly test generation has been used before in the verification of real-time communication protocols [44]. This approach is considered suitable for real-time systems, specially when the test case generation can react to the actual outputs of the SUT while running under the operation environment (Further on we will explain how QuickCheck does this through the simplification of failed test cases). Since real-time systems are characterized as being non-deterministic, offline testing (the opposite approach to on-the-fly testing) is limited in its capacity to react to changes in the environment and identify faults that are linked to such changes.

4 Case study

This section provides the details of the study. We start by describing the medical SUT. Secondly, we present a description of QuickCheck and motivate our approach for testing. Subsequently, we present the properties tested along with a description of the testing set-up.

4.1 Position tracking device

The SUT is called 4DRT (Four Dimension Radio Therapy) and is a real-time organ position tracking system intended for supporting radiotherapy. It is able to locate the position of an organ in four dimensions, three-dimensional space and time. This enables one to monitor the position of tumors in prostate cancer patients and thereby helps to improve the accuracy of the radiation during radiotherapy treatments.

The SUT is based on radio frequency transmission. The measurement of the position is done through an implantable device in the organ (or nearby), which acts as a transmitter. The transmitter emits a radio frequency, which is captured by multiple receivers, typically arranged in a plate on the treatment table under the patient (See Fig. 1). The software uses the signal captured by the receivers as input to calculate the position of the organ. A set of floating-point values (representing the measured signals) is continuously sent to the software. The software maps the floating-point values received from the Receivers to a specific coordinate position in the real world. This coordinate position is given in a coordinate system specific to the SUT, which has a predetermined range for each axis (X, Y, Z) and two angles (Vy, Vz), i.e., rotation over y and z-axes. The "mapping process" or the algorithm for calculating the position uses a mathematical model not discussed here. The non-functional requirement described in Table 1 explicates the accuracy required, and it is expressed in terms of confidence intervals; i.e., positions calculated by the SUT should be within a radial distance (in Euclidean space) of 2 ± 1mm from the actual position to ensure that the tumor receives radiation and not the healthy tissue around it. That is, the calculated position should not divert from the actual position with more than 2mm with a standard deviation of 1mm.

Table 1. Description of the functional requirement (and its corresponding non-functional requirement) tested in the SUT

Functional requirement:	The software component should calculate the 5D positioning of the transmitter (X, Y, Z, Vy, and Vz, where Vy is rotation around Y axis and Vz is rotation around Z axis)
Non-functional requirement:	The system should achieve 3D difference or radial accuracy of 2 ± 1 mm

The software of the SUT is the result of migrating a prototype from LabView [47] to a commercial platform language i.e., Microsoft .NET C#. Pseudo-code describing the underlying algorithm for position estimation and the LabView

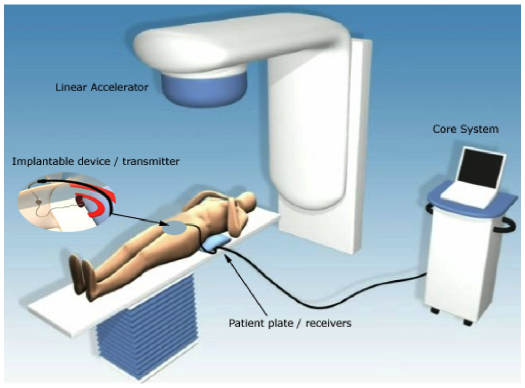

Fig. 1. View of the 4DRT system in a treatment environment. Elements such as the Linear Accelerator, the implantable device, and the patient plate are depicted.

code were used for performing the migration. Even if the equivalence of the algorithm implementation (between LabView and C#) can be reviewed through code inspection, it is still a challenge to ensure correct behavior during its execution. Micropos Medical was mainly interested in a solution that may enable testing in real life conditions and identify problems at system level. Due to signal fluctuations that are dependent on the environment, on-site calibration is also required. Micropos needs a cost-effective solution for performing system level testing on a regular basis during development and after deployment.

4.2 QuickCheck and the proposed approach

QuickCheck is a tool that combines random test generation, with a flexible language for specifying generators and the use of properties to verify test results [8]. The properties can be written in a restricted logic, and then QuickCheck can be invoked to test the property in a large number of cases. Properties are specified in Erlang [48]. Among other things, one can quantify over sets and express preconditions. For example, the property

```
prop_positive() ->
   ?FORALL({Pos1,Pos2},{coordinate(),coordinate(),
      ?IMPLIES(Pos1 =/= Pos2,
               radial_distance(Pos1,Pos2) > 0)).

radial_distance({XP,YP,ZP},{XC,YC,ZC}) ->
   math:sqrt(
        math:pow(XP-XC,2)+math:pow(YP-YC,2)+math:pow(ZP-ZC,2)).

coordinate() ->
   {int(),int(),int()}.
```

checks whether for two generated coordinates `Pos1` and `Pos2` that if they are not equal, then there distance is positive. Here, `FORALL` and `IMPLIES` are examples of logic operators provided by the QuickCheck library. QuickCheck generates test cases according to the provided generators, in this case coordinates, which only uses randomly generated integers in three dimensions. QuickCheck allows focusing on the properties that code should satisfy, rather than on the selection of individual test cases. As mentioned before, QuickCheck also performs the automated simplification of failing test cases. Details concerning this last feature can be found in [26].

Property-based testing. From a Risk-Based Analysis outset, verifying the accurate position calculation is key in assuring a safe treatment delivery. A QuickCheck property was formulated and corroborated through execution (See Fig. 2). The property should hold if the radial distance (See Formula 1) between the position estimated by the software and the actual position is less or equal to 2mm. The advantage of QuickCheck over other tools in this context is that the input to QuickCheck (the property modeled) is very close to the mathematical specification that one would expect. Hence, it is easy to inspect that the right aspect of the SUT has been tested) (cf. Fig 2).

$$\sqrt{((X_p - X_c)^2 + (Y_p - Y_c)^2 + (Z_p - Z_c)^2)} \qquad (1)$$

Random testing. In terms of coverage in the underlying test domain, it is clear that due to the nature of the software we are testing, the process of requesting only one single position calculation will cover the critical path of the modules. Thus, if the transmitter is located in {0,0,0,0,0} and then we request the position, we would have full code coverage without revealing any failure in the SUT. Therefore, we need to test many data points. The test set-up is such that we mounted an antenna to a kind of three-dimensional plotter. We can instruct this plotter to move the antenna to a certian position, specified in millimeter precision and we could then turn the antenna in a certain angle. The parameters needed to place the antenna in a given position are given in natural numbers, which correspond to the number of millimeters the antenna is moved away from the origin. Typically an area much larger than a potential tumor is tested, say a cube of 200 millimeters on each side. Together with a potential angle of the antenna, this results in about 720 million positions to test. Moving the antenna in a certain position takes up to 45 seconds, thus testing all positions is rather impractical. Moreover, the same position may return different values at different points in time, thus testing positions more than once is not superfluous.

System level testing. Given the simplicity of the property (the accuracy) that we want to test and the dependency of the whole SUT to correctly pass a large number of tests; we estimate that we can catch all failures that otherwise would be caught by unit testing. Thus, it seems that starting with system level testing and leaving out unit testing is cheaper in this case than designing dedicated tests for each unit. Because of the simplicity of the underlying formula for correctness (Formula 1), the ease with which this formula can be expressed in QuickCheck, and the kind of errors we can expect (typical for floating point handling), we decided to use system level testing as the only way of testing.

4.3 Testing environment and tested properties

The testing set-up. The lab setting used during testing consisted of a transmitter, a receiver, software and an additional mechanical device called Auto Setup[3] to which the transmitter is attached. QuickCheck generated coordinates as integer values within a range supported by the SUT. The coordinates were then used to control the Auto Setup, which, in turn, moved the transmitter to a corresponding position. The software of the SUT calculated the position of the transmitter and "sent" the calculated X, Y, Z, Vy, Vz coordinates back to QuickCheck. These calculated values are floats. QuickCheck then determined the radial distance between the initially generated position and the position calculated by the SUT. A test fails if this distance is more than 2mm. The property is depicted in Fig. 2. QuickCheck communicated via TCP/IP with a sort of request broker that we implemented in C#. This broker receives commands from QuickCheck and requests the Auto Setup to move the transmitter to a specified coordinate and then calls the software component of the SUT to request the position estimation. Details of the testing set-up are provided in [49].

```
prop_within_margin(Margin) ->
   ?FORALL(Coordinate, antenna_coordinate(),
           begin
              move_antenna_to(Coordinate),
              Position = read_position(),
              radial_distance(Position, Coordinate) =< Margin
           end).
```

Fig. 2. Accuracy Property tested in QuickCheck

Accuracy Property. In Fig. 2 `antenna_coordinate()` is a function that generates a random triplet of x, y and z coordinates and a fourth value, which is the angle of the transmitter relative to the specific surface. The generated value is bound to the variable `Coordinate`. First the transmitter is moved to a certain position. The function called `move_antenna_to(Coordinate)` returns a value when the transmitter has reached the desired point. After that the most recently estimated position is fetched from the SUT, it is then compared to the actual coordinates. Whenever a test fails, i.e., any of the actions fails or the

[3] A simplified version of a Coordinate Measurement Machine (CMM)[52], referred to here as Auto Setup is used. A CMM consists of a workspace where parts (a sensor and a mechanical assembly for moving the sensor around the workspace) are fixtured. In our case, the sensor consists of the transmitter and the mechanical assembly situates the transmitter at specific coordinates indicated through an external software interface.

result of the last inequality is `false`, then QuickCheck will automatically search for simplified failing test cases. An example of a generated simplification of failing test case constituted one of the border cases i.e., {0,0,0,0,0}. QuickCheck randomly generated each test from a QuickCheck property such as the one presented in Fig. 2. Typically, integer values specifying millimeters were used to move the transmitter to a given position (the Auto setup can be moved in steps of 1mm). QuickCheck could for instance generate a test from the property in which the transmitter is steered to position: X=58, Y=127, Z=94, Vy=0, Vz=0. The estimated position: X=58.15106462, Y=126.9147189, Z=94.82734652, Vy=-2.582979671, Vz=-3.070729491 is then registered. The distance to the real value is computed: 0.84533759 and since it is less than 2mm, the test passes successfully[4].

QuickCheck uses a uniform distribution in its random generation of coordinates. For the purpose of testing the software, we are satisfied by that. The non-functional requirement in Table 1 does indicate, however, to use a normally distributed set of sample points and to generate a normally distributed sample from them. Since patient data is unavailable at this point, we decided to be stricter than that and use a uniform distribution, requiring an accuracy of 2mm, without leaving space for points in one standard deviation. We analyzed the few failing tests (i.e., those with a distance larger than 2mm) to see by how much they deviated.

```
prop_symmetric(Margin)->
   ?FORALL(Coordinate,antenna_coordinate(),
           begin
              Extrapolated = extrapolate(Coordinate),
              move_antenna_to(Coordinate),
              Pos1 = read_position(),
              move_antenna_to(Extrapolated),
              Pos2 = read_position(),
              Distance1 = radial_distance(Coordinate, Pos1),
              Distance2 = radial_distance(Extrapolated, Pos2),
              abs(Distance1 - Distance2) =< 1
           end).

extrapolate({X,Y,Z})->
      {?upper_x - abs(X-?lower_x),
       ?upper_y - abs(Y-?lower_y),
       ?upper_z - abs(Z-?lower_z)}.
```

Fig. 3. Symmetry property tested in QuickCheck

[4] It is important to point out that the Vy and Vz are only considered for performing the position calculation and not for computing the radial distance. Hence the radial distance shown in the example only contemplates X, Y and Z.

Symmetry Property. The SUT works under the assumption that the underlying mathematical model used by the position calculation algorithm is symmetric. This means that given a coordinate, a similar accuracy on the corresponding extrapolated coordinate is attained with SUT (i.e. if the transmitter position is {36,41,73}, the SUT will give similar results in accuracy as if the transmitter was located in the extrapolated value {134,139,171}). This is presented in property `prop_symmetric()`which is depicted in Fig. 3. We verify that the accuracy distance between a given coordinate value and its corresponding extrapolated coordinate is less than 1mm. We used 1mm as the delimitation value for practical reasons. The value was experimentally determined by a test simplification that helped us to determine that the major difference between results of extrapolated coordinates in the SUT didn't exceed 1mm.

5 Results and Analysis

In this section, results from the testing process, perceived benefits from our approach, and possible areas for improvement are presented and discussed.

5.1 Test results

Within the given coverage range, the SUT provided even better accuracy than that specified by the non-functional requirement. One large sample of generated tests had a mean of 1.528505mm for the radial distance, with a standard deviation of ±0.477921, where 87% of test cases passed and 13% failed; from the failed test cases, 2% had between 2.4mm and 3.4mm for radial distance and 11% between 2mm and 2.4mm. Others were even more accurate. Others were even more accurate and only 2% of the test cases failed, displaying a radial distance of 2.02mm to 2.04mm. The set of test cases proved that the SUT had better accuracy than the requirement, and we felt very satisfied considering the results above.

Most of the failures were detected in the first test cases QuickCheck produced from the main property described in Sect. 4.3. In all cases, it was possible to trace the failures back to the code. So, adequate corrections could be performed. Typical issues involved floating point operations, type conversion, and the use of erroneous types in the drivers' interfaces. For instance, we found out that the hardware driver for the Auto Setup did not accept decimal points as parameters in one of the interfaces. This problem was identified when using QuickCheck for sending the coordinates to the Auto Setup and it was observed that the latter did not move the transmitter as expected. We could trace this problem back to a division operation in the Auto Setup interface, which was performed prior to sending the coordinates to the actual Auto Setup controller. This division produced decimal values occasionally instead of just integers. Consequently, the Auto Setup only moved when the resulting division was a whole number.

Another problem came about because of the use of incorrect casting operations (i.e. truncating decimals instead of rounding), which was detected while observing a set of failed test cases showing a very similar radial distance. We found that conversion in LabView is implicitly managed, in contrast to C#, which requires a specific conversion method.

In addition, errors due to misinterpretations of the pseudo-code (i.e. declaration of global variables and static values interpreted as local and dynamic variables) could be detected by observing failed test cases that showed a very big radial distance. A similar error was found in the same test cases, where an incorrect constant value for one of the algorithms was used (due to the mistakes during the migration process where an outdated version of LabView code was used for a specific module).

The aforementioned issues are typical when performing migration from two different platforms (in this case from LabView to C#), where some assumptions (such as typing and management of decimal values) in the old platform are not longer valid in the new platform. They are also related to a typical situation in the medical domain when specifications regarding interfaces for hardware drivers as well as software COTS (components off the shelf) are not so clear [50]. QuickCheck facilitates code refinement and simplifies the task of detecting those errors (mostly within a couple of property executions).

Note that we found all these errors by specifying just one property and generating random test cases from it. Therewith, the work of creating test cases is dramatically simplified in contrast to more traditional testing approaches. Also note that most of these errors are implementation errors and no matter how well the model is formally verified, such errors can appear.

It was moreover possible to determine which of the underlying mathematical models (See Section 4.1) for calculating the position support a given radial accuracy. Whenever a new model is introduced, it is possible to test it with QuickCheck and its adequacy being visible almost immediately. For instance, one day we had introduced a model, which was stated to provide better accuracy than the previous one; we ran QuickCheck on it and found a coordinate with an unacceptable accuracy. Hence, the model was further improved before being introduced again.

Issues in the communication protocol between QuickCheck and the C# request broker were also detected with QuickCheck. Incidentally, a problem due to the overwriting of instructions from the C# request broker into the Auto Setup driver was detected while executing the testing (this overwriting issue resulted in a series of incomplete test executions). We found that the Auto Setup driver demands a lapse of 40ms in order to process one instruction and read the next one. Although the communication with QuickCheck and C# is not part of the SUT, this last example gives us an insight of how QuickCheck could support integration testing as well, where the communication between software components needs to be verified.

5.2 Perceived contributions from the approach

Improved coverage in regression testing. We perceived that it was possible to introduce changes in other parts of the SUT (e.g. hardware, since this product is evolving constantly; making devices smaller and faster) and afterwards use QuickCheck to perform high-level testing. This enabled us to detect any incongruence or errors that might result as a consequence from those changes. The same situation applies to code enhancements performed in order to improve performance. Some data processing in LabView could be implemented in C# in a more efficient way. We run QuickCheck to make sure that these enhancements in the code give the same results as the original algorithms written in LabView. Furthermore, the coverage of the side effects resulting from changes introduced in the software (or hardware) is more comprehensive with QuickCheck since it generates new random test cases each time. In that sense, QuickCheck constitutes a good asset for a product that is constantly evolving (a scenario very typical in medical device development [51]) in contrast to regression testing which will run the same tests-suites every time.

Improving the system quality. An example of how QuickCheck helped in improving the quality of the software occurred when we utilized various mathematical models to see which ones gave better results (as explained previously in Sect. 5.1). Furthermore, having a formal specification of the SUT that can actually be run and corroborated constitutes a significant advantage for certification processes (as pointed out by [6] and mentioned in Sect. 2).

Cost effectiveness. Faults related to testing the mathematical model (accuracy checking) were detected after on average 12.85 test cases, and abnormal cases were detected after approximately 78 tests. It is very unlikely that one would manually write test cases with the same results, but it shows that several cases would have to be written for a good test suite, whereas here we only write one property once.

Each time a test case is run, the transmitter must be positioned before performing the measurement, and this is a rather expensive task if an automated tool does not support it. In our case, it took in around 5-7 seconds per each test case; depending on to which position the Auto Setup was moved. QuickCheck requires relatively less amount of effort, and supports repetitiveness and generation of new values every time. Accordingly, it was very useful for the type of testing performed.

Support for exploring heterogeneous fault sources. Another benefit of QuickCheck is that it generates simplified counter-examples (or failed test cases), which helps to analyze the nature of the error, sometimes leading to the detection of problems at software level and system level. This was found to be particularly useful when new technologies were involved (as in this context) and the main goal is to explore as much as possible the behavior of the SUT and uncover unexpected effects of the environment over the results as well as exploring heterogeneous fault sources.

Support for detection of atypical faults. Sometimes you want to run the property for a longer period and use extreme values (including boundary cases) on the test parameters in order to find atypical results. By extending the margin tolerance (increasing the radial accuracy limit), we could detect atypical cases related to the transmitter angles (angles very close to the negative or positive borders brought about significant radial distances). For instance, when we modified the accuracy property and set the accuracy tolerance up to 6mm; an apparently normal position (in the sense that it was within the coverage-range of the SUT) resulted in a radial distance of almost 6mm. Following some more tests, we found out that one version of the underlying mathematical model used in the SUT was sensitive to strongly angled positions (in terms of Vy and Vz). This finding lead to adjustments in the mathematical model in order to improve its robustness against angling. It must be mentioned that the parameters used for performing this type of testing exceeded the limits of what could be called a normal scenario (e.g. test parameters derived from real patient data).

5.3 Areas for improvement

We have identified a number of limitations in our case study. This study does not cover the necessity of having a given distribution (in this case a normal distribution) and usage of sample data from patients.

When a test fails, we want to obtain the coordinates that give the highest possible measurement fault, in other words, for which the distance to the position is greatest. This is not possible to perform automatically with the current version of QuickCheck. QuickCheck provides simplification of input data but cannot yet connect it to the results of the actual test.

It is worth mentioning that one of the limitations of working in a lab is the presence of sporadic radio transmission noise due to research activities taking place at nearby companies. This also enforces a sufficient number of tests in order to assure the robustness of the SUT in less than ideal situations. We can store the test case sequences in QuickCheck and redo the property execution in order to see any behavior that can be influenced by the environment and signal fluctuations. This would be particularly good if we want to improve the robustness of the system to external noise factors, which is very common in an environment like a hospital.

Throughout this study, we have observed a potential for QuickCheck to support statistical functionalities (e.g. to test confidence intervals). Some planned features for future releases include control or specification of the number of test cases, and generation of test cases by sampling from a defined set of data (i.e. real patient data). Also, improving the logging capabilities for QuickCheck could notably expand the potential of using QuickCheck for test results analysis. Logging not only failed test cases but also the asserted test cases would potentially upgrade the tool.

6 Conclusion

We have described a case study on testing a medical device by using a formal model as a basis for the automatic generation of test cases with the tool QuickCheck. We found a number of errors in the code we developed and were able to spot inaccuracies in prototype models. Early detection and correction of these errors has lead to a high quality product being developed by the medical company at which the case study was performed.

This case study assembles adequate conditions for using formal models. The model is simple, clear and based on a mathematical formula. Verifying medical devices may not always be like this case, and there may be a need for more complex modeling for system behavior. Nevertheless we believe that is it worthwhile to try the technology on more medical equipment. We intend to continue this work involving more complex properties than the ones presented here.

The most remarkable aspects of this study focus on several positive results: First, property-based testing proved to be feasible and cost-effective within this domain in contrast to the normal tendency of using test suites. This is of great value particularly for those projects with high-pace development, typically involving continuous modifications in the code in order to improve performance and constant incorporation of new features. QuickCheck's approach on lightweight formal specification has great potential to be used in proof-based certifications for medical devices as recognized by several medical practitioners involved with the project. Automated simplification of test cases supported the exploration of atypical cases, and was found useful for testing real-time systems (which are commonly present in the medical domain). Finally, the capacity of QuickCheck for performing high level testing can be regarded as a potential tool that could facilitate the process of integration and system testing within the highly complex domain of medical systems.

References

1. J. L. Cyrus, J. Daren and P. D. Harry: Formal Specification and Structured Design in Software Development. Hewlett-Packard Journal, 1991
2. J.W. Brakel: Formal Verification of a Medical Protocol. ISIS Technical Report, University of Twente, 2005
3. V. Kasurinen and K. Sere: Integrating action systems and Z in a medical system specification. Industrial Benefit and Advances in Formal Methods, LNCS **1051**, (1996) 105–19
4. Food and Drug Administration (FDA). Online: *http://www.fda.gov*
5. Medical device directive (MDD). Online: *http://www.mdss.com/MDD/mddtoc.htm*
6. I. Lee, G. Pappas, R. Cleaveland, J. Hatcliff, B. Krogh, P. Lee, H. Rubin and L. Sha: High-Confidence Medical Device Software and Systems. Computer, **39(4)**, (2006) 33–38
7. D. Jackson and J. Wing: Lightweight Formal Methods. IEEE Computer, **29(4)**, (1996) 22–23
8. T. Arts, K. Claessen, J. Hughes, and H. Svensson: Testing implementations of formally verified algorithms. *Proc. 5th Conf. on Soft. Eng. Research and Practice in Sweden*, (2005) 20–21
9. M. Kim, S. Kannan, I. Lee and O. Sokolsky: Java-MaC: a Run-time Assurance Tool for Java. *Proc. Runtime Verification*, Electronic Notes in Theoretical Computer Science, **55**, Elsevier Science, 2001.
10. S. Nelson and C. Pecheur: V&V of advanced systems at NASA, Produced for the Space Launch Initiative 2nd Generation RLV TA-5 IVHM Project, 2002

11. M. Taghdiri and D. Jackson: A lightweight formal analysis of a multicast key management scheme. *Proc. 23rd IFIP Int. Conf. on FTNDS*, (2003) 240–256
12. S. Easterbrook, R. Lutz, R. Covington, J. Kelly, Y. Ampo, D. Hamilton: Experiences using lightweight formal methods for requirements modeling. IEEE Soft. Eng. **24(1)**, (1998) 4–14
13. Micropos Medical AB. Official web site: *http://www.micropos.se*
14. K. Claessen and J. Hughes: QuickCheck: a lightweight tool for random testing of Haskell programs. *Proc. 5th Int. Conf. on Functional Programming*, (2000) 268–279
15. P. Jones: Assurance and Certification of Software Artifacts for High-Confidence Medical Devices. High Confidence Medical Device Software and Systems Workshop, Philadelphia, USA, 2005.
16. J. Bowen and V. Stavridou: Safety-critical systems, formal methods and standards. Soft. Eng. Journal **8(4)**, (1993) 189–209
17. D.R. Wallace, D.R. Kuhn and L.M. Ippolito: An analysis of selected software safety standards. *Proc. 7th Conf. on Computer Assurance*, (1992) 123–136
18. R. Jetley and S. P. Iyer: Enabling Certification through an Integrated Comprehension Approach. In [15].
19. J.P. Bowen and V. Stavridou: Formal methods and software safety. Safety of Computer Control Systems, Pergamon Press, (1992) 93–98
20. M. Marcos, M. Balser, A. ten Teije and F. van Harmelen: From informal knowledge to formal logic: a realistic case study in medical protocols. *Proc. 13th Int. Conf. EKAW*, LNCS **2473**, (2002) 49–64
21. J. Jacky, J. Unger, M. Patrick, D. Reid and R. Risler: Experience with Z developing a control program for a radiation therapy machine. *Proc. of 10th Int. Conf. of Z Users*, LNCS **1212**, (1996) 317–328
22. J. Rushby. Formal Methods and the Certification of Critical Systems. Computer Science Laboratory, SRI International, Menlo Park, CA. Number SRI-CSL-93-7, December 1993.
23. R. Jetley, P. Iyer and P. Jones: A Formal Methods Approach to Medical Device Review. IEEE Computer, **39(4)**, (2006) 61–67
24. Food and Drug Administration (FDA). "General Principles of Software Validation: Final Guidance for Industry and FDA Staff", FDA, 2002.
25. Medical Device Directory (MDD). "Council directive 93/42/EEC of 14 June 1993 concerning medical devices", Medical Device Directory, 2003.
26. T. Arts, J. Hughes, J. Johansson, and U. Wiger: Testing telecoms software with Quviq QuickCheck. *Proc. ACM SIGPLAN Workshop*, (2006) 2–10
27. Encyclopedia on Software Engineering (edited by J.J. Marciniak), Wiley, 2001. Ibrahim K. El-Far and James A. Whittaker: "Model-Based Software Testing"
28. H.S. Hong, I. Lee, O. Sokolsky and H. Ural: A temporal logic based coverage theory of test coverage and generation. *Proc. 8th Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems*, LNCS **2280**, (2002) 327–339
29. I. Gronau, A. Hartman, A. Kirshin, K. Nagin and S. Olvovsky: A methodology and architecture for automated software testing. IBM Research Laboratory in Haifa Technical Report, MATAM Advanced Technology Center, Haifa 31905, Israel
30. J. Dick and A. Faivre: Automating the generation and sequencing of test cases from model-based specifications. *Proc. of Industrial-Strength Formal Methods*, LNCS **670**, (1993) 268–284
31. P. Ammann and A. J. Offutt: Using formal methods to derive test frames in category-partition testing *Proc. 9th Conf. on Computer Assurance*, IEEE Computer Society Press, (1994) 69–80

32. R. A. Kemmerer: Testing formal specifications to detect design errors. IEEE Trans. on Soft. Eng., **11(1)**, (1985) 32–43
33. G. Laycock: Formal specification and testing: A case study. The Journal of Software Testing, specification, and reliability. **2(1)**, (1992) 7–23
34. P. Stocks and D. Carrington: A framework for specification-based testing. IEEE Trans. on Soft. Eng., **22(11)**, (1996) 777–793
35. W. T. Tsai, D. Volovik, and T. F. Keefe: Automated test case generation for programs specified by relational algebra queries. IEEE Trans. on Soft. Eng., **16(3)**, (1990) 316–324
36. A. J. Offutt, Yiwei Xiong, Shaoying Liu: Criteria for generating specification-based tests. *5th IEEE Int. Conf. ICECCS*, (1999) 119–129
37. E. Weyuker, T. Goradia, and A. Singh: Automatically generating test data from a boolean specification. IEEE Trans. on Soft. Eng., **20(5)**, (1994) 353–363
38. B. Nielsen and A. Skou: Automated Test Generation from Timed Automata. *Proc. Tools and Algorithms for the Construction and Analysis of Systems*, (2001)
39. M. Utting, A. Pretschner and B. Legeard: A Taxonomy of model-based testing. University of Waikato, Department of Computer Science. No. 04/2006.
40. S. Butler, S. Chalasani, S. Jha, O. Raz and M. Shaw: The Potential of Portfolio Analysis in Guiding Software Decisions. *EDSER-1 as part of ICSE'99*, (1999)
41. B. Legeard, F. Peureux, and M. Utting: Automated Boundary Testing from Z and B. *Proc. Int. Symposium of Formal Methods.* LNCS **2391**, (2002) 21–40
42. N. Kosmatov, B. Legeard, F. Peureux, M. Utting: Boundary Coverage Criteria for Test Generation from Formal Models. *ISSRE*, (2004) 139–150
43. F. Belli, M. Eminov and N. Gokce: Prioritizing Coverage-Oriented Testing Process – An Adaptive Learning Based Approach and Case Study. *Proc. 31st Int. Conf. Computer Soft. and Applications.* IEEE Computer Society, 197–203
44. R. Castanet, O. Koné, P. Laurençot: On-the-fly test generation for real-time protocols. *Proc. 7th Int. Conf. in Comp. Comm. and Networks*, IEEE Computer, 1998.
45. R. Alur, C. Courcoubetis and D. Dill: Model-checking for real-time systems. Logic in Computer Science, (1990) 414–425
46. T. Jéron, P. Morel: Test generation derived from model-checking. *CAV'99, Italy*, LNCS **1633**, (1999) 108–122
47. National Instruments: "NI LabVIEW" Online: *http://www.ni.com/labview/whatis/*
48. J. L. Armstrong, M. Williams, R. Virding, and C. Wilkström: ERLANG for Concurrent Programming. Prentice-Hall, 1993.
49. A. Fallas Yamashita and A. Bergqvist: Testing a radiotherapy support system with QuickCheck. Masters thesis, IT University of Göteborg, Sweden, 2007.
50. G. Sharp and N. Kandasamy: A Dependable System Architecture for Safety-Critical Respiratory-Gated Radiation Therapy. *Proc. Int. Conf. on Dependable Systems and Networks*, **00**, DSN, IEEE Computer Society, June 2006.
51. M. Poonawala, S. Subramanian, W. Tsai, R. Mojdehbakhsh and L. Elliott: Testing Safety-Critical Systems – A Reuse-Oriented Approach. *Proc. 9th Int. Conf. Software Eng. and Knowledge Eng.*, Knowledge Systems Institute, (1997) 271–278
52. W. Singhose, N. Singer and W. Seering: Improving repeatability of coordinate measuring machines with shaped command signals. Precision Engineering **18**, (1996) 138–146