A COMPARISION OF RANDOM TESTING AND BOUNDED EXHAUSTIVE TESTING

MASTER'S THESIS - WIETSE VENEMA

Centrum voor Wiskunde en Informatica
Universiteit van Amsterdam

SUPERVISOR:
Prof. Dr. Paul Klint

20 augustus 2012

ABSTRACT

Random testing and bounded exhaustive testing are two forms of uninformed software testing. In contrast with informed testing strategies that are based on knowledge about the system under test, uninformed strategies only use knowledge about the input domain of the system. Automated uninformed testing requires an automated oracle to judge whether the behavior of the system is correct. We used property-based testing to provide these oracles. In a case research study we compared the practical value of random and bounded exhaustive testing using relative failure detecting ability as a measure for efficiency. We constructed 26 properties for different parts of the Rascal project. Of those 26, 25 were refuted using random testing and 19 also using bounded exhaustive testing. In 8 of those 19, different failures were found by both tools. We present a detailed investigation and interpretation of a selection of test cases. We conclude that both random testing and bounded exhaustive testing are useful and possibly complementary tools in the context of our study. Random testing was slightly more efficient.

CONTENTS

1 INTRODUCTION

A popular story in computer science is that of the first bug ever encountered. In 1946 an operator noticed a malfunction in the the Harvard Mark II, an electromechanical computer. Further investigation led him to a dead moth trapped inside a relais. We know this because this 'bug' was carefully removed and taped into the logbook. Now, 60 years later, bugs are still with us. Only now they are not dead insects, but inadvertent mistakes made by programmers which manifest as failures of software systems.

Software bugs can have severe consequences. For example, in 1996, the European Ariane 5 rocket exploded 37 seconds after launch because of a software bug. In 2011, car manufacturer Honda had to recall 2,5 million cars since cars would spontaneously shift out of park due to a software bug. And then there is the horrifying story about the Therac-25. This was a medical device used for radiation therapy. It malfunctioned because of a subtle and not often triggered software bug. At least three patients died from exposure to approximately 100 times the intended radiation dose. This is now a textbook example of software development bad practice. As it seems, software bugs are now part of everyday life.

Due to the possible adverse consequences of software bugs, the software engineering industry tries to validate whether a software system behaves as expected. Usually this validation involves software testing. A test consists of a sequence of inputs accepted by a program. An oracle then determines whether the output is correct. Such an oracle can be manual of automated.

Modern software systems can be very large and are often developed by many people. The different parts of the software can interact in unexpected ways. To reach a certain level of confidence about the correct behavior of the software the whole system has to be tested, even after a small change to the system. This means that the use of a manual (human) oracle can be very costly and time-consuming. Automated testing trumps human testing in many aspects. Tests can be executed overnight or even during development and they are always executed in precisely the same way. Consequently, a programmer gets immediate feedback during development, which is desirable from the point of view of software quality [2].

Generally, a program has a great volume of possible test inputs. Most software testing strategies execute the program on a limited subset of inputs only since it is often not feasible to try all inputs. Automated testing strategies can be divided in two categories: *informed* or *uninformed*. Uninformed strategies use no knowledge about the system under test at all to select test inputs, in contrast to informed strategies. This knowledge can for instance be the specification of the program,

a list of all features, statement coverage or even developer suspicion (*"Test what you fear might break"* [2]). Most testing strategies can be considered informed strategies in some way [27].

Uninformed testing has long been regarded as the worst case of program testing [7]. When testing without knowledge about the program, it seems that there is a very low chance of generating valid of even 'interesting' test inputs. However, empirical experiments with random testing, a kind of uninformed testing, show otherwise. Random testing selects test inputs randomly from the entire input domain. Researchers used random testing to validate a multiprocessor cache controller [28], to test C compilers [21] and to validate JavaScript code [14]. They reported promising and useful results. In some cases, random testing even outperformed informed testing [13].

Next to random testing, another class of uninformed testing strategies is *bounded exhaustive testing*. This only examines a small and finite part of the input domain, usually the first part, according to some notion of *depth*. The motivation for bounded exhaustive testing can be summarized by the following hypothesis; *"If a program fails to meet its specification, it almost always fails in some simple case"* [24]. This is similar to the *'small scope hypothesis'* that motivates model-checking tools such as Alloy [15]. Also, the *'competent programmer hypothesis'* has been adopted in mutation testing. It assumes that if a program has no errors that can be exposed using a few simple mutations, no errors exist at all [6].

According to Hamlet [13], it is still not clear whether bounded exhaustive testing or random testing is more useful in a real world setting. This is a relevant question to pursue. All steps taken towards the validation or refutation of the small scope hypothesis are beneficial since validating the small scope hypothesis would void the need for complex test cases. This would simplify software testing by great lengths.

In this thesis we explore the question whether random testing or bounded exhaustive is more useful in a real-world setting.

2 METHODS

2.1 *Study design*

In this thesis, we investigated whether uninformed testing strategy is more useful in a practical setting: random testing or bounded exhaustive testing. We constructed tests for several features of a software project. We then generated test input for these tests using both random and bounded exhaustive and compared the results using a suitable measure which we propose in section 2.2.

The independent variable we examine is the strategy we use to choose values from the input domain, either bounded exhaustive or random. This is the only variable we change. The controlled variables or context is characterized by the program we test, the programming language and the programmers that built the software. There are a lot of factors in this context that can possibly influence the outcome of the comparison between testing strategies, including, but not limited to, the domain of the software system, its maturity and isolation of individual features within the product, the size of the team and the number of people working concurrently on the same feature, etc. We are only just beginning to understand what factors influence the emergence of software bugs and their effect on software quality. This indicates that it is most appropriate to make our comparison of random testing and bounded exhaustive testing in a natural setting as opposed to a more controlled environment. It has proven very difficult to create artificial faults with the same characteristics as 'natural' faults. We want to compare the effect of the testing strategies with real faults, occurring in real software written by real programmers.

It is hard to make a comparison of bounded exhaustive testing and random testing on theoretical grounds. Bounded exhaustive testing is motivated by the hypothesis that when no bugs can be found with small and simple cases, there are no bugs at all. This motivation is empirical, not theoretical. It is straightforward to think of an example where there is a failure just outside of the range of bounded exhaustive testing. This is the second reason why the comparison between bounded exhaustive testing and random testing is best done in a natural setting.

We chose to design our study as a case study. With a case study, rather than trying to seek a general law ('nomothetic'), you examine a single entity in a particular context ('idiographic') and attempt to understand it [3].

To summarize, there are two main reasons to choose a case research study design:

1. It is not possible to examine the phenomenon outside of the natural setting of a real-world software project.
2. It is hard to favor either testing approach on theoretical grounds, due to the empirical principle that motivates bounded exhaustive testing.

2.2 *Conceptual model*

To compare the practical value of bounded exhaustive testing and random testing we need to define a measure.

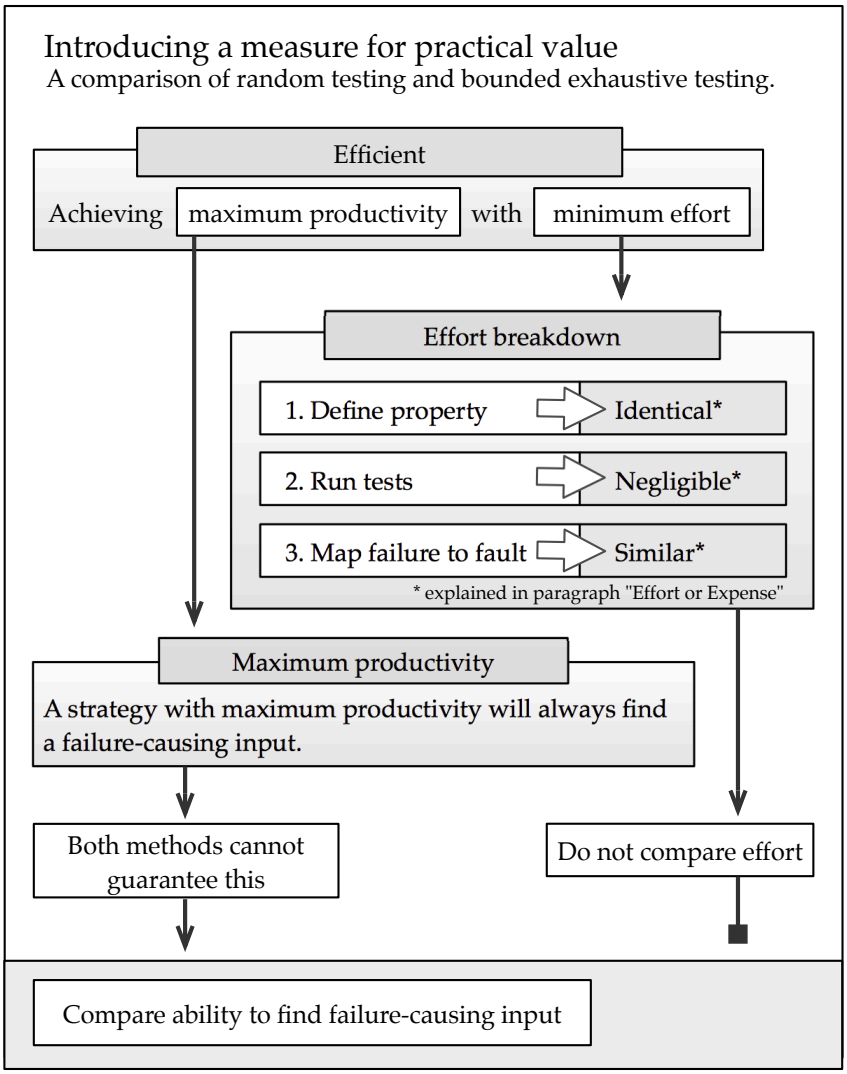

Figure 1: Illustration of conceptual model

2.2.1 *Conceptual model*

An important goal of most software quality initiatives is to reduce the time between fault introduction and fault detection, since this lowers costs and leads to higher software quality [8]. Efficiency is the term that accurately reflects this balance between results and time. It is paramount that a testing strategy must be very effective at finding failures, but it should also produce results within the shortest amount

of time possible. In this section we present a conceptual model which outlines what constitutes a good measure for efficiency.

EFFICIENT Achieving maximum productivity with minimum wasted effort or expense.

This definition of efficient concerns 'productivity' and 'effort or expense'. We will now explore both concepts.

MAXIMUM PRODUCTIVITY A testing strategy determines if a program is correct with respect to its specification. Therefore, a testing strategy with *maximum productivity* will guarantee that a program is correct. But, as Dijkstra already observed, testing can prove the presence, but not the absence of bugs. We limit ourselves to software testing that relies on observation of the behavior of a software system. These kind of strategies can only observe the difference between the specified and the observed behavior of a program. The IEEE standard definition [1] defines this difference as a *failure*. A failure can be caused by a *fault*, which is the difference between a correct and an incorrect program. These are the actual mistakes programmers make. The mapping from failures to faults is a complex task that typically can only be performed by developers with intimate knowledge of the system. The number of failures detected by a testing method is an unreliable measure, since multiple failures can be caused by one fault [4]. We argue that the best way to compare random testing with bounded exhaustive testing in terms of productivity is to compare their ability to find a failure-causing test input for a given test.

When one strategy finds a failure-causing input and the other doesn't, the first is considered having a greater productivity.

EFFORT OR EXPENSE We defined efficient as *"achieving maximum productivity with minimum wasted effort or expense"*. We already introduced a measure for productivity. The other part of the definition concerns 'effort or expense'. We consider effort from the point of view of the developer, testing software. A good measure for effort in this context is time [8]. Time that is spent working, thinking or waiting for tests to complete. With testing strategies that rely on observing behavior, the time spent can be broken down into the following three steps;

1. Manually specify test cases for of the system under test. This requires a developer to thoroughly define the intended behavior of their program.
2. Execute the tests (developer waiting time).
3. Inspect the failures manually and reduce them to faults.

Since we do a comparative evaluation, we now need to investigate whether random testing or bounded exhaustive testing spends more time in either of these three steps.

1. TEST SPECIFICATION The first step is identical for both strategies, since the test cases are a controlled variable in our experiment. We change only the way we generate input for the tests, not the tests themselves.

2. TEST EXECUTION The second step concerns the execution of the tests. We make the following assumptions;

- Both bounded exhaustive testing and random testing have similar short execution times. Bounded exhaustive testing is designed that way. It tests only a small and finite part of the input domain. With regard to random testing, we argue in section 2.5 that running random tests for an extended period of time does not add much more confidence.
- We assume a usage scenario in which a developer will run the tests while writing code. Testing during development is a paradigm that is popular and familiar (Test Driven Development [2]). Therefore, we view test execution time as effort; developers have to wait for the tests to complete.

We argue that the time both strategies spent in the second step is similar. The differences in execution time are negligible and the duration of the second step is very short when compared to the first and third step. Constructing tests or debugging failures can take hours, and running the tests takes approximately 10 seconds or less (section 2.5)

3. REDUCE FAILURE TO FAULT Finally, the third step concerns reducing failures to faults. This is largely a manual process. A developer has to manually inspect the failure. He will probably try to reproduce the fault and then try to simplify the input value, so that it forms a minimal example of what causes the failure. Then he will try to deduce the cause of the failure and find the fault(s) causing it. Random testing typically produces larger failure-causing input values than bounded exhaustive testing since the latter only tests small cases. Reducing large values to a minimal example takes a lot of time when done manually, but this can be automated using delta debug-

ging [29]. With delta debugging, the failure-causing test inputs are similar in size and so should be the time needed to reduce the failures to faults.

With regard to effort, we assert that both methods are associated with a similar amount of effort. Our experience while developing tests using both methods reinforces this.

MEASURE FOR EFFICIENCY To conclude, our goal is to make a relative comparison based on efficiency, since this a valuable measure from an engineering perspective. We introduced a definition of efficiency, *"achieving maximum productivity with minimum wasted effort or expense"*. This definition is comprised of two parts, productivity and effort or expense. We argued that the effort associated with both methods is similar. Therefore we will only compare the *productivity* of random testing and bounded exhaustive testing. Earlier in this section we stated that when one strategy finds a failure-causing input and the other doesn't, the first is considered having a greater productivity and, following the above rationale, *efficiency*.

2.3 *Context*

In this section we will further characterize the context in which we conducted our experiment. We developed tests that exercised parts of the programming language Rascal and languages developed with Rascal.

Rascal [16] is a meta programming language that is used to create tools that analyze, transform, generate or visualize source code of software systems. A strong point of Rascal is that it integrates source code analysis and manipulation on a high level. Rascal is based on a Java interpreter and runtime and is well integrated with the IDE Eclipse. In terms of maturity, Rascal is still a relatively young project and development is ongoing. Features are being added and removed on a regular basis. The development team considers Rascal to be in 'alpha' stage.

Rascal is being developed by researchers of the Software Analysis and Transformation group at Centrum Wiskunde & Informatica in Amsterdam. The engineers who contribute to Rascal are highly educated and can be considered experienced software developers. This is an important aspect with regard to generalizing this research.

The team developing Rascal is of small to moderate size; between 10 or 15 people do active development. This probably has a positive effect on software quality [20].

Since the developers are also researchers, they have to perform a constant balancing act between publishing papers in peer-reviewed papers and releasing production quality software. Both demands can reinforce each other but may also conflict at times.

We developed tests for parts of the Rascal distribution and for Derric, a declarative domain specific language written with Rascal. We made a subdivision in four different groups based on maturity, complexity and purpose of the code. Those are outlined below.

A. RASCAL SYNTAX FEATURES AND CORE MODULES The Rascal distribution contains a set of modules that complement the different datatypes such as Set, List, String and Map. This part is called the Prelude. A lot of the functionality has been around since the inception of Rascal, so we can consider this functionality as quite mature. A substantial part of the Prelude is written in Java: 2610 lines of Java vs 6610 lines of Rascal code. The complexity of this part is limited.

B. RASCAL UTILITY LANGUAGE IMPLEMENTATIONS Contained within the Rascal distribution are several language parsers. They provide a way to parse files and return an abstract data type which can

be used for further manipulation or analysis. Most are written in Rascal, such as the dot parser, and some are only a front for Java libraries, such as the XML implementation. Most implementations have been around for a while. Whether they have been in active use or not is not clear so it is hard to judge the maturity of this part.

C. RASCAL DEMONSTRATION LANGUAGES Rascal contains several language implementations with an instructive purpose, to demonstrate features or good practice. Therefore, the code is not 'hardened' by production use, but is very well readable and properly documented.

D. DERRIC Derric [25] is a domain specific language for declaratively specifying data structures. Based on these declarations, Derric has already been used to construct file carvers. File carvers are used in Digital Forensics to recover data from files. Derric is the most mature DSL implementation we examined. The source code of Derric has been publicly released.

2.4 *Tools*

With uninformed testing, it is important to have an automated oracle which can judge whether the behavior of the system under test is correct. Property-based testing is a convenient way to provide automated oracles. We have built a property-based testing framework to compare bounded exhaustive testing with random testing, inspired by the tools QuickCheck [5] and SmallCheck [24]. The tool `quickcheck` has been integrated with the Rascal test framework and has already been released for public use.

Property-based testing uses testable specifications. A developer specifies a property that should be true for all input values of a certain type. The testing framework then generates values of that type and checks if the property holds. If it does not this is reported as a counter example.

Listing 1: Example property

```
public test bool proptextreadwrite( value a ){
        loc file = |tmp:///xxx|;
        writeTextValueFile(file, a);
        value b = readTextValueFile(file);
        return( a == b );
}
```

Listing 1 shows an example of a property specification for the ValueIO module in Rascal. The property states that any value should remain unchanged when writing it to a file and then reading it in again. The procedure to test this property is;

1. Inspect the parameters of the property specification.
2. Generate a certain number of test inputs based on the parameters.
3. Evaluate the test inputs, using the property specification as an automated oracle. Stop when the oracle signals a failure and report the failure-causing input.

Property-based testing is very suitable for use with random testing and bounded exhaustive testing, since the specification can be used as an automated oracle. It is also a right fit for this experiment as we can change the strategy used to generate the test cases while keeping the other aspects constant.

2.4.1 *Illustration of tool usage*

We take the property in listing 1 as a example. In the next listing we try to refute this property using random input values. The function

`quickcheck` takes the property function as an argument. `quickcheck` will inspect the parameters of the property. Based on the types of the parameters it will generate a preset number of random values and evaluate the property with these values until the property fails or all values have been evaluated. In the latter case it reports that the property has not been refuted. A greater number of cases results intuitively in a greater confidence that the property can never be refuted, but this is not necessarily true, as we propose in section 2.5.

In the next listing we use `quickcheck` to validate the property in listing 1. The tool reports a counterexample very quickly. When the year-part of a date is greater than 9999, the system is able to write it to a file but is unable to read it in again. This results in the error displayed below. An interesting observation is that bounded exhaustive testing would never test a date with such a high year since it is out of its scope.

Listing 2: Test property with quickcheck

```
rascal>quickcheck(proptextreadwrite);
std:///ValueIO.rsc:35,9: Java("Error reading date, expected
    \'\\-\'\\, found: 48")
failed with [(-375103r84676:$53870469-08-28T22:35:03.634+0100)]
bool: false
```

Since the tool `quickcheck` has been integrated with the Rascal distribution, the property in listing 1 could also have been tested by running the shell command `:test`. Properties are indicated with the function modifier `test`.

In the next listing we try to refute the same property using bounded exhaustive testing. We will explain more thoroughly what happens here in section 2.4.3. In this example, `smallcheck` did not find a counter example.

```
rascal>smallcheck(proptextreadwrite, 3, 1000);
1: Checked with [-0.75]: true
2: Checked with [-12.0]: true
3: Checked with [-1.5]: true
...
...
37566: Checked with [<0>]: true
37567: Checked with [<true>]: true
37568: Checked with [<false>]: true
Not refuted up to depth 3
bool: true
```

2.4.2 *Random Testing*

The generation of random input values consists of choosing pseudorandom numbers and mapping them into the domain of the parameters of a property. An important consideration when generating random values is the distribution of values to use. A common advice is to use a distribution that mimics the operational profile [7]. However, during development an operational profile is often not available. Moreover, when testing at unit level, as opposed to system level, it is often impossible to construct such a profile. An uniform distribution of input values is often used instead [13]. This gives rise to another problem; when choosing values from an infinite domain it is impossible to use an uniform distribution. One approach to these problems is to shift the responsibility for choosing a suitable distribution to the developer of the tests through user-programmable value generators [18, 5]. The use of such custom generators has some benefits:

- A developer of tests can control the distribution of input values with user-programmable generators. It makes no sense to choose a distribution beforehand without any knowledge of the desired distribution, so this is a choice best made by the developer of the tests.
- With user-programmable generators a developer can also state preconditions on properties. For example if you define a property that only holds for sorted lists, you could use the default generator and ignore all unsorted lists. But then you would mainly test very short lists, since the probability of a short random list to be sorted is greater than that of a longer random list. In this case it would be wiser to construct a generator which builds random *sorted* lists.

However, there are also some drawbacks to user-programmable generators.

- Creating a good user-programmable generator for a custom datatype is difficult [24]. Developer errors can lead to subtle bugs which can result in a distribution of values other than intended or even cause non-termination. These kind of bugs tend to be hard to notice. Moreover, if you have a lot of custom datatypes, you also have to write a lot of generators.
- When using a generator to express preconditions, the definition of the precondition is in the generator, but the postcondition is defined separately in a property. This is undesirable from a software engineering perspective.
- It is tempting to build random value generators that favor a subset of the input domain based on some knowledge about the program. This undermines the strength of uninformed test-

> ing, since the assumptions underlying this preference might be wrong [13]. Uninformed testing strategies are by definition unsystematic. They have a small chance of testing 'interesting' cases, but they cannot be deceived by false assumptions. This is one of the strong points of uninformed testing.

Providing a default random generator for all types has benefits that outweigh the drawbacks, particularly in the context of languages with a lot of custom datatypes. As well, we do not want to create an opinionated tool that forces decisions on its users. For instance, we have no data on the actual need for properties with preconditions. Therefore we took a pragmatic approach. The tooling provides a default random value generator for all values, even for user defined data types. Developers are then encouraged to inspect the generated distributions using built-in Rascal functions such as `classify` and possibly redefine the default behavior using a custom generator. The tool contains a framework to create composable monadic generators to aid in the development of user-programmable generators. One advantage of using monads is that the necessary depth limiting to handle recursive datatypes can happen behind the scenes. We based the design of this framework on work in [18].

In the next listing we define an algebraic data type `Exp` and call the function `arbitrary` which returns a random `Exp` value, constructed using the default random generator. The two arguments to `arbitrary` are the type of the value we want it to return and a depth limit to limit the recursive depth.

Listing 3: Generating arbitrary values

```
rascal>data Exp = con(int n) |
>>>>>>>mul( Exp e1, Exp e2) |
>>>>>>>add( Exp e1, Exp e2 );
ok

rascal>arbitrary(#Exp, 4);
Exp: mul(
  con(-2),
  add(
    con(-1),
    mul(
      con(217),
      con(-19))))
```

In the remainder of this subsection we explain how we generate random values using a type-driven generator and why we need a depth limit.

PRIMITIVE TYPES We use the `Random` class provided with Java to generate pseudorandom numbers. For all simple types in Rascal such

as number, tuple, datetime and booleans it is fairly straightforward to implement a mapping from a pseudorandom number to a value.

DATATYPES Rascal allows developers to define custom types through algebraic data types. Once an algebraic data type is declared, it can be constructed using a call to one of its constructor functions.

```
rascal>data Bool = t() | f();
ok

rascal>t();
Bool: t()
```

We generate a random datatype value by choosing one of the constructors at random, generating random arguments. However, since datatypes can be recursive such as the `Exp` datatype in listing 3, we need some mechanism to ensure termination. A common strategy is to add probabilistic weights to the alternatives with larger weights added to the 'leaves' [19]. This ensures eventual termination. Since this would require the developer to annotate all datatypes this is an impractical solution in our case. A second approach is to decrease the probability that an alternative will be selected each time it is used, as reported in [17]. This can still result in very large data structures. Therefore we chose to limit the recursive depth [5, 18]. This is the depth limit you have to provide when calling `quickcheck` or `arbitrary`. Following the common definition of depth of tree, the depth of an algebraic datatype with zero arguments is 0 and the depth of an alternative with more than 0 arguments is defined as the maximum depth of its of arguments plus 1.

LISTS, SETS, RELATIONS AND MAPS A list can be viewed as a pretend datatype with two constructors. It is either empty, or a head element followed by a tailing list.

Listing 4: Imaginary list datatype

```
data MyList = emptyList() | myList( value head, MyList tail);
```

Using this interpretation, lists can be generated the same way as datatypes. Strings are lists of characters so they can be handled in a similar way. Due to the way we generate datatypes, a list has an expected length of one. Off course there are cases when this is undesirable. This is when a developer has to resort to building a user-programmable generator.

Sets, despite being unordered, are generated in the same fashion; a head element is randomly chosen from the set of elements that are not yet in the 'tail' of the set. This extends to relations because they are

sets of tuples with the same static type. Finally maps are generated using a similar approach.

VALUES OF TYPE VALUE In Rascal there is a super type `value` which stands for all possible Rascal values. It is a container type and has no value of its own. The generator selects a random type and then generates a random value of that type. The system generates random types using a similar approach as it generates values. The recursive depth of the type is limited by the same depth limit.

PARAMETERIZED TYPES Functions and data types in Rascal can be defined using parameterized types. It is useful to express properties that hold for values of more than one type so we allow for parameterized types. At a binding occurrence, a random type is selected within the type constraint of the parameterized type.

2.4.3 *Bounded Exhaustive Testing*

With bounded exhaustive testing, a small and finite part of the input domain is exhaustively enumerated. As with the implementation of random values, the generation of values is type driven. This means that for each type in Rascal there has to be a definition of depth. In the remainder of this section we explain the exhaustive generation of values limited by depth. We start with algebraic datatypes and extend this to other types.

```
rascal>exhaustive(#Exp, 2);
set[Exp]: {
  mul(
    con(0),
    con(0)),
  add(
    con(0),
    con(0)),
  con(1),
  con(-1),
  con(0)
}
```

DATATYPES As with random generation, we follow the common definition of depth of tree. The depth of an algebraic datatype with zero arguments is 0 and the depth of an alternative with more than 0 arguments is defined as the maximum depth of the list of arguments plus 1. The set of all instances of a particular datatype up to some depth consists of the union of all alternatives of that datatype up to that depth. To construct all instances of an alternative up to some

depth, one computes the cartesian product of all arguments up to that depth minus one, and uses the resulting matrix to build all instances of the alternative.

LISTS AND STRINGS A list can be seen as a pretend datatype (listing 4). This means that an empty list has depth 0 and a list with one element has a depth one greater than the depth of the element.

TUPLE We treat tuples the same as constructors of datatypes. An empty tuple has depth 0, and tuples with arguments have a depth one greater than the maximum argument depth.

NUMBERS Integers can also be interpreted as pretend datatypes.

```
data myInt = Zero() | Successor( myInt i );
```

This means that the depth of an integer is given by its absolute value. The common representation of a floating point number can is $x \cdot 2^y$, where x is the number of significant digits and 2 the base for scaling using the exponent y. The depth of a floating point (`real` type) is then the same as the depth of `tuple[int x,int y]`.

BOOLEAN AND DATETIME We model a boolean as a datatype with two alternatives. The depth of a datetime value is given by the number of seconds since the epoch.

RELATION AND SET Sets are different from lists in the sense that they have no order. An empty set has depth 0, and all other sets have a depth one greater than the maximum element depth, similar to tuples. We handle relations as sets of tuples with the same static type.

MAP A map is a set of pairs consisting of a key and value. Only a single value can be associated with each key. The depth of an empty map is zero, and the depth of a non-empty map is the maximum pair depth plus 1. We model a pair as if it is a tuple.

TYPE VALUE AND PARAMETERIZED TYPES Using the same depth limit we first generate a list of types exhaustively and use them to generate the corresponding values.

2.4.4 *Implementation details*

Both tools were implemented as type driven value generators using a visitor pattern. An user programmable generator consists of a function with a depth limit as an argument. When generating a value when no custom generator is defined for the given type, a new generator function for the type is generated at runtime, implemented in Java. Both tools are lightweight, concise and well isolated. Using a simple metric counting the number of newline characters, only 1848 lines of Java and 312 lines of Rascal were used to implement both tools, with 707 lines of Rascal to test the functional aspects of the implementation. The code is available online [26].

CONFIGURATION The function `quickcheck` takes the property function as an argument. There are two ways to override the default settings for the depth limit and number of tries. First, a property can be annotated using the `@maxDepth{...}` and `@tries{...}` annotations. Second, the function `quickcheck` allows for two optional parameters that override all other settings for depth limit and number of tries. Custom generators can be set at runtime using the two functions `setGenerator(&T (int) generator );` and `resetGenerator( type[&T] reified );` found in the module `cobra::quickcheck`. The function `smallcheck` takes the property function as an argument, followed by the depth limit and timeout value in milliseconds. Custom generators can be set at runtime using two corresponding functions to set and unset generators, found in the module `cobra::smallcheck`.

2.5 *Configuration*

Both `quickcheck` end `smallcheck` can be configured by the user. The following parameters can possibly effect the productivity of the tools. We handle this in two ways.

1. First, our goal is to compare both methods in a natural setting. Therefore we try to choose settings that arguable would have been chosen in a typical development situation and we document these choices.
2. Second, we are careful not to choose settings that inadvertently cripple a strategy.

There are several ways to configure the behavior of both tools;

1. Bounded exhaustive testing expects a timeout parameter and a depth limit.

2. Random testing expects a maximum number of values to generate and a depth limit.
3. It is possible to create user-programmable generators to build input values.

1. BOUNDED EXHAUSTIVE TESTING Bounded exhaustive testing terminates after the supplied timeout value has been reached. Following the motivation of bounded exhaustive testing, only a small part of the domain has to be explored. We iteratively increase depth, until a timeout of 10 seconds is triggered. Due to combinatorial explosion, there is often a clear cut off. For instance, depth 3 can contain just 10 cases, while depth 4 hits the timeout after ~60.000 cases. The depth limit is governed by the timeout setting. It turned out to be uncommon to reach a depth further than 5.

2. RANDOM TESTING With regard to random testing you would expect that the more values you test, the better the results. The longer you generate values, the greater the coverage of the input domain. However, there is evidence that if random testing finds bugs at all, it will do so quickly. After random testing a multiprocessor cache controller in the late 1980's, Wood et al. [28] reported that the bugs they found generally showed up within 300 simulation cycles. Ciupa et al. [4] performed contract-based random testing of Eiffel programs and reported that random testing detected a fault within at most 30 seconds. Forrester and Miller [9] subjected Windows NT programs to random testing and found that if an application was going to fail, it did so after a small number of test cases. Lindig [18] tested C calling conventions with a random testing framework inspired by QuickCheck [5]. He reported that most bugs show up quickly, and running the tool over a longer period of time rarely produced new bugs. We found that a limit of 100 case delivered consistent results and a convenient short runtime. Considering the depth limit, imagine testing a property with both bounded exhaustive testing and random testing, with the *same* depth limit. Bounded exhaustive testing will test the set of all input values within the depth bound. Random testing will only sample a subset thereof. So in this case, random testing obviously has a lower chance at finding failure-causing input values. This shows us that we need to make sure that the depth limit of random testing is sufficiently larger than that of bounded exhaustive testing when testing the same property. Moreover, a strength of random testing is to test complicated test cases. We found that a maximum depth of 10 delivered a convenient balance between runtime and results.

3. USER-PROGRAMMABLE GENERATORS Both tools allow for custom generators that override the behavior of the default generators. With a case study, it would make sense to allow for custom generators since we compare both methods in a natural setting. We chose not to do this, since it would hurt the reproducibility of our results and increase the risk at bias-introducing bugs. Also, with random testing a user programmable generator serves to change the distribution of input values. With bounded exhaustive testing, a user programmable generator serves to change the interpretation of depth and possibly ordering.

3 RESULTS

We constructed 26 properties for different parts of the Rascal project. Of those 26, 25 were refuted using random testing and 19 also using bounded exhaustive testing. In 8 of those 19, different failures were found by both tools. A detailed report of our results is attached at the end of this thesis. A summary of the results can be found in table 1. It reports the number of properties, number of failing properties detected by random or bounded exhaustive testing and the number of properties where one method returned a different failure than the other. In this section we explore some of the tests and interpret the results. We will also comment on the usage of the tools.

PART OF RASCAL	PROPERTIES	RT	BET	DISTINCT
A. Rascal syntax and Prelude	4	4	2	1
B. Rascal utility language implementations	12	12	11	5
C. Rascal demonstration languages	5	4	4	1
D. Derric	5	5	2	1
Total	26	25	19	8

Table 1: Summary of results

CONFIGURATION In section 2.5 we outlined the importance of configuration settings. In that section we documented the settings we used for our experiments.

3.1 *Observations*

DERRIC Derric is a declarative DSL. It is used to define the structure of a file format. One application of Derric is to create file carvers to recover information from files, which has already been used in digital forensics.

We constructed 5 tests for Derric. These test cases all conformed to the following pattern; given a function `f` in Derric with an argument of type `a`, `f` should not throw an Exception when called with any value of type `a`.

This turned out to be a rather powerful pattern, since we were able to trigger Exceptions for all functions we tested. Test D1 (appendix) is most notable, since it led us to submit a bug fix for Derric. At one

point in the development of Derric, the developer decided to change the signature of the function `writeExpression`. The random testing tool `quickcheck` discovered that there was still at least one instance where the old signature was called. Our bug report led to further investigation by the developer, resulting in the discovery of another instance were the old signature was called. This was regarded as a valuable contribution.

Another interesting example is test D5. With bounded exhaustive testing we found no failure, but with random testing we did. The function `generateGlobal` throws an exception *"Missing return statement"* when called with `gdeclV(float(big(),129001004),"")`. Further investigation led us to the following function:

```
public str generateFloatDeclaration(int bits, str name) {
        if (bits <= 32) return "float <name>;";
        else if (bits <= 64) return "double <name>;";
}
```

When this function is called with a value for `bits` that is larger than 64, it throws a *Missing return statement*. In the strict sense this is a failure, but its relevance is questionable, since float values with more than 64 bits are not sensible. Note that this is a bug that could never be found with bounded exhaustive testing since the failure-causing input is far outside the range of values that bounded exhaustive testing examines.

READAUT, WRITEAUT AUT files contain relations of type `rel[int, str, int]`. Rascal provides two complementary functions to read and write AUT files which are implemented in Java. We defined the following property (B1):

All values of type rel[int, str, int] can be written with writeAUT and consequently read by readAUT, without changing the value.

With both tools we quickly found a failure-causing input. The function throws an error `field access not supported on strat`. This test failed early and repeatable with both methods.

COMPILE PICO Pico is one of the demonstration languages in Rascal. This is the test we constructed (C3).

For all values of type PROGRAM, compileProgram should not throw an exception

Both random testing and bounded exhaustive testing we found a failure-causing input.

```
program(
        [],
        [
                whileStat(strCon(""),[])
        ]
)
```

However, it is not possible to create a program that results in this particular abstract datatype when parsing pico files; it is impossible to construct program that results in a `strCon("")` being parsed.

Although the exact mechanics of this failure or not as interesting, it does show an important limitation of uninformed testing in general. Uninformed testing will try all cases, even if they make no sense.

TURING MISSING ALTERNATIVE We tested the function turing2box (C5) with the following property;

For all values of type Program, turing2box should return a value of type Box.

Both tools quickly found a counter-example. The datatype Program defined in the file we tested is an extension of a type Program defined in a separate file. The function `turing2box` handles only the alternatives defined in the latter file. It is conceivable that this kind of mistakes tend to occur more often when definitions of datatypes that are split over multiple files. In Rascal it is impossible to describe the intention to handle all alternatives of a given datatype, so being complete in handling all alternatives is the responsibility of the programmer. This is a kind of failure that is typical for bounded exhaustive testing to find.

UNICODE FAILURES Recently, Unicode support was introduced to Rascal. Most functions in the String library were not yet ready since in Java, the function `String.length()` can not handle Unicode strings. With random testing it is easy to find this kind of failures, with bounded exhaustive testing it is infeasible since this kind of failures only show with high range characters. Bug A4 is an example.

DIFFERENTIAL COMPARISON The newly added statistics library contained an implementation 'sum'. We defined the following property:

For all lists of numbers, the function sum in Prelude should return exactly the same result as the function sum in the statistics library.

Differential testing is a powerful construct. With random testing, it has been used to test C compilers with a reference implementation [21]. One drawback is that it is not clear which implementation is at fault when a failure is triggered. In our setting, bounded exhaustive testing showed that the new implementation was not able to handle an empty list and random testing showed that there one implementation used a different precision when rounding.

PATTERNS IN PROPERTIES It turned out that it was not always straightforward to write good properties. Often it is easy to specify correct behavior in a specific case and to abstract to a more general property that holds for all possible input values is much harder. We discovered some patterns in the type of properties we defined.

- Differential testing: when two implementations of the same functionality exist, they should have the same results with the same input.
- Code should not fail, regardless of the input values. This is simple but very powerful pattern.
- When one operation is the inverse of another: for example; `unparse(parse(text)) == text`
- General invariants: after a set of operations, a certain property should still hold. For example, after simplifying an expression, the result should still be the same for all input values.

4 DISCUSSION

In this thesis we set out to compare two uninformed testing strategies in a practical setting. We argued that it was reasonable to compare both methods based on their relative efficiency. As a measure for efficiency, we compared the productivity of both methods. If one method finds a failure causing input for a given test and the other method does not, the first is considered having a greater productivity.

In 8 out of the 19 properties that failed using both random testing and bounded exhaustive testing, the failure triggered by random testing was different from the failure triggered by bounded exhaustive testing. This might indicate that both tools are complementary.

With respect to this measure, we discovered that random testing had a slight advantage. We defined several properties for the language Derric which were refuted with random testing but not with bounded exhaustive testing. One explanation could be that the implementation of Derric is more mature and production hardened than the other software we tested. One would expect that most bugs triggered by simple input values have surfaced already. Random testing tends to test unexpected and complicated cases. Off course this explanation goes against the motivation of bounded exhaustive testing, which states that when there are no failures that can be triggered by simple cases, there are none at all.

In our own experience, random testing was slightly more satisfying to use. Bounded exhaustive testing is deterministic, which means that when you run the tool twice with the same property it always delivers the same result. Random testing generally delivers a different result each time you run the tool, due to the non-deterministic nature of random testing. Another convenient quality of random testing is that the runtime is always more or less the same. With bounded exhaustive testing this depends on the breadth of the input domain, so the runtime varies from case to case. This is a minor inconvenience when using the tool, even though the timeout setting asserts that there is an upper limit to the waiting time.

A disadvantage of uninformed testing methods in general is that they only use knowledge about the input domain and not about the program. We encountered some cases where the testing method would generate input values that made no semantic sense, but caused the test to fail. It is a matter of opinion whether this kind of failures are bugs or not. On the other hand, the lack of knowledge is actually a strong point. Although uninformed testing will test cases that do not make any semantic sense, it will most certainly try cases the developer of the system has never thought of.

The process of defining properties was valuable during the development process as it increased our understanding of the code we tested. We think that property-based testing helps in the development of reliable code. The fact that we found failures with the code in part C, Rascal Demonstration Languages, might attribute to this observation, since this code can be considered less mature. The goal of this code is clarity and readability, not reliability.

4.1 *Limitations*

CONSTRUCT VALIDITY We chose to measure efficiency by comparing the relative ability to detect failures. With regard to construct validity you can argue that this measure is incomplete, since we do not weigh the impact a fault has on the user of the system under test. Some failures can severely limit the functionality of the system from the perspective of a user while others are just a nuisance. We argue that it is very unlikely for one method to find only faults that have a higher impact. Another threat to construct validity concerns the choice to consider the effort associated with both methods as similar. Our results showed that bounded exhaustive and random testing did find different failure causing inputs for the same tests in some cases. This might indicate that one method finds a different kind of failures that are more difficult to reduce to a fault. This warrants further investigation.

INTERNAL VALIDITY In terms of internal validity a substantial factor is the instrumentation we developed. One developer error in our tools could introduce a bias towards either testing strategy. Therefore, great care was taken to ensure the quality and accuracy of both tools. A smaller code size usually means less bugs [20]. Therefore, a design goal was to keep the tools lightweight and concise and we developed a body of automated functional tests. Moreover we tried to give a comprehensive and thorough description of our implementation to allow for our results to be replicated. This detailed characterization of the implementation can be found in section 2.4.4. Furthermore, an important aspect is the configuration of our tools. We documented how we tried to resolve this extensively in section 2.5.

EXTERNAL VALIDITY Since this is a case research study, the external validity is limited by design. The results can probably be replicated in software projects with a comparable domain, maturity and context using property-based testing.

4.2 *Related Work*

Random testing has enjoyed a fair amount of attention as a research subject in the past decades. As early as in 1984, Duran [7] investigated the merits of random testing when compared with conventional functional testing methods, through some empirical explorations. They identified random testing as a viable alternative testing strategy.

Miller et al. were able to crash 25-33% of the utility programs on several UNIX distributions using random testing in 1990 [23]. In 2000 they repeated the same research with Windows utilities and reported similar results [9].

Bounded exhaustive testing is motivated by the following observation: *"If a program does not fail in any simple case, it hardly ever fails in any case"* [24]. Variations of this hypothesis have been coined in related fields.

- The *'small scope hypothesis'* motivates model-checking tools such as Alloy [15]. When no counter example has been found in a small scope, the specification is probably correct.
- The *'competent programmer hypothesis'* has been adopted in mutation testing. It assumes that if a program has no errors that can be exposed using a few simple mutations, no errors exist at all [6].

Furthermore, much attention has been devoted to the investigation of techniques that seek to improve uninformed testing by adding some knowledge about the system under test. Korat [22] is a tool for bounded exhaustive testing of Java programs using a hard-coded specification of the scope. In more systematic random testing approaches, model checking, static program analysis and automated constraint solving are used to increase the relevance of the cases [12, 10, 11].

Based on the prior work done on this subject we certainly uniformed testing to be successful in this domain, since large input domains with complex input values and the possibility to define simple invariants to use as automated oracles are common traits in reports on the successful application of uninformed testing. Within the source code analysis and manipulation domain, complex data structures are common. To the best of our knowledge, we are the first to present an empirical comparison between bounded exhaustive testing and random testing.

An interesting observation with random testing is that most properties tended to fail quickly, or not at all [4, 9, 18]. Our results collaborate these findings, which provides some evidence that random

testing and bounded exhaustive testing can be used to provide continuous feedback during development.

4.3 *Implications*

Uninformed testing is a useful tool for software testing. It is complementary to more systematic approaches. Random testing and bounded exhaustive testing are two uninformed testing strategies that can provide useful testing results. Our results indicate that both tools are possibly complementary. Environments with large input domains and the possibility to define useful properties are especially suited for both techniques.

Finally, we integrated random testing with the testing framework in the Rascal distribution. It has already been released for public use and first reports have been enthusiastic.

4.4 *Conclusion*

We have taken two uninformed testing strategies and compared them in a practical setting. To our best knowledge we are the first to make this comparison.

The majority of the properties we defined for different parts of the Rascal project were refuted using both random testing and bounded exhaustive testing. When the properties were refuted with both tools, in approximately half of those cases, random testing found a failure different from bounded exhaustive testing.

We demonstrated that in the context of this study, random testing was slightly more valuable from a practical perspective using relative failure detecting ability as a measure for efficiency. Also, we experienced random testing to be slightly more user-friendly.

	Module	Description of property	Tested functions	RT failure	BET failure	different failure	RT failure	BET failure	# of BET tests
A. Rascal syntax features and core libraries									
A1	*builtin*	/^<var>$/ := var should hold for all values of type string.	string interpolation in regular expressions	yes	no		Syntax error: Illegal repetition near index 0		623530
A2	*builtin*	{_*, a} := {*as , a} should hold for all value a and set[value] as.	set matching	yes	yes	no	BoolValue cannot be cast to ISet failed with [{}, false]	BigDecimalValue cannot be cast to ISet failed with [{}, -1.0]	1
A3	*stat::Descriptive Prelude*	Prelude::sum(numList) == stat::Descriptive::sum(numList)	Prelude::sum(list[num]) stat::Descriptive::sum(list[num])	yes	yes	yes	failed with [[0.49944240363078163, 991090161r1967202527]]	Java("java.lang.NumberFormatException") failed with [[]]	1
A4	*Prelude*	The string function left should return a string with the right length for all characters c: size(left(c, 4, “x”)) == 4	left	yes	no		failed with high range Unicode characters.		623530
B. Rascal utility language implementations									
aut									
B1	lang::aut::IO	All values of type rel[int, str, int] can be written with writeAUT and consequently read by readAUT, without changing the value	writeAUT(rel[int, str, int]) readAUT(rel[int, str, int])	yes	yes	no	field access not supported on strat	field access not supported on strat	4
box									
B2	lang::box::util::HighlightToLatex	The function highlight2latex will produce a string for all values of type list[Box]	highlight2latex(list[Box] bs)	yes	yes	no	- Unhandled box	- Unhandled box KW(NULL())	3
B3	lang::box::util::Box2Text	For all values of type Box, box2latex should not throw an Exception	box2latex(Box b)	yes	yes	no	The called signature: RR(list[Box], Box, options, foptions, int),does not match the declared signature:list[list[Box]] RR(list[Box], Box, options, int);	The called signature: RR(list[Box], Box, options, foptions, int),does not match the declared signature:list[list[Box]] RR(list[Box], Box, options, int);	32
B4	lang::box::util::Box2Text	For all values of type Box, box2html should not throw an Exception	box2html(Box b)	yes	yes	yes	Unexpected error in Rascal interpreter: out of memory caused by Java heap space. Failed with [SPACE(205495894)]	Failed with [R([])] The called signature: RR(list[Box], Box, options, foptions, int), does not match the declared signature: list[list[Box]] RR(list[Box], Box, options, int);	32
B5	lang::box::util::Box2Text	For all values of type Box, box2text should not throw an Exception	box2text(Box b)	yes	yes	no	The called signature: RR(list[Box], Box, options, foptions, int),does not match the declared signature:list[list[Box]] RR(list[Box], Box, options, int);	Failed with [R([])] The called signature: RR(list[Box], Box, options, foptions, int), does not match the declared signature: list[list[Box]] RR(list[Box], Box, options, int);	32

	Module	Description of property	Tested functions	RT failure	BET failure	different failure	RT failure	BET failure	# of BET tests
B6	lang::box::util::Box2Text	For all values of type Box, format should not throw an Exception	format(Box b)	yes	yes	yes	Ambiguous code (internal error)	Failed with [R([])] The called signature: RR(list[Box], Box, options, foptions, int), does not match the declared signature: list[list[Box]] RR(list[Box], Box, options, int);	32
			csv						
B7	lang::csv::IO	All relations of type rel[value, value] can be written with writeCSV and read with readCSV, without changing the value.	writeCVS(&T relation, loc location), readCSV(type[&T] result, loc location)	yes	yes	yes	failed with [{<[],0.2999704585154547>,<{}, $23430841-11-02T14:27:25.654+0100>,<-1923486447r1216146914,1089748965r1186299286>}]	Failed with [{<0.0,<0.0>>}]	5
B8	lang::csv::IO	All relations of type rel[rat,int] can be written with writeCSV and read with readCSV, without changing the value.	writeCVS(&T relation, loc location), readCSV(type[&T] result, loc location)	yes	yes	no	failed with [{<328677286r566427353,802878002>}]	Failed with [{<0r,0>	2
B9	lang::csv::IO	All relations of type rel[str,str] can be written with writeCSV and read with readCSV, without changing the value.	writeCVS(&T relation, loc location), readCSV(type[&T] result, loc location)	yes	no		Fails with Unicode strings		512
B10	lang::csv::IO	All relations of type rel[datetime,str] can be written with writeCSV and read with readCSV, without changing the value.	writeCVS(&T relation, loc location), readCSV(type[&T] result, loc location)	yes	yes	yes	failed with [{<$279338066-06-11T01:24:47.085+0100,-500741497>}]	Failed with [{<$1970-01-01T01:00:00.000+0100,-1>}]	2
			dot						
B11	util::Dot	For all values of type DotGraph, toString should yield a string that is not empty.	toString(DotGraph g)	yes	yes	no	Undeclared variable, function or constructor: oStmt (should be oStm)	Undeclared variable, function or constructor: oStmt (should be oStm)	162744
			xml						
B12	lang::xml::DOM	For all values of type Node, xmlPretty should return a string, excluding expected Exceptions.	xmlPretty(Node n)	yes	yes	yes	Java("StringValue cannot be cast to IConstructor") failed with [document(entityRef(""))]	Java("IntegerValue cannot be cast to IConstructor") failed with [document(charRef(0))]	1
			C. Rascal demonstration languages						
			missgrant						
C1	demo::lang::MissGrant::CheckController	For all values of type Controller, checkController should not throw an Exception	checkController(Controller c)	yes	yes	yes	get-annotation not supported on strat	Failed with [controller([],[],[],[])] : EmptyList()	1
C2	demo::lang::MissGrant::ParallelMerge	For all values of type Controller, parMerge should not throw an Exception	parMerge(c)	yes	yes	no	Undeclared variable, function or constructor: unique	Undeclared variable, function or constructor: unique	1
			pico						

	Module	*Description of property*	*Tested functions*	*RT failure*	*BET failure*	*different failure*	*RT failure*	*BET failure*	*# of BET tests*
C3	demo::lang::Pico::Compile	For all values of type PROGRAM, compileProgram should not throw an Exception	compileProgram(PROGRAM p)	yes	yes	no	IndexOutOfBounds(1)	IndexOutOfBounds(1)	1
C4	demo::lang::Pico::TypeCheck	For all values of type PROGRAM, checkProgram should not throw an Exception	checkProgram(PROGRAM p)	no	no				65474
			turing						
C5	demo::lang::turing::l2::format::Format	For all values of type Program, turing2box should return a value of type Box.	turing2box(Program p)	yes	yes	no	The called signature: stat2box(Statement), does not match the declared signature: void (void);	The called signature: stat2box(Statement), does not match the declared signature: void (void);	5
			D. Derric						
D1	lang::derric::GenerateDerric	For all values of type Format, generate should not throw an unexpected exception	generate(Format f)	yes	yes	yes	The called signature: writeExpression(Expression), does not match the declared signature: str writeExpression(Expression, bool);	2: Failed with [format("",[],[],[node()],[])] Missing return statement	2
D2	lang::derric::BuildValidator	For all values of type Format, build (Validator) should not throw an Exception	build(Format f)	yes	no		The called signature: buildStatements(Field), does not match the declared signature: void (void);		4986
D3	lang::derric::CheckFileFormat	For all values of type Format, check should not throw an Exception	check(Format f)	yes	no		Undeclared variable, function or constructor: t		4986
D4	lang::derric::GenerateGlobalJava	For all values of type Global, generateGlobal should return a non empty string	generateGlobal(Global g)	yes	no		Missing return statement failed with [gdeclV(float(big(),129001004),"")]		82879
D5	lang::derric::GenerateStructureJava	For all values of type Structure, generateStructure not throw an Exception	generateStructure(Structure s)	yes	yes	no	Missing return statement failed with [structure("",[ldeclV(float(big(),880446176),"a")])]	Missing return statement Failed with [structure("",[readValue(float(little(),0),"")])]	5

REFERENCES

[1] Ieee standard glossary of software engineering terminology. (1): 1, 1990.

[2] K. Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[3] I. Benbasat, D.K. Goldstein, and M. Mead. The case research strategy in studies of information systems. *MIS quarterly*, pages 369–386, 1987.

[4] I. Ciupa, A. Pretschner, A. Leitner, M. Oriol, and B. Meyer. On the predictability of random tests for object-oriented software. In *Software Testing, Verification, and Validation, 2008 1st International Conference on*, pages 72–81. IEEE, 2008.

[5] K. Claessen and J. Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. volume 35, pages 268–279. ACM, 2000.

[6] R.A. DeMillo, R.J. Lipton, and F.G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4): 34–41, 1978.

[7] J.W. Duran and S.C. Ntafos. An evaluation of random testing. *Software Engineering, IEEE Transactions on*, (4):438–444, 1984.

[8] P. Duvall, S. Matyas, and A. Glover. *Continuous integration: improving software quality and reducing risk*. Addison-Wesley Professional, 2007.

[9] J.E. Forrester and B.P. Miller. An empirical study of the robustness of windows nt applications using random testing. In *Proceedings of the 4th conference on USENIX Windows Systems Symposium-Volume 4*, pages 6–6. USENIX Association, 2000.

[10] P. Godefroid, P. de Halleux, A.V. Nori, S.K. Rajamani, W. Schulte, N. Tillmann, and M.Y. Levin. Automating software testing using program analysis. *Software, IEEE*, 25(5):30–37, 2008.

[11] P. Godefroid, A. Kiezun, and M.Y. Levin. Grammar-based whitebox fuzzing. In *ACM SIGPLAN Notices*, volume 43, pages 206–215. ACM, 2008.

[12] P. Godefroid, M.Y. Levin, D. Molnar, et al. Automated whitebox fuzz testing. In *Proceedings of the Network and Distributed System Security Symposium*. Citeseer, 2008.

[13] D. Hamlet. When only random testing will do. In *Proceedings of the 1st international workshop on Random testing*, pages 1–9. ACM, 2006.

[14] P. Heidegger and P. Thiemann. Contract-driven testing of javascript code. *Objects, Models, Components, Patterns*, pages 154–172, 2010.

[15] D. Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11 (2):256–290, 2002.

[16] P. Klint, T. van der Storm, and J. Vinju. Rascal: A domain specific language for source code analysis and manipulation. In *Source Code Analysis and Manipulation, 2009. SCAM'09. Ninth IEEE International Working Conference on*, pages 168–177. IEEE, 2009.

[17] AS Kossatchev and MA Posypkin. Survey of compiler testing methods. *Programming and Computer Software*, 31(1):10–19, 2005.

[18] C. Lindig. Random testing of c calling conventions. In *Proceedings of the sixth international symposium on Automated analysis-driven debugging*, pages 3–12. ACM, 2005.

[19] P.M. Maurer. Generating test data with enhanced context-free grammars. *Software, IEEE*, 7(4):50–55, 1990.

[20] S. McConnell. Code complete: A practical handbook of software construction. 2004.

[21] W.M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

[22] A. Milicevic, S. Misailovic, D. Marinov, and S. Khurshid. Korat: A tool for generating structurally complex test inputs. In *Proceedings of the 29th international conference on Software Engineering*, pages 771–774. IEEE Computer Society, 2007.

[23] B.P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12): 32–44, 1990.

[24] C. Runciman, M. Naylor, and F. Lindblad. Smallcheck and lazy smallcheck: automatic exhaustive testing for small values. In *ACM SIGPLAN Notices*, volume 44, pages 37–48. ACM, 2008.

[25] J. van den Bos and T. van der Storm. Bringing domain-specific languages to digital forensics. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 671–680. IEEE, 2011.

[26] Wietse Venema. Source code of tools. URL `https://github.com/cwi-swat/rascal/tree/master/src/org/rascalmpl/library/cobra`.

[27] E.J. Weyuker and B. Jeng. Analyzing partition testing strategies. *Software Engineering, IEEE Transactions on*, 17(7):703–711, 1991.

[28] D.A. Wood, G.A. Gibson, and R.H. Katz. Verifying a multiprocessor cache controller using random test generation. *Design & Test of Computers, IEEE*, 7(4):13–25, 1990.

[29] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, pages 183–200, 2002.