Tests from Traces: Automated Unit Test Extraction for R

Filip Křikava
Czech Technical University
Czech Republic

Jan Vitek
Northeastern University and CTU
USA

ABSTRACT

Unit tests are labor-intensive to write and maintain. This paper looks into how well unit tests for a target software package can be extracted from the execution traces of client code. Our objective is to reduce the effort involved in creating test suites while minimizing the number and size of individual tests, and maximizing coverage. To evaluate the viability of our approach, we select a challenging target for automated test extraction, namely R, a programming language that is popular for data science applications. The challenges presented by R are its extreme dynamism, coerciveness, and lack of types. This combination decrease the efficacy of traditional test extraction techniques. We present GENTHAT, a tool developed over the last couple of years to non-invasively record execution traces of R programs and extract unit tests from those traces. We have carried out an evaluation on 1,545 packages comprising 1.7M lines of code. The tests extracted by GENTHAT improved code coverage from the original rather low value of 267,496 lines to 700,918 lines. The running time of the generated tests is 1.9 times faster than the code they came from.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**;

KEYWORDS

Test extraction, Program tracing, R

ACM Reference Format:
Filip Křikava and Jan Vitek. 2018. Tests from Traces: Automated Unit Test Extraction for R. In *Proceedings of 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'18).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3213846.3213863

1 INTRODUCTION

Testing is an integral part of good software engineering practices. Test-driven development is routinely taught in Computer Science programs, yet a cursory inspection of projects on GitHub suggests that the presence of test suites cannot be taken for granted and even when tests are available they do not always provide sufficient coverage or granularity needed to easily pinpoint the source of errors.

This paper explores a rather simple idea, namely, can we *effectively and efficiently extract test cases from program execution traces?* Our motivation is that if programmers do not write comprehensive unit test suites, then it may be possible for a tool to extract those for them, especially when the software to test is widely used in other projects. Our approach is as follows: For each project and its reverse dependencies, gather all runnable artifacts, be they test cases or examples, that may exercise the target, run the artifacts in an environment that records execution traces, and from those traces produce unit tests and, if possible, minimize them, keeping only the ones that increase code coverage.

The key question we aim to answer is: *how well can automated trace-based unit test extraction actually work in practice?* The metrics of interest are related to the quality of the extracted tests, their coverage of the target project, as well as the costs of the whole process. To answer this we have to pick an actual programming language and its associated software ecosystem. Any combination of language and ecosystem is likely to have its quirks, we structure the paper so as to identify those language specific features.

Concretely, this paper reports on our implementation and empirical evaluation of GENTHAT, a tool for automated extraction of unit tests from traces for the R statistical programming language [11]. The R software ecosystem is organized around a curated open source software repository named CRAN. For our purposes, we randomly select about 12% of the packages hosted on CRAN, or 1,545 software libraries. These packages amount to approximately 1.7M lines of code stripped of comments and empty lines. The maintainers of CRAN enforce the presence of so-called vignettes, these are documentation with runnable examples, for all hosted libraries. Some libraries come equipped with their own test cases. These are typically coarse grained scripts whose output is compared for textual equality. Our aim with GENTHAT is to help R developers extract unit tests that can easily pinpoint the source of problems, are reasonably small, provide good coverage, and execute quickly. Furthermore, we want to help developers to automatically capture common usage patterns of their code in the wild. In the R code hosted on CRAN, most of the code coverage comes from examples and vignettes, and very little is already in the form of tests. In the corpus of 1,545 packages we selected for this work, tests provide only an average of 19% coverage, whereas when examples and vignettes are executed coverage is boosted to 68%.

As of this writing, we are not aware of any other tool for automatically extracting test cases for R. This likely due to the limited interest that data analysis languages have garnered in our community, and also due to features of R that make it a challenging target for tooling. The language is extremely dynamic: it has no type annotations to structure code, each and every operation can be redefined during execution, values are automatically coerced from type to type, arguments of functions are lazily evaluated and can be coerced back to source expressions, values are modified using a

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ISSTA'18, July 16–21, 2018, Amsterdam, Netherlands
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-5699-2/18/07...$15.00
https://doi.org/10.1145/3213846.3213863

copy-on-write discipline most of the time, and reflective operations allow programs to manipulate most aspect of a program's execution state [9]. This combination of features has allowed developers to build complex constructions, such as support for object-oriented programming, on top of the core language. Furthermore, large swaths of the system are written in C code and may break any reasonable invariants one may hope for.

The contributions of this paper are the description of a tool, GENTHAT, for automatically extracting unit tests for R, as well as an empirical evaluation of that tool that demonstrates that for a large corpus, 1,545 packages, it is possible to significantly improve code coverage. On average, the default tests that come with the packages cover only 19%. After deploying GENTHAT we are able to increase the coverage to 53%. This increase mostly comes from extracting test cases from all the available executable artifacts in the package and the artifacts from packages that depend on this package. GENTHAT is surprisingly accurate, it can reproduce 80% of the calls executed by the scripts and it is also able to greatly reduce the number and size of test cases that are retained in the extracted suite, running 1.9 times faster than package examples, tests and vignettes combined with only 15% less code coverage (53% vs 68%). The reduction in coverage comes from limitations in what code can be turned into tests.

Artifact. The code of our tools, the analysis scripts used to generate the reports and sample data are available in the validated artifact accompanying this paper[1].

2 BACKGROUND AND RELATED WORK

This section starts with a brief overview of the relevant characteristics of the R programming language, as well as of its ecosystem, before discussing previous work on automated test extraction and its application to R.

2.1 The Language

The R programming language is a challenging language for tooling. The relevant features are the following:

- R does not have type annotations or a static type system. This means there is nothing to suggest what the expected arguments or return values of a function could be, and thus there is little to guide test generation.
- Symbols such as `+` and `(` can be redefined during execution. This means that every operation performed in a program depends on the state of the system, and no function call has a fixed semantics. This holds even for control flow operations such as loops and conditionals. While redefinitions are not frequent, they do occur and tools must handle them.
- Built-in types are automatically and silently coerced from more specific types to more general types when deemed appropriate.
- R is fully reflective, it is possible to inspect any part of a computation (e.g. the code, the stack or the heap) programmatically, moreover almost all aspects of the state of a computation can be modified (e.g. variables can be deleted from an environment and injected in another).
- All expressions are evaluated by-need, thus the call `f(a+b)` contains three delayed sub-expressions, one for each variable and one for the call to plus. This means that R does not pass values to functions but rather passes unevaluated promises (the order of evaluation of promises is part of the semantics as they can have side effects). These promises can also be turned back into code by reflection.
- Most values are vectors or lists. Values can be annotated by key-value pairs. These annotations, coupled with reflection, are the basic building blocks for many advanced features of R. An example of this are the four different object systems that use annotations to express classes and other attributes.
- R has a copy-on-write semantics for shared values. A value is shared if it is accessible from more than one variable. This means that side effects that change shared values are rare. This gives a functional flavor to large parts of R.

R is a surprisingly rich language with a rather intricate semantics, we can't do it justice in the space at hand. The above summary should suffice for the remainder of this paper.

2.2 The Ecosystem

The largest repository of R code is the Comprehensive R Achive Network (CRAN).[2] With over 12,000 packages, CRAN is a rapidly growing repository of statistical software[3]. Unlike sites like GitHub, CRAN is a curated repository. Each program deposited in the archive must come with documentation and abide by a number of well-formedness rules that are automatically checked at each submission. Most relevant for our purpose, packages must have documentation that comes in the examples and vignettes, and, optionally, tests. All executable artifacts are run at each commit and for each new release of the language. Vignettes can be written using a variety of document processors. For instance Figure 1 contains a (simplified) vignette for a package named A3. That particular vignette combine text formatting comments, bits of documentation with code. For our purposes, the key part is the section tagged `examples`. That section contains code which is a runnable example intended to showcase usage of the A3 package.

While it is remarkable that every package comes with data and a least one vignette, this nevertheless does not necessarily make for good tests. There are two issues with R's approach, vignettes are coarse grained, they typically only exercise top-level functions in a package, and, they do not specify their expected output, sometimes that output is graphics, often it has some textual output. As shown above, vignettes are long-form guides to packages: they describe the problem that the package is designed to solve, and then show how to solve it. Their primary audience is developers. A vignette should divide functions into useful categories, and demonstrate how to coordinate multiple functions to solve problems. Code can be extracted from a vignette and run automatically. One can therefore only assert whether code extracted from examples or vignettes ran without throwing any exceptions and whether the output is similar to the last time this was run, but nothing can be said about the correctness of its output and if there is a difference in output

[1] https://github.com/fikovnik/ISSTA18-Artifact

[2] http://cran.r-project.org

[3] CRAN is receiving about 6 new packages a day [8]

```
% Generated by roxygen2

\name{a3}\alias{a3}

\title{A3 Results for Arbitrary Model}

\usage{a3(formula, data, modelfn, model.args = list(),...)}

\arguments{
 \item{formula}{the regression formula}
 \item{data}{a data frame containing data for model fit}
 \item{modelfn}{the function that builds the model}
}

\description{Calculates A3 for an arbitrary model.}

\examples{
 summary(lm(rating ~ ., attitude))
 a3(rating ~ ., attitude, lm, p.acc = 0.1)
 require(randomForest)
 a3(rating ~ .+0, attitude, randomForest, p.acc = 0.1)
}
```

Figure 1: Sample vignette.

whether this is a substantial departure or if it is an accident of the way numbers are printed.

In our experience, most R developers are domain experts, rather than trained software engineers; this is a possible explanation for the relatively low proportion of unit tests in packages hosted in CRAN. Most tests are simply R scripts that are coupled with a text file capturing expected textual output from the interpreter running the given test [12]. The most popular unit testing framework, called `testthat`, is used by only 3,017 packages (about 23% of the total number of packages in CRAN).[4] In our experience, even when `testthat` is used, only a few tests are defined, covering only a subset of the interface of the package.

2.3 Automating Test Extraction

Our goal is to extract black-box unit tests for packages from a large body of client code. The benefit we are aiming for is to reduce the manual effort involved in the construction of such regression testing suites and not necessarily to find new bugs. To the best of our knowledge there are no tools that address this issue for R, and the previous work requires substantial adaptation to translate to our context.

Test generation is a topic with an extensive body of work. We will only touch on the research directly relevant to our development. We focus on automated techniques that do not require users to write specifications or to provide additional information. The literature can be split into work based on static techniques, dynamic techniques and a combination of both.

Static techniques use the program text as a starting point for driving test generation [1, 5, 16]. The difficulty we face with R is its extreme dynamism. Consider the following expression `f(x/(y+1))`. The semantics of R entails that definitions of `f`, `/`, `+`, and even `(` have to be looked up in the current environment. Given that `x` and `y` may have class attributes, any of those functions could have to dispatch on those. At any time, any of these symbols can be redefined (and in practice they are, sometimes for good reasons [9]). It is also possible for code to reflectively inject or remove variable bindings in any environment and turn any expression back into text that can be manipulated programmatically and then turned back into executable code.

One approach that would be worth exploring, given that sound static analysis for R appears to be a non-starter, is some combination of static analysis and machine learning. For instance, MSeqGen uses data obtained by mining large code bases to drive test generation [19]. So far we have not gone down that road.

Dynamic approaches for generating unit tests often rely on some form of record and replay. Record and replay has been used to generate reproducible benchmarks from real-world workloads [13], and, for example, capture objects that can be used as inputs of tests [6]. Joshi and Orso describe the issues of capturing state for Java programs [7]. Test carving [4] and factoring [15] have very similar goals to ours, namely to extract focused unit tests from larger system tests. These works mostly focus on capturing and mocking up a sufficiently large part of an object graph so that a test can be replayed. Rooney used similar approach to extract tests during an actual use of an application through instrumentation [14]. While R has objects, their use is somewhat limited, they are typically self=contained, and capturing them in their entirety has worked well enough for now.

In terms of pushing our work towards bug finding, we pine for the sanity of languages such as Java or even JavaScript, as in these languages it is not too hard to get an errant program to throw an exception—null pointers or out of bound errors are reliable oracles that something went wrong [3]. In R, most data type mismatches cause silent conversions, out of bound accesses merely extend the target array, and missing values are generated when nothing else fits; none of this causes the computation to stop or is an obvious sign that something went wrong.

Finally, there are few tools dealing with languages that support call-by-need. The most popular tool in that space is QuickCheck [2] for Haskell. It leverage types as well as user-defined annotations to drive random test generation.

3 GENTHAT: DESIGN & IMPLEMENTATION

An *execution trace* is the sequence of operations performed by a program for a given set of input values. We propose to record sets of execution traces of clients of a target package and from those traces extract unit tests for functions in either the public interface or the internal implementation of that target package. To create those unit tests, only data needed to execute individual function calls is required. This boils down to capturing function arguments and any global state the function may access or rely on. To validate that the function ran successfully, outputs must be recorded so that the value observed in the trace can be compared with results of running the generated test.

Consider Figure 2 which is illustrates generation of a test for function `subset` in some target package. It features code from a client of that target package `(a)`, the target package `(b)` that is

[4] https://cran.r-project.org/web/packages/testthat

```
> a <- c("Mazda", "Fiat", "Honda")
> b <- c(110, 66, 55)
> cars <- data.frame(name=a, hp=b)


> subset(cars, cars$hp >= 60)

     name  hp
1   Mazda 110
2    Fiat  66
```

(a) Client code

```
subset <- function(xs, p) {
  # filter the data frame rows
  xs[p, ]
}
```

(b) Package

```
test_that("subset", {
  cars <- data.frame(
    name=c("Mazda", "Fiat", "Honda"), hp=c(110, 66, 55)
  )

  expect_equal(
    subset(xs=cars, p=cars$hp >= 60),
    data.frame(name=c("Mazda", "Fiat"), hp=c(110, 66))
  )
})
```

(c) Extracted test

Figure 2: Extracting a test case for `subset`.

```
> a <- c("Mazda", "Fiat", "Honda")
> b <- c(110, 66, 55)
> cars <- data.frame(name=a, hp=b)

# hp is only defined in the cars data frame
> subset(cars, hp >= 60)

     name  hp
1   Mazda 110
2    Fiat  66
```

(d) Client code

```
subset <- function(xs, p) {
  # capture the expression without forcing promise
  expr <- substitute(p)
  with(xs, xs[eval(expr), ])
}
```

(e) Package

```
test_that("subset", {
  cars <- data.frame(
    name=c("Mazda", "Fiat", "Honda"), hp=c(110, 66, 55)
  )

  expect_equal(
    subset(xs=cars, p=hp >= 60),
    data.frame(name=c("Mazda", "Fiat"), hp=c(110, 66))
  )
})
```

(f) Extracted test

Figure 3: Tracing lazy-evaluation and promises.

to be tested, and the corresponding generated test (`c`). The client code creates a data frame with information about vehicles and then filters out cars with horse power less than 60. The `subset` function subsets the given data frame (the `cars$hp >= 60` returns a vector of booleans `c(TRUE,TRUE,FALSE)` that is used to filter out data frame rows in `xs[p, ]`). The generated test first recreates the argument of the call, then it calls the `subset` function and checks the return value.

In a simpler language, all one would need to do would be to record argument and return values as shown in the example of Figure 2. Not so in R. Lazy evaluation complicates matters as argument are passed by *promise*, a promise is a closure that is evaluated at most once to yield a result. Thus, at function call one cannot record values as they are not yet evaluated. The example in Figure 3 shows an improved version of the `subset` function which allows programmers to reference the columns directly so the sub-setting predicate in the client code (`d`) can be more directly written as `hp>=60` instead of `cars$hp>=60`. But, in (`d`) the is no definition for `hp`, so the code would be incorrect if the argument was evaluated eagerly. Instead, in (`e`) reflection is used to reinterpret the argument. The expression `substitute(p)` does not force a promise, instead it simply gets the source code of the expression passed as argument `p`. This is subsequently used in the `with` function that evaluates its arguments in an environment that has been enriched by content of the first argument. Thus `hp>=60` is evaluated in the context of `cars` which has a column named `hp`. Now, the problem for our tool is that we cannot simply record the values of the passed arguments because doing so would force promises that cannot be evaluated in the current frame. In our example, trying to record `p` would force the evaluation of `hp>=60` which would fail since there is not variable binding called `hp` at the scope of the call. In general, manually forcing a promise is dangerous. It may alter the behavior of the program since promises can have side effects. Moreover the reflective capabilities of R dictate that the generated expression must be as close to original call as possible. Any syntactic change may be observable by user code. The generated test thus attempts to retain the structure of the client source code as much as possible.

3.1 Genthat Overview

Figure 4 presents an overview of the main phases of Genthat:

(a) **Install**: given a target package, the package is downloaded from CRAN and installed in the current R environment. Furthermore, any package in CRAN that transitively depends on the target package is also acquired.
(b) **Extract**: executable code is extracted from the examples and vignettes in the installed packages. Target package functions that are to be tested are instrumented. Depending on which option is selected, either the public API functions or the private ones are decorated. All executable artifacts are turned into scripts. Each script is a self-contained runnable file.

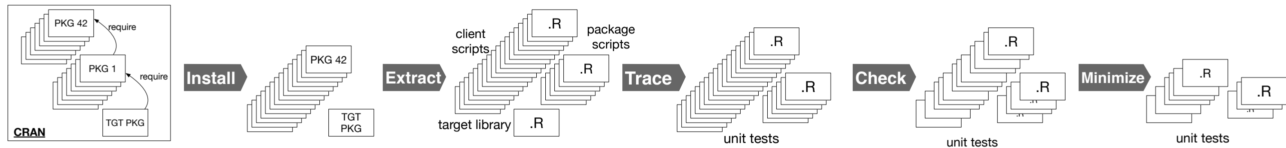

Figure 4: Overview. GENTHAT **installs packages from CRAN, it extracts scripts, traces their execution, and generates unit tests. Unit tests are checked for correctness and validity, and finally tests are minimized.**

(c) **Trace**: for each script, run that script and record trace information. From these traces generate unit tests for all calls to target functions.
(d) **Check**: all unit tests are checked for validity and correctness. Valid tests are those that execute without error. Correct ones are those that return the expected result. Invalid and incorrect tests are discarded. Code coverage data is recorded for the tests that are retained.
(e) **Minimize**: optionally, minimize the test suite. Minimization uses simple heuristics to discard tests that do not increase code coverage. Coverage being equal, tests that are, textually, smaller are preferred.

Test extraction can fail. Section 4.4 details the reasons, but at a high-level, the failures can occur during (a) tracing because the instrumentation perturbs the behavior of the program, (b) generation because some value could not be serialized, (c) validation because deserialization fails, and (d) correctness checking because the test was non-deterministic or relied on external state that was not properly captured. These failures account for the difference in coverage between the original traces and the extracted tests.

We will now describe salient points of our implementation. R is an interpreted language. A straightforward solution would therefore be to modify the R interpreter. However, one of our design goals was to make GENTHAT available as a package in the CRAN repository so that developers can use it without having to modify their distribution. This limits the implementation of GENTHAT to publicly available APIs of the R environment. The implementation is mostly written in R. The only exception is the serializer which is written in C++ for performance reason; a serializer written in R turned out to be an order of magnitude slower.

The whole system consists of 4,500 lines of code and 760 lines of C++ code. The software is released in open source[5].

3.2 Tracing

Tracing involves capturing function invocations and recording the inputs and outputs, including any additional details need to reproduce the function call in an isolation. Consider the following example of a function that filters a vector `xs` by a predicate `p`:

```
filter <- function(xs, p) xs[sapply(xs, p)]
```

Tracing is realized by instrumentation of R functions. Code is injected in target function bodies to record argument values, return value, and the state of the random number generator.[6] After instrumentation, the above function body is rewritten to:

```
`_captured_seed` <- get(".Random.seed", env=globalenv())
on.exit({
  if (tracing_enabled()) {
    disable_tracing()
    retv <- returnValue(default = `_deflt_retv`)
    if (normal_ret(retv)) {
      record_trace(name = "filter", pkg = NULL,
                   args = as.list(match.call())[-1],
                   retv = retv,
                   seed = `_captured_seed`,
                   env = parent.frame())
    }
    enable_tracing()
}})
xs[sapply(xs, p)] # original function body
```

The inserted code has a prologue that runs before any of the original function's code and an epilogue that runs right before the function exits. The latter relies on the `on.exit` hook provided by R. The prologue records the random seed. Due to R's lazy nature, recording of the argument values is done at the end of the function to avoid forcing a promise before it is needed—that might cause a side effect and change the result of the program or cause an error. With the exception of environments, R values are immutable with a copy-on-write semantics, so any modification of function arguments in the function's body will trigger their duplication. There are two things that need to be done before recording a call. First, tracing must be temporally disabled as it could recursive. Next, we need to check whether the function terminated normally or threw an exception. A call is only recorded in the case the function did not fail. While most values can be easily recorded, three type of values must be handled specially: symbols, expressions, and closures. Consider this example:

```
m <- 0.5; n <- 1;
filter(runif(10) + m, function(x) x > n)
```

The code has a call to `filter` with a nested, anonymous function definition which captures the variable `n` from the environment at the call site. Captured variables need to be resolved to their actual values. R has a built-in code analysis toolkit[7] that can report the list of free variables used by a closure. In the above call the set of free variables is `` `{`, runif, `+`, m, n, `>` ``. They are resolved to their values using the hierarchy of environments starting with the enclosing environment of the traced function. Extra care is needed for symbols coming from other packages. For those, we do not store values but instead a call to get their values at runtime. In the example, `runif` is a function from the stats package so the

[5] https://github.com/PRL-PRG/genthat
[6] In statistics, random number generators are omnipresent.
[7] https://cran.r-project.org/web/packages/codetools/

call `stats::runif` will be kept. Since it is possible to redefine any symbols including `` `+` `` and `` `>` ``, we have to resolve them as well. However, from the base library, we only keep the symbols that were modified. The complete trace for the above call will therefore be:

```
$ :List of 6
 ..$ fun   : chr "filter"
 ..$ pkg   : NULL
 ..$ args  :List of 2
 .. ..$ xs : language runif(10) + m
 .. ..$ fun: language function(x) x > n
 ..$ globals:List of 3
 .. ..$ runif: language stats::runif
 .. ..$ m   : num 0.5
 .. ..$ n   : num 1
 ..$ seed  : int [1:626] 403 20 -1577024373 1699409082 ...
 ..$ retv  : num [1:3] 0.5374310 1.4735399 0.9317512
```

The very same has to be done for global variables that are closures. For example, if the filter function is called as follows:

```
gt <- function(x) x > n
filter(runif(10) + m, gt)
```

The tracer needs to capture the free variable `n` and store it in the environment of the `gt` function. The `args` and `globals` of this call become:

```
 ..$ args  :List of 2
 .. ..$ xs : language runif(10) + m
 .. ..$ fun: symbol gt
 ..$ globals:List of 3
 .. ..$ runif: language stats::runif
 .. ..$ m   : num 0.5
 .. ..$ gt  : language function(x) x > n
```

The value of `n` is stored in the enclosing environment of the `gt` function.

3.3 Generating Tests

To generate unit tests out of the recorded trace we need to write out arbitrary R values into a source code file and to fill a template representing a unit test with the generated snippets of code. We first describe the general mechanism for serializing values to source code also known as *deparsing*.[8] The `deparse` function turns some values into a character string. Unfortunately, it handles only a subset of the 25 different R data types [10]. Since any of those data types can show up as an argument to a function call, we had to provide our own deparsing mechanism to support them all. In general we strive to output textual forms of arguments because they can be inspected and modified by developers. But there are values for which it is either impractical (*e.g.* large vectors or matrices) or not possible (*cf.* below) to turn them back into source code, for those we fallback to built-in binary serialization. The data types can be divided into the following groups:

Basic values There are 6 basic vector types (logical, integer, numeric, character, complex, and raw). Together with types representing internal functions (*e.g.*, `if`, `+`) as well as the type representing the `NULL` value, they can be turned into source code using `deparse`. The subtleties of floating-point numbers and Unicode characters are therefore handled directly by R.

Lists and symbols Lists are polymorphic, they can contain any value and therefore we cannot use `deparse`. For symbols, `deparse` does not handle empty symbol names and does not properly quote non-syntactic names.

Environments Environments are mutable hash maps with links to their parent environments. They are not supported by `deparse`. Serialization is similar to lists; however, we need to keep track of possible cycles. It is also important to properly handle named environments and exclude the ones that were imported from packages.

Language objects A language object contains the name of the function and a list of its arguments values. Additionally to ordinary functions, there are also unary functions, infix functions, `()`, `[]`, `[[]]` and `{}` all of which needed to be properly handled.

Closures Closures are triples that contain a list of formal parameters, code, and a closing environment. To serialize them properly, we need to resolve all their dependencies which is a transitive closure over its global symbols. We used the same mechanism for recording closures.

S4 S4 is one of the builtin R object systems. S4 classes and objects are internally represented by a special data type. They are complex and are currently serialized in a binary form.

Promises If a promise is encountered, the serialization tries to force it while hoping for the best. They are infrequent, because, when we serialize all the arguments to the function which have been used have been forced.

External pointers External pointers are used for system objects such as file handles or in packages that expose native objects. They are linked to the internal state of the interpreter and there is no general mechanism for persisting them across R sessions. They are currently not supported.

R internal values These values are used within R's implementation. There is no reason these should be exposed in a unit test and thus they are not supported.

Some values might be large in their serialized form. For example, the only way to capture the random number generator is to write out its entire state, which is a vector of over 600 floating-point numbers. If we were to add that to the generated test, it would create ungainly clutter; 126 lines of the test would be full of floating point values that are of no obvious advantage for the reader. Such values are therefore kept separately in an environment that is saved in a binary form next to the test file.

With all the trace values serialized, generating a unit test is merely filling a string template and reformatting the resulting code for good readability. For the latter we use the formatR package[9]. The extracted unit test from the second example call `filter(runif(10) + m, gt)` is listed in Figure 5. The `.ext.seed` referencing an external value which will be loaded into the test running environment at runtime.

[8] In the R world, serialization refers to a binary serialization while deparsing denotes turning values into their source code form.

[9] https://cran.r-project.org/web/packages/formatR/index.html

```
library(testthat)

.Random.seed <<- .ext.seed

test_that("filter", {
  m <- 0.5
  gt <- genthat::with_env(function(x) x > y, list(y=5))
  expect_equal(
    filter(xs=stats::runif(10) + m, fun=gt),
    c(0.5374310 1.4735399 0.9317512)
  )
})
```

Figure 5: Extracted test for `filter(runif(10)+m,gt)`.

3.4 Minimization

Once tests have been created, GENTHAT checks that they pass and discards the failing ones. At the same time we record their code coverage. Given the code coverage we can minimize the extracted test suite, eliminating redundant test cases [21]. Our minimization strategy is a very simple heuristic that aims to decrease the number of tests and to keep the size of individual tests small. In future work we will investigate alternative goal functions such as minimizing the combination of test size, test number, and execution time.

The current heuristic works as follows. Tests are sorted by increasing size (in bytes), with the intuition that, all things being equal, we prefer to retain smaller tests over the larger ones. We use the `covr` package to compute code coverage during minimization. The approach is to take each test, compute the coverage and compare that coverage to the union of all the previously retained tests. If the new test's coverage is a subset of the coverage so far, the test can be discarded. If the test increases coverage, it is retained. In practice, only a small fraction of tests are retained.

It is technically possible to compute the code coverage during tracing, discarding redundant traces. However, running R code with code coverage is slow and since there are many duplicate calls (only a third of the calls in our experiment were unique) this would impose a significant overhead. Instead we discard duplicate calls during tracing and then run the minimization at the end.

4 EVALUATION

To evaluate GENTHAT, we were interested in finding out how much we can improve test coverage by extracting tests from documentation and reverse dependencies and how efficient such an extraction can be. We were also interested in finding what proportions of the functions calls can be turned into test cases and how large the resulting test suites would become.

For these experiments, we selected 1,700 packages from CRAN. We picked the 100 most downloaded packages from RStudio CRAN mirror[10] including their dependencies and then 1000 random selected CRAN packages, again including their dependencies. This added up to some 1,726 packages. The motivation for this choice is to have some well established and popular packages along with a representative sample of CRAN. From the 1,726, some 1,545 ran succesfully. The remaining 181 failed either because of a timeout (5 hours during tracing), a runtime error, or failure to compute code coverage. The packages amounted to 1.7M lines of R (stripped of comments and empty lines). There were 158,208 lines of examples (on average 102 lines per packages), 32,524 lines of code extracted from vignettes (on average 21 lines per package) and 163,835 lines of tests (on average 106 lines per package).

The experiments were carried out on two virtual nodes (each 60GB of RAM, 16 CPU at 2.2GHz and 1TB virtual drive) using GNU parallel [17] for parallel execution. We used GNU R version 3.4.3[11] from Debian distribution.

[10]Used data for June 2017 from http://cran-logs.rstudio.com.

[11]Released in November 2017

4.1 Scale

Table 1 presents an overview of the scale of the experiment in terms of number of function calls and number of extracted tests. Out of 5.3M of function calls, GENTHAT traced 1.6M unique calls, calls with distinct arguments and return values. Out of the recorded traces, 93.6% were saved as unit tests. On average, 86.2% of the generated tests were valid and correct. From the total of 1.3M correct tests 97.9% were redundant, *i.e.* not increasing code coverage. Finally, some 26,838 tests were retained.

Table 1: Function calls & extracted tests for 1,545 packages (*s*=standard deviation, *m*=median). Reproducibility is the ratio between the number of valid and correct tests and the number of traced unique calls.

	Overall	Average per package	
Total number of calls	5,274,108	3,413	(s=13,375 m=141)
Traced unique calls	1,615,151	1,045	(s=3,145 m=82)
Extracted tests	1,512,555	979	(s=3,067 m=68)
Valid & Correct tests	1,303,114	843	(s=2,823 m=50)
Non-redundant tests	26,838	17.4	(s=33 m=9)
Reproducibility	0.8	0.75	(s=0.3 m=0.9)

4.2 Coverage

Figure 6 compares the code coverage obtained with the default tests and the coverage obtained using GENTHAT using code from examples, vignettes and tests. The default tests provide very little coverage for many packages. GENTHAT is able to increase the coverage from an average of 19% to 53% per package. This is a significant improvement.

4.3 Performance

Table 2 presents the average time of tracing and generating tests for our 1,545 packages (tracing), running the generated tests (testing w. GENTHAT), and running all of the code that comes with the package (default tests, vignettes and examples); *s* is standard deviation and *m* is median. In our testbed, running 16 packages in parallel on each node, the tracing took 1d 18h, running genthat tests 19m and running the package code 1h 09m.
Tracing clearly has a significant overhead. The median time is 60 seconds. Given that test generation is a design time activity, this is likely acceptable. The main portion of the time is spent in running

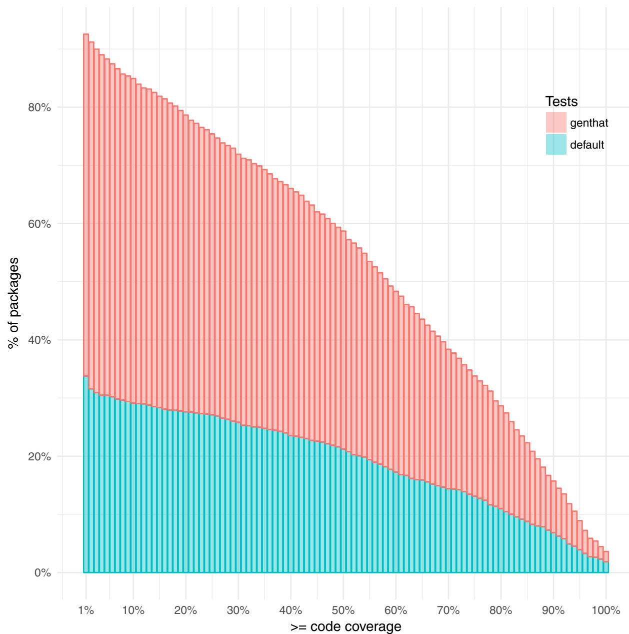

Figure 6: Code coverage from package tests and GENTHAT extracted unit tests. Each bar shows the ratio of packages that have at least that code coverage.

Table 2: Average running time for 1,545 packages.

Task	Average per package
Tracing	420s (s=1500, m=60)
Testing w. GENTHAT	15s (s=29, m=8)
Testing (default)	28s (s=60, m=10)

the generated tests which are numerous. On average, each package yields 1000 tests. They all have to be run to determine validity and correctness, as well as test minimization to determine whether they increase code coverage. On the other hand, testing with the minimized tests runs quickly. We are able to process all the packages in 19 minutes. Thanks to test minimization, this is faster than the time it would take to run all of the original examples, tests and vignettes. Concretely, running GENTHAT extracted tests for the 1,545 is 1.9 times faster. The resulting coverage is 15% smaller, but what we provide are actual unit tests while the code in examples and vignettes (and often in R tests as well) is just a plain R code with no assertion of expected behavior other than that it runs.

4.4 Accuracy

In terms of accuracy, GENTHAT managed to recreate 80% of all the unique function calls. Here we discuss the remaining 20%. Problems can be divided into three groups: tracing, generation, and replay. They are summarized in Table 3.

The tracing inaccuracies happen while recording function arguments. They account for 4.3% of the failures. Most are due to trace size; we intentionally discard values larger than 512KB in the R binary data format. Our reasoning is that tests which are too large are awkward to work with. The size limit was chosen empirically such that no single package loses more than 10% of its tests. The largest test was 887 MB (on average, skipped tests were 3.5 MB). Among the other tracing inaccuracies are the triple-dots (. . .) used in an incorrect context and missing symbols. Both problems are related to non-standard evaluation [20] in which the function arguments are not meant to be resolved in the usual scope. In most cases, GENTHAT records arguments correctly, but sometimes (0.2%) it fails to resolve them properly. The remaining problems are difficult to classify; many issues are related to some of the corner cases and issues with GENTHAT itself. They contribute very little to the overall failure rate.

Table 3: Issues in 312,037 failed test extraction.

	Overall	%
Tracing	70,020	4.3%
- skipped traces (size > 512kB)	60,360	
- . . . used in an incorrect context	2,150	
- unable to resolve symbol	1,419	
- unclassified	6,091	
Test generation	32,576	2.0%
- environments with cycles	24,200	
- other serialization errors	1,655	
- unclassified	6,721	
Test replay	209,441	12.9%
- incorrect tests (value mismatch)	77,850	
- invalid tests (execute with error)	131,591	

The test generation inaccuracies are related to environment serialization in the presence of cycles. This happens for example if there is a variable in an environment that references the same environment or any of its parents. This is a known issue that shall be fixed in future version of GENTHAT. Other serialization problems are related to unsupported values such as weak references or byte code. Similarly, to the unclassified tracing errors, the rest of the problems does not have a common cause. Some problems come from the `deparse` function. Again, they occur very rarely.

The final category is responsible for most of the inaccuracies. They are the most difficult to reason about since they require knowledge of the actual package code. However, there are some common causes:

- External pointers (pointers to native code) are not supported, so any functions that use them explicitly or transitively will not work.
- We do not keep any state, so if there is an expected sequencing for function calls GENTHAT will not work (for example, `close` must follow `open`).
- Non-standard evaluation and serialization do not always play well together. Currently, GENTHAT does not have any heuristics to guess in which evaluation mode a function uses and thus the serialization might fail.

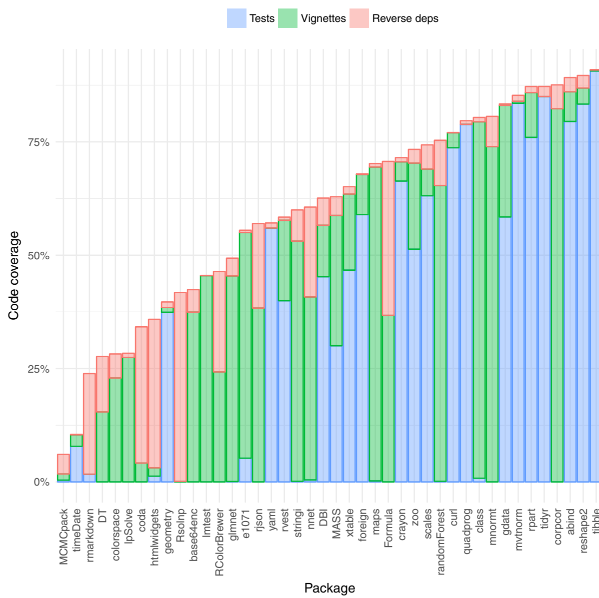

Figure 7: Code coverage obtained with reverse dependencies.

– Out of the 4 popular object systems used in R, we only cover the builtin S3 and S4. Reference classes and R6[12] are currently not supported.

There is one last source of inaccuracy. The function calls tracing happens in the `on.exit` hook that is injected into method bodies. While this mostly works, the hook can be overwritten and the entire call will be missed. To quantify the frequency of such event, we instrumented the code with a counter to count the number of function calls. In the 1,545 packages, the counter registered 5,274,108 function calls, on average 3,413 per package. The `on.exit` hook registered 245,815 fewer calls (5% less). Currently, the `on.exit` hook is the only way to inject code into a R function without changing the R interpreter. We have raised this question of how to avoid hooks being overridden with the Core R team and a solution will be hopefully included in an upcoming release.

4.5 Reverse Dependencies

One of the ideas we wanted to explore is to use the code found in clients of a package to generate additional unit tests. As computing test for the transitive closure of a package's reverse dependencies can be quite expensive we randomly selected 65 packages among all packages that had at least 20 reverse dependencies. For each of these package, we downloaded all of the packages that depend on it, extracted scripts from those packages tests, vignettes and examples, and used those scripts to extract test cases. In 42 cases, using reverse dependencies improved code coverage. On average, the coverage increased from 52% to 60%. Figure 7 is a histogram of the cumulative test coverage obtained from only the default tests, the GENTHAT results for the package only, and the GENTHAT results including all reverse dependencies. The improvements are significant in some cases.

[12] https://cran.r-project.org/web/packages/R6/

5 DISCUSSION

Following the evaluation we present the threats to validity of the selected approach, propose some of its application and discuss a possible generalization.

5.1 Threats to Validity

There are certain limitations to our work. Some can be mitigated by more engineering work, some is intrinsic to the approach.

Large tests. Programs in R naturally tend to use large vectors and matrices. Serializing them into source code results in rather a large unreadable chunks of tests which have a little value for the reader. On average the generated tests are 18.58 kB (median 490.00 B). The largest test that was generated was 1.55 MB of R code. GENTHAT can already be tuned to discard traces that exceed certain size already at runtime (*cf.* Section 4.4). Further, large values can be separated into the `.ext` file keeping only a reference in the unit test code.

Non-determinism. If a function call includes non-determinism which is not captured by the value of calling arguments, the generated tests will most likely fail.

Over specific testing oracle. The resulting unit tests directly represents the code from which they were extracted. They are as good as the source code it. If a package does not include much runnable artifacts or does not have many clients, GENTHAT will not do much.

Brittleness. Extracting tests from existing code can in general produce false positives (extracted tests passes while the original code from which it is generated fails) or false negatives (extracted tests fails while the original code runs) [14]. False positives do not occur in GENTHAT since if a function call fails, GENTHAT does not keep the trace (*cf.* Section 3.2). False negatives, apart from the reasons discussed in 4.4, also occur when the tests are run on a different version of a package than the one that was used to extract the test suite. If the code base changes, the extracted tests naturally becomes out of date. The same happens for manually written tests. The advantage of the automatically extracted tests is that the process can be simple repeated. On the other hand, GENTHAT does not currently have any support for simple regenerating tests that are out of date.

Tracing time. The tracing comes with some overhead. On average, tracing is 21.5 (s=97.1 m=4.9) times slower than simply running the package code artifacts. For most packages, the running time reasonably short, 75% of them runs under 4m. But there are packages for which the overhead makes the running very long, with some exceeding the set timeout of 5h. Commonly, this happens in packages that contains some iterative algorithms that run on a sufficiently large matrices. Since most of these calls are from the execution code path very similar, it shall be possible to mitigate this problem by using stochastic tracing. Instead of tracing each an every function call, the tracing should be guarded by some probability with long-tail distribution.

5.2 Utility

There are multiple applications for the extracted tests. First, they help regression testing. Especially in the case when it is possible to extract code from package clients. Genthat can also help to bootstrap manual tests by first generating an initial test suite which then package authors updates by hand. This approach can also be useful for Core R developers and package repositories where packages need to be often tested (for example with every change in the R interpreter). Since the extracted tests runs faster than package artifacts it might be beneficial to use them. They might also help to narrow root cause of a failure since they are actual unit tests.

5.3 Generalization

Text extraction from runtime program traces is naturally going to work better in language/programming models that limit mutable global state and where arguments to functions tend to be self contained. We would expect that scientific programming languages like MATLAB or Julia, and functional languages like Haskell or OCaml should behave in similar ways to R. Imperative languages are likely to be less conducive to this approach. Whether our approach would yield reasonable results in C or C++ is an open question.

6 CONCLUSION

We have presented the design, implementation and evaluation of Genthat, a tool for the automated extraction of unit tests for the R language. Our evaluation of this tool on a large corpus of packages suggests that it can significantly improve coverage of the code being tested.

In a way our results are surprising, as the limitations of our tool are quite severe. Genthat does not keep any global state or external state, so any test that changes values in a lexically scoped environment is bound to fail. Any code that manipulates pointers to C data structures is likely to fail. And lastly, any code that is not deterministic will surely fail. Our intuition is that Genthat works well because of the functional nature of R, mutation is rarely used, function access values passed in, and produce new ones.

For future work, we aim to explore techniques geared towards finding new bugs. This can be achieved by fuzzing the inputs to a function. We also will look for more changes that allow to reduce the inaccuracies of Genthat as well as to speed up the process of handling reverse dependencies.

ACKNOWLEDGMENT

The authors thank Petr Maj, Roman Tsegelskyi, Filippo Ghibellini, Michal Vácha and Lei Zhao for their work on previous versions of Genthat. We also thank Tomáš Kalibera from the R Core Team for his feedback. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 695412) and the NSF Darwin Award. The experiment presented in this paper was conducted using GNU parallel [18].

REFERENCES

[1] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. 2002. Korat: Automated Testing Based on Java Predicates. In *International Symposium on Software Testing and Analysis (ISSTA)*. https://doi.org/10.1145/566171.566191
[2] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *International Conference on Functional Programming (ICFP)*. https://doi.org/10.1145/351240.351266
[3] Christoph Csallner and Yannis Smaragdakis. 2004. JCrasher: An Automatic Robustness Tester for Java. *Softw. Pract. Exper.* 34, 11 (2004). https://doi.org/10.1002/spe.602
[4] Sebastian G. Elbaum, Hui Nee Chin, Matthew B. Dwyer, and Jonathan Dokulil. 2006. Carving differential unit test cases from system test cases. In *International Symposium on Foundations of Software Engineering (FSE)*. https://doi.org/10.1145/1181775.1181806
[5] Michael Ernst, Sai Zhang, David Saff, and Yingyi Bu. 2011. Combined Static and Dynamic Automated Test Generation. In *International Symposium on Software Testing and Analysis (ISSTA)*. https://doi.org/10.1145/2001420.2001463
[6] Hojun Jaygarl, Sunghun Kim, Tao Xie, and Carl Chang. 2010. OCAT: object capture-based automated testing. In *nternational Symposium on Software Testing and Analysis (ISSTA)*. https://doi.org/10.1145/1831708.1831729
[7] Shrinivas Joshi and Alessandro Orso. 2007. SCARPE: A Technique and Tool for Selective Capture and Replay of Program Executions. In *International Conference on Software Maintenance (ICSM)*. https://doi.org/10.1109/ICSM.2007.4362636
[8] Uwe Ligges. [n. d.]. 20 Years of CRAN (Video on Channel9. In *Keynote at UseR!*
[9] Floréal Morandat, Brandon Hill, Leo Osvald, and Jan Vitek. 2012. Evaluating the Design of the R Language: Objects and Functions for Data Analysis. In *European Conference on Object-Oriented Programming (ECOOP)*. https://doi.org/10.1007/978-3-642-31057-7_6
[10] R Core Team. 2017. R Internals 3.4.3. https://doi.org/3-900051-14-3
[11] R Core Team. 2017. R Language 3.4.3. https://doi.org/10.1016/0164-1212(87)90019-7
[12] R Core Team. 2017. Writing R Extensions 3.4.3. https://doi.org/10.2135/cropsci1998.0011183X003800020005x
[13] Gregor Richards, Andreas Gal, Brendan Eich, and Jan Vitek. 2011. Automated Construction of JavaScript Benchmarks. In *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*. https://doi.org/10.1145/2048066.2048119
[14] Thomas Rooney. 2015. *Automated Test Generation through Runtime Execution Trace Analysis.* Master's thesis. Imparial College London.
[15] David Saff, Shay Artzi, Jeff H. Perkins, and Michael D. Ernst. 2005. Automatic test factoring for Java. In *International Conference on Automated Software Engineering (ASE)*. https://doi.org/10.1145/1101908.1101927
[16] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A Concolic Unit Testing Engine for C. In *European Software Engineering Conference and International Symposium on Foundations of Software Engineering (ESEC/FSE)*. https://doi.org/10.1145/1081706.1081750
[17] O. Tange. 2011. GNU Parallel–The Command-Line Power Tool. *;login: The USENIX Magazine* 42, 47 (2011).
[18] O. Tange. 2018. GNU Parallel 2018. https://doi.org/10.5281/zenodo.1146014
[19] Suresh Thummalapenta, Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. 2009. MSeqGen: Object-oriented Unit-test Generation via Mining Source Code. In *European Software Engineering Conference and Symposium on The Foundations of Software Engineering (ESEC/FSE)*. https://doi.org/10.1145/1595696.1595725
[20] Hadley Wickham. 2014. *Advanced R* (1 edition ed.). Chapman and Hall/CRC.
[21] Shin Yoo and Mark Harman. 2012. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability* 22, 2 (2012), 67–120. https://doi.org/10.1002/stv.430