The dataref versuchung*

Saving Time through Better Internal Repeatability

Christian Dietrich
Friedrich–Alexander University (FAU)
Erlangen–Nuremberg, Germany
dietrich@cs.fau.de

Daniel Lohmann
Friedrich–Alexander University (FAU)
Erlangen–Nuremberg, Germany
lohmann@cs.fau.de

ABSTRACT

Compared to more traditional disciplines, such as the natural sciences, computer science is said to have a somewhat sloppy relationship with the external repeatability of published results. However, from our experience the problem starts even earlier: In many cases, authors are not even able to replicate *their own* results a year later, or to explain how exactly that number on page three of the paper was computed. Because of constant time pressure and strict submission deadlines, the successful researcher has to favor timely results over experiment documentation and data traceability.

We consider internal repeatability to be one of the most important prerequisites for external replicability and the scientific process. We describe our approach to foster internal repeatability in our own research projects with the help of dedicated tools for the automation of traceable experimental setups and for data presentation in scientific papers. By employing these tools, measures for ensuring internal repeatability no longer waste valuable working time and pay off quickly: They *save* time by eliminating recurring, and therefore error-prone, manual work steps, and at the same time increase confidence in experimental results.

1. INTRODUCTION

Over the course of centuries, traditional disciplines, such as the natural sciences, have developed a strong sense for reproducibility. Compared to that, computer science, as a much younger discipline, is said to have a somewhat sloppier view on reproducibility – and thereby on the scientific process itself. There seems to be some truth in this: Basically, every computer scientist could share an anecdote of a failed attempt to replicate the results of a some top-notch paper. In a recent report, Collberg et al. [3] describe their (in many cases failed) attempts to systematically replicate artifacts from highly ranked research papers. Their methodology to assess repeatability and their quantitative results, however, have been criticized (and also partly rebutted) by the community [17] and Collberg and colleagues are working on a new version of their report. Nevertheless, their report triggered an important public discussion about the issue of repeatability in computer science.

In the area of reproducible research, researchers often argue about the correct terminology. Although the differences between repetition, replication, and reproduction of scientific findings is subtle, it exists and is import. In this article, we use the terminology defined by Feitelson [6], which appeared together with this article in the same issue of the *SIGOPS Operating System Review*. Briefly summarized: For repetition, the researcher reruns the experiment with the original artifacts; For replication, the researcher re-builds and uses the experiment apparatus in order to retain equivalent artifacts; For reproduction, the researcher implements the gist of the original experiment in order conceive results that confirm or refute the original findings in a different setting. In this article, we mainly focus on repeatability.

Commonly, if we talk about repeatability, what we typically have in mind is *external* (or *universal*) repeatability: Eventually, *other* scientists should be able to run the experiments and repeat the *published* results. This view, naturally, supports the understanding of repeatability as task to be done at the end of a research project – as some kind of keystone. This task is scheduled shortly before the paper has been submitted (in the best case), accepted, or "when there is time". Unfortunately, the latter appears to be the standard: The results are half-hearted, sloppy, or just incomplete experiment descriptions that are neither validated nor reviewed by the community.

However, the problem starts earlier: Besides external repeatability, there is also *internal* (or *personal*) repeatability: At any time, the researcher and her colleagues should be able to repeat *their own results* consistently, without too much overhead, and in a comprehensible manner. Compared to the natural sciences, who teach their students rigorous standard procedures for experimental work (such as maintaining a laboratory protocol that details each and every step and input parameter), practical computer science as a discipline has not yet developed a notion of internal repeatability: Our experiments (run and measure a piece of software) are, in comparison, very cheap to carry out, but also employ a very large number of (easy to change) external input variables, such as the employed compiler, OS configuration, and so on. Our major venues are prestigious conferences with strict deadlines. All this fosters, especially if we come close to a paper deadline, a quick-and-dirty trial-and-error experimentation approach that favors quick results over documentation and traceability.

Clearly, internal repeatability is an important prerequisite for external replicability (and a quality-driven scientific process). Nevertheless – deadlines are deadlines and time *is* short. We need automation and tool support to reduce the *short-term* costs of internal repeatability.

1.1 Our Contributions

We describe our experience with internal repeatability and how it culminated in the development of two tools:

versuchung is a Python-based scripting framework to code executable and *traceable* experiments for systems software. Although it is easy to use, versuchung provides powerful mechanisms for orchestrating and journalizing complex experimental setups and their dependencies, from bootstrapping the tools involved over executing the experiment to aggregating

*This work was supported by the German Research Council (DFG) under grants no. LO 1719/1-2 and LO 1719/3-1

the resulting data and metadata.

dataref is a LaTeX package for the symbolic and intensional description of data points (i.e., numbers) in LaTeX documents. During document compilation, symbolic data points and their derivatives (such as differences, ratios, or percentages) are substituted by their actual or calculated value. Data points can be annotated with further metadata, and their usage is tracked throughout the document. Each data point can even be validated through assertions to uncover discrepancies from the expected results.

Basically, `versuchung` provides automation of internal repeatability for all experimental results, while `dataref` extends the scope of this automation into the documents that *describe* results, such as the paper, technical report, or thesis currently being authored. In our own research we came to the conclusion that – thereby – the goal of internal repeatability no longer waste valuable work time, but on the contrary quickly pays off: It *saves* time and headache and increases ones own confidence in experimental results.

1.2 Structure

The rest of the article is structured as follows: We start with our problem analysis in Section 2, which is followed by the description of the `versuchung` and `dataref` software packages in Section 3 and Section 4. In Section 5 and Section 6, we present two case-studies using these packages in data-intensive research projects. We discuss our experience with this approach in Section 7, present some related work in Section 8, and finally conclude in Section 9. After the list of references, we additionally provide a *datagraphy*, which contains all data points used within the article.

2. PROBLEM DESCRIPTION

Scientists often find themselves in an ongoing conflict between a constant lack of time and their own aspiration to do good and valid research. To illustrate this problem, we collected three scenarios that have occurred repeatedly in our scientific process:

SCENARIO 1 (FOUR HOURS TO THE PAPER DEADLINE).
Robin is a young PhD student and her supervisor has asked her to write about her latest research for a systems conference. Shortly before the deadline, she discovers a bug in the implementation – she has to rerun all experiments. Although each tool runs without much interaction, Robin has to orchestrate them manually. Afterwards, she has to update all numbers in the LaTeX code of her paper and resubmit the document in time. But how was that percentage on page 3 computed again?

SCENARIO 2 (THE CONTRIBUTION IS ACCEPTED).
*After the notification, a very happy Robin reads the reviews for her first accepted contribution. All reviewers were impressed by the approach and only one reviewer complains about the outdated compiler version Robin has used. Therefore, Robin reruns all analyses **again** with a more recent compiler. **Again**, she replaces all numbers, updates all tables, and recalculates all percentages.*

SCENARIO 3 (WRITING THE PHD THESIS).
Years later, Robin is working hard on her PhD thesis. In chapter 5, she wants to explain the approach from her first published paper. Unfortunately, Robin does not remember which version of her software and which parameters were used. While grepping through her home directory for magic comments and README files, she discovers two datasets, but only one seems to match the numbers in the paper. Robin bisects different version of her tool to replicate the numbers, until she spots a source-code repository with changes that were never put under version control.

All described scenarios are examples of a scientific process that lacks internal repeatability. Of course, the researcher can publish the raw datasets and some description, but this will not automatically lead to results that are easy to replicate. The external repeatability suffers significantly from the poor internal repeatability. In order to improve both, internal and external repeatability, we will identify the problems in the described scientific process.

Her constant lack of time tempted Robin to track the experiment work flow manually. Work-flow automation takes time and it is not obvious the cost will amortize later on. Several problems arise from this manual work-flow tracking. The correct invocation and parametrization of the tools is often kept in short-term memory; months or years later this knowledge has vanished. Additionally, the stack of software used is often an arcane mixture of self-developed programs and common-off-the-shelf software. When invoked manually, their orchestration needs expert knowledge, takes time, and is prone to error. We state that this is especially true for systems software, because here repeatability cannot be achieved as easily as in a closed system.

In the scenarios described, not only the work flow but also the datasets were tracked manually. When replicating the scientific process, questions such as "Which version of the tool was used to produce this dataset?" or "What command-line arguments were used?" arise. Automatic tracking of datasets and input parameters answers these questions and assists archiving and retrieving the raw data.

The next problem for repeatability arises from unclean execution environments. The researcher starts the analysis in her own working copy of the code and in its own working environment. Environment variables might influence the analysis execution and changes to the code base might not be put under version control, or even the executable in the changed code base is not rebuilt. The problem of unclean build environments has already been recognized by the *continuous-integration* community [8]. In systems, however, we typically have to face an additional burden: Most of our software is highly configurable at compile time. Linux, for instance, provides thousands of compile-time configuration options – the configuration actually employed may have a larger impact on the results than the Linux version number, which in many cases is the only information provided. In their "Ten years later" paper [20], Palix and colleagues describe the enormous difficulties to figure out the Linux v2.4.1 configuration used by Chou in [2] in order to reproduce their results. Eventually, they had to apply source-code statistics and other heuristic measures to figure out some configuration "that is closest to that of Chou *et al.*" [20]. The takeaway here is: Our experiments have many *many* input variables – often more than we are aware of.

Besides the problem with the dataset generation, repeatability includes also the tracking of *data points* within the published document. Often data points are inserted as explicit numbers, and not symbolically, into the document. For a high information density and to assist the fast comprehension of the results, authors have to aggregate data, draw graphs, and calculate percentages. These document-preparation tasks are also part of the scientific process and are time consuming and error prone, when done manually. This is especially true, if the underlying data is updated during the document writing process. Additional tension comes into the process when data acquisition and writing is done by different researchers. Communication overhead and misunderstandings in the interpretation of numbers are a possible result.

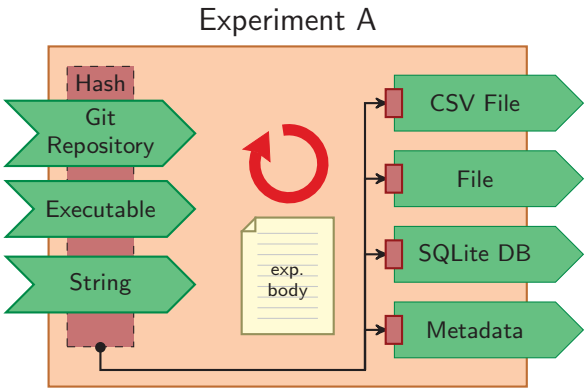

Figure 1: Schema of a `versuchung` experiment. Three input parameters are defined here: a Git repository, an executable, a plain string. For each invocation, these parameters are used to produce a hash value. The formal output parameters, which are produced by the experiment body, are tagged with this hash. Additional metadata provides further information about the experiment invocation.

3. THE VERSUCHUNG FRAMEWORK

In the natural sciences, experiments often involve interaction with the physical world that has to be done manually. Therefore, these disciplines developed the notion of laboratory protocols as part of good laboratory practice. In these documents, standardized procedural methods are described with all input parameters, the experiment setup, and the desired measurement points. During the experiment application, standardized notes and observations are written down. These note artifacts can be shared with other researchers from the same or another laboratory to replicate the experiment.

In computer science, we are often in a much more comfortable situation: Our experiments neither involve the physical world nor manual interaction. Therefore, we did not evolve such a sophisticated notion of experiment protocols. Nevertheless, we already have strict formalisms at hand that suit the purpose of expressing experiment protocols well: programming languages. Computer scientists have the chance to make their experiment protocols executable.

`versuchung`[1] is a framework written in the Python scripting language to encode experiment protocols, which can be instrumented and executed afterwards. `versuchung` provides an application programming interface that allows the researcher to codify the experiment procedure. Figure 1 depicts the schema of a `versuchung` experiment. Every experiment consists of three parts: Input parameters, output parameters, and an experiment body. The input parameters, which may include Git repositories, executables, ZIP archives, and many more, are made available to the experiment body. Additionally, a hash value, which is calculated over the input parameters, is used to identify this concrete experiment invocation. For the placement of the output artifacts, a directory is named with the experiment name and the calculated hash value. An additional metadata file contains information about the input parameters and is placed within the artifact directory.

Figure 2 shows the experiment description as a `versuchung` experiment. Every experiment is a Python class that inherits from

[1] `versuchung` is a pun in the German language. "Versuch" translates to "experiment" and "versuchung" translates to "temptation". Normally, the suffix "-ung" makes a noun from a verb (entdecken → Entdeckung; discover → discovery).

```
1  class AnalyzeLinux(Experiment):
2      inputs = {
3        "linux": GitArchive("/srv/git/linux.git"),
4        "tool" : Executable("~/bin/analyzer"),
5        "arch" : String("x86")
6      }
7      outputs = {
8        "results": CSV_File("results.csv"),
9        "logfile": File("terminal.log"),
10     }
11     def run(self):
12         linux_path = with self.linux.path
13         stdout = shell("cd %s; %s -a %s 2> %s",
14                        linux_path,
15                        self.analyzer.path,
16                        self.arch.value,
17                        self.logfile.path)
18         for line in stdout:
19             fields = line.split(" ")
20             name = fields[0]
21             count = len(fields[1:])
22             self.results.append([name, count])
23
24 if __name__ == "__main__":
25     # Initialize and Execute
26     experiment = AnalyzeLinux()
27     experiment(sys.argv)
```

Figure 2: A `versuchung` Experiment. The experiment defines three input parameters and assigns them a name. When accessing these parameters in the experiment body, the underlying data is made available. For example, the Git archive is checked out into a temporary directory. All underlined words are defined by `versuchung`.

```
Usage: experiment.py <options>
[...]
  --tool=TOOL       (default: ~/bin/analyzer)
  --arch=ARCH       (default: x86)
  --linux-clone-url=LINUX-CLONE-URL
                    (default: /srv/git/linux.git)
  --linux-ref=LINUX-REF
                    (default: refs/heads/master)
```

Figure 3: The command-line interface. For the input parameters from Figure 2, a command-line interface is automatically generated by `versuchung`.

a `versuchung` base class (line 1). At the class level, the input and output parameters are declared and filled with default values (line 2ff.). These parameters are common among all experiment invocations. The `run()` method (line 11) contains the experiment body and is executed after the input parameters are set up.

Upon experiment invocation (line 27), the framework constructs a command-line interface from the declared parameters (see Figure 3). Through this interface, the default values can be overridden to choose different input parameters. For example, `--linux-clone-url` determines the location of the Git source-code repository for the `linux` input parameter.

The identifier for one concrete experiment invocation is calculated from the input parameters and stored in the metadata file, which is located in the artifact directory. An example metadata file, which is a plain-text representation of a Python data structure, is shown in Figure 4. Besides the start and the end time of the experiment invocation and information about the experiment file itself, detailed information about the input parameters is recorded. For example,

```
{
 "date-end": "2014-09-03 15:18:00.819147",
 "date-start": "2014-09-03 15:18:00.815746",
 "experiment-hash": "cfe02370[...]",
 "experiment-name": "AnalyzeLinux",
 "experiment-version": 1,
 "arch": "x86",
 "linux-clone-url": "/srv/git/linux.git",
 "linux-hash": "ddb68e510612c1[...]",
 "linux-ref": "refs/heads/master",
 "tool-md5": "970be6a05c1ccbadbcece0c6db9b3882"
}
```

Figure 4: `AnalyzeLinux-cfe0237[...]/metadata` – Each experiment artifact is accompanied by a metadata file, which gives information about the concrete arguments used.

```
1 class VisualizeResults(Experiment):
2     inputs = {
3       "analysis": AnalyzeLinux()
4     }
5     outputs = { ... }
6     def run(self):
7         for row in self.analysis.results.value:
8             filename, count = row
9             ...
```

Figure 5: A chained `versuchung` Experiment. Experiments can use other experiments as input parameters. The depicted experiment references the "AnalyzeLinux" experiment from Figure 2.

the hash of the commit used from the Git source repository identifies the software version exactly.

Before and within the experiment body, `versuchung` automates various tasks: First, `versuchung` creates a temporary directory as a working space for the experiment. When the experiment requests the path of the `linux` input (line 12), the source-code repository is checked out into the temporary directory. This ensures a clean working copy of the used source-code base. `versuchung` provides similar automated accessors for different classes of input parameters (e.g., ZIP archives, automatically downloaded URLs).

The `versuchung` API provides tools to write very dense experiment code. For example, the `shell()` function (line 13) wraps the invocation of programs and takes care of proper argument escaping and return-value checking; optionally, all `shell()` commands can be tracked in detail. Another tool provided by `versuchung` is the `MachineMonitor` that monitors the environment (e.g., network activity, processor utilization,...) during the experiment.

Another key aspect of `versuchung` are the rich output types, which abstract different output formats and provide easy-to-use interfaces to the experiment author. For example, the `CSV_File` output type (line 8, 22) abstracts the construction of a Comma-Separated–Value file. `versuchung` ensures that the collected data is correctly dumped after the successful termination of the experiment. Different rich output types include SQLite[2] databases, compressed files, and data points used by `dataref`.

In addition to the input parameter types already mentioned, the artifacts of other `versuchung` experiments can be used as input parameters. This mechanism allows the decomposition of complex experiment and analysis work-flows into smaller reusable parts. Figure 5 shows an example experiment that uses the experiment from

[2]An SQL database that uses a single file as data storage.

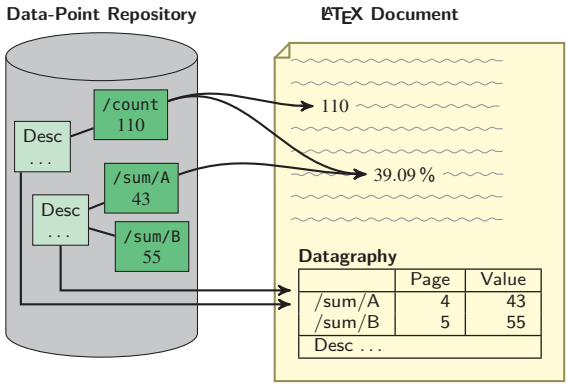

Figure 6: Schema of `dataref`. Symbolic names for data points from the repository can be referenced by several `dataref` commands within the output document.

Figure 2 as input parameter (Figure 5, line 3). The experiment body uses the name of the output parameter (Figure 2, line 8) as an input parameter (Figure 5, line 7). By using the experiment hash of the `AnalyzeLinux` experiment to calculate the hash of the `VisualizeResults` experiment, `versuchung` tracks all input parameters of all referenced experiments. The dependency is recorded in the metadata file of "VisualizeResults":

```
{
 "date-end": "2014-09-03 15:22:00.819147",
 "date-start": "2014-09-03 15:24:00.815746",
 "experiment-hash": "12d5fa[...]",
 "experiment-name": "VisualizeResults",
 "analysis": "cfe02370[...]"
}
```

The hash of the referenced experiment (cfe02370...) identifies the experiment artifact employed and is used to calculate the hash of the current experiment (12d5fa...). Result directory names contain the experiment's hash, and therefore the researcher can retrieve the correct experiment artifact with little effort.

`versuchung` tries not to capture all possible influences the environment has on the experiment execution, but rather leaves the explicit parameter declaration to the experimenter. Software packages like CDE [11] are a possible addition to make experiments fully self-contained.

The `versuchung` framework is freely available[3] and published as Free Software under the GPLv3.

4. THE DATAREF LaTeX PACKAGE

The `dataref` package for the LaTeX typesetting system is a companion to the `versuchung` framework. Both projects can be used individually, but they work nicely with each other, since `versuchung` has support to output `dataref` data-point definitions directly.

`dataref` supports the author of a LaTeX document to express data points intensionally; during the document compilation, `dataref` replaces the symbolic data-point names with the actual numbers. Figure 6 shows the general approach of `dataref`. The symbolic data points are declared in the data-point repository, which is a usual TeX file with `\drefset{<key>}{<value>}` commands (see Figure 7).

[3]`https://github.com/stettberger/versuchung`

```
\drefset{/linux features/hardware/v2.6.12}{4122}
\drefset{/linux features/software/v2.6.12}{583}
\drefset{/linux features/hardware/v3.15}{12076}
\drefset{/linux features/software/v3.15}{1880}

\drefsethelp{/linux features/hardware/.*}{%
  The number of Linux features that are related
  to hardware abstraction. This includes drivers/,
  arch/, and  sound/. Numbers are raw data from
  \textcite{ruprecht:14:gpce}.}

\drefsethelp{/linux features/software/.*}{%
  The number of Linux features that are included
  in software components that are not directly
  related to hardware. This includes kernel/ and
  net/.}
```

Figure 7: A `dataref` Data-Point Repository. Besides the data points, `dataref` gives the researcher the possibility to give descriptions for the results.

Within the document, several `dataref` commands are available to reference the values from the data-point repository. `\dref{}` references a single value and inserts it directly into the text. Numbers can be formatted in different formats (e.g., scientific notation, digit grouping, precision,...).

The author can use the `\drefrel` command to express the intention to calculate some relation, such as percentages, between two data points. `dataref` calculates the desired value directly within LaTeX during the compilation process. For example, the growth of features from Linux v2.6.12 to Linux v3.15 as a factor is expressed as

```
\drefrel[factor,percent,
         prefix=/linux features/software/,
         base=v2.6.12]{v3.15}
```

and results in "322.46" in the document. By using this intensional approach, mixing up numbers is less likely, everyone with access to the document's TeX source understands where the numbers come from, and the percentages are always up-to-date with the underlying data. If more complex calculations are desired, the `\drefcalc` command evaluates arbitrary arithmetic expressions, which in turn can reference `dataref` data points.

Results are not always inserted as bare numbers, but also qualitative statements, such as "more than half", are expressed. With `dataref`, the bare numbers are updated when the underlying data is replaced. This may lead to inconsistencies between qualitative and quantitative statements. `dataref` does not aim to avoid this problem, but with `\drefassert` it provides a command to detect such inconsistencies (see Figure 8). The command works like an assertion in other programming languages and ensures that an arithmetic expression evaluates to true. The author can add an assertion to the qualitative statement, and `dataref` ensures that the assertion holds when the document is compiled.

Often, results are presented as tables. The generation of complex tables filled with data is an annoying and error-prone task in LaTeX. `dataref` supports the (partial) generation of data-filled tables (see Figure 9). Most of the control over the table formatting is left in the user's hands, since only single table cells are generated. In Figure 9, more columns can be shown by editing the comma-separated list `\versions`.

With the `\drefsethelp` command, `dataref` supports the provi-

```
\drefassert{(  data("/linux features/hardware/v3.15")
             + data("/linux features/software/v3.15"))
            >13000}
The Linux kernel exposes more than 13000 features...
```

Figure 8: Usage of `\drefassert`. Assertions validate qualitative statements by checking quantitative data points.

```
% A comma-separated list of Linux versions
\def\versions{v2.6.12,v3.15}
% Set a dref prefix for shorter keys
\drefprefix{/linux features/}
\begin{tabular}{rll}
              & \drefrow*{\versions}{#1}   \\
   HW-Features & \drefrow{\versions}{hardware/#1} \\
   SW-Features & \drefrow{\versions}{software/#1} \\
\end{tabular}
```

	v2.6.12	v3.15
HW-Features	4,122	12,076
SW-Features	583	1,880

Figure 9: A table filled with `\drefrow`. The `\drefrow` command iterates over a comma-separated list and replaces `#1` in the template (second argument) with the value. Each list item results in one table cell.

sioning of annotated descriptions for groups of data points. These descriptions can be used to communicate the semantics of a data point between the experimenter, who might be using `versuchung`, and the author of the paper. Since the description is stored together with the data points, it cannot get lost over the years.

The data-point descriptions are also used when `dataref` is ordered to typeset a usage report. The usage report lists all data points that were used in the document with page number, value and description. Figure 10 shows an example usage report for two data points that are grouped by one data-point description. The `dataref` usage report is similar to a bibliography, which lists referenced literature, but for used data points; a *datagraphy*. At the end of this document, a datagraphy for this paper is included. Of course, the datagraphy is not necessarily intended to be included into the publication, but it could be published as a separate document.

For proof-reading purposes, `dataref` supports an annotation mode; referenced keys and derived values are commented with the origin of the value: 43[4], 0.78[5], 0.39[6]. These annotations are intended to help the author during the writing process.

`dataref` is a self-contained LaTeX package. No additional tools or compilation steps are required. Everything is calculated within the typesetting system. `dataref` is freely available from the *Comprehensive TeX Archive Network* (CTAN)[7] and is published as open-source software under the LPPL1.3.

[4] `\dref{/sum/A}`
[5] `\drefcalc{data("/sum/A")/data("/sum/B")}`
[6] `\drefrel[base=/count,factor]{/sum/A}`
[7] `http://www.ctan.org/pkg/dataref`

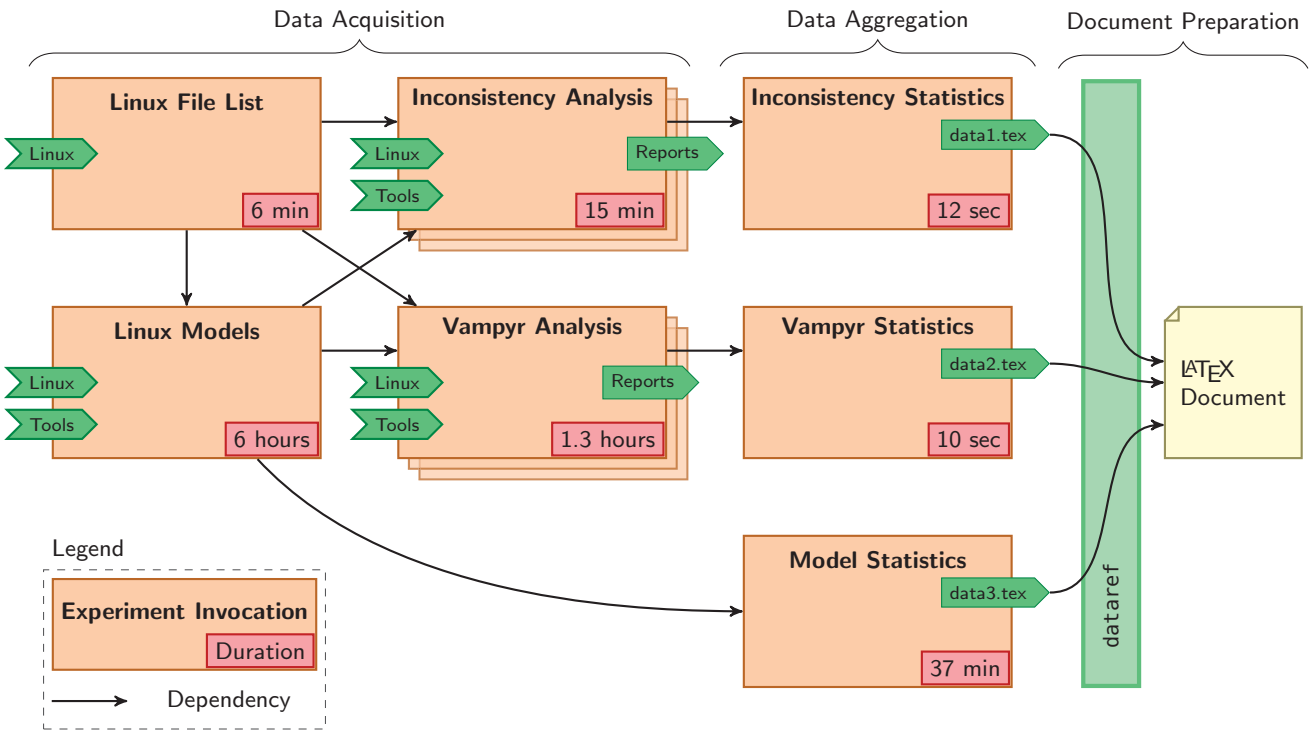

Figure 11: The VAMOS `versuchung` pipeline. The `versuchung` experiments are automatically executed by the Jenkins continous-integration tool. Jenkins distributes the experiment invocations onto several machines and keeps track of the dependencies.

	Pages	Value
/linux features/hardware/v2.6.12	5	4122
/linux features/hardware/v3.15	5	12076
The number of Linux features that are related to hardware abstraction. This includes `drivers/`, `arch/`, and `sound/`. Numbers are raw data from Ruprecht, Heinloth, and Lohmann [22]		

Figure 10: Example of a `dataref` usage report which prints the page numbers of the used data points, their values, and their description.

5. CASE STUDY 1: VAMOS

The presented software packages, `versuchung` and `dataref`, support the management of experiment work flows, datasets, and the connection to research articles. `versuchung` manages the data acquisition and aggregation steps, while `dataref` assists the document-preparation phase. Although both packages can be used separately, they integrate nicely with each other. In order to demonstrate a possible work flow, using `versuchung` and `dataref`, we will present the VAMOS tool pipeline.

As a first case study, we will examine the use of the presented tools in the context of the VAMOS project [29]. VAMOS examined the static variability, in terms of compile-time configurable features, in Linux. The configuration model of Linux exposes over 13,000 features [27, 28], which are interconnected by a complex web of dependencies and constraints. Among these features, the user chooses a subset at compile time to tailor a specific Linux variant. All configurations that fulfill the model constraints form the software family of Linux. Goals of VAMOS were: (1) Extraction of a formal variability model [25, 4]. (2) Identification of inconsistencies between the model and the implemented variability[8]. (3) Automatic generation of configurations towards different nonfunctional properties [28, 22].

Over the years, the team members developed a rich stack of tools to analyze the variability in Linux. The orchestration of these tools and their repeatable application became an increasingly complicated problem. During the research, we developed `versuchung` and `dataref` as a response to the identified hassles. With `versuchung`, we constructed an automated experiment pipeline consisting of several chained experiments.

The (simplified) VAMOS build pipeline is shown in Figure 11. The run time of the experiments varies significantly and scatters from a few seconds on a single machine to several hours on multiple machines. We manage the pipeline activation and the distribution onto several machines with the Jenkins[9] continous-integration service. Jenkins monitors the execution of repeated jobs and distributes different invocations onto a pool of machines. It is mainly focused on the automatic building of software products, but in our case it allows us to trigger the whole VAMOS pipeline with a single click.

`versuchung` assists us in tracking the dependencies between the experiments. For example, the "Vampyr Analysis" requires a dataset of the "Linux File List" and the "Linux Models" experiment. In reverse, we know that we have to retrigger the "Inconsistency Analysis" experiment when one of the dependencies was updated. The metadata, stored by `versuchung`, indicates outdated result datasets. With an automatic schedule, we trigger the pipeline once a week to analyze the current development version of Linux. This is a trade-off between resource consumption and density of the result sets.

The dependency tracking allows the reuse of datasets for follow-

[8]Static variability in Linux is expressed with Makefiles and the C preprocessor

[9]`http://jenkins-ci.org`

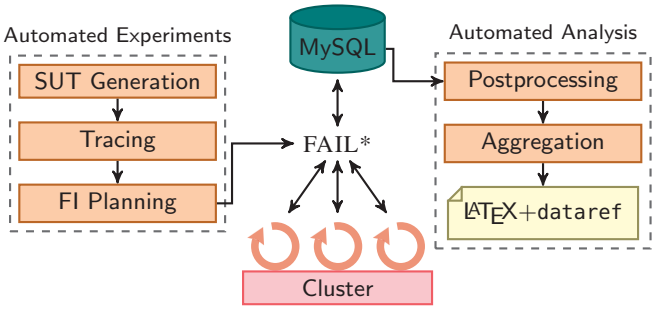

Figure 12: The DanceOS experiment work flow. The fault-injection campaigns contain a rupture in experiment automation, since the operation of the computing cluster requires manual intervention. After this rupture, the automated analysis is resumed.

up experiments. At all times, we know exactly which tool version or which Linux version was used. The reuse of datasets is of special interest for long-running experiments, such as "Linux Models", which extracts the variability models from the Linux source base.

Another key principle which has become obvious over the years is the separation between *analysis* and *aggregation* experiments. Analysis experiments are long-running experiments that generate many detailed reports; aggregation experiments aggregate the reports into comprehensible results. This separation allows the fast adaption of buggy aggregation experiments and exposes the raw tool reports as well as the aggregated results. In the VAMOS pipeline, we used `dataref` data-point repositories as a common aggregation format.

The generated data-point repositories have been used in two theses and four peer-reviewed publications [19, 4, 28, 5]. `dataref` connects the data points to our LaTeX documents. These documents include not only research papers, but also detailed graphical reports. The placement of `dataref` at the end of this pipeline allows a quick update of all numbers in the connected documents. For example, the draft version of Dietrich et al. [4] included data for Linux v3.1, while the final version used updated data for Linux v3.2. This update process entailed almost no manual intervention. For the final version, the data-point repositories were committed alongside the document to freeze the result.

6. CASE STUDY 2: DANCEOS

In the DanceOS research project, software-based measures are developed against transient hardware faults, such as bit flips. The project focuses mainly on dependability measures in system software. For example, the researchers investigated the influence of the general operating-system API design on the rate of silent data corruptions [15] and developed an aspect-oriented object protection for operating-system data structures [1]. Currently, we are working on a fault-tolerant operating system [13].

One of the most common evaluation scenarios is a fault-injection campaign. For this, the *system-under-test* (SUT) is traced without faults to generate a golden run. From the golden run, we determine which fault injections have to be done to cover the entire fault space. For each of the planned fault injections, we start the SUT, stop its execution at the time of the fault, and inject the fault pattern in the right location. Afterwards, we observe the SUT's behavior and record the result alongside with the planned experiments. For covering the entire fault space, we developed the FAIL* framework [23].

FAIL* provides tools for all steps of the fault-injection campaign. The experiment work flow is depicted in Figure 12. We used `versuchung` to automate the construction of the SUTs, the tracing of the golden run, and to generate the experiment plan. Nevertheless, our fault-injection campaigns take a lot of computing power and are carried out on our local computing cluster consisting of several hundred machines. As yet, the cluster operation is not automated with `versuchung` because the cluster's management software requires manual interaction. Although this step could be automated in the future, the need for manual operation opened a rupture in our `versuchung` work flow.

Nevertheless, after this rupture, we resumed the usage of `versuchung` and `dataref` to postprocess and aggregate the results. Using the file-system hierarchy as storage unit for experiment artifacts enabled us to easily invoke steps manually within the rupture. `versuchung` allows the step-wise migration to an automated work flow and does not force the experimenter to switch completely to `versuchung` in the first step. Of course, in a partially automated work flow, `versuchung` cannot track the parameters completely. Nevertheless, we can manually add a dependency between the last experiment before the rupture and the first one after. This will bridge the gap in terms of parameter tracking.

For the document-preparation phase, not all data points are the results of automated `versuchung` experiments; some data must be gathered manually – for example, referenced data from other publications and manual classifications. The data-point–repository interface between `versuchung` and `dataref` is well suited to inject manually gathered data points as a separate repository. Hence, manual and automatic experiment results can be treated equally in the document-preparation phase, while the origin of the data is still well documented.

Within the DanceOS project, `versuchung` has been used in four peer-reviewed publications [12, 15, 14, 16], with a fifth currently being under review.

7. DISCUSSION

As a team, `versuchung` and `dataref` assist the computer scientist in his everyday life. `versuchung` provides a rich toolbox for expressing traceable experimental work flows. The explicit definition of input and output parameters provides a unified interface for experiment invocation and dataset management. Additionally, `versuchung` tracks the software versions used and the given input parameters together with the resulting datasets.

The usage of `versuchung` gives the researcher more confidence in his own results, since the formalized experiment procedure will always be executed in the same sequence. Manual, error-prone tool invocation is not necessary. Even if there is a bug in the experiment description or in any other component, at least the misbehavior is consistent and the stored metadata helps to identify defective datasets.

The experiment artifacts generated by `versuchung` as datasets are self-contained. Within a single directory, metadata as well as experiment outcomes are collected. The artifacts are easy to move around and can be stored alongside research articles in a source-code control system, such as Git or Subversion.

An automatic pipeline reveals another benefit of automating experiments: Results can be generated continuously. For VAMOS, the analysis basis was Linux, which is a moving target. The weekly pipeline execution did not only test our software continuously, but also delivered continuously updated data. In the VAMOS project, we were able to monitor the development progress of Linux. It also made it easy to provide the reviewers of our papers with simple-to-use access to all experiments that provided the data of the paper under review. Even though we never got any explicit feedback on that, we are convinced that being able to observe these experiments

in action greatly increases the credibility of our results.

Using `dataref` closes the semantic gap between experiment results and the numbers in a LaTeX document. The symbolic reference of data points makes not only the update of results easy, but also prohibits confusion of data points. The intensional notation of datapoint relations, like percentages and factors, makes the connection between the data points used visible to all writers. Annotations and the datagraphy make the connections even visible to a reviewer.

Nevertheless, using `versuchung` does not come for free. Like any other software package, `versuchung` needs an initial training. The researcher, or the student who is forced by his supervisor to use `versuchung`, has to learn Python and has to get used to the `versuchung` API. Although the learning curve is not very steep, it is understandable to be afraid of the effort.

Another hurdle we often encounter in our everyday lives is the continuous consideration whether some task is complicated enough to be worth the effort of automation. Wrapping a single invocation of a single tool into `versuchung` results in another level of abstraction. If the benefits of `versuchung`, such as automatic dataset tracking, are not needed, it is tempting to invoke the toolchain manually this once. Nevertheless, often the invocation frequency does not stay at the level of "this once", but tends to be dozens of times.

`versuchung` does not prevent the developer from writing experiments that are not easily repeatable. It only provides a toolbox which *can* be used to write more robust and repeatable experiments. As another drawback, `versuchung` does not encapsulate the complete execution environment of the experiment. Here, CDE [11], which captures the whole execution environment, is a noteworthy addition and can be used together with `versuchung`.

`dataref` misuses TeX as a mathematical and numerical engine, therefore it inherits the limitations of TeX. TeX was not intended as a number-crunching engine, and the extensive use of in-document calculations will slow down the document compilation. Also, the use of arbitrarily large integers or floats is not supported. For more complex operations on data points, such as statistical analyses, the user should use either `versuchung` or other a proper programming language.

Literate-programming packages, such as Sweave [18], are used to make statistical research repeatable. Sweave mangles the statistical language R and LaTeX together into one document; it couples the data-aggregation and the document-preparation phases. Although we think this tight coupling is problematic for long-running aggregation steps, the idea of literate programming can be an addition to `versuchung`: Making the experiment's source code literate opens up the possibility to create a detailed experiment description as a well-formatted document, while still being able to use all `versuchung` features for the actual work flow.

In this work, we mainly focused on the repeatability of experiments, as defined by Feitelson [6]. Automation of experiments and document preparation allow an easy rerun of the work flow in the same environment; we achieve *internal repeatability*. Nevertheless, the codified experiment description and the parameter tracking make it easier to run the actual `versuchung` experiment in a different environment; there, other scientists can *replicate* the experimental artifacts. The explicit definition of input parameters and the generated command-line interface allow the *variation* of the experiment execution. Actual *reproducibility* is indirectly fostered by `versuchung`, since it acts as an executable laboratory protocol that catches the coarse-grained essence of the experimental apparatus.

8. RELATED WORK

Similar to the dataset-hash calculated by `versuchung`, Gavish and Donoho [10] identify *verifiable computational results* (VCRs) with a unique identifier, the *verifiable resource identifier* (VRI). Each input parameter is retrieved from a VCR repository by its VRI and the experiment outcome is uploaded again to a repository, retrieving a new VRI. Together with the VCR, the process of computation is recorded in terms of called function, arguments, and code. VCRs can be included directly into LaTeX by referencing their VRI. Unlike `versuchung`, this approach depends heavily on a complex infrastructure and does not assist the researcher in the process of experiment creation. In contrast to `dataref`, the inclusion of VCRs is on the granularity of tables and graphics, and not on single numbers and relations within in the text body.

Schwab and Schroeder [24] use GNU make to model the experiment work flow. Each step is implemented as a make target that contains the invocation sequence, and dependencies are expressed as make prerequisites. Their approach does not assist the experimenter in doing complex operations within the experiment description. Also, no tracking of input or output parameters is done.

With VisTrails, Freire et al. [9] capture the whole experiment work flow. It supports multiple scripting languages to write experiment steps, records a history of results, and can include visualizations in LaTeX. VisTrails is an all-in-one solution with a strong focus on visualization. Furthermore, it relies heavily on its graphical interface for defining work flows. In the document preparation phase, VisTrails works on the granularity of whole figures, and not within the text body itself. Compared to VisTrails, `versuchung` and `dataref` are light-weight, focused on single problems, and do not force the whole work flow into one system that is only accessible with a single user interface. Nevertheless, it should be possible to use `versuchung` within VisTrails experiment steps, since Python is already supported as a scripting language.

Quite a number of LaTeX packages are available to help the authors of a data-intensive document to automate the transformation of raw data into graphs or formatted LaTeX tables. Examples include `pgfplots` [7] for automatic graph creation, or `pgfplotstable` [7] and `datatool` [26] for tables. These packages also provide means to calculate and typeset derived values such as sums, averages or percentages. Others provide an interface to external tools for more complex calculations and analyses, such as the `reporttools` package [21] that interfaces to the powerful R system for statistical computing. However, while all these packages automate the process of updating larger tables and graphs, they do not reach into the main text of a document. In a computer-science paper, we typically do not only present "the data", but also discuss single data points in the main text, subsume them into qualitative statements, and even repeat the most interesting data points in the introduction and conclusions. For these cases, `dataref` provides automated consistency – and additionally assists the authors through the possibility to annotate data points or sets, a hierarchical naming scheme of data points, automated bookkeeping of all referenced data points, and assertions to ensure the validity of qualitative statements.

9. CONCLUSIONS

Practical computer science is missing a strong attitude towards the repeatability of results, a keystone of the scientific process. Even though our experiments should in theory be easy to replay – they are just computer programs – actual experiments tend to reside in some former student's home folder, have never been documented thoroughly, and require arcane knowledge about tool `X` regarding "that workaround applied shortly before the paper deadline". The reasons for this sloppiness are manifold, but time pressure is one of them: Explicit measures for experiment documentation and data traceability take valuable work time, while "publish or perish" and strict conference deadlines call for timely results.

We have presented tooling and automation support to reduce the short-term costs of internal repeatability: `versuchung` provides easy but powerful means to formalize complex executable and *traceable* experimental setups in practical computer science, while `dataref` brings automated updating, documentation, and traceability of data points and their derivatives into scientific documents written in LaTeX. Both are freely available and have been published under open-source licenses.[10]

In our own research projects, the efforts for automation and internal repeatability of results have quickly paid off exactly through this combination: We frequently discovered bugs that required to rerun the experiments while writing the respective paper. However, thanks to `versuchung` and `dataref` it was sufficient to fix the bug and restart the experiment to update all results – including the paper being written – automatically.

References

[1] Christoph Borchert, Horst Schirmeier, and Olaf Spinczyk. "Generative Software-based Memory Error Detection and Correction for Operating System Data Structures". In: *43rd Int. Conf. on Dep. Systems & Networks (DSN '13)*. (Budapest, Hungary). Washington, DC, USA: IEEE, June 2013. DOI: 10.1109/DSN.2013.6575308.

[2] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. "An empirical study of operating systems errors". In: *18th ACM Symp. on OS Principles (SOSP '01)*. (Banff, Alberta, Canada). Ed. by Keith Marzullo and M. Satyanarayanan. New York, NY, USA: ACM, 2001, pp. 73–88. ISBN: 1-58113-389-8. DOI: 10.1145/502034.502042.

[3] Christian Collberg, Todd Proebsting, Gina Mraila, Akash Shankaran, Zuoming Shi, and Alex M Warren. *Measuring Reproducibility in Computer Systems Research*. Tech. rep. Dept. of Computer Science, The University of Arizona, Mar. 2014. URL: http://reproducibility.cs.arizona.edu/tr.pdf.

[4] Christian Dietrich, Reinhard Tartler, Wolfgang Schröder-Preikschat, and Daniel Lohmann. "A Robust Approach for Variability Extraction from the Linux Build System". In: *16th Software Product Line Conf. (SPLC '12)*. (Salvador, Brazil, Sept. 2–7, 2012). Ed. by Eduardo Santana de Almeida, Christa Schwanninger, and David Benavides. New York, NY, USA: ACM, 2012, pp. 21–30. ISBN: 978-1-4503-1094-9. DOI: 10.1145/2362536.2362544.

[5] Christian Dietrich, Reinhard Tartler, Wolfgang Schröder-Preikschat, and Daniel Lohmann. "Understanding Linux Feature Distribution". In: *2nd AOSD W'shop on Modularity in Systems Software (AOSD-MISS '12)*. (Potsdam, Germany, Mar. 27, 2012). Ed. by Christoph Borchert, Michael Haupt, and Daniel Lohmann. New York, NY, USA: ACM, 2012. ISBN: 978-1-4503-1217-2. DOI: 10.1145/2162024.2162030.

[6] Dror G. Feitelson. "From Repeatability to Reproducibility and Corroboration". In: *SIGOPS Oper. Syst. Rev.* 50.1 (2015).

[7] Christian Feuersänger. *The Pgfplots package. Create normal/logarithmic plots in two and three dimensions*. Version 1.11. URL: http://www.ctan.org/pkg/pgfplots (visited on 09/12/2014).

[8] Martin Fowler. *Continuous Integration*. 2006. URL: http://martinfowler.com/articles/continuousIntegration.html.

[9] Juliana Freire, David Koop, Fernando Seabra Chirigati, and Cláudio T Silva. "Reproducibility using VisTrails". In: *Implementing Reproducible Research* (2014), p. 33.

[10] Matan Gavish and David Donoho. "A Universal Identifier for Computational Results". In: *Procedia Computer Science* 4.0 (2011). Proceedings of the International Conference on Computational Science, ICCS 2011, pp. 637 –647. ISSN: 1877-0509. DOI: http://dx.doi.org/10.1016/j.procs.2011.04.067. URL: http://www.sciencedirect.com/science/article/pii/S1877050911001256.

[11] Philip J. Guo. "CDE: A Tool for Creating Portable Experimental Software Packages". In: *Computing in Science and Engineering* 14.4 (2012), pp. 32–35. ISSN: 1521-9615. DOI: http://doi.ieeecomputersociety.org/10.1109/MCSE.2012.36.

[12] Martin Hoffmann, Christian Dietrich, and Daniel Lohmann. "Failure by Design: Influence of the RTOS Interface on Memory Fault Resilience". In: *2nd Int. W'shop on Softw.-Based Methods for Robust Emb. Sys. (SOBRES '13)*. LNCS. Koblenz, Germany: Gesellschaft für Informatik, 2013. URL: http://danceos.org/sobres/2013/papers/SOBRES-550-Hoffmann.pdf.

[13] Martin Hoffmann, Christian Dietrich, and Daniel Lohmann. "dOSEK: A Dependable RTOS for Automotive Applications". In: *19th Int. Symp. on Dependable Computing (PRDC '13)*. (Vancouver, British Columbia, Canada). Fast abstract. Washington, DC, USA: IEEE, Dec. 2013. DOI: 10.1109/PRDC.2013.22.

[14] Martin Hoffmann, Peter Ulbrich, Christian Dietrich, Horst Schirmeier, Daniel Lohmann, and Wolfgang Schröder-Preikschat. "A Practitioner's Guide to Software-based Soft-Error Mitigation Using AN-Codes". In: *15th IEEE Int. Symp. on High-Assurance Systems Engineering (HASE '14)*. (Miami, Florida, USA). IEEE, Jan. 2014, pp. 33–40. ISBN: 978-1-4799-3465-2. DOI: 10.1109/HASE.2014.14.

[15] Martin Hoffmann, Christoph Borchert, Christian Dietrich, Horst Schirmeier, Rüdiger Kapitza, Olaf Spinczyk, and Daniel Lohmann. "Effectiveness of Fault Detection Mechanisms in Static and Dynamic Operating System Designs". In: *17th IEEE Int. Symp. on OO Real-Time Distributed Computing (ISORC '14)*. (Reno, Nevada, USA). IEEE, 2014, pp. 230–237. DOI: 10.1109/ISORC.2014.26.

[16] Martin Hoffmann, Peter Ulbrich, Christian Dietrich, Horst Schirmeier, Daniel Lohmann, and Wolfgang Schröder-Preikschat. "Experiences with Software-based Soft-Error Mitigation using AN-Codes". In: *Software Quality Journal* (2015). (accepted).

[17] Shriram Krishnamurthi. *Examining "Reproducibility in Computer Science"*. [Accessed: 5. November 2014]. 2014. URL: http://cs.brown.edu/~sk/Memos/Examining-Reproducibility/.

[18] Friedrich Leisch. "Sweave: Dynamic Generation of Statistical Reports Using Literate Data Analysis". In: *Compstat 2002 — Proceedings in Computational Statistics*. Ed. by Wolfgang Härdle and Bernd Rönz. ISBN 3-7908-1517-9. Physica Verlag, Heidelberg, 2002, pp. 575–580. URL: http://www.stat.uni-muenchen.de/~leisch/Sweave.

[19] Sarah Nadi, Christian Dietrich, Reinhard Tartler, Ric Holt, and Daniel Lohmann. "Linux Variability Anomalies: What Causes Them and How Do They Get Fixed?" In: *10th Conf. on Mining Software Repositories (MSR '13)*. Washington, DC, USA: IEEE, May 2013, pp. 111–120. ISBN: 978-1-4799-0345-0. DOI: 10.1109/MSR.2013.6624017.

[20] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia L. Lawall, and Gilles Muller. "Faults in Linux: Ten years later". In: *16th Int. Conf. on Arch. Support for Programming Languages & Operating Systems (ASPLOS '11)*. New York, NY, USA: ACM, 2011, pp. 305–318. DOI: 10.1145/1950365.1950401.

[10] See https://github.com/stettberger/versuchung and http://www.ctan.org/pkg/dataref for details.

[21] Kaspar Rufibach. *reporttools: Generate LaTeX tables of descriptive statistics. Loads, rounds, formats and postprocesses numerical tables*. Version 1.1.1. URL: http://cran.r-project.org/web/packages/reporttools/ (visited on 09/12/2014).

[22] Andreas Ruprecht, Bernhard Heinloth, and Daniel Lohmann. "Automatic Feature Selection in Large-Scale System-Software Product Lines". In: *13th Int. Conf. on Generative Programming and Component Engineering (GPCE '14)*. (Västeås, Sweden). Ed. by Matthew Flatt. New York, NY, USA: ACM, Sept. 2014, pp. 39–48. ISBN: 978-1-4503-3161-6. DOI: 10.1145/2658761.2658767.

[23] Horst Schirmeier, Martin Hoffmann, Rüdiger Kapitza, Daniel Lohmann, and Olaf Spinczyk. "FAIL*: Towards a Versatile Fault-Injection Experiment Framework". In: *25th Int. Conf. on Architecture of Computing Systems (ARCS '12), Workshop Proceedings*. (Munich, Germany). Ed. by Gero Mühl, Jan Richling, and Andreas Herkersdorf. Vol. 200. Lecture Notes in Informatics. Gesellschaft für Informatik, Mar. 2012, pp. 201–210. ISBN: 978-3-88579-294-9.

[24] Matthias Schwab and Joel Schroeder. "Reproducible research documents using GNU-make". In: *SEP-89: Stanford Exploration Project* (1995), pp. 217–226.

[25] Julio Sincero, Reinhard Tartler, Daniel Lohmann, and Wolfgang Schröder-Preikschat. "Efficient Extraction and Analysis of Preprocessor-Based Variability". In: *9th Int. Conf. on Generative Programming and Component Engineering (GPCE '10)*. (Eindhoven, The Netherlands). Ed. by Eelco Visser and Jaakko Järvi. New York, NY, USA: ACM, 2010, pp. 33–42. ISBN: 978-1-4503-0154-1. DOI: 10.1145/1868294.1868300.

[26] Nicola Talbot. *The Datatool package. Tools to load and manipulate data*. Version 2.22. URL: http://www.ctan.org/pkg/datatool (visited on 09/12/2014).

[27] Reinhard Tartler, Julio Sincero, Christian Dietrich, Wolfgang Schröder-Preikschat, and Daniel Lohmann. "Revealing and Repairing Configuration Inconsistencies in Large-Scale System Software". In: *International Journal on Software Tools for Technology Transfer (STTT)* 14.5 (Feb. 2012), pp. 531–551. DOI: 10.1007/s10009-012-0225-2.

[28] Reinhard Tartler, Christian Dietrich, Julio Sincero, Wolfgang Schröder-Preikschat, and Daniel Lohmann. "Static Analysis of Variability in System Software: The 90,000 #ifdefs Issue". In: *2014 USENIX ATC*. (Philadelphia, PA, USA). Berkeley, CA, USA: USENIX, June 2014, pp. 421–432. ISBN: 978-1-931971-10-2. URL: https://www.usenix.org/system/files/conference/atc14/atc14-paper-tartler.pdf.

[29] *VAMOS - Variability Management in Operating Systems*. FAU Erlangen-Nuremberg, 2012. URL: http://www4.informatik.uni-erlangen.de/Research/VAMOS/.

Datagraphy

	Page	Value
/splc2012/linux version/original	7	v3.1
/splc2012/linux version/final	7	v3.2
The final version of Dietrich et al. [4] was delivered with a more recent version of Linux than it was originally written for.		

	Page	Value
/linux features/hardware/v2.6.12	5	4122
/linux features/hardware/v3.15	5	12076
The number of Linux features that are related to hardware abstraction. This includes `drivers/`, `arch/`, and `sound/`. Numbers are raw data from Ruprecht, Heinloth, and Lohmann [22].		

	Page	Value
/linux features/software/v3.15	5	1880
/linux features/software/v2.6.12	5	583
The number of Linux features that are included in software components that are not directly related to hardware. This includes `kernel/` and `net/`. Numbers are raw data from Ruprecht, Heinloth, and Lohmann [22].		

Keys without Description	Page	Value
/count	4, 5	110
/sum/A	4, 5	43
/sum/B	4, 5	55
For these keys, no description was given		