Improving Software Engineering Research Through Experimentation Workbenches

Klaus Schmid(✉), Sascha El-Sharkawy(✉), and Christian Kröher(✉)

Institute of Computer Science, University of Hildesheim, Hildesheim, Germany
{schmid,elscha,kroeher}@sse.uni-hildesheim.de
https://sse.uni-hildesheim.de/en/

Abstract. Experimentation with software prototypes plays a fundamental role in software engineering research. In contrast to many other scientific disciplines, however, explicit support for this key activity in software engineering is relatively small. While some approaches to improve this situation have been proposed by the software engineering community, experiments are still very difficult and sometimes impossible to replicate.

In this paper, we propose the concept of an *experimentation workbench* as a means of explicit support for experimentation in software engineering research. In particular, we discuss core requirements that an experimentation workbench should satisfy in order to qualify as such and to offer a real benefit for researchers. Beyond their core benefits for experimentation, we stipulate that experimentation workbenches will also have benefits in regard to reproducibility and repeatability of software engineering research. Further, we illustrate this concept with a scenario and a case study, and describe relevant challenges as well as our experience with experimentation workbenches.

Keywords: Experimentation workbench · Empirical software engineering · Static analysis · Software product line analysis

1 Introduction

A significant part of software engineering is experimental in nature. This holds both for method-oriented research, which typically requires humans-in-the-loop, as well as more implementation-oriented research (related to program analysis, verification, software generation, etc.), which is the focus of this contribution.

The challenges to experimental research in software engineering are very similar to these in other experimental disciplines, like physics or psychology. Those include replicability of research results, efficient support for the experimental process, like conducting variations, or enabling others to reuse the scientific results. In some disciplines these issues have gained wide-spread attention, like in psychology due to the reproducibility crisis [2]. In large-scale physics, like the Large Hadron Collider (LHC), creating documentation solutions and supporting

© Springer Nature Switzerland AG 2019
M. H. ter Beek et al. (Eds.): Gnesi Festschrift, LNCS 11865, pp. 67–82, 2019.
https://doi.org/10.1007/978-3-030-30985-5_6

many variations of experiments is considered well before any experiments are actually built, i.e., creating the experiments are major systematic engineering activities in their own right. This inspired us to compare this situation with software engineering research, in particular experimental research based on software tools.

In software engineering, deficiencies in the systematic support of the research process are increasingly recognized as an issue. In our own experience (and that of others), even if the relevant software is provided, e.g., as open-source, it is very difficult and sometimes impossible to replicate the experiments as they may rely on (unavailable) third party tools or undocumented execution details. Thus, the replication of a single evaluation may require several days or weeks of work only for reverse engineering missing information or assets. This has also influenced organizations, like the Association for Computing Machinery (ACM), to address this need and provide guidelines to improve the situation, e.g., with assessing publications [1]. As part of these guidelines, ACM defines a terminology that distinguishes repeatability, replicability, and reproducibility. In this paper, we will follow this terminology and, hence, use these terms as follows:

- *Repeatability* means that researchers receive the same results with their own experimental setup on multiple trials.
- *Replicability* means that a different person receives the same results with the same experimental setup as reported by a researcher on multiple trials.
- *Reproducibility* means that a different person receives the same results as reported by a researcher with their own experimental setup on multiple trials.

A typical way to improve repeatability, replicability, and reproducibility is the publication of all artifacts relevant to an experiment. For instance, conferences increasingly provide the possibility to back up publications with artifacts and assess their quality [1]. Other measures include the use of docker or virtual machines to improve replicability [3]. However, these approaches are typically applied after the fact, i.e., after the experiments are finished, as opposed to practices in established experimental disciplines. This post-mortem approach may lead to threats to validity as it leads to the risk of missing important details in the documentation artifacts. These solutions do also not address other issues in the scientific process, like exploration of experimental variation.

Here, driven from our own experiences in conducting technical research experiments, we propose the concept of an *experimentation workbench* for software engineering to remedy this situation and make the scientific workflow and its requirements a central aspect in the tools we build. A key motivation for our proposal is the question:

> *"How would a support environment for software engineering research look like, if we would specifically engineer one?"*

Today, we are used to *development workbenches* like Eclipse [32], but while they are heavily used in research, they (only) aim at supporting the software

development process in general. They do not address any specific research-oriented requirements. Other uses of the term workbench include artifacts, like *language workbenches* [7]. Again, this term is more directed towards (language) development, not so much towards research. We choose the term *experimentation workbench* in analogy to these uses of the term. The term experimentation workbench is also not completely new. It has already been used in networking [9], however, with slightly different semantics, namely to denote a specific form of simulation environment.

An experimentation workbench, as we envision it, is not only about replicability, but about supporting the scientific process at large (e.g., rapid variation, reuse in new research), as we will discuss in the following sections. Thus, among other things, it should also support general reproducibility. This would move software engineering more in line with other experimental sciences. The requirements we put forward for defining the concept of experimentation workbenches are our main contribution. We believe thinking in these terms from the beginning and supporting the scientific process with such environments can be a major contribution to our community. In summary, our contributions are:

- The definition of the concept of an experimentation workbench along with a description of its defining requirements.
- An illustrative scenario highlighting the benefits of experimentation workbenches.
- An example implementation (KernelHaven).
- A discussion of challenges for creating experimentation workbenches.
- A report of our experiences with realizing and using experimentation workbenches in our research on product line analysis.

Below, we will further refine the concept of experimentation workbenches in a scenario (Sect. 2), before we define the fundamental requirements in Sect. 3. We illustrate the defined concept based on KernelHaven in Sect. 4, discuss major challenges to realizing experimentation workbenches in Sect. 5, and provide our experiences in Sect. 6. Finally, we conclude in Sect. 7.

2 Usage Scenario

In this section, we describe a scenario to clarify our expectations on how experimentation workbenches support the experimentation workflow. We assume experimentation workbenches to be constructed for a specific research domain. For our scenario we use the domain of static product line analysis as a reference, which is a rather active field of research [34]. It aims at questions like detecting code that can never be part of a product, as there is no product configuration that would allow this, or detecting type inconsistencies that only arise for specific code configurations. All these analyses have a certain structure: the different inputs like a variability model, source code, etc. must be analyzed and transformed into appropriate formats for integration and analysis. We choose this domain to match it to the example experimentation workbench discussed in Sect. 4. Figure 1

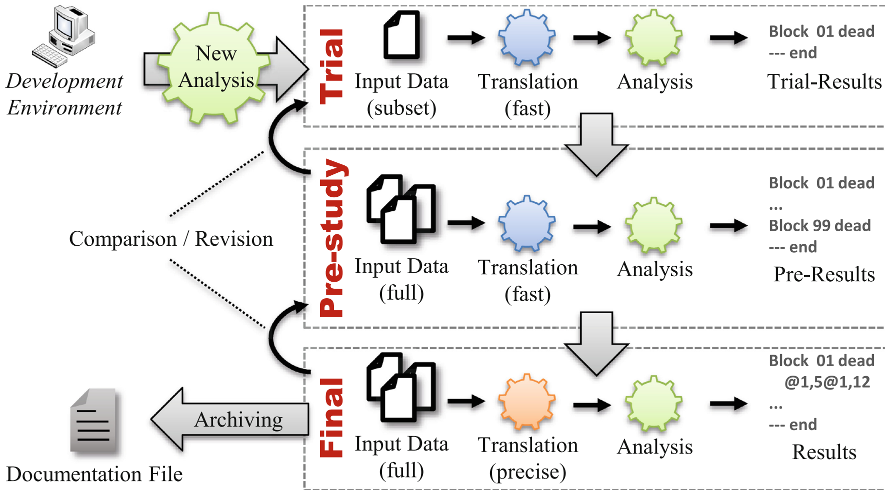

Fig. 1. Example workflow using an experimentation workbench.

shows an illustration of an example workflow in this domain as supported by an experimentation workbench. We discuss this workflow below by means of a scenario.

Preparation. Stefania wants to test her new analysis approach. She implemented it as a plugin for an experimentation workbench. This analysis works on an abstract representation of a product line and requires inputs from the variability model, the variability-enhanced build model, and C-code files. She uses Linux as a case study, which is often used in research, but huge. For translating the source code into an appropriate format for her analysis, two techniques are available: a fast, but not so precise one, and one precise, but rather slow.

Trial. First, Stefania wants to perform a trial with a small subset of the data. Thus, she defines the case study subset, the fast translation technique, and her analysis by configuration of the experimentation workbench. This is possible as data format standardization (along with necessary translations) and other services are offered by the experimentation workbench. The workbench also addresses parallelization and other technical issues regarding resource utilization allowing her to focus only on the realization of her analysis. In particular, no coding is required (except for implementing her analysis). This run gives the expected results after a few minutes.

Pre-study. Stefania changes the configuration to include all input data for her pre-study. She starts the analysis, which finishes already in a few hours. The results are again positive, but some files have not been correctly processed as she still used the fast but imprecise translation technique of her first trial.

Final. Stefania switches from the fast translation to the more precise one simply by configuration of the experimentation workbench. This technique uses a different approach and produces different outputs, but the experimentation workbench

handles format translations transparently. Hence, changing the complete analysis is again as easy as simply modifying a configuration option. This helps to avoid introducing accidental changes of the experiment that could occur if more complex programming would be involved. Stefania compares the final results with her pre-study. This is easy to do as she used the documentation feature, which results in automatic archiving of all input and output data, implementation artifacts, source code, and the entire configuration of the experimentation workbench. Apart from the impact of the more detailed analysis, the results match. Hence, Stefania shares the documentation file with her fellow researchers, who can directly rerun the analysis and compare the results or do further studies.

It is exactly this kind of fast, iterative changes along with the comprehensive documentation that the concept of an experimentation workbench is about.

3 Concepts and Requirements

As illustrated in the usage scenario above, experimentation workbenches should support researchers in easily performing experiments, explore the space of possibilities, document them, and share them with others, who then can build on them, refine them, apply their own techniques or create further derived experiments. These goals partially overlap with other approaches to improve the scientific process in software engineering.

For example, benchmarking as a scientific approach can support community building and can help to accelerate scientific advancement [29]. However, it does not address aspects like replication, supporting the experimentation process itself, etc. Concepts like Jupyter notebooks [26] support experimentation and to some limited degree replication and sharing, so they already come close. We could consider them as one specific instance of an experimentation workbench for data science, but this is usually not applicable to software engineering experimentation and it is still very generic, leaving the major burden of programming to the researchers. Other concepts like using docker images or virtual machines in software engineering address replication [3], but not other experimentation-oriented capabilities. Thus, while various approaches exist that address related topics, so far no one fully addresses the problems of the software engineering researcher as we do here with the concept of experimentation workbenches.

In our vision, experimentation workbenches provide key capabilities to support typical research activities in the scientific workflow. However, we do not expect that there will be a single experimentation workbench for all kinds of software engineering research just as there is no single experimentation facility in physics. Rather, we expect that the generic requirements, we present below, will be instantiated in domain-oriented ways. For clarity, we abstract here from any activities that are already well-supported, e.g., by development environments, and focus on those, for which there is typically no automated support available. In our view these are, in particular, the following ones:

R1 Support the setup (definition) of experiments.
R2 Support the analysis of experiments.
R3 Support the fast execution of variants of the experiment, including applying the experiment setup to different cases.
R4 Support the documentation of all relevant artefacts for replication.
R5 Support the reuse of experiments (by third parties).
R6 Support the extension and specialization of experiments by third parties.

Supporting the *setup of experiments* (*R1*) means, in particular, that technical issues that are not relevant to the study, but only required to ensure its execution, are handled by the workbench as far as possible. These could include providing initialization code, process coordination, and parallelization. Platform independence could be another aspect, which is not mandatory, but rather a design decision made by the developers and judged according to the requirements of the type of experimentation to be supported. Ideally, researchers only need to focus on the algorithmic aspects of their contributions. Thus, the front end to the researcher should provide a configuration interface or a Domain-Specific Language (DSL) or a combination of both to assist in these tasks.

After an experiment execution an *experiment analysis* (*R2*) must be done in order to determine what the results mean in relation to the initial research question. This could be provided by visualization tools, by providing certain kinds of tabularization, or simply by analysis scripts. The needs in this area are strongly domain-dependent as the analysis will depend on the types and amounts of data produced, requirements on statistics, and so forth. However, in many cases it will be possible to address these requirements using environments for data analysis like R [33]. Thus, if appropriate interfaces are available, there is no need to re-implement this for each workbench.

In experimentation it is often the case that one wants to *analyze variations in the data or in algorithms* to determine their impact on the overall outcome. This requires the possibility to set up new versions of an experiment with little effort and to easily go back to the previous analysis, if an experiment turns out to be not successful (*R3*). Sometimes such a variation can also be driven by performing a simplified version to improve turn-around time.

Finally, an experimentation workbench should support *documentation of experiments* such that automated replication is easily facilitated (*R4*). Such a replication package should at least include all inputs, outputs, code, and analysis results, if applicable. Thus, the package should directly support the inspection of any results, but also the direct replication of the experiments by any third-party.

Ideally, it should be possible to directly *reuse* not only the results, but even the experiments (*R5*). While this reusability enables repeatability by allowing researchers to always receive the same results with the same experimental setup, it also supports replicability and reproducibility by different persons. In particular, third parties should be able to easily re-conduct an experiment by reusing the experimental setup either directly, or with only slight adaptations, e.g., to fit their environment, which still conform to the initially documented experiment.

The direct reuse of an experiment (*R5*) may not always be sufficient to enable reproducibility. For example, if a third party aims at conducting a previous experiment of other researchers using a different case study. This may require variations like different algorithms to provide the necessary data from software artifacts as the new case study consists of different types of artifacts than the initial one (e.g., Java source code instead of C source code). This may require *extensions or specializations* of the initial experimentation, which ideally should be directly supported by the experimentation workbench (*R6*). Moreover, from the perspective of the overall scientific process that should be supported along the lines of the well-known adage of "standing on the shoulders of giants", this requirement is actually particularly important. Today, such an extension is extremely difficult, even if all the code is available as open source as existing experimental implementations are typically not created for reuse or even extension by third-parties. Thus, we want to emphasize this here due its importance to the scientific process.

4 An Experimentation Workbench for Static Product Line Analysis

In this section, we discuss KernelHaven[1] as an example of an open source experimentation workbench [20,21]. We do not argue that it is the ideal or perfect implementation of an experimentation workbench, but we use it here as a reference to describe some properties and technical implications of the concepts and requirements introduced in Sect. 3. KernelHaven instantiates these generic requirements for the domain of static analysis of software product lines. While we focus on this domain here, a specialized instance[2] of KernelHaven exists, which addresses metrics for software product lines [12] as a subset of static product line analysis (cf. requirement *R6* in Sect. 3).

In order to abstract from technical details and allow to rapidly set up new experiment variants (*R3*), it is necessary to take a domain-oriented perspective. The resulting workbench will only support experiments in this domain. In our case of product line analysis, Fig. 2 shows the resulting structure of that workbench. It consists of various *extractors*, which transform the *available assets* into a *common data model* that provides a good basis for *analysis*. In our domain, the relevant information is typically derived from three categories of assets: the variability model, the build system, and code assets. Hence, a *code pipeline*, a *build pipeline*, and a *variability model pipeline* further structure the workbench in Fig. 2, which perform this derivation for the respective category of assets individually.

While this workbench was initially developed for experiments on Linux, its architecture is much broader as all analysis and extractor components are implemented by a flexible plugin system. Thus, for example, the application to a proprietary variational build system only requires the development of an

[1] Available at GitHub: https://github.com/KernelHaven/KernelHaven.
[2] Available at GitHub: https://github.com/KernelHaven/MetricHaven.

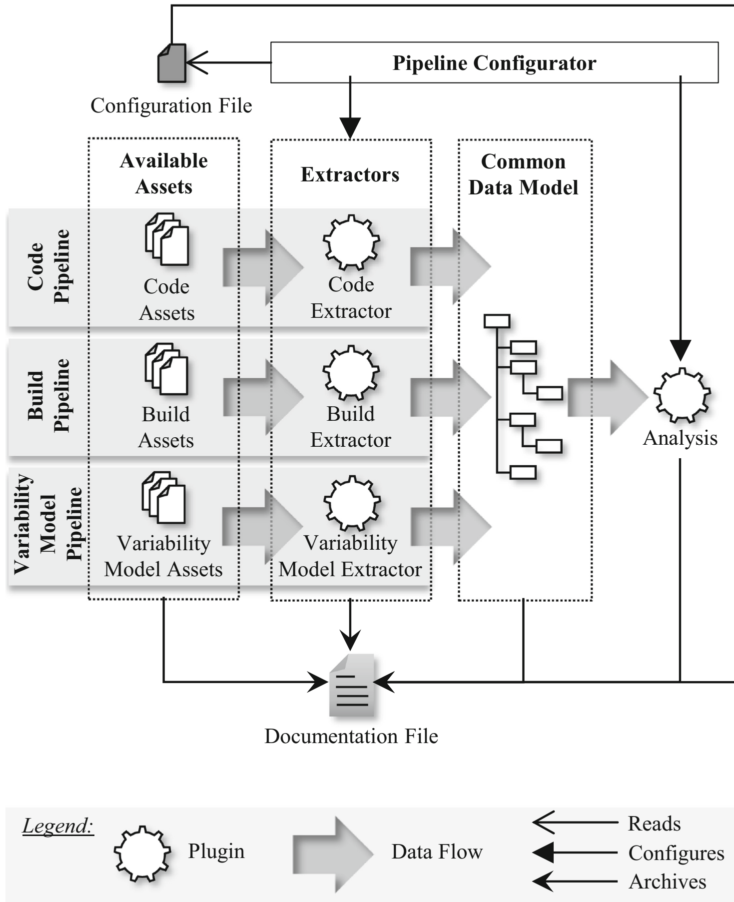

Fig. 2. KernelHaven Overview.

appropriate extraction plugin.[3] The common data model, which is used to represent the collected data of the various extractors, allows the reuse of existing analysis plugins without additional work.

Figure 2 shows that the *pipeline configurator* reads a *configuration file* to configure the whole infrastructure. It performs initialization of all subsystems (in particular the wiring, initialization and starting of the components), creates the corresponding processes, and allocates hardware resources. Also issues like parallelization of the various processes are handled by the infrastructure. Initially an adequate number of processes are created and throughout data dependencies are used to manage the parallel processing.

[3] In the case of minor variations, of course, also variations of existing plugins or even parameterized instances can be used. In order to support this a parametrization approach for plugins exists.

The configuration-oriented approach, which leads to an open ecosystem platform, directly addresses requirements *R1* to *R3* (cf. Sect. 3). The platform can also be configured to directly invoke documentation-related activities like archiving all relevant data, sources, implementations, configuration information, and so forth. This addresses requirement *R4* in Sect. 3. Requirements *R5* and *R6* are addressed by combining that (a) other researchers can rerun the experiments due to auto-documentation and (b) build on them by changing the configuration using either existing or self-developed plugins. The auto-documentation feature of KernelHaven therefore produces the experiment documentation in terms of an archive that contains all input, intermediate, and output data, as well as the main infrastructure, all plug-ins, and the configuration file. This feature directly supports reproducibility as a core task of research: the archive provides the original experimental setup to other researchers, enables them to rerun the same experiment on the same input data, and allows inspection of the previous results.

Initially, KernelHaven plugins were mostly derived from existing research prototypes. For example, they wrap a pre-existing tool and handle all the details of driving these tools (e.g., particular parametrization or environment needs). This has two major effects:

– The (re-)use of successful tools has been tremendously simplified: while for some tools, like TypeChef [18], people typically need several days to make it work reliably, the plugin embeds the relevant knowledge to make it reusable in minutes.
– The combination of tools is now possible simply by configuration: while combining existing tools as well as integrating with existing ones requires a lot of work and tool-knowledge, it is now a matter of defining the desired plugins as a parameter in a configuration file.

An important part of the domain design is the definition of the data structures and relevant data transformations to make extractors interchangeable. This can also be illustrated with KernelHaven. The toolset provides several extractor plugins, which can operate on C-Source code and can provide variability-tagged source-code fragments. One is derived from Undertaker [6], another one from TypeChef [18]. They differ, however, very significantly in terms of the level of detail they provide: Undertaker scans the source-file, identifies code blocks as sequences of lines and tags them with the relevant variability derived from any `#ifdef`-command. In the process it ignores header-files. On the other hand, TypeChef performs full variability-aware parsing, including header-files and macro expansions. As a consequence, it provides a complete AST adorned with variability information.

While the results of the two tools differ fundamentally, they share some information. Both extract the included variability information from source-files using preprocessor directives, i.e., the *presence conditions*. This commonality is sufficient for some types of analyses, like the identification of dead code [31]. In KernelHaven, all entities for representing extracted code information inherit from a class, which stores this common information. This allows to exchange

```
code.extractor.class = UndertakerExtractor
code.extractor.file_regex = .*\.c
build.extractor.class = KbuildMinerExtractor
variability.extractor.class = KconfigReaderExtractor
analysis.class = DeadCodeAnalysis
...
```

Listing 1.1. Excerpt of a KernelHaven configuration file.

code extractors as long as the desired analysis does not require the specific output of a certain extractor. The plugin system knows about these dependencies and takes care of them. An example is illustrated in Listing 1.1, which shows the relevant part to perform a dead code analysis on Linux with the Undertaker-extractor. Only the configuration file, in particular Line 1, must be modified in order to use TypeChef instead of Undertaker. However, this can be seen as a refinement of the Undertaker-information as this also corresponds to code-blocks. This is actually how the information is represented: a source-code processor may provide variability-adorned code-blocks, which may contain more detailed information (e.g., AST). The data structures are defined in a way that further steps may ignore levels of detail that are not required in their processing increasing composability of the various plugins.

While the analysis of results itself is not part of KernelHaven, the infrastructure supports *R2* by supporting the export of the resulting data in analysis-friendly formats like text-files (e.g., csv) or Excel. The core analysis is then typically done either with Excel or using R-scripts.

This allows to execute the scenario described in Sect. 2. One can first test new analysis concepts based on the rather fast, but not so detailed Undertaker-extractor, which extracts variability elements as line ranges. After the analysis has been positively evaluated, one can perform a more detailed analysis using the macro-aware parser of TypeChef, which provides a code block as an AST-fragment where all elements have the same presence condition. So, what both extractors have in common is to provide source code elements tagged with presence conditions, which is sufficient for dead code analysis. An AST-based analysis like type-analysis requires code extractors, which extract an AST containing variability information. Currently KernelHaven supports this with TypeChef or srcML [30]. It is important to note that (a) these different types of analysis are all supported by KernelHaven, and (b), for switching among them, it is sufficient to change some configuration options; no implementation change for any extractor plugin or the analysis plugin is required as long as they all adhere to the interface conventions.

Here, KernelHaven realizes two different perspectives on experimentation workbenches. On the one hand, KernelHaven is a platform that provides support for various experiments in the domain of static product lines analyses. On the other, we derived different KernelHaven instances based on this platform. These instances consist of the common experimentation workbench,

configuration parameters, and if necessary also experimentation-specific plugins that realize one specific analysis. In Sect. 6, we exemplary show some of the experiments, i.e., KernelHaven instances, which we realized based on the KernelHaven platform.

5 Challenges

While we believe the concept of experimentation workbenches is very fruitful for the research community and our own experiences with the KernelHaven implementation of it are so far very positive, there are still some challenges associated with the realization of an experimentation workbench.

Domain specifity. The first and most obvious challenge to experimentation workbenches is that they need to be constructed for a specific domain of experimentation. Thus, the requirements, we presented here, must be interpreted in the corresponding context and the capabilities of the workbench need to be scoped in terms of types of experiments (variations) to take into account. Thus, an experimentation workbench can be regarded as some form of product line [23] or open ecosystem [4]. Similar to product lines, of course, incremental development of it is possible.

Freedom of Implementation. It is fundamentally hard to guarantee full replicability, without significantly restricting the expressiveness used for realising specific parts of an experimental implementation. This is particularly the case for an experimentation workbench, like KernelHaven, that even allows pre-existing systems written in different languages and with arbitrary infrastructures as plugins. This issue is further compounded as in different domains different aspects may be important for replicability. For example, KernelHaven is purely functionality-related, i.e., as long as the same outputs are achieved for the same input, we can assume replicability. In other areas like performance engineering, the issue is different as similar timing behavior is required for replicability [8]. Hence, a corresponding experimentation workbench will have to address different issues. In this special case, special performance-rated environments have been proposed to promote replication [25].

Scope of Documentation. An important issue is the scope of an implementation that needs to be archived for replicability. In the example given in Sect. 4 only code artifacts related to the workbench implementation and the plugins are considered. The Java virtual machine and the operating system are not included. This yields rather lightweight packages where multiple archives can easily be stored locally. However, in other contexts the replication may require a copy of the virtual machine and the operating system. In such cases, an experimentation workbench may of course directly create a docker image or a virtual machine [3]. One can even imagine cases where a complex multi-machine setup needs to be archived like in large-scale adaptive systems.

Controlled Experiment Variation. A related issue are experiment variations. If some part of the analysis is replaced by something else, then this will result in changes to the experiment. Typically, this will also invoke undesirable changes. In

the example given, a switch from the simple code extractor to the more detailed one does not only lead to a more precise analysis of variability information in header files, but can also impact the details of the analyzed blocks as they are analyzed in a different way. Whether these changes are acceptable or not, will depend very much on the specifics of the analysis performed. In case plugins are used that have been engineered from the beginning with an experimentation workbench in mind, we expect this also to be less of an issue than it is currently the case with the reengineered plugins that KernelHaven uses.

These challenges basically come down to the need of achieving a sufficient domain understanding. Either prior to the construction of such an environment or as part of the experimental process. In this regard the development of an experimentation workbench can be compared to the development of a software product line.

6 Experiences

So far, we described the general requirements for experimentation workbenches and how the research community can take advantage of them. In this section, we share our experiences when working with KernelHaven (cf. Sect. 4). We used this experimentation workbench for our own research in the ITEA3 project REVaMP2[4], which focuses on round-trip engineering of software product lines. Since KernelHaven supports the definition of various experiments (*R1*) in the domain of static SPL analysis, we were able to use KernelHaven for many different research activities, like for example reverse engineering of variability information for bootstrapping of SPL development, evolution support, and verification tasks. We provide an overview of the variety of analyses supported by KernelHaven and show how we could realize these very diverse analyses with limited development resources in a short time. Further, we present lessons learned when working with KernelHaven.

Together with the Robert Bosch GmbH, we worked on reverse engineering of a dependency management system for a large-scale industrial product line [10]. For this, we decided to adapt the feature effect analysis described by Nadi et al. [24] to the needs of Bosch. This kind of analysis requires usually much effort to combine various parsers that extract variability information from different information sources. By means of KernelHaven, we were able to develop a first prototype very quickly, since the combination of data from different sources is a major concern of KernelHaven and first suitable parsers were already present. As a result, we could focus on the integration of parsers specific to the development environment of Bosch [10], lifting the propositional analysis of feature effects to integer-based variability [19], and on providing visualization support for reverse engineered dependencies [17]. In addition, KernelHaven's reproduction support (*R4*) simplified the execution of configured algorithms at the two partners. Thus, we also achieved a significant benefit for industrial transfer of our research results.

[4] http://www.revamp2-project.eu/.

KernelHaven also supports the verification of various properties of SPLs through its data extraction and analysis capabilities. For instance, we can reproduce and freely combine a large number of published product line metrics [11] resulting in more than 23,000 variations of metrics for single systems and SPLs, many of which are not handled by any other tool [12]. Another very important aspect for SPLs, is the analysis of (un-)dead code with respect to its variability model [31]. This is a very time consuming task as it analyzes whether the variability model allows the (de-)selection of all configurable code parts, e.g., `#ifdef`-blocks. Thus, this kind of analysis is more suitable for daily builds than for a continuous analysis during the development. However, a commit analysis of the Linux kernel has shown that changes to variability information occur infrequently and only affect small parts [22]. Based on this insight, we implemented an incremental verification approach to reduce the overall time consumption by about 90% [14], which is suitable to be applied in a continuous development environment. The incremental verification is realized by combining previous results of an already available dead code analysis with a new analysis that detects changed variability information (*R5* and *R6*).

Through the broad range of conducted experiments in combination with tested variations of algorithms, KernelHaven evolved quickly to a highly configurable system. For this, we realized a documentation system that provides the user, based on installed plugins, a list of available configuration options, supported values, and default settings. However, this system does not scale well as it neither supports a documentation of suggested settings arising through the combination of multiple plugins nor does it provide a dependency management among the plugins, e.g., the metric analysis plugin requires code extractors that extract a variability-aware AST rather than a simple block structure as needed by most other analyses (cf. Sect. 4). Thus, for the future, we plan to address this issue by (1). limiting the amount of configuration possibilities for stable plugins and by (2). integrating dependency management systems suitable for software ecosystems [5], e.g., based on our EASyProducer implementation [28][5].

This does also strongly suggest that experimentation workbenches can be regarded as a special form of product line or open software ecosystem [27].

7 Conclusion

In this paper, we introduced the concept of an experimentation workbench as a way of thinking about scientific experimentation artifacts with a focus on the needs of the scientific process. We believe that thinking about experimental research software in terms of this concept provides significant advantages when developing research systems in software engineering. In the future, we believe that some powerful experimentation workbenches for specific software engineering domains may provide a major contribution and foster the development of better ecosystems that drive software engineering research.

[5] https://sse.uni-hildesheim.de/en/research/projects/easy-producer/.

Our main contributions besides the concept itself are the characterizing requirements, which define an "ideal" experimentation workbench along with an illustrative scenario. We further described KernelHaven as an example experimentation workbench situated in the domain of product line analysis. KernelHaven may provide a basis for a research ecosystem for product line analysis as it integrates already today a number of existing research tools and makes them significantly more accessible than is otherwise the case. Besides achieving already significant research benefits, as discussed, we also found that this approach significantly improves our potential of working with industrial partners.

We assume that the concept of an experimentation workbench always needs to be interpreted relative to the specific scientific area. However, we hope the general requirements we presented may guide the creation of such systems and thus support the scientific progress by fostering the creation of ecosystems around experimentation workbenches in a number of software engineering fields. For example, one may interpret our concept presented in this paper in the context of Natural Language Processing (NLP) in requirements engineering [13,15,16]. In particular, the NLP tool for requirements analysis [16] may provide an excellent foundation for extending it to an experimentation workbench for that domain in future.

Acknowledgements. This work is partially supported by the ITEA3 project REVaMP[2], funded by the BMBF (German Ministry of Research and Education) under grant 01IS16042H. Any opinions expressed herein are solely by the authors and not by the BMBF.

References

1. Association for Computing Machinery: Artifact review and badging (2018). http://www.acm.org/publications/policies/artifact-review-badging. Accessed 03 May 2019
2. Baker, M.: Over half of psychology studies fail reproducibility test. News article in Nature - International Weekly Journal of Science (2015). https://www.nature.com/news/over-half-of-psychology-studies-fail-reproducibility-test-1.18248. Accessed 03 May 2019
3. Boettiger, C.: An introduction to docker for reproducible research. ACM SIGOPS Operating Syst. Rev. **49**(1), 71–79 (2015)
4. Bosch, J.: From software product lines to software ecosystems. In: 13th International Software Product Line Conference (SPLC 2009), pp. 111–119 (2009)
5. Brummermann, H., Keunecke, M., Schmid, K.: Formalizing distributed evolution of variability in information system ecosystems. In: 6th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS 2012), pp. 11–19 (2012)
6. CADOS / VAMOS Team: Undertaker (2015). https://vamos.informatik.uni-erlangen.de/trac/undertaker. Accessed 03 May 2019
7. Dyer, R., Nguyen, H.A., Rajan, H., Nguyen, T.N.: Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In: 35th International Conference on Software Engineering (ICSE 2013), pp. 422–431 (2013)

8. Eichelberger, H., Sass, A., Schmid, K.: From reproducibility problems to improvements: a journey. In: Symposium on Software Performance (SSP 2016), Softwaretechnik-Trends, vol. 36, no. 4, pp. 43–45 (2016)
9. Eide, E., Stoller, L., Lepreau, J.: An experimentation workbench for replayable networking research. In: 4th USENIX Conference on Networked Systems Design & Implementation (NSDI 2007), pp. 16–16 (2007)
10. El-Sharkawy, S., Dhar, S.J., Krafczyk, A., Duszynski, S., Beichter, T., Schmid, K.: Reverse engineering variability in an industrial product line: observations and lessons learned. In: 22nd International Systems and Software Product Line Conference (SPLC 2018), vol. 1, pp. 215–225 (2018)
11. El-Sharkawy, S., Krafczyk, A., Schmid, K.: MetricHaven – more than 23,000 metrics for measuring quality attributes of software product lines. In: 23rd International Systems and Software Product Line Conference, SPLC 2019, vol. B. ACM (2019). Accepted
12. El-Sharkawy, S., Yamagishi-Eichler, N., Schmid, K.: Metrics for analyzing variability and its implementation in software product lines: A systematic literature review. Inf. Softw. Technol. **106**, 1–30 (2019)
13. Ferrari, A., Dell'Orletta, F., Esuli, A., Gervasi, V., Gnesi, S.: Natural language requirements processing: a 4D vision. IEEE Softw. **34**(6), 28–35 (2017)
14. Flöter, M.: Prototypical realization and validation of an incremental software product line analysis approach. Master thesis, University of Hildesheim (2018)
15. Gnesi, S., Ferrari, A.: Research on NLP for RE at CNR-ISTI: a report. In: 1st Workshop on Natural Language Processing for Requirements Engineering, vol. 4, pp. 1–5 (2018)
16. Gnesi, S., Trentanni, G.: QuARS: A NLP tool for requirements analysis. In: 2nd Workshop on Natural Language Processing for Requirements Engineering and NLP Tool Showcase, 1, pp. 1–5 (2019). Tool Demonstrations
17. Grüner, S., et al.: Demonstration of tool chain for feature extraction, analysis and visualization on an industrial case study. In: 17th IEEE International Conference on Industrial Informatics (INDIN 2019) (2019). Accepted
18. Kästner, C.: TypeChef (2013). https://ckaestne.github.io/TypeChef/. Accessed 03 May 2019
19. Krafczyk, A., El-Sharkawy, S., Schmid, K.: Reverse engineering code dependencies: converting integer-based variability to propositional logic. In: 22nd International Systems and Software Product Line Conference (SPLC 2018), vol. 2, pp. 34–41 (2018)
20. Kröher, C., El-Sharkawy, S., Schmid, K.: Kernelhaven – an experimentation workbench for analyzing software product lines. In: 40th International Conference on Software Engineering: Companion Proceedings (ICSE 2018), pp. 73–76 (2018)
21. Kröher, C., El-Sharkawy, S., Schmid, K.: Kernelhaven - an open infrastructure for product line analysis. In: 22nd International Systems and Software Product Line Conference (SPLC 2018), vol. 2, pp. 5–10 (2018)
22. Kröher, C., Gerling, L., Schmid, K.: Identifying the intensity of variability changes in software product line evolution. In: 22nd International Systems and Software Product Line Conference (SPLC 2018), vol. 1, pp. 54–64 (2018)
23. van der Linden, F., Schmid, K., Rommes, E.: Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering, 1st edn, 333 pp. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71437-8
24. Nadi, S., Berger, T., Kästner, C., Czarnecki, K.: Where do configuration constraints stem from? An extraction approach and an empirical study. IEEE Trans. Softw. Eng. **41**(8), 820–841 (2015)

25. Oliveira, A., Petkovich, J.C., Reidemeister, T., Fischmeister, S.: Datamill: rigorous performance evaluation made easy. In: 4th ACM/SPEC International Conference on Performance Engineering (ICPE 2013), pp. 137–149 (2013)
26. Project Jupyter: The Jupyter Notebook (2019). http://jupyter.org. Accessed 03 May 2019
27. Schmid, K.: Variability modeling for distributed development – a comparison with established practice. In: 14th International Conference on Software Product Line Engineering (SPLC 2010), pp. 155–165 (2010)
28. Schmid, K., Eichelberger, H.: Easy-producer: from product lines to variability-rich software ecosystems. In: 19th International Conference on Software Product Line (SPLC 2015), pp. 390–391 (2015)
29. Sim, S.E., Easterbrook, S.M., Holt, R.C.: Using benchmarking to advance research: a challenge to software engineering. In: 25th International Conference on Software Engineering (ICSE 2003), pp. 74–83 (2003)
30. srcML Team: srcML (2017). http://www.srcml.org/. Accessed 03 May 2019
31. Tartler, R., Lohmann, D., Sincero, J., Schröder-Preikschat, W.: Feature consistency in compile-time-configurable system software: facing the Linux 10,000 feature problem. In: 6th Conference on Computer Systems (EuroSys 2011), pp. 47–60 (2011)
32. The Eclipse Foundation: Eclipse IDE (2019). https://www.eclipse.org/. Accessed 03 May 2019
33. The R Foundation: R Project (2019). https://www.r-project.org/. Accessed 03 May 2019
34. Thüm, T., Apel, S., Kästner, C., Schaefer, I., Saake, G.: A classification and survey of analysis strategies for software product lines. ACM Computing Surveys 47(1), p. 45 (2014). Article 6