ing.grid preprints

RESEARCH ARTICLE

Towards a Specification for Recombinable Benchmarks and Software Tools

Simon Dierl [1], Falk Howar [1, 2]

1. Department of Computer Science, TU Dortmund University, Dortmund, Germany.
2. Research Coordinator, Fraunhofer ISST, Dortmund, Germany.

Date Submitted:
2024-12-31

Keywords:
Artifacts, Benchmarks, Recombination, Schemas, Software Tools

Data availability:
Benchmark and tool artifacts for all case studies

Software availability:
ReReSo schemata and JVM tooling Maven artifact
Case study source code

Corresponding Author:
Simon Dierl
simon.dierl@tu-dortmund.de

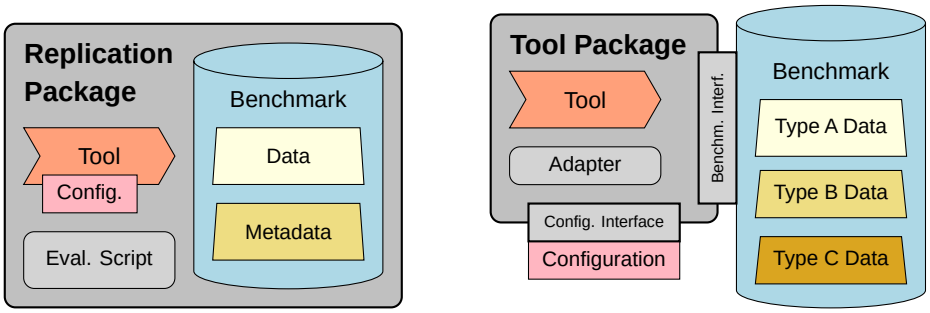
Abstract. We consider research that is based on implementing software tools from novel ideas and evaluating them by executing the tools against benchmarks. Subsequent research project would benefit from freely recombining existing and novel tools and benchmarks to acquire data. However, existing approaches for distributing tools and artifacts do not allow recombination. We propose an approach for packaging and describing tools and benchmark that allows and supports recombination. We specify a tool packaging format, an approach to structuring benchmarks, and a descriptor file format to link the two. We applied these specifications in two case studies in Software Engineering and Environmental Engineering. For each case study, we considered the benefit gained and the overhead induced by introducing our approach, e.g., for file format conversions. While minor limitations became apparent, we showed both the viability of our proposed specification and its low overhead in terms of development effort.

1 Introduction

In many disciplines, research involves the implementation of ideas and algorithms into (prototypal) *tools*. Claims about the correctness or performance of the implemented algorithms are evaluated by executing these tools on *benchmarks*, i.e. non-executable collections of data and accompanying metadata. For example, a machine learning-based object recognition algorithm would be evaluated using a collection of images labeled with the objects visible in them. The closer the implementation's outputs and the labels are, the better the tool performs.

Being able to re-run such experiments is essential for open science. In Software Engineering and other fields of study, publications are therefore often accompanied by peer-reviewed[1] *replication packages* that contain the tool(s) and benchmarks used in the study and that are long-term archived on, e.g., Zenodo. Such packages are usually containers (e.g., OCI containers or virtual machines). An example is shown in Figure 1a. With respect to the FAIR [3] and FAIR4RS [4] principles, accessibility is guaranteed by the long-term archival platform's facilities, while findability is dependent on the quality of the packages' metadata.

1. Successful replication is usually denoted on the publication using ACM [1] or EAPLS [2] badges.



(a) replication package **(b)** recombinable tool and benchmark

Figure 1: Conceptual overview of our idea. While replication artifacts (a) combine benchmarks and tools into a fixed bundle, our approach (b) allows recombination of benchmark and tool bundles via defined interfaces.

However, research building upon existing, already-implemented ideas greatly benefits from not only being able to repeat past studies, but *recombine* either "historic" tools with new or extended benchmarks or measure the performance of new tools on benchmarks used in previous work. The former yields a picture of "tool evolution", while the latter enables well-supported comparisons between competing tools.

Replication packages – due to being closed systems – do not meet this need. Introducing new benchmarks into a container requires conversion into a supported format, may necessitate modifying evaluation scripts, etc. Transferring a tool into a new execution environment may be an even more daunting task. As a result, such containers are not interoperable or resusable. In the case of open-source tools, building "from scratch" instead is not be a viable alternative: in a survey of publications backed by software [5], Collberg and Proebsting failed to acquire and build even half of the tools.

In this work, we propose a novel approach to packaging benchmarks and tools that enables recombination. Instead of monolithic replication packages, we propose the system of *recombinable* benchmarks and tools sketched in Figure 1b: tools and benchmarks are independently packaged and combined via standardized interfaces "just in time". For tools, this necessitates a form of containerization that allows users to pass benchmarks, configurations, etc. to the tool without having to modify the container's inner workings. Benchmarks instead undergo a paradigm shift with respect to their contents. To facilitate maximum reuse, they must aggregate the greatest amount of data possible. This necessitates standardization of the exchange formats used, i.e., how to containerize tools and how to describe the contents of tool and benchmark packages. Note that we do *not* intend to prescribe the actual *data* formats, but instead defer to research communities to agree on them. Such recombinable tools and benchmarks will, by definition, be interoperable and reusable, while descriptors increase their findability.

Contribution We have drafted an *abstract* specification that defines a) a standardized format for containerizing tools, b) a descriptor language for benchmarks that can describe the data contained therein, and c) a descriptor language for tools that defines the input and output formats. We implemented a support library for working with such format in JVM programming languages. We instantiated our abstract specification in case studies in Software Engineering and Environmental

Engineering, estimated the benefit gained, and measured the overhead required for using our approach. Results show the viability of our approach, which induces only a low amount of overhead.

Related Work Recombinable benchmarks should be *technically interoperable* according to the EOSC framework [6]. Our descriptor files for tools share approaches with the files used by the Common Workflow Language [7], however, that initiative is focused on describing tools and intermediary artifacts, not long-lived benchmarks.

In Software Engineering and adjacent fields, there have been multiple successful field-specific standardization efforts, but no *high-level* specification similar to ours. We will discuss three examples. First, the Automata Wiki [8] collects and archives automata models (e.g., DFAs or Mealy machines) from research projects. However, the storage formats are only informally specified and metadata is only available as HTML pages. Second, the yearly Competition on Software Verification (SV-COMP) [9] compares software verification tools, using standardized benchmarks and tool formats. However, the tool format does not include the full ecosystems; instead, tools target a pre-installed software stack. This makes reuse of tools on foreign systems difficulty. Third, in the domain of security research, many benchmark collections aggregate web applications to be analyzed for security vulnerabilities, e.g., the OWASP benchmark [10], Juliet [11], and STONESOUP [12]. Each benchmark defines different program and metadata formats, showcasing the need for increased standardization. However, tool output formats have mostly been standardized in the SARIF [13] format.

Structure The remainder of this work is structured as follows: Section 2 will introduce our proposed specification for packaging and describing benchmarks, while Section 3 will outline the same for tool containers. Next, we will discuss two case studies: Section 4 will describe our experiences in applying our framework in Software Engineering, while Section 5 describes a study in Environmental Engineering. We summarize our results in Section 6 and discuss potential future work.

2 Recombinable Benchmarks

In this section, we will describe our proposed benchmark format. We will first discuss our preferred approach of storing benchmarks as structured data. Next, we will consider the case of unstructured benchmarks. We will conclude by describing our proposed descriptor format and storage recommendations.

2.1 Structured Benchmarks

Ideally, recombinable benchmarks are stored in structured, preferably hierarchical data formats. Such formats lay out data by nesting primitives like lists, key-value mappings, strings, and numbers. Examples of structured data formats include JSON [14], YAML [15], and XML [16]. When using these formats, benchmark authors must describe both the *syntax* (i.e., the data format being used) and the *semantics* (i.e., what data is encoded into which structures).

```
[
  {
    author: "da Vinci, Leonardo",
    title: "La Gioconda",
    description: "A woman, sitting in front of a landscape",
    image: "/9j/4RJ0", // truncated
  },
  {
    author: "Raffael",
    title: "La scuola di Atene",
    description: "A room with scientists and philosophers",
    image: "/9j/4AAQ", // truncated
  },
]
```

Figure 2: Example Renaissance painting database in JSON5 format. Each entry contains the painter, title, a description, and the digitized painting.

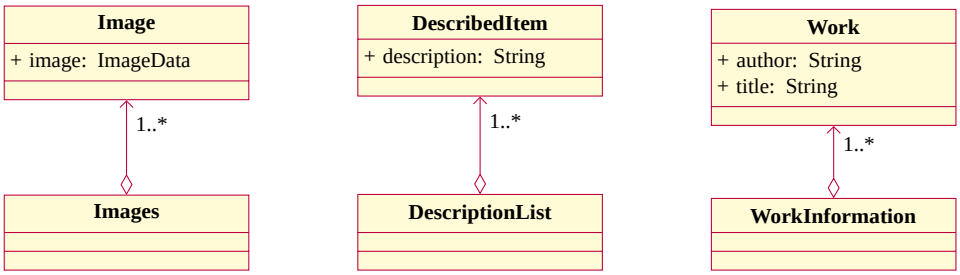
As a running example, we will use the data set shown in Figure 2 aggregating information about Renaissance paintings. For each painting, the file contains painter's name, the work's title, a human-readable description of the subject, and the image itself. The data is structured as JSON5[2] [17]. A data scientist might use this benchmark to train a neural network to recognize the painter from a given image, using both the image data and the painter's name. The same benchmark might be of interest to an art historian studying a painter's preferred subjects from the names and content descriptions.

For describing a benchmark's syntax, we propose using *media types* [18], the quasi-standard for describing file types. For most formats, a standard identifier already exists – in our example, we would use `application/json5` – and for other formats, non-standard identifiers can be defined by prepending the second component with `x-`.

As discussed in the introduction, we envision benchmarks to aggregate as much data as possible, possibly spanning multiple domains, with consumers being responsible for extracting information relevant to them. When describing the semantic content of a benchmark, we therefore need to account for multiple types of data. We describe each type of data as a *facet*. Each subset of facets must be independently accessible in the benchmark, so users will be able to ignore facets irrelevant to their use case. Note that we do not designate facets as "data" or "metadata", but leave this distinction to the benchmarks' consumers. In our example, we would identify three facets: images, descriptions, and work information (i.e., names and titles). While the facet data is stored in proximity, it can still be independently accessed.

In our proposed specification, we independently describe each facet using a schema language, *JSON Schema* [19]. Contrary to its name, this schema language can describe most hierarchical data formats, including JSON, but also a substantial fragment of XML. This stands in contrast to, e.g., XML Schema [20], which is designed to only describe XML. JSON Schemata are uniquely identified by an URL (conventionally, the location where the schema file is hosted). In the example, the facets would be described using the object model shown in Figure 3. Note that each facet independently describes the basic list-of-objects structure and then defines different attributes of each object.

2. An extension to JSON with some convenience features such as comments.



(a) list of images **(b)** list of described items **(c)** list of work metadata

Figure 3: Schemata for three facets of the painting database.

This layout ensures that facets can be composed and accessed at will. The facets contained in a benchmark can be described by the set of schemata the benchmark satisfies. The availability of a large set of open-source validation tools ensures that compliance with a schema can be automatically checked. Finally, since JSON Schema permits the definition of schemata as the intersection of subsidiary schemata, common combinations of facets can be succinctly identified by a single combining schema.

2.2 Unstructured Benchmarks

In practice, not all benchmarks can easily be stored as structured data. If a benchmark is only handled by applications excepting a single, well-established data format, enforcing the use of another format would mean wrapping each tool into a conversion layer, ultimately yielding little benefit. E.g., neural networks are commonly exchanged in the ONNX format [21]. Other data formats can not be well represented by schema. E.g., SV-COMP's [9] benchmarks include C programs, however, "valid C source code" can not be captured by any schema language.

For these cases, we do not recommend conversion to a structured data format and description of facets via schemata. Instead, we suggest standardizing the format and describing it via a domain-specific media type instead of the structured data type – e.g., ONNX could be represented as `application/x-onnx`. Since the media type would prescribe both syntax and semantics, schemata would not be required.

2.3 Storage and Descriptors

To employ recombinable benchmarks in research, researchers must be able to locate and identify benchmarks they can use in their studies. We propose identifying recombinable benchmarks by accompanying them with a metadata file, `rereso-benchmark.yml`. This file would enumerate

- essential metadata, such as name and license information,
- the benchmark's syntax as a media type,
- for structured benchmarks, the URLs of all satisfied schemata, and
- the relative location of the benchmark files.

Interpretation of the location is somewhat dependent on the concrete storage used. However, most research data management services as well as informal storage solutions such as Git repositories offer a file system-like view of their contents, so relative paths can be interpreted here.

```yaml
rereso-benchmark-version: '0.3'
metadata:
  name: Renassiance Painting Database
  description: A collection of Renassiance paintings.
  references:
    - https://paintings.art/
license:
  spdx: CC0-1.0
format:
  media-type: application/json5
  schemas:
    - https://images.art/images-2.json
    - https://generic-schemas.org/described-items-0.1.json
    - https://authorship.com/work-information-1.1.json
benchmarks:
  - path: renassiance-paintings.json5
```

Figure 4: Benchmark descriptor for the Renaissance painting database.

A sample descriptor file for the painting example that showcases most fields of the proposed format is shown in Figure 4. Note that the format specifies both syntax (`application/json5`) and the facets. Each facet has been assigned a (fictitious) URL. The benchmark itself would be placed alongside this descriptor as `renaissance-paintings.json5`.

3 Recombinable Tools

This section describes our proposed tool format. We will first explain our reasoning in choosing a container format and then describe our descriptor format and its interaction with benchmark descriptors.

3.1 Container Format

For tools, the crucial feature is the capability to easily execute the tool on any machine. This necessitates bundling most of a tool's execution environment in a container that can the be distributed. While in industry, Docker is likely the most prevalent technology for containerization, we instead recommend the use of Singularity [22] due to three reasons. First, the tool is already popular in scientific computing. Workflow management systems already offer native support for executing Singularity containers, and issues such as GPU usage are mostly abstracted away by the Singularity executor. Second, containers are intended to be distributed as a single file instead of via an external server, which simplifies long-term archiving as single-file artifacts. Third, Singularity offers easy interaction with the host file system. This satisfies our need for interfaces to inject configuration and benchmarks is easily met be the file system integration: benchmark and configuration files can be placed into a host directory and are immediately visible to the container. The same route can be used to return execution results.

Unfortunately, this format comes with a major downside. Since we distribute compiled code, the resulting containers are bound to a specific host instruction set architecture (e.g, `x86_64` or `aarch64` on modern Apple devices). Even when distributing containers for multiple architectures, there is no guarantee that the software performs identically, and support for future architectures is

essentially impossible to achieve. However, we are not aware of any alternative containerization approach solving these issues.

3.2 Storage and Descriptors

Similar to benchmarks, our proposed descriptor file is stored in a `rereso-tool.yml` next to the tool container file. This descriptor contains

- essential metadata, such as name and license information,
- all ways to invoke the tool, each of which including input and output formats, and
- the relative location of the tool container.

The major difference to benchmark containers is the invocation syntax. We assume that a tool may offer multiple *commands*, each of which performs a different task. Each command is described informally. Therefore, users have to integrate a tool manually – offering an abstract, universal description of invoking programs is beyond the scope of this work. For each command, the descriptor specifies a number of input and output formats in the same syntax as used for benchmarks. A command may have zero inputs (e.g., a benchmark generator), zero outputs (e.g., a test that only detects errors), or both.

This allows for two forms of automation. First, since a command defines the required syntax and necessary semantics, identifying usable benchmarks can be done by checking the input constraint. Second, when building multiple-step command pipelines, each command's output and input formats allows for a compatibility test.

An example for a tool descriptor is shown in Figure 5, packaging the neural network learner from our motivating example as the file `imlt.sif`. The tool offers two commands, the neural network training (`train-classifier`) and a tool to apply the trained network to images (`classify-all`). For trainer, our example benchmark is a valid input and a neural network in the ONNX format is created. The classifier applies the network to a list of images and yields a list of inferred author-title-combinations. For this tool, we could use our benchmarks as the first input and the first command's output as the second input.

4 Case Study A: Heat Pump Mining

We will first discuss a case study in a Software Engineering research project involving the authors. The goal of this project was to apply automata learning methods to reverse engineer (i.e., "mine") a heat pump's control logic from a long-running log of system events. The project is conducted in collaboration with a group at Queen's University, Canada. The learning approach selected is based on the pipeline shown in Figure 6: the log is first processed using a discretizer based on the `nfer` framework [23]. The resulting log is then pared down to a subset of events. Then, a passive learner based on LearnLib [24] is used to derive a automaton model, which is evaluated on a validation data set. The selection, learning, and evaluation components are developed by the authors.

The components developed by our collaborators use a data format that is a tree of CSV files inside a ZIP archive. For this case study, we developed a JSON-based structured benchmark

```yaml
rereso-tool-version: '0.3'
metadata:
  name: Image Metadata Learning Toolbox
  description: A toolbox for learning authors and titles from images.
  references:
    - https://image-recognition.tools/
license:
  spdx: AGPL-3.0-or-later
commands:
  - name: train-classifier [input.json] [classifier.onnx]
    input-formats:
      - media-type: application/json5
        schemas:
          - https://images.art/images-2.json
          - https://authorship.com/work-information-1.1.json
    output-formats:
      - media-type: application/x-onnx  # not formally specified
  - name: classify-all [images.json] [classifier.onnx] [result.json]
    input-formats:
      - media-type: application/json5
        schemas:
          - https://images.art/images-2.json
      - media-type: application/x-onnx
    output-formats:
      - media-type: application/json5
        schemas:
          - https://authorship.com/work-information-1.1.json
container: imlt.sif
```

Figure 5: Tool descriptor for the machine learning tool to recognize creators from images.

format for exchanging logs that was used by our tools and a converter from the ZIP-based format into our JSON-based format that is run between the two phases. To validate the suitability of this format, we also developed converters for five existing log benchmarks in other formats. Finally, we bundled the software implementing both our work and our collaborators' into two tools.

4.1 Log Benchmark Format

To create a suitable benchmark format, we surveyed existing data sets that have a log-like structure (i.e., they contain sequences of events). We defined five facets, shown in Figure 7 to describe the various features present in these benchmarks:

Basic logs are plain lists of objects with a string value, as shown in Figure 7a. All log benchmarks are assumed to satisfy this facet, so criteria such as "at least one log per benchmark" are not repeated by other facets.

Classified logs are shown in Figure 7b. This facet requires that each log has an attached classifier – e.g., a collection of system logs may classify them as "erroneous" or "error-free". Note that this facet does not even demand the existence of logs, instead relying on the base facet will add in this constraint. Hypothetically, this facet could be generalized to classified objects in general.

Normalized logs are shown in Figure 7c. For many logs, the events are free-form string, i.e., not from a limited set of events. For example, the log message `login by user example@company.invalid` could occur in infinite varieties, one for each mail address.

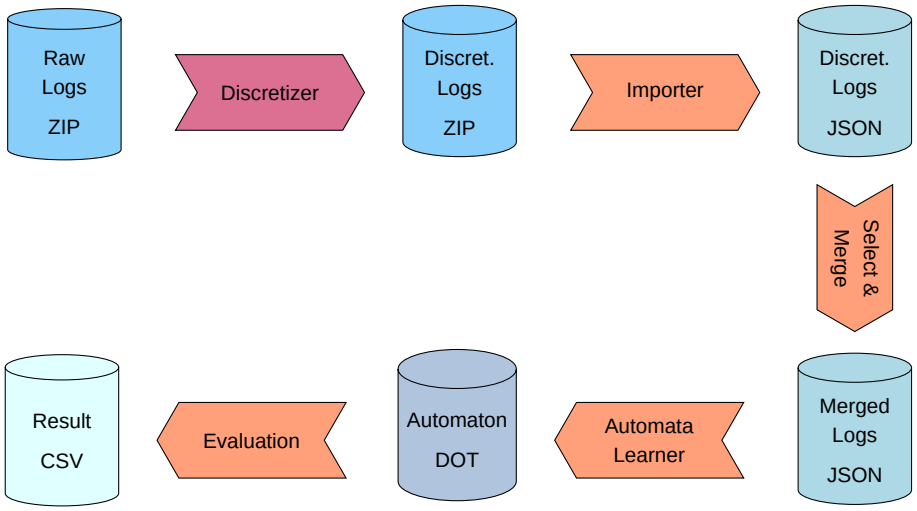


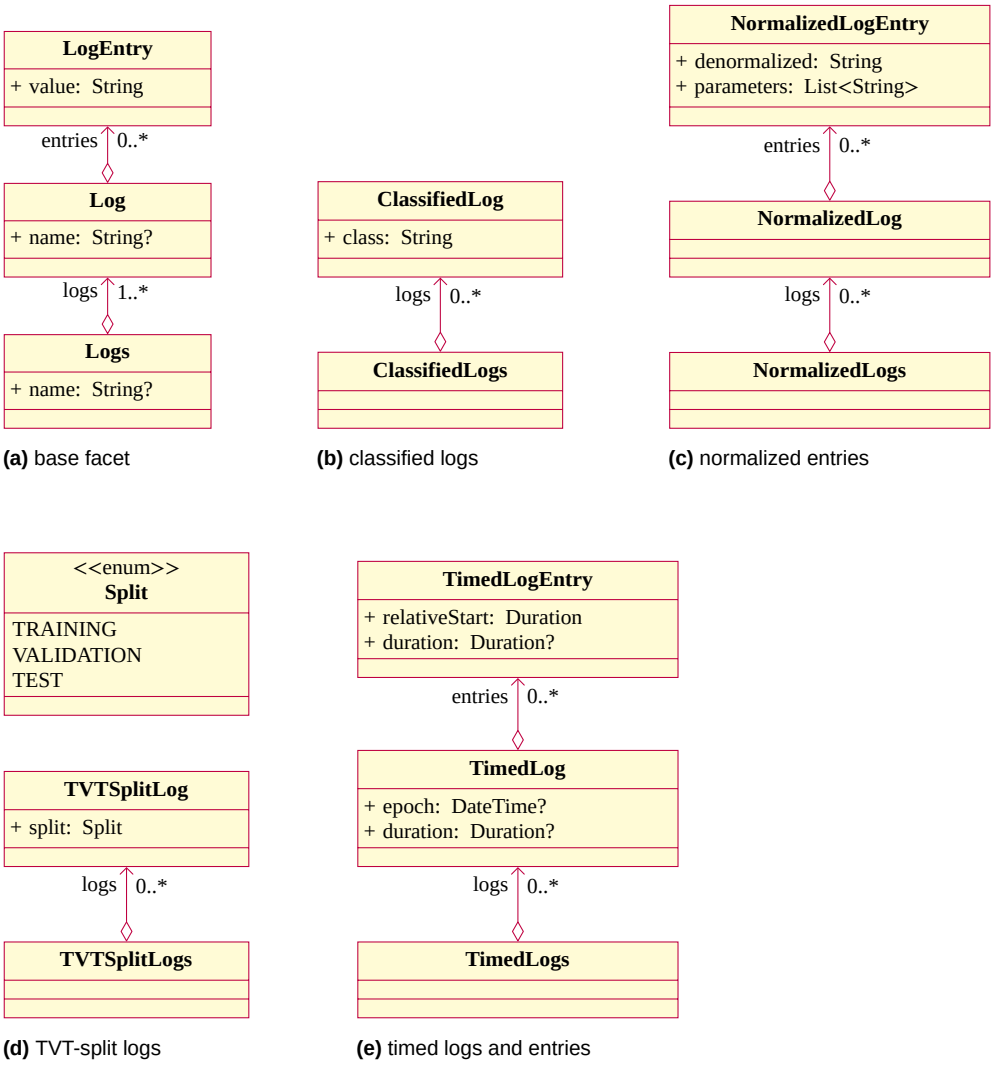
Figure 6: Simplified architecture of the heat pump miner. Multiple tools are applied to the input data in sequence. The last four tools are developed by the authors, while the first is developed by collaborators.

Most automata models cannot represent such inputs. Normalization of such a log maps each entry to a finite set of normalized values plus parameters from a infinite set. In the example, this might result in the representation `login(example@company.invalid)` with the parameter being the email address. In a normalized log, the entries' values are set to the normalized values and the original value plus the parameters are attached to each entry.

TVT-split logs are logs that have already been split into training, validation and test sets, as shown in Figure 7d. This is useful to ensure reproducibility of experiments. Again, this does not refer to logs and could be generalized to split objects.

Timed logs are sequences of values with relative or absolute timestamps (e.g., a classic system log), shown in Figure 7e. This facet requires each entry to define its start point relative to a log-specific value. If the absolute value of the latter is known, it can be stored as an epoch. Additionally, if available, both entries and logs may record their duration. This facet demonstrates how to handle variations in available information by using optional values.

We codified each facet as a schema. For our use, we used (compressed) JSON files to encode logs. In total, our input data after discretization contained 2 268 logs with a total of 1 099 176 entries that only satisfied the base and timed facets. Since the logs were gathered from a real system, the are all examples of expected behavior, so they can not be meaningfully classified. By leveraging libraries for serialization logic, we could implement the converter in only 35 lines of code using the Kotlin programming language.



(a) base facet **(b)** classified logs **(c)** normalized entries

(d) TVT-split logs **(e)** timed logs and entries

Figure 7: The five facets of logs defined in our case study. Note that the facets b–e are intended to be used in combination with the base facet a.

4.2 Additional Log Benchmarks

As a proof of concept, we also developed importers for several of the benchmark sets we considered when defining our format. Table 1 gives an overview of the benchmark sets and the complexity of the importer logic. We developed importers for the following five data sets:

Abbadingo One The training data for the Abbadingo One [25] automata learning competition. These are a set of execution traces from synthetic automata, with participants having to infer a matching automaton from the traces. The test data do not contain classifiers and are therefore omitted.

ALFRED The training and validation data from the ALFRED [26] benchmark's reference actions. The benchmark is designed to evaluate domestic robots; the reference data contains simplified action specifications as logs. The test data is not classified in this benchmark, as well.

Data Set	Features				# of		LOC
	Cls.	Norm.	TVT	Time	Logs	Entries	
Heat Pump				✓	2 268	1 099 176	35
Abbadingo One	✓				440 000	8 390 011	37
ALFRED	✓				7 080	53 094	75
DISC	✓		✓		106 620	5 371 800	53
LogHub Hadoop	✓	✓		✓	978	179 994	101
MIT-AR	✓			✓	425	3 652	183

Cls.: classified logs **Norm.**: normalized entries **TVT**: training-validation-test split **Time**: timestamp information **LOC**: lines of code in the importer

Table 1: Additional data sets converted into the ReReSo log format.

DISC The replication data for the DISC tool [27], a collection of processed benchmarks for replicating the referenced publication's results. We retain the authors' training-validation-test split.

LogHub Hadoop The Hadoop server logs [28] from the LogHub data set [29], processed with the normalization data from the LogHub-2.0 companion data set [30]. These are real-world server logs with timestamps, classified as "normal operation" or by observed errors.

MIT-AR The MIT Activity Recognition [31] benchmark, two multi-day timestamped logs of interactions with household objects, classified by the domestic activities the household's occupants performed at the moment.

We also implemented these converters in our Kotlin framework. On average, a converter required less than 90 lines of code. The MIT-AR importer was the most complex to implement, resulting in 183 lines, mostly due to the difficulties of processing the Matlab data format on the JVM. Anecdotally, most converters could be implemented in a few hours. This demonstrates that the overhead of format conversion, even for existing benchmarks in unusual formats, is usually acceptable.

4.3 Heat Pump Mining Tools

To conclude this case study, we created recombinable tools from both our locally-developed software and the software by our collaborators. Since we implemented our functionality in a single application and its build system already supported creation of a Singularity container, conversion required only 10 lines of code to trigger the build process. For our collaborators' tool, only marginally more work (28 lines of code) was required. In summary, packaging required only minimal effort, even for tools not developed with our recombinable format in mind.

5 Case Study B: DuMu^x

Our second case study is on an existing tool from Environmental Engineering, DuMu^x [32], [33]. DuMu^x is an open-source simulation library based on DUNE [34] and can be used to realize flow and transport simulations in and around porous media. The DuMu^x library bundles a set

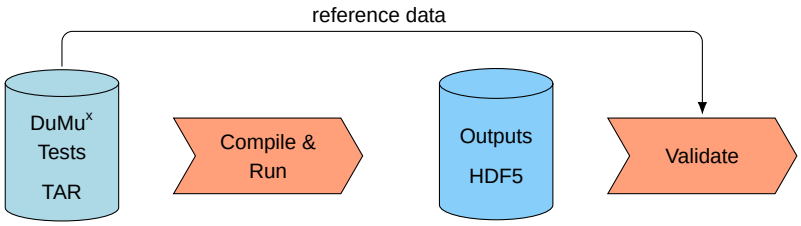


Figure 8: Architecture of the split-tool DuMu^x integration tests.

of integration tests that are deeply integrated into the library's build process. These tests are two-stage: first, an small DuMu^x-based simulation application in C++ is compiled against the DuMu^x library and executed on test data; second, the result data (in HDF5 format) is compared against reference data using `fieldcompare` [35].

In our case study, we made these tests recombinable. We define a non-structured benchmark format for DuMu^x integration tests and convert both the existing tests and a "real" DuMu^x-based tool into it. We then created three tools: one for the build-and-execute step for both DuMu^x 3.8.0 and 3.9.0 and one for the reference data comparison step. Next, we obtained feedback from the DuMu^x development team on the work. Finally, we considered recombinability of DuMu^x-based tools by transforming a DuMu^x-based replication package into a benchmark and tool component.

5.1 Integration Test Benchmark Format

The existing integration tests for DuMu^x each consist of a set of C++ sources, input data and reference data. Additionally, research based on DuMu^x may be performed using a development build that incorporates features missing from the latest release version. Therefore, a benchmark might also contain a patch to DuMu^x that must be applied before compilation.

The resulting benchmarks are therefore proprietary to DuMu^x– even without patching, only a fully API-compatible software could hypothetically run them. We therefore do not use a structured benchmark format, but instead define a benchmark as a compressed TAR archive containing the above-mentioned components and a descriptor file with file names etc.

We implemented a generic converter for the DuMu^x-internal test cases. Written in Python, the converted comprises 372 lines of code and can transform all tests fully automatically into 15 benchmarks. We also converted the `Koch2024a` replication artifact for a simulation of flow in brain tissue [36] into a benchmark using bespoke conversion logic (13 lines of code plus 145 lines in patches and metadata). While development of the generic converter was more complex than the software developed for our first case study, the total effort remained limited.

5.2 Integration Test Tools

Transforming the test pipeline into recombinable tools was greatly simplified by the existence of DuMu^x Docker containers that could be used as the basis for our Singularity artifacts for building and executing. The `fieldcompare` wrapper bundled with DuMu^x, `runtest.py`, needed to be modified to handle our use case, but could be extracted from DuMu^x and integrated into a simple

container for Python applications. In total, this required 109 lines of packaging code, including changes to `runtest.py`.

However, since our benchmark format deviates from the file layout expected by DuMu^x and `runtest.py`, we also needed to implement two adapter scripts that handle unpacking of the benchmark and invocation of the necessary commands inside the container. These amount to 71 lines of shell script.

5.3 Evaluation by DuMu^x Developers

We gathered feedback on our CI work from the DuMu^x development team. While the technical aspects worked as expected, feedback on the use of the proposed CI architecture in the scope of the DuMu^x project was more critical, with the unstable API of DuMu^x being seen as a hindrance to using our approach at the current state. This can be verified using our approach: most integration tests taken from DuMu^x 3.9.0 fail to build with DuMu^x 3.8.0.

It was emphasized that the approach in general appears sound and could be used in conjunction with a more limited, stable API (e.g., a Python API). Also, interest in applying recombinability to DuMu^x-based tools such as replication packages was expressed.

5.4 DuMu^x-Based Replication Package

We applied our previous experiences to transform the DuMu^x-based `Koch2024a` replication package [36] (which we already used as a benchmark) to a tool-benchmark-combination. Here, the tool is the complete simulation application, whereas the benchmark is its input data. Hypothetically, this data could be consumed by other tools performing the same simulation task. Since we can fully compile the tool during packaging, this is simpler to implement than the integration tests.

We used a simplified version of the non-structured IT benchmark format for the benchmark that relied on naming conventions instead of a metadata file. Creation of the tool was greatly helped by the presence of a Docker build file in the replication package, which only required the addition of a 15-line wrapper script to handle unpacking. In total, the packaging code required 34 lines of code. This showcases how, with increasing experience, the overhead imposed by our approach becomes almost negligible in combination with good software engineering practices.

6 Conclusion

We have developed formats for both benchmarks and tools that enable *recombination*. Benchmarks are stored as structured data, with facets described using JSON Schema or, less preferably, in domain-specific formats combining syntax and semantics. Tools are to be distributed as Singularity containers. We also developed descriptor formats that describe such artifacts in a machine-readable manner. This contributes to findability, interoperability, and reusability w.r.t. the FAIR and FAIR4RS principles.

By applying our approach in two case studies, we demonstrated that it is useful for the tasks at hard, agnostic of research-disciplines, and only causes a very limited overhead. As summarized

	Case Study	# of Benchm.	# of Tools	Lines of Code Packaging	Lines of Code Runtime
A	HPM phase 1 tools	0	1	28	0
A	HPM phase 2 tools	0	1	10	0
A	other log benchmarks	140	0	68	0
B	DuMu^x IT tools	0	3	109	71
B	DuMu^x IT examples	15	0	372	0
B	`Koch2024a`-based IT benchmark	1	0	13	145
B	`Koch2024a` reproduction package	1	1	34	15

Table 2: Effort required to convert the DuMu^x artifacts to our specification.

in Table 2, no artifact required more than 400 lines of code to create. While we did not precisely track development time, anecdotally, every conversion required less than a day of effort to implement.

Future Work We envision multiple directions for future work. First, our proposed specification is ready to be applied to a wide variety of use cases in which research projects could benefit from recombinable artifacts. For example, our `ralib-benchmarking` framework [37] performs evaluations via a series of shell scripts that would benefit from proper packaging. Second, integration of our descriptors into systems such as into Betty's (Re)Search Engine [38] would increase findability of recombinable tools. Third, in domains such as autonomous mobility research, tools are often combined into complex workflows, with data being distributed, merged etc. Our tool specification could serve as a building block to modeling complex workflows in this and other fields of study.

7 Acknowledgements

The authors would like to thank the Federal Government and the Heads of Government of the Länder, as well as the Joint Science Conference (GWK), for their funding and support within the framework of the NFDI4Ing consortium. Funded by the German Research Foundation (DFG) – project numbers 442146713 (NFDI4Ing); 495857894 (STING).

We thank Bernd Flemisch for many productive discussions and his assistance with the DuMu^x tool. We also would like to thank both Sean Kauffman and Nastaran Kianersi for their feedback on case study A and the DuMu^x development team for theirs on case study B.

8 Roles and contributions

Simon Dierl: Conceptualization, Investigation, Methodology, Software, Writing – original draft

Falk Howar: Conceptualization, Funding acquisition, Supervision, Writing – review & editing

References

[1] Association for Computing Machinery, *Artifact review and badging - current*, Version 1.1, Aug. 2020. Accessed: Jan. 17, 2025. [Online]. Available: https://www.acm.org/publications/policies/artifact-review-and-badging-current.

[2] European Association for Programming Languages and Systems, *EAPLS artifact badges - May 2021*, May 2021. Accessed: Oct. 6, 2024. [Online]. Available: https://eapls.org/pages/artifact_badges/.

[3] M. D. Wilkinson et al., "The FAIR Guiding Principles for scientific data management and stewardship," *Scientific Data*, vol. 3, no. 1, Mar. 2016, Art. no. 160018, ISSN: 2052-4463. DOI: 10.1038/sdata.2016.18.

[4] M. Barker et al., "Introducing the FAIR Principles for research software," *Scientific Data*, vol. 9, no. 1, Oct. 2022, Art. no. 622, ISSN: 2052-4463. DOI: 10.1038/s41597-022-01710-x.

[5] C. Collberg and T. A. Proebsting, "Repeatability in computer systems research," *Communications of the ACM*, vol. 59, no. 3, pp. 62–69, Feb. 2016, ISSN: 0001-0782. DOI: 10.1145/2812803.

[6] O. Corcho et al., *EOSC interoperability framework: report from the EOSC Executive Board Working Groups FAIR and Architecture*. Publications Office of the European Union, 2021, ISBN: 978-92-76-28949-4. DOI: 10.2777/620649.

[7] M. R. Crusoe et al., "Methods included: Standardizing computational reuse and portability with the Common Workflow Language," *Communications of the ACM*, vol. 65, no. 6, pp. 54–63, May 2022, ISSN: 0001-0782. DOI: 10.1145/3486897.

[8] D. Neider, R. Smetsers, F. Vaandrager, and H. Kuppens, "Benchmarks for automata learning and conformance testing," in *Models, Mindsets, Meta: The What, the How, and the Why Not?* Ser. Lecture Notes in Computer Science, T. Margaria, S. Graf, and K. G. Larsen, Eds., vol. 11200, Cham: Springer International Publishing, 2019, pp. 390–416, ISBN: 978-3-030-22348-9. DOI: 10.1007/978-3-030-22348-9_23.

[9] D. Beyer and J. Strejček, "Improvements in software verification and witness validation: SV-COMP 2025," in *Tools and Algorithms for the Construction and Analysis of Systems*, A. Gurfinkel and M. Heule, Eds., ser. Lecture Notes in Computer Science, vol. 15698, Cham: Springer Nature Switzerland, 2025, pp. 151–186, ISBN: 978-3-031-90660-2. DOI: 10.1007/978-3-031-90660-2_9.

[10] The OWASP Foundation, *OWASP Benchmark*, Oct. 1, 2016. [Online]. Available: https://owasp.org/www-project-benchmark/.

[11] NSA Center for Assured Software, *Juliet Java 1.3*, Oct. 1, 2017. [Online]. Available: https://samate.nist.gov/SARD/test-suites/111.

[12] IARPA, *IARPA STONESOUP phase 3 - virtual machine 3.0*, May 1, 2015. [Online]. Available: https://samate.nist.gov/SARD/test-suites/113.

[13] OASIS Open, "Static analysis results interchange format (SARIF) version 2.1.0 plus errata 01," OASIS, OASIS Standard incorporating Approved Errata, Aug. 28, 2023. [Online]. Available: https://docs.oasis-open.org/sarif/sarif/v2.1.0/errata01/os/sarif-v2.1.0-errata01-os-complete.html.

[14] T. Bray, "The JavaScript object notation (JSON) data interchange format," RFC Editor, Internet Standard 90, Dec. 2017. DOI: 10.17487/RFC8259.

[15] YAML Language Development Team, "YAML ain't markup language (YAML™) version 1.2," Tech. Rep., Oct. 2021, Revision 1.2.2. [Online]. Available: https://yaml.org/spec/1.2.2/.

[16] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, F. Yergeau, and J. Cowan, "Extensible markup language (XML) 1.1 (second edition)," W3C, W3C Recommendation, Aug. 2006. [Online]. Available: https://www.w3.org/TR/2006/REC-xml11-20060816.

[17] A. Kishore and J. Tucker, "The JSON5 data interchange format," Tech. Rep., Mar. 2018. [Online]. Available: https://spec.json5.org/.

[18] N. Freed and N. S. Borenstein, "Multipurpose internet mail extensions (MIME) part two: Media types," RFC Editor, Request for Comments 2046, Nov. 1996. DOI: 10.17487/RFC2046.

[19] A. Wright, H. Andrews, B. Hutton, and G. Dennis, "JSON Schema draft 2020-12," JSON Schema, Tech. Rep., Jun. 16, 2022. [Online]. Available: https://json-schema.org/draft/2020-12.

[20] P. Walmsley and D. Fallside, "XML Schema part 0: Primer second edition," W3C, W3C Recommendation, Oct. 2004. [Online]. Available: https://www.w3.org/TR/2004/REC-xmlschema-0-20041028/.

[21] The Linux Foundation, *ONNX 1.19.0 documentation*, 2024. [Online]. Available: https://onnx.ai/onnx/index.html.

[22] G. M. Kurtzer, V. Sochat, and M. W. Bauer, "Singularity: Scientific containers for mobility of compute," *PLOS ONE*, vol. 12, no. 5, May 2017, Art. no. e0177459, ISSN: 1932-6203. DOI: 10.1371/journal.pone.0177459.

[23] S. Kauffman, K. Havelund, and R. Joshi, "`Nfer` – a notation and system for inferring event stream abstractions," in *Runtime Verification*, Y. Falcone and C. Sánchez, Eds., ser. Lecture Notes in Computer Science, vol. 10012, Cham: Springer International Publishing, 2016, pp. 235–250, ISBN: 978-3-319-46982-9. DOI: 10.1007/978-3-319-46982-9_15.

[24] M. Isberner, F. Howar, and B. Steffen, "The open-source LearnLib," in *Computer Aided Verification*, D. Kroening and C. S. Păsăreanu, Eds., ser. Lecture Notes in Computer Science, vol. 9206, Cham: Springer International Publishing, 2015, pp. 487–495, ISBN: 978-3-319-21690-4. DOI: 10.1007/978-3-319-21690-4_32.

[25] K. J. Lang, B. A. Pearlmutter, and R. A. Price, "Results of the Abbadingo One DFA learning competition and a new evidence-driven state merging algorithm," in *Grammatical Inference*, V. Honavar and G. Slutzki, Eds., ser. Lecture Notes in Computer Science, vol. 1433, Berlin, Heidelberg: Springer, 1998, pp. 1–12, ISBN: 978-3-540-68707-8. DOI: 10.1007/BFb0054059.

[26] M. Shridhar et al., "ALFRED: A benchmark for interpreting grounded instructions for everyday tasks," in *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2020, pp. 10 737–10 746, ISBN: 978-1-7281-7168-5. DOI: 10.1109/CVPR42600.2020.01075.

[27] M. Shvo, A. C. Li, R. Toro Icarte, and S. A. McIlraith, "Interpretable sequence classification via discrete optimization," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, no. 11, pp. 9647–9656, May 2021. DOI: 10.1609/aaai.v35i11.17161.

[28] Q. Lin, H. Zhang, J.-G. Lou, Y. Zhang, and X. Chen, "Log clustering based problem identification for online service systems," in *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, New York, NY, USA: Association for Computing Machinery, May 2016, pp. 102–111, ISBN: 978-1-4503-4205-6. DOI: 10.1145/2889160.2889232.

[29] J. Zhu, S. He, P. He, J. Liu, and M. R. Lyu, "Loghub: A large collection of system log datasets for AI-driven log analytics," in *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*, Oct. 2023, pp. 355–366, ISBN: 979-8-3503-1594-3. DOI: 10.1109/ISSRE59848.2023.00071.

[30] Z. Jiang et al., "A large-scale evaluation for log parsing techniques: How far are we?" In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, New York, NY, USA: Association for Computing Machinery, Sep. 2024, pp. 223–234, ISBN: 979-8-4007-0612-7. DOI: 10.1145/3650212.3652123.

[31] E. M. Tapia, S. S. Intille, and K. Larson, "Activity recognition in the home using simple and ubiquitous sensors," in *Pervasive Computing*, A. Ferscha and F. Mattern, Eds., ser. Lecture Notes in Computer Science, vol. 3001, Berlin, Heidelberg: Springer, 2004, pp. 158–175, ISBN: 978-3-540-24646-6. DOI: 10.1007/978-3-540-24646-6_10.

[32] B. Flemisch et al., "DuMu^x: DUNE for multi-{phase, component, scale, physics, …} flow and transport in porous media," *Advances in Water Resources*, vol. 34, no. 9, pp. 1102–1112, Sep. 2011, ISSN: 0309-1708. DOI: 10.1016/j.advwatres.2011.03.007.

[33] T. Koch et al., "DuMu^x 3 – an open-source simulator for solving flow and transport problems in porous media with a focus on model coupling," *Computers & Mathematics with Applications*, vol. 81, pp. 423–443, Jan. 2021, ISSN: 0898-1221. DOI: 10.1016/j.camwa.2020.02.012.

[34] P. Bastian et al., "The DUNE framework: Basic concepts and recent developments," *Computers & Mathematics with Applications*, vol. 81, pp. 75–112, Jan. 2021, ISSN: 0898-1221. DOI: 10.1016/j.camwa.2020.06.007.

[35] D. Gläser, T. Koch, S. Peters, S. Marcus, and B. Flemisch, "fieldcompare: A Python package for regression testing simulation results," *Journal of Open Source Software*, vol. 8, no. 81, 2023, Art. no. 4905, ISSN: 2475-9066. DOI: 10.21105/joss.04905.

[36] T. Koch and K.-A. Mardal, "Estimation of fluid flow velocities in cortical brain tissue driven by the microvasculature," *Interface Focus*, vol. 15, no. 1, Apr. 2025, Art. no. 20240042, ISSN: 2042-8901. DOI: 10.1098/rsfs.2024.0042.

[37] S. Dierl, P. Fiterau-Brostean, F. Howar, B. Jonsson, K. Sagonas, and F. Tåquist, “Scalable tree-based register automata learning,” in *Tools and Algorithms for the Construction and Analysis of Systems*, B. Finkbeiner and L. Kovács, Eds., ser. Lecture Notes in Computer Science, vol. 14571, Cham: Springer Nature Switzerland, 2024, pp. 87–108, ISBN: 978-3-031-57249-4. DOI: 10.1007/978-3-031-57249-4_5.

[38] V. Seibert, A. Rausch, and S. Wittek, “Betty’s (Re)Search Engine: A client-based search engine for research software stored in repositories.,” *ing.grid*, vol. 1, 2 May 2024, ISSN: 2941-1300. DOI: 10.48694/inggrid.3953.