"Automatic Control Knowledge Repository" – A Computational Approach for Simpler and More Robust Reproducibility of Results in Control Theory

Carsten Knoll[1,*] and Robert Heedt[1]

[1] *Institute of Control Theory, Dresden University of Technology*
[*] Corresponding author: <firstname>.<lastname>@tu-dresden.de

2020

Abstract

As many other disciplines, control theory to some degree suffers from a reproducibility crisis. In particular, since computational methods like simulation, numeric approximation or computer algebra play an important role, the reproducibility of results relies on implementation details, which are typically out of scope for written papers. While some publications do reference the source code of the respective software, this is by far not standard in industry and academia. Additionally, having access to the source code does not imply reproducibility due to dependency issues w. r. t. hardware and software components. This paper proposes a three-component approach to mitigate the problem: a) an open repository with a suitable data structure to publish formal problem specifications and problem solutions (each represented as source code) along with necessary metadata, b) a web service that automatically checks the solution methods against the problem specifications and auxiliary software for local testing and c) a peer-oriented process scheme to organize both the contribution process to that repo and formal quality assurance. The proposed concept offers simplified and more robust reproducibility, as well as increased visibility of published results and a community-curated set of reference implementations for control-related methods.

1 Introduction

The scientific advances of the past have been possible because researchers either added their own ideas to earlier ones or modified them to draw different conclusions. This process is mainly facilitated by the publication of results. For the creative individual it is necessary to fully understand published results in their important details like assumptions, reasoning, conclusions – both for improving and for criticizing. Despite omnipresent digital technology, the main medium for the necessary information transport is still (written) natural language. Like many other disciplines, scientific communication in control theory typically is enriched with an ample stock of mathematical symbols and other formalisms such as block diagrams, but natural language still provides the necessary context.

However, since more and more computational methods play a crucial role in systems and control research, it becomes increasingly difficult to fully understand the essence of many publications, instead of merely believing their statements and claims. The difficulties of software in relation to reproducibility have been recognized early on [1] and are still a frequent point of discussion [2–6]. Recently, the journal *ReScience C* [7] was founded specifically to document examples of these issues. From the perspective of the authors, full understanding implies the ability to independently reproduce the results from a ground of common assumptions. For (conventional) mathematical proofs and similar reasoning, natural language and written formal symbols therefore still seem appropriate. On the other hand, for results that practically cannot be reproduced (i. e. understood) without using computation (like simulation, numerical linear algebra, or symbolic calculations with computer algebra systems), it appears naturally that all information that is necessary to reproduce these results should be provided along with the published description in natural language. This includes experimental data [8–10] but also, and arguably more important in our field, source code for simulations and other types of calculations [11–17]. Even with the information pieces seemingly available in published texts and source code, interested readers still often run into road blocks caused by differences in the en-

vironment, configuration, or missing documentation. Recent efforts therefore propose employing so called Continuous Integration (CI) [18–20], a concept explained below.

A further problem, which might also dissipate significant research resources, is to manually search for already existing solutions of the problem one currently investigates (or similar/related problems). In other words, the question here reads: "*Are there any published relevant results in the first place, whose reproduction should be attempted?*" While classical search engines are doubtlessly helpful, they operate mainly on the *syntactical* level (e. g. searching for matching words) and thus are of limited use. An improvement can be expected by the use of *semantic* information as this would allow searching via the *meaning of words* instead of their syntactical representation as a string of characters. However, publishing results with such suitable semantic information requires certain standardization. Additionally, efficient searching would benefit greatly from the existence of a *central* knowledge base.

Assuming that an upcoming publication owes (part of) its content to computational methods, the present contribution proposes the following: a) As a supplement to the usual natural-language-based publications (typically published as PDF file), the related source code along with semantic metadata should be published to a central and open repository with a suitable data structure. To facilitate this, we further suggest b) a web service connected to that repository that automatically checks the solution methods against the problem specifications and a suite of auxiliary software for local testing. Finally, we propose c) a peer-oriented process scheme to organize the contribution process (acceptance/refusal) and formal quality assurance without the need for a central authority.

From the viewpoint of the authors, the proposed approach of an "active knowledge[1] repository" interconnects innovative technical and social aspects. However, to have any noteworthy effect on the above mentioned replication problems it must gain some significant support and adoption by other researchers and therefore must be open to ideas, criticism and improvement suggestions by the community from an early stage. Consequently, the presented approach should not be considered a "final solution" but instead a conceptual draft, which might be the point of departure for further discussion and iterative development.

[1]The exact meaning of the term 'knowledge' is a philosophical issue and thus far beyond the scope of this paper. In this contribution we mainly refer to instrumental aspects of knowledge: The *know how* suitable to solve certain specific problems. This kind of knowledge can beneficially be represented as source code.

The remainder of this paper is structured as follows: Section 2 summarizes relevant established technologies and a concept for how to integrate them. Section 3 proposes the above mentioned dedicated structure of the repository, while Section 4 describes how the repository and the proposed web service can be combined to perform a `contribution` as the basic use case of the whole platform. In Section 5 we briefly comment on what elements of the concept are already implemented (see also [21]) and in Section 6 answer some critical questions, which are expected to arise for the reader.

2 Existing Solutions and Integration Concept

This section briefly summarizes already existing technologies and techniques. Then these elements are combined to form an integrated process concept.

2.1 Collaboration via Distributed Version Control Systems

A key feature for digitally represented information is the ability to perform changes (including adding or removing content) with very little effort. However, keeping track of such changes is a challenge on its own, especially if changes are asynchronously made by several individuals. In software engineering, version control systems (VCS) are used to solve this problem. During the last decade, decentralized version control systems (DVCS) and most prominently the software *git* have arguably become the de facto standard. A set of information that is under version control is called a *repository*. It typically represents a file system (directories and files) along with its evolution history. This history is represented by a collection of incremental snapshots, stored along with metadata like time stamp or author as a collection of *commits*. The order relation inside such a collection is ensured by using a hash tree (also called Merkle tree [22]) as the data structure: For each commit, a unique cryptographic hash is calculated and stored as part of the metadata. Additionally, each commits metadata also contains the hash code(s) of its parent(s) commit(s), i. e. the state of the repository to which the incremental snapshot applies.

This data structure allows for temporary parallelism. Each node (commit) in the tree can have multiple child nodes (*branches*). Moreover, each node can have multiple parent nodes too, which enables branches to be *merged* together after a period of independent evolution, see Fig. 1. Git pro-

Preprint: Only for personal use.

vides mechanisms to resolve merge conflicts automatically or manually.

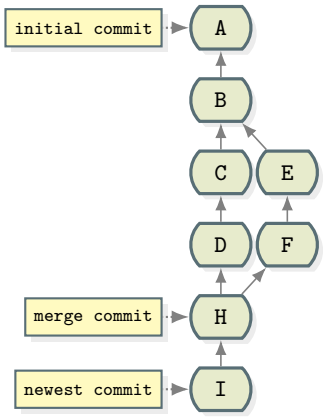

Figure 1: Schematic visualization of branching and merging in the distributed version control system git.

This technology facilitates a distributed development model, which is adopted by many open source projects: Potential contributors *clone* the original repository, i. e. create a local copy of it, make their own contributions in the form of commits, and turn in a *merge request* for the original repository. Typically, some review iterations take place before the external contribution is merged into the main repository.

2.2 Automated Tests and Continuous Integration Services

An *automated test* is a piece of software that executes other pieces of software and compares their results to (hardcoded) expected results [23]. Roughly, tests can be subdivided into *unit tests*, checking the isolated functioning of a small piece of software, and *black box tests*, checking correct input-output behavior of a larger piece of software.

While creating and maintaining such "unproductive" code takes additional effort, this investment usually is justified even for projects of modest complexity by the increased efficiency of identifying unintended behavior ("bugs"). Additionally, automated tests are an important technology if heterogeneous execution environments (e. g. different version of dependency libraries) should be supported and greatly help to facilitate distributed software development, see section 2.1. Due to this surplus value of suitable automated testing, so called *continuous integration services* were developed. Such a service basically monitors the state of a repository and whenever this state changes (e. g. by some uploaded commits) it executes the set of automated tests that are defined in the repository itself and reports the result in a suitable way.

2.3 Integration Concept: "Automatic Control Knowledge Repository"

The aforementioned technologies and approaches do exist and have been applied for years or even decades. However, to the best of the authors' knowledge, there does not exist a combination of these components suitable for the needs of the scientific community and in particular the requirements of researchers in the field of automatic control and systems theory. Thus, the purpose of this contribution is to propose such a combination and to expose these ideas to critical comments from the research community.

As mentioned in the introduction, we propose in particular

1. to establish a central repository in the sense of Sec. 2.1, which collects content-related software (e. g. control theoretic algorithms), organizational software (the automated tests) and metadata in a formal structure,
2. to establish a supportive web service for that repository, providing CI functionality tailored to its structure and to the needs of systems and control engineering, plus a suite of software tools to simplify contributions to that repository e. g. by allowing for local testing,
3. to establish a set of rules that distribute the decision making about submitted contributions (merge requests) and other maintenance work among the previous active contributors.

We call this concept *active knowledge repository* to reflect the fact that it is more than a passive store of information. Due to the testing carried out by the web service, it is guaranteed that the contained software had worked under clearly specified conditions on computational infrastructure that is independent from the content contributor (and thus not prone to undiscovered dependency issues). The term "knowledge repository" distinguishes from the term "knowledge base" because it suggests dynamic evolution by contributions from multiple sources and from the mere "repository" because it hints that it does not just contain source code but by the kind and structure of this source code (and metadata), the repository content can be regarded as a representation of domain specific problem solving knowledge (which can be transferred to other problems).

Preprint: Only for personal use.

3 Proposed Repository Structure

To reproduce a result, it is necessary to reduce it to a binary yes-no question. Our approach is based on the observation that the typical (computational) result in control theory can be split up into a precise description of the problem (or class of problems), e. g. computing an input trajectory for an equilibrium transition of a state-space system, and a description of how to solve these problems. This corresponds to the concept of a unit test, where the problem description is the test case and the solution description is the software to be tested. As a test case can succeed or fail, a problem solution can either solve a problem (i. e. calculate the expected result from the provided data) or fail to do so. To achieve flexibility and modularity, the repository structure requires a formal split of contributions into independent (but linked) *entities* of different types, see Fig. 2. The most important entities are `ProblemSpecification` and `ProblemSolution`[2].

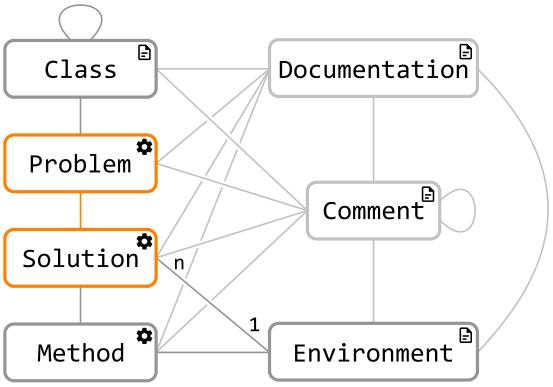

Figure 2: Relations between the various entities. All cardinalities are m-to-n with one exception: Each `ProblemSolution` is associated with exactly one `EnvironmentSpecification`. Note that entities of types `ProblemClass` and `Comment` can also reference entities of their own type. The gear wheel symbolizes executable code associated with the entity, other (meta)data is marked by the file icon.

Note that in this structure a `ProblemSpecification` is also given by executable code (indicated by the gear wheel). The solution entity is expected to represent the concrete application of zero or more `MethodPackages`, which for example contain the code for solving boundary value problems. Such methods might be complex pieces of software, which are compatible with one or more runtime `EvironmentSpecifications`.

In the other direction, to facilitate the discoverability of concrete `Problem` entities, they can be associated to one or more `ProblemClasses`, e. g. "trajectory planning of mechanical systems" or "trajectory planning of nonlinear state-space systems". This entity type can be considered as an elaborated keyword structure – it does not contain code, but it can be the parent entity of concrete `Problems` as well as other `ProblemClasses`. This allows the construction of a taxonomy, i. e. a semantic categorization of solved (and also open) problems in the field of systems and control theory. Such a taxonomy could then be a starting point for an even deeper ontological coverage of the field [24–27], which, from the perspective of the authors, promises further advances for accessibility and reproducibility of control-related knowledge.

Finally there are entity types for `Documentation` (with obvious use) and `Comments`. The latter provides a formal possibility for transparent long term communication, such as improvement suggestions or criticism related to other entities.

In the repository, each entity is represented by a directory containing at least one file named `metadata.yml`, which must specify some defined[3] attributes in the widespread and easy to edit YAML format. The most important of these attributes is the *entity key*, which is required to be a five character string of numbers and capital letters and serves to uniquely identify every entity in the repository. While most attribute fields are generic to all entity types, some are specific to one type such as the field `estimated_runtime` for a `ProblemSolution`. Entities that contain source code obviously need to include the respective files in their entity directory and reference it in the metadata, e. g. `solution_file="solution.py"` for the type `ProblemSolution`.

While the type for each entity is specified in the metadata, it is a convention to have a top level directory structure that maps to the different types. Inside these top level directories, arbitrary subdirectories can exist, but entities are not allowed to be nested.

[2] Short forms: `Problem` and `Solution`. Other short forms are similarly self-explaining.

[3] For details see the `README.md` file of [28].

Preprint: Only for personal use.

4 General Architecture and Process Scheme for Contribution

4.1 Architecture Overview

So far, we only have a passive repository with a suggested structure (in the following named `ackrep_data`). However, the main advantage of the proposed approach should be that a contribution to the repo will only be merged if it is *consistent*, i. e. obeys the formal structure, and all (old and new) `ProblemSolutions` are able to solve the related `Problems`. In other words: all tests must pass. To check this, we propose a suitable interoperation between different software components. This architecture is visualized in Fig. 3.

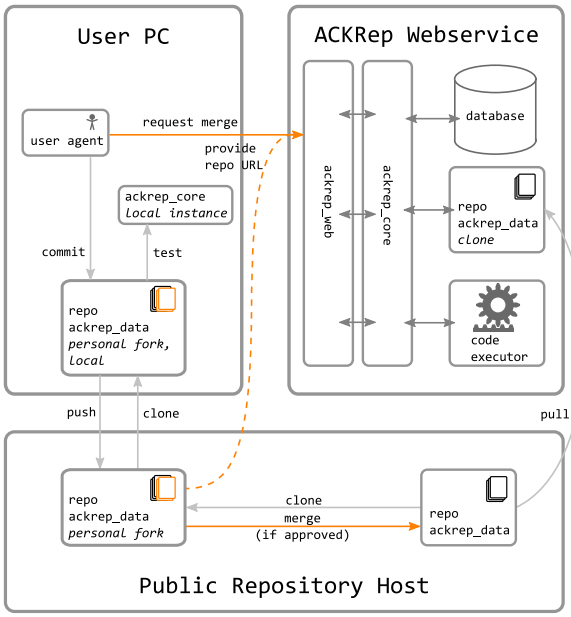

Figure 3: Schematic visualization of the proposed component architecture and their interaction during the use case of a contribution (see Sec. 4.3). The orange elements are associated directly with a merge request.

The basic component is a software that crawls through the repository, checks the consistency of each entity, collects all solutions and related problems, runs the individual test cases and reports the results. In the current prototype implementation this software is called `ackrep_core`. It is available with a command line interface, which allows local testing on personal IT infrastructure. However, the relevant test run, which decides whether a contribution is acceptable, must take place on independent infrastructure, i. e. on a server. To comfortably interact with `ackrep_core` on that server, there is a web interface (`ackrep_web`). To simplify the processing and interconnection of the entities, the metadata from the repository is loaded into a database. When a solution check is triggered via the web service, `ackrep_core` creates an executable script from a template. This script is then executed by a component named `code executor`, which is configured according to the relevant `EnvironmentSpecification`. The script passes the `ProblemSpecification` object to the `ProblemSolution` function. Then the script passes the result into the `evaluate_solution` function, which finally compares the numerical/symbolical values of the solution with the expected ones and decides whether the test was passed or failed. The result is displayed to the user via `ackrep_web` along with optional graphical results that were created by the solution, cf. Fig. 4.

Obviously, there must be a copy of the `ackrep_data` on the server. However, the "canonical place to live" for this repo should rather be a public repository host such as *github.com*, which we chose for the prototype. Alternatives would be e. g. *gitlab.com*, *bitbucket.com* or *codeberg.org* (non-profit). Such a service provides features for browsing the repository history and creating a personal *clone*, which can evolve into a *fork*, i. e. a version of the repository that shares the history but after a branching point differs from the official repo state, e. g. due to additional commits. For such a fork (which gets its own URL from the repo host), a potential contributor can then file a merge request via `ackrep_web`, which triggers the consistency and solution checking described above.

4.2 Basic Interaction Scheme – Constitutional Rules

The merge request described above only has to be triggered by a potential contributor and the formal quality assurance in the form of consistency and solution checking takes place automatically. However, these formal conditions are only necessary but not sufficient. The final decision whether a contribution should be merged to the official (i. e. canonical) `ackrep_data` should be made by humans based on relevance of the contribution and its scientific and educational value.

The aim to establish peer-based decision making in potentially controversial situations while minimizing both conflict and bureaucratic overhead poses some social challenges, which are tackled in the form of a set of rules called *constitution*. The basic idea is that any decision, both regarding contributions and the constitution itself, should

Preprint: Only for personal use.

be made consensually by the *group of active contributors* (see below). For initial simplicity, there is no other body of decision-making, but as this group grows, it can and should (consensually) decide to establish a more elaborated structure. To facilitate this idea, the proposed initial constitution describes a *scalable rejection-minimizing consensus mechanism*, see `Constitution.md` in [28].

From the viewpoint of the authors, rejection minimization is important for the following reason: In contrast to rules implemented in program code, which are applied to inanimate and indifferent pieces of information ("data"), the subject of the constitutional rules are human individuals with their own interests and preferences. Thus, these rules are not followed due to natural laws (as is the case for data processed by software on a digital device) but due to free decision. If a majority vote displeases group members, it is likely they leave and will not contribute anymore.

4.3 Basic Use Case: Making a Contribution

For the sake of understandability, the basic use case is described as a concrete example. We assume a) that there is a set distinct of individuals $\{\mathcal{C}_1, \ldots, \mathcal{C}_n\}$ who have already contributed to the repository ("active contributors") and b) a *prospective contributor* $\mathcal{P}$ who has developed a method for trajectory planning of nonlinear state-space systems and wants to contribute this method to the `ackrep_data` repository. To accomplish this, the following steps are required:

1. $\mathcal{P}$ creates a personal fork of `ackrep_data` and locally installs the supporting software (`ackrep_core`).
2. $\mathcal{P}$ splits up their code into entities according to the required data structure[4], see Sec. 3.
3. With the help of the `ackrep` command line tool, $\mathcal{P}$ tests locally whether all consistency and solution checks pass and, if necessary, fixes all reported formal issues.
4. If no issues are reported, $\mathcal{P}$ pushes the changes to their personal fork.
5. $\mathcal{P}$ files a *merge request*[5] via the *ackrep.org* web service.
6. The web service verifies that all tests pass.
7. $\mathcal{C}_1 \ldots \mathcal{C}_n$ get notified about the merge request and express their opinion about accepting or rejecting the request and optionally suggest improvements.
8. After optional review iterations, the (final) merge request is accepted or refused based on the constitutional rules. If accepted, it is part of the official version of `ackrep_data` and if $\mathcal{P} \notin \{\mathcal{C}_1, \ldots, \mathcal{C}_n\}$ then $\mathcal{P}$ becomes $\mathcal{C}_{n+1}$ and $n := n + 1$.

It is worth mentioning that the decision makers $\mathcal{C}_1 \ldots \mathcal{C}_n$, although they could, are not expected to perform a careful review of the code, which would clearly be too much effort. However, they are expected to check that the code is relevant to the scope of `ackrep_data` and does not contain obvious errors. If a problem with a contribution later arises, this could be documented e. g. via the `Comment` entity.

5 Results: Current Implementation Status

The aim of the authors is to develop a practically working concept, therefore a prototype of the proposed software `ackrep_core` and `ackrep_web` has been implemented and is publicly available as Free Software (GPLv3). However, not all of the described features are already implemented. For example, currently there is only one default `Environment`, which only supports Python code. Furthermore, the *code executor* component in Fig. 3 is not yet realized as a separate container as it should be for performance and security reasons but instead as a plain process run by the web service. Nevertheless, the current state of the code and the running instance at *testing.ackrep.org* (see Fig. 4) should be sufficient to illustrate the proposed concept to a certain degree and to generate some feedback.

6 Discussion

It seems probable to the authors that the proposed approach raises questions and maybe skepticism. In this section we try to anticipate and answer some of them.

What is the added value to related publications? As stated in the introduction, earlier publications ceased either with a description (and lamentation) of reproduction hurdles or with an appeal to publish source code and to connect it with CI services. While the latter doubtlessly fosters reproducibility, this appeal is based on mere

[4] The main code will be a `MethodPackage` while `ProblemSpecification` and `ProblemSolution` can easily be created based on existing examples

[5] Often also called *pull request*, although *merge request* better reflects the essence of the concept: the request to merge new content into the existing repository.

Preprint: Only for personal use.

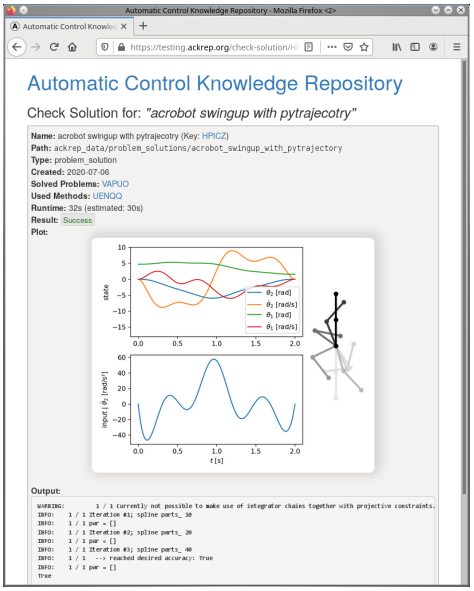

Figure 4: Screenshot of `ackrep_web`. Currently displayed is the result page of the `check solution` command for the acrobot swingup example. The relevant metadata is shown along with the calculation result.

comprehension of the underlying problem. Given typical resource constraints and incentive structures in the research system, even strong intention for reproducibility might be outweighed by the necessary *additional* effort to transform the code into a "presentable" state and to deploy CI. The proposed approach of the *active knowledge repository*, however, might change this balance in favor of code publication due to the following expected effects:

1. Basing own work on existing examples reduces necessary effort and helps adjust the own ambition to coding style and quality.

2. Linking to the own code with an `https://ackrep.org/<key>`-URL proves that published code works on independent infrastructure ($\hat{=}$ formal quality assurance), which might provide some extra incentive.

3. Making the own results easily findable on a platform dedicated to the own research field increases peer-related visibility much more than just publishing code among millions of unrelated projects on a public repository host (which of course does not conflict with the proposed approach). Increased visibility is expected to be a strong extra incentive.

How does an ACKRep-publication relate to "classical publications"? Publishing source code and metadata is supposed to be a *supplement to classical publications* such as scientific reports, conference or journal papers. However, both publication types would benefit from linking each other: the paper/pdf publication becomes much easier to reproduce with linked (working) source code and the published source code becomes easier to comprehend with the background information from a peer reviewed scientific paper. Indirectly, the proposed approach might also increase the content quality of final publications because the respective peer reviewers will be the first beneficiaries of low-barrier reproducibility and thus are able to assess the submission more precisely and, if necessary, point out weaknesses and possible improvements.

Is the proposed *central* platform not a single point of failure or otherwise problematic? Experience shows that a limited (and often short) lifetime is attached to almost all human creations. This holds even more for digital creations. As the proposed concept does not create hard dependencies it can be considered as expandable as useful[6]: Every contributor is free (and encouraged to) publish the own contributions elsewhere (cf. public repository host in Fig. 3). With the features of distributed version control this redundancy is easily manageable and by default every contributor has at least one local and one public copy of the *whole data* under control. Furthermore, the constitutional rules (see Sec. 4.2) of the platform should prevent that centralization implies concentration of influence and the availability of all the data and all the code under a Free Software License gives anybody the low-effort possibility to clone/fork the whole project – which in turn acts as a feedback in favor of finding compromises.

The most sensitive part would be the URLs hard-coded in publications, which in the worst case might become unreachable. But on the other hand, this could happen with any other web service as well, and those are typically not as simple to replace, do not grant open access to all their content and do not offer easy identification of contributions with unique entity keys.

Does the proposed solution have scalability issues? In this paper we suggest to establish *one* repository and a central web service (currently running on a single virtual machine). This results in two obvious limitations when many contributions are made: a) The repository might become quite big, which complicates further contributions and other interactions and b) the com-

[6] Even in the worst case scenario of a sudden irreversible shutdown after a period of successful operation this would only destroy (some of) the benefits it added before but not cause additional problems.

Preprint: Only for personal use.

putational resources of the current web server will not suffice if confronted with several requests in parallel. However, many other platforms and projects show that such problems are solvable. This contribution only aims to propose the basic concept to raise feedback. If scalability becomes an problem, then there necessarily is a group of several contributors who can then also provide the resources to solve that problem.

How can the correctness of the content be ensured in the long term? The proposed solution offers two layers of validation: First, an automatic formal verification, i. e. a test whether submitted code runs without errors and produces the asserted results. This is required before inclusion into the repository. Additionally, content-related flaws such as implementation bugs or conceptual errors can only be detected by domain experts – as is the case for classical publications. However, the `Comment` entity (cf. Fig. 2), which by definition is associated with specific other entities, offers an accountable and traceable communication channel that can be used to precisely identify and document issues. In contrast, for classical publications there is no predefined mechanism to inform the audience about errors or caveats.

7 Conclusion

In this contribution, we proposed a concept to increase the reproducibility of published results in system and control theory. To this end, we first summarized the reproducibility problem, which affects our research discipline as many others. After that we gave a brief overview on relevant existing technologies and how to combine them. The core of the contribution is the description of the repository structure, the overall architecture and the process scheme for the basic use case (along with the constitutional rule set). To facilitate the experimental validation of the proposed concept, a prototype implementation was briefly introduced. Finally, some expected concerns were discussed. The authors want to close with the normative claim that reproducibility should be a key feature of scientific results. Published results should be as easy as possible to reproduce by interested individuals or groups, e. g. peer researchers, graduate students or business and industry. The proposed approach of the active knowledge repository and much more its current implementation status is not expected to be the optimal solution or even close to that. But nevertheless it might be a relevant step in the ongoing optimization process.

References

[1] J. F. Claerbout and M. Karrenbach. “Electronic Documents Give Reproducible Research a New Meaning”. In: *SEG Technical Program Expanded Abstracts 1992*. 0 vols. SEG Technical Program Expanded Abstracts. Society of Exploration Geophysicists, 1992, pp. 601–604.

[2] D. L. Donoho et al. “Reproducible Research in Computational Harmonic Analysis”. In: *Computing in Science Engineering* 11.1 (2009), pp. 8–18.

[3] R. R. Downs et al. “Community Recommendations for Sustainable Scientific Software”. In: *Journal of Open Research Software* 3.1 (1 2015), e11.

[4] “Devil in the Details”. In: *Nature* 470.7334 (7334 2011), pp. 305–306.

[5] Z. Merali. “Computational Science: ...Error”. In: *Nature* 467.7317 (2010), pp. 775–777.

[6] A. Morin et al. “Shining Light into Black Boxes”. In: *Science* 336.6078 (2012), pp. 159–160.

[7] N. P. Rougier et al. “Sustainable Computational Science: The ReScience Initiative”. In: *PeerJ Computer Science* 3 (2017), e142.

[8] J. C. Molloy. “The Open Knowledge Foundation: Open Data Means Better Science”. In: *PLOS Biology* 9.12 (2011), e1001195.

[9] O. J. Reichman, M. B. Jones, and M. P. Schildhauer. “Challenges and Opportunities of Open Data in Ecology”. In: *Science* 331.6018 (2011), pp. 703–705.

[10] V. Gewin. “Data Sharing: An Open Mind on Open Data”. In: *Nature* 529.7584 (7584 2016), pp. 117–119.

[11] G. Boulton et al. “Science as a Public Enterprise: The Case for Open Data”. In: *The Lancet* 377.9778 (2011), pp. 1633–1635.

[12] B. Hanson, A. Sugden, and B. Alberts. “Making Data Maximally Available”. In: *Science* 331.6018 (2011), pp. 649–649.

[13] V. Stodden. “Trust Your Science?: Open You Data and Code”. In: *AMSTAT news: the membership magazine of the American Statistical Association* 409 (2011), pp. 21–22.

[14] D. C. Ince, L. Hatton, and J. Graham-Cumming. “The Case for Open Computer Programs”. In: *Nature* 482.7386 (2012), pp. 485–488.

Preprint: Only for personal use.

[15] D. Bailey, J. Borwein, and V. Stodden. "Set the Default to "Open"". In: *Notices of the Ams, Accepted March* (2013), p. 2013.

[16] S. M. Easterbrook. "Open Code for Open Science?" In: *Nature Geoscience* 7.11 (11 2014), pp. 779–781.

[17] L. A. Barba. "Praxis of reproducible computational science". In: *Computing in Science & Engineering* 21.1 (2019), pp. 73–78.

[18] F. Chirigati et al. "ReproZip: Computational Reproducibility With Ease". In: *Proceedings of the 2016 International Conference on Management of Data*. SIGMOD '16. San Francisco, California, USA: Association for Computing Machinery, 2016, pp. 2085–2088.

[19] T. Crick, B. Hall, and S. Ishtiaq. "Reproducibility in Research: Systems, Infrastructure, Culture". In: *Journal of Open Research Software* 5.1 (1 2017), p. 32.

[20] M. Krafczyk et al. "Scientific Tests and Continuous Integration Strategies to Enhance Reproducibility in the Scientific Software Context". In: *Proceedings of the 2nd International Workshop on Practical Reproducible Evaluation of Computer Systems*. P-RECS '19. Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 23–28.

[21] *ACKRep Testing Instance*. URL: `http://testing.ackrep.org/`.

[22] R. C. Merkle. "Method of Providing Digital Signatures". U.S. pat. 4309569A. Univ Leland Stanford Junior. 1982.

[23] G. Fraser and J. M. Rojas. "Software Testing". In: *Handbook of Software Engineering*. Ed. by S. Cha, R. N. Taylor, and K. Kang. Cham: Springer International Publishing, 2019.

[24] C. M. Keet. *An Introduction to Ontology Engineering*. Cape Town: College Publications, 2018.

[25] P. Borst, H. Akkermans, and J. Top. "Engineering Ontologies". In: *International Journal of Human-Computer Studies* 46.2 (1997), pp. 365–406.

[26] T. Tudorache. "Employing Ontologies for an Improved Development Process in Collaborative Engineering". 2006.

[27] S. Fathalla, S. Auer, and C. Lange. "Towards the semantic formalization of science". In: *Proceedings of the 35th Annual ACM Symposium on Applied Computing*. 2020, pp. 2057–2059.

[28] C. Knoll and R. Heedt. *ACKRep Data Repository on GitHub*. URL: `https://github.com/cknoll/ackrep_data`.

Preprint: Only for personal use.