Automatically Assessing Complexity of Contributions to Git Repositories

Rolf-Helge Pfeiffer[1]

IT University of Copenhagen, Rued Langgaards Vej 7, Copenhagen, Denmark
ropf@itu.dk

Abstract. Lehman's second law of software evolution suggests that under certain conditions software *"becomes more difficult to evolve"*. Similarly, Technical Debt (TD) is often considered as technical compromises that render future changes of software more costly. But how does one actually assess if modifying software becomes more difficult or costly? So far research studied this question indirectly by assessing internal structural complexity of successive software versions arguing that increasing internal complexity renders evolution tasks more difficult and costly too. Our goal is to assess complexity of evolution tasks *directly*. Therefore, we present an algorithm and tool that allows to automatically assess Contribution Complexity (CC), which is the complexity of a contribution respecting difficulty of integration work. Our initial evaluation suggests that our proposed algorithm and readily available tool are suitable to automatically assess complexity of contributions to software in `Git` repositories and the results of applying it on 8 686 contributions to two open-source systems indicate that evolution tasks actually become slightly more difficult.

1 Introduction

Software is usually evolving to adapt to changing environments or requirements, to correct errors, to address *"problems ... that are not carried out adequately during ... development"* [18], etc. Software is said to become increasingly difficult to evolve over time unless it is continuously refactored to decrease internal complexity. For example, Lehman's second law of software evolution says: *"As [software] is changed its complexity increases and [it] becomes more difficult to evolve unless work is done to maintain or reduce the complexity."* [14] Similarly, the term *technical debt* is used to describe software constructs that render future contributions increasingly complex. For example, the participants of Dagstuhl Seminar 16162 agreed to define TD as: *"a collection of design or implementation constructs that are expedient in the short term, but set up a technical context that can make future changes more costly or impossible."* [3].

But how does one actually assess if software *"becomes more difficult to evolve"* or if *"future changes [are] more costly"*? Either human developers possess a skill allowing them to assess if modifying existing software becomes more difficult,

managers unearth such knowledge from business/process data, or such assessments are conducted indirectly by assessing complexity of successive versions of software. For example, researchers apply various complexity metrics, such as, size in LOC, cyclomatic (McCabe) complexity [16], coupling of functions or modules [12] to assess change of internal complexity of software over time and argue by induction that more complex systems are more complex to evolve. Using these metrics researchers invalidate Lehman's second law, see [10] for an overview or, e.g., [2].

To the best of our knowledge, there is no research or tool available that allows to directly and automatically assesses if the actual *work of evolving* a software system is getting more difficult/complex. In this paper we present an algorithm that allows to directly asses if evolution work becomes more complex and thereby more costly. Our Contribution Complexity (CC) algorithm computes a score that indicates mainly how difficult it is to integrate work (a set of commits) into an existing software system. The CC score is computed on basic size-based and entropy-based metrics on commit and file level, i.e., number of modified lines/files, degree of scattered work across files/methods, etc. Together with this paper, we publish a tool (`https://pypi.org/project/contribution-complexity/`) that implements the proposed algorithm. The tool can be used by practitioners to enhance CI/CD chains and by researchers to study software evolution and TD.

The contributions of this paper are *a)* presentation of an algorithm to automatically assess CC of contributions to `Git` repositories (Sec. 3), *b)* implementation of that algorithm in a readily installable open-source tool, *c)* initial evaluation of the CC demonstrating its suitability for the task (Sec. 4), and *d)* together with initial results of applying CC to two open-source database systems (Sec. 4.2), we provide a corresponding dataset containing CC scores together with the tool.

2 Background, Terminology, and Motivation

In this section we explain the terminology that we use in the remainder of the paper and motivate our CC score. In this and the following sections we refer to examples from development of the graph database `Gaffer`, which is created mainly by the British Government Communications Headquarters (GCHQ). More details about `Gaffer` and why it appears in this paper follow in Sec. 4.

Terminology: Work in software projects is often organized via issue trackers, e.g., Atlassian's `Jira`(`https://www.atlassian.com/software/jira`) or Github's integrated issue tracker, and work on files is handled via VCS like `Git`.

Tickets in issue trackers describe work, such as, perfective or adaptive maintenance task, new features, etc. Tickets may be resolved without any *contribution* to the developed software. For instance, unwanted features or not reproducible bugs are marked accordingly and respective tickets are closed without modification of the software. Other tickets get resolved by implementing a required change via one or more commits to a VCS repository. Commonly, a commit refers to a corresponding ticket via a ticket identifier in the commit message,

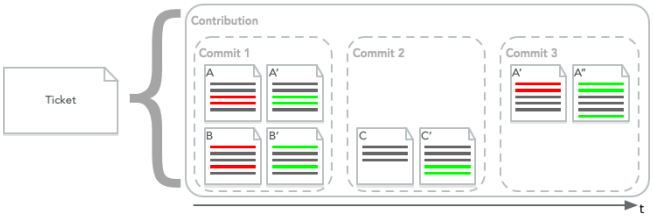

Fig. 1. A contribution resolving a ticket. The contribution consists of three commits with in total four modifications (file changes) over time.

i.e., multiple commits can refer to one ticket. To present a clean development history(`https://git-scm.com/book/en/v2/Git-Tools-Rewriting-History`) multiple commits are sometimes *squashed* into a single commit when merging branches. Consequently, there exists a one to many relation between tickets and commits.

Each commit consists of zero or more file *modifications* where conflict-free merges contain zero modifications. The term *modification* is synonymous to a change of a file, see Spadini et al. [20]. *Modifications* can be considered as edit deltas even though `Git` stores commits as snapshots of entire files. Most `Git` tools present *modifications* as deltas or patches too, see e.g., commit `ee3e2a` in the `Gaffer` repository on Github. Modifications carry information about the kind of change that was applied. For example, `Git` records if a file is *added*, *deleted*, *modified*, *copied*, or *renamed*.

In this paper, we call one or more commits to a `Git` repository, which consist of one or more *modifications* a *contribution*. Contributions, contain the *work* that eventually resolves tickets. Fig. 1 shows a conceptual illustration of *tickets*, *contributions*, *commits*, and *modifications*, where two file *modifications* (`A to A´` and `B to B´`) form commit 1, commits 2 and 3 are formed by one file *modification* respectively, and the three commits form a *contribution* that resolves a *ticket*.

Complexity: This work is based on two conceptions of *complexity*. Basili [4], describes *complexity* as the difficulty a developer faces when performing tasks like coding, debugging, or modifying software. Clearly, different kinds of work on existing software are differently complex. For example, implementing a new feature in an object-oriented system via inheritance and conformance to interface specifications is less complex than implementing a feature for which existing abstractions have to be refactored or a patch has to be woven into existing classes and methods. This varying cognitive complexity of tasks is described by Dörner via the *"...existence of many independent variables in a given system. The more variables and the greater their interdependence, the greater that system's complexity. Great complexity places high demands on a planner's capacities to gather information, integrate findings, and design effective actions."* [6] The exemplary extension of an object-oriented system via inheritance deals with a low amount of independent variables (conformance to class interfaces and interface specifications provide a low number of integration points) compared to a higher

amount of strongly interdependent variables when integrating scattered changes during refactoring of existing abstractions.

We believe that besides the inherent complexity of a contribution (conceptual difficulty of realizing it), the complexity of *integration work* (scattered changes to integrate a solution into an environment) characterizes to a large degree the complexity of a contribution. We call the *complexity* of a *contribution* respecting the work of integrating it into existing software Contribution Complexity (CC).
Motivational example: Before formally defining CC in the next section, we illustrate it on three examples from the `Gaffer` project:

Ticket `gh-1808` describes a bug which prevents release of a package to the Maven Central package store.
Ticket `gh-2228` specifies that copyright headers in all code files need to be updated to point to the correct time range.
Ticket `gh-190` asks to refactor `Gaffer`'s storage engine to be better encapsulated and more descriptive.

`Gaffer`'s `Git` repository contains a single commit (`31e23a`) that refers to ticket `gh-1808`, another commit (`ee3e2a`) that refers to ticket `gh-2228`, and 21 commits that refer to ticket `gh-190`. Of the 21 commits, 17 contain modifications and 4 are empty merge commits. Due to constrained space, we refer to the online representations of the respective contributions (`https://github.com/gchq/Gaffer/commits`). The bug fix that resolves ticket `gh-1808` excludes a conflicting dependency from a file with project meta-information (`pom.xml`). The contribution is of low complexity since it consists only of six contiguous lines, which are added in a single commit to a single file. More complex than this minuscule contribution is the update of all copyright headers (ticket `gh-2228` with commit `ee3e2a`). Even though conceptually only four digits need to change, they are changed across 1 975 code files replacing 1 977 lines with a new line. Note, `Git` operates with lines as smallest unit of change. Even changing one character of a line in a file, first deletes that entire line and subsequently adds its new version.

Certainly the most complex contribution of the three examples, is the refactoring of `Gaffer`'s storage engine (ticket `gh-190`). Over multiple commits multiple hundreds of lines in dozens of files are modified and the changes are scattered within files and across methods, see e.g., commit `2874da`.

These three examples shall illustrate that traditional complexity metrics, such as, size of change in LOC/number of files or change of McCabe complexity alone are not suitable to assess CC. For example, size-wise the largest contribution (modifying 1 977 lines in 1 975 files) updates the copyright headers, see commit `ee3e2a`. Contrary, only some dozens of files with some hundreds of lines are modified to refactor the storage engine. However, these changes are scattered within files and across methods (high entropy). When considering *complexity* as the difficulty of performing tasks like coding or software modification [4], then only the high entropy of the modifications in refactoring of the storage engine suggests higher complexity than the size-wise bigger copyright header update. Furthermore, complexity measures like McCabe complexity would not yield insightful results for the three examples. Either it is not applicable to relevant

artifacts (McCabe complexity of a `pom.xml` file?), not all relevant changes are analyzed by it (update of copyright headers), or it 'overlooks' complexity caused by distributed nature of changes.

To overcome these restrictions of traditional complexity metrics when assessing complexity of contributions, we develop an algorithm that should mimic human intuitions as presented above. We implement it in a tool and we call both the algorithm in the next section and the tool uniformly Contribution Complexity (CC).

3 Computing Contribution Complexity

In this section we describe how to compute a discrete Contribution Complexity (CC) score ($c_{contrib}$) for a set of commits using basic metrics on *modification*, *commit*, and *contribution* level. A priori, we decided that a CC score should map a contribution to the discrete values *low*, *moderate*, *medium*, *elevated*, or *high*. To facilitate presentation, we use the following notation: $\mathbb{C}$ denotes the set of all commits of a contribution and $\mathbb{M}$ denotes the set of all modifications of all commits of a contribution. The CC score is computed in two stages. First, a set of metrics is applied to modifications, whose results are aggregated and subsequently merged with the results of metrics computed for commits. Our presentation follows these two stages.

Per modification ($m \in \mathbb{M}$), i.e., per modified file in a commit, the following basic metrics are computed:

Number of lines added ($m_{l+}(mod)$) The total number of lines added to the file in this modification.

Number of lines removed ($m_{l-}(mod)$) The total number of lines removed from the file in this modification.

Number of hunks ($m_h(mod)$) The total number of blocks that are modified contiguously. For example, `A´´` in Fig. 1 contains two hunks and `C´` contains one hunk. The number of hunks indicates how scattered a change is and thereby how difficult is it to integrate it into the file.

Number of modified methods ($m_{mth}(mod)$) In case a modified file contains programming language source code, the number of modified methods (or functions) is counted. For non-programming language artifacts the metric evaluates to zero. Similar to $m_h(mod)$, the rationale is that work with changes scattered over multiple methods is more difficult. Note, since our tool depends internally on `pydriller`(`https://pydriller.readthedocs.io/`) which uses the `lizard`(`https://github.com/terryyin/lizard`) tool to parse source code, the number of methods can only be non-zero for the 16 languages that are currently supported by `lizard`.

Modification kind ($m_{mk}(mod)$) This metric returns the kind of file modification in `Git` terms, i.e., one of the values *added*, *deleted*, *modified*, *copied*, or *renamed*.

Before the final CC score is computed, a separate *modification complexity* score (c_{mod}) is computed for each modification (mod) separately. It is defined as the arithmetic mean of the lines added complexity (c_{l+}), the lines removed complexity (c_{l-}), the hunk complexity (c_h), and the method complexity (c_{mth}), see Eq. 1. In case a file is *deleted* or *copied* in a commit, its *modification complexity* is low, since there is no 'real' work behind all the removed or newly added lines.

$$c_{mod} = \begin{cases} low & \text{if } m_{mk}(mod) = deleted|added \\ \frac{1}{4} \times (c_{l+}(m_{l+}(mod)) + c_{l-}(m_{l-}(mod)) + c_h(m_h(mod)) + c_{mth}(m_{mth}(mod))) & \text{if } otherwise \end{cases} \tag{1}$$

The lines added and the lines removed complexity (c_{l+} and c_{l-} respectively) are computed via the same model as in Eq. 2, i.e., $c_{l+}(l) = c_{l-}(l)$, and only the former is presented here. The hunk and the method modified complexity (c_h and c_{mth}) are both computed via the mapping in Eq. 3, i.e., $c_h(n) = c_{mth}(n)$, and only $c_h(n)$ is presented here. Note, that we assume that the complexity values low to $high$ are equivalent to the numerical values 1 to 5, so that we can use them in calculations.

$$c_{l+}(l) = \begin{cases} low & \text{if } 0 \leq l \leq 15 \\ moderate & \text{if } 15 < l \leq 30 \\ medium & \text{if } 30 < l \leq 60 \\ elevated & \text{if } 60 < l \leq 90 \\ high & \text{if } l > 90 \end{cases} \tag{2}$$

$$c_h(n) = \begin{cases} low & \text{if } 0 \leq n \leq 2 \\ moderate & \text{if } 2 < n \leq 5 \\ medium & \text{if } 5 < n \leq 7 \\ elevated & \text{if } 7 < n \leq 9 \\ high & \text{if } n > 9 \end{cases} \tag{3}$$

Rationale for the modification complexity models: The line modification complexity models (c_{l+} and c_{l-}) are adapted from Visser et al. [22], where the authors argue that maintainable methods shall contain less than 15 LOC and higher values render a method progressively more complex. We re-use their thresholds only extending them by a fifth level for *high* complexity. The hunk and method complexity model (c_h and c_{mth}) are based on Miller [17] who argues that human short-term memory usually deals well with is 7 ± 2 entities. Visser et al. use similar thresholds [22], e.g., for assessing complexity of method signatures.

All modification complexities c_{mod} contribute to the CC score ($c_{contrib}$) not individually but in aggregated form. First, the frequencies of all modification complexity values are collected into a set of pairs ($\mathbb{K} = hist\left([c_{mod}(m) : m \in \mathbb{M}]\right)$). For example, $\mathbb{K} = \{(low, 12), (moderate, 14), (medium, 5), (elevated, 3), (high, 0)\}$ would mean that a contribution consists of 12 modifications with low complexity, 14 with *moderate* complexity, etc. The modification complexity frequencies are aggregated into a single value $c_{\forall mod}$ as a weighed average of all the frequency pairs $c_{\forall mod} = \frac{1}{5} \times \sum_{(i,j) \in \mathbb{K}} i^i \times j$. The exponential weights (i^i) are inspired by using Fibonacci numbers for time estimation [21]. We use exponentials to express that it is way harder to work on high complexity modifications than low complexity modifications.

To compute the overall CC score, the following metrics are computed over all commits ($c \in \mathbb{C}$), i.e., for the entire contribution:

Number of modified files in commit ($m_{|f|}(c)$) The total number of files that were either *added*, *deleted*, *modified*, *copied*, or *renamed* in a commit.

Number of lines in commit ($m_{|l|}$) The sum of all added and removed lines in all modifications of a commit.

The CC score ($c_{contrib}$) is defined as the arithmetic mean of the modified files complexity ($c_{|f|}$), changed lines per file complexity ($c_{l/f}$), modification kind complexity (c_{mk}), and the overall modification complexity ($c_{\forall m}$), see Eq. 4. There, n_{files} is the total number of modified files in all commits ($n_{files} = \sum_{n_f \in \{m_{|f|}(c):c \in \mathbb{C}\}} n_f$), n_{lines} is the total number of modified lines in all commits ($n_{lines} = \sum_{n_l \in \{m_{|l|}(c):c \in \mathbb{C}\}} n_l$), and the cardinality of all work kinds ($n_{mk} = |\{m_{mk}(m) : m \in \mathbb{M}\}|$) encodes the variety of work in a contribution. n_{mk} ranges from 1 to 5 denoting, e.g., if files were only added or only modified ($n_{mk} = 1$) or if files were added, deleted, renamed, copied, and modified ($n_{mk} = 5$).

$$c_{contrib} = \tfrac{1}{4} \times c_{|f|}(n_{files}) + c_{l/f}\left(\tfrac{n_{lines}}{n_{files}}\right) + c_{mk}(n_{mk}) + c_{\forall m}(c_{\forall mod}) \tag{4}$$

The modified files complexity model ($c_{|f|}$) and the changed lines per file complexity ($c_{l/f}$) use the same thresholds as the line modification complexity models c_{l+} and c_{l-}, see Eq. 2, and are therefore omitted here. The modification kind complexity (c_{mk}), and the overall modification complexity ($c_{\forall m}$) are computed via the mappings in Eq. 5 and Eq. 6 below.

$$c_{mk}(n) = \begin{cases} low & \text{if } n = 1 \\ moderate & \text{if } n = 2 \\ medium & \text{if } n = 3 \\ elevated & \text{if } n = 4 \\ high & \text{if } n = 5 \end{cases} \tag{5}$$

$$c_{\forall m}(n) = \begin{cases} low & \text{if } 0 \leq n \leq 195 \\ moderate & \text{if } 195 < n \leq 390 \\ medium & \text{if } 390 < n \leq 781 \\ elevated & \text{if } 781 < n \leq 1562 \\ high & \text{if } n > 1562 \end{cases} \tag{6}$$

Computation example: For brevity, we illustrate calculation of CC on a small contribution to `Gaffer`. Consider ticket `gh-2304`, which reports a bug on lost status information when certain exceptions are caught and re-raised. The contribution resolving the issue consists of a single commit (`291111`) that modifies two files, i.e., $n_{files} = 2$. In both files, in total 29 lines are modified ($n_{lines} = 29$). In the first file 5 lines of a method are replaced by two new lines in one hunk, i.e., $m_{l+}(mod) = 2$, $m_{l-}(mod) = 5$, $m_h(mod) = 1$, and $m_{mth}(mod) = 1$. In the second file 20 lines of a method, and two new import lines are newly added over three hunks, i.e., $m_{l+}(mod) = 22$, $m_{l-}(mod) = 0$, $m_h(mod) = 3$, and $m_{mth}(mod) = 1$. That is, we have one modification (c_{mod_1}) of *low* and another one (c_{mod_2}) of *moderate* modification complexity, see below.

$$\begin{aligned} c_{mod\_1} &= \frac{c_{l+}(2) + c_{l-}(5)) + c_h(1) + c_{mth}(1)}{4} = \frac{1+1+1+1}{4} = \frac{4}{4} = 1 = low \\ c_{mod\_2} &= \frac{c_{l+}(22) + c_{l-}(0)) + c_h(3) + c_{mth}(1)}{4} = \frac{2+1+2+1}{4} = \frac{6}{4} = 1.5 = moderate \\ \mathbb{K} &= [(low, 1), (moderate, 1), (medium, 0), (elevated, 0), (high, 0)] \\ c_{\forall mod} &= \frac{\sum_{(i,j) \in \mathbb{K}} i^i \times j}{5} = \frac{1^1 \times 1 + 2^2 \times 1 + 3^3 \times 0 + 4^4 \times 0 + 5^5 \times 0}{5} = \frac{5}{5} = 1 = low \end{aligned} \tag{7}$$

Since both files exist before the contribution the only modification kind is *modified*, i.e., $n_{mk} = 1$. That leads to a contribution of *low* CC via the final $c_{contrib}$ formula.

$$c_{contrib} = \frac{c_{|f|}(2) + c_{l/f}\left(\frac{29}{2}\right) + c_{mk}(1) + c_{\forall m}(1)}{4} = \frac{4}{4} = \underline{\underline{\textbf{low}}} \tag{8}$$

4 Evaluation

In this section, we evaluate to which degree the CC scores that our tool computes are aligned with human assessment of complexity of selected contributions and we provide results of an initial experiment of distribution of CC scores of all contributions to two open-source case systems.

Case Systems: To evaluate our CC score and for initial experimentation, we need software projects as cases that have publicly available Git repositories and issue trackers so that we can compute CC scores of actual contributions. Furthermore, these cases should be of a certain size and age so that contributions of various complexities exist. Due to our work for the Research Center for Government IT at IT University of Copenhagen, we are interested in studying software that is developed and deployed at public agencies. To identify possible case systems, we manually search Github's list of public agencies that use the platform for development(`https://government.github.com/community`). There, we identify `Gaffer` (`https://gchq.github.io/gaffer-doc`) as a suitable case. It is a graph database, that is created mainly by the British signals intelligence agency GCHQ. A first version of it was open-sourced in 2015 (with 125 releases since then), and the project's issue tracker is available on Github(`https://github.com/gchq/Gaffer/issues`). Since we cannot identify a software project from the same domain on the mentioned list, we choose Apache `Cassandra` (`https://cassandra.apache.org`) as a second case. It is an open-source, distributed, wide-column store, NoSQL DBMS that was originally developed by Facebook [13]. It was open-sourced in 2008 (with 265 releases since then). The project uses Jira(`https://issues.apache.org/jira/projects/CASSANDRA`) as issue tracker. Both systems are written mainly in Java, are licensed under Apache License 2.0, and their sources are available as `Git` repositories (`https://github.com/gchq/Gaffer`, `https://github.com/apache/cassandra`.

Even though, `Gaffer` (version 1.9.1) and `Cassandra` (version 3.9) consist of approximately the same amount of files (2 294 and 2 316 respectively), `Cassandra` is circa twice as large as `Gaffer` (588 017 lines with 424 733 LOC versus 291 071 lines with 199 816 LOC). Statistics are generated with the Succinct Code Counter tool (version 2.13.0) `https://github.com/boyter/scc`.

Dataset Creation: With two Python scripts, we export all tickets, ticket identification keys, ticket resolution dates, etc., from the respective issue trackers. Using our tool, we compute a mapping from ticket identifiers to commits. The mapping is created by matching ticket identifiers via regular expressions in commit messages. For example, the regular expression for `Gaffer` is `(Gh |gh-)<issue_key>( |$)` and for `Cassandra` it is `CASSANDRA-<issue_key>( |$)`,

where `<issue_key>` is an integer in both cases. We identify these regular expressions by brief manual inspection of the commit histories, and via the contribution guidelines of the respective project. The resulting dataset with all tickets and corresponding contributions contains 2 403 tickets for `Gaffer`, of which 2 300 are resolved and 820 of these are resolved with contribution, i.e., with at least one commit attached to the respective ticket. For `Cassandra`, the dataset includes 16 485 tickets, of which 14 158 are resolved, and 7 866 are resolved with contribution. We let our CC tool compute the a CC score for each resolved ticket with contribution from both projects. The resulting datasets are stored as CSV files and are available online (`https://raw.githubusercontent.com/HelgeCPH/contribution-complexity/master/data/cassandra_contrib_compl.csv`, `https://github.com/HelgeCPH/contribution-complexity/blob/master/data/gaffer_contrib_compl.csv`).

For the manual evaluation, see Sec. 4.1, we sample 25 contributions (five from each possible CC score) from `Cassandra` and 23 for `Gaffer` (five from each CC score except for *high*, where there are only three contributions). The author of this paper manually classifies the CC scores for each of these 48 contributions, that consist in total of 247 commits. Thereafter, we compare our manually classified CC scores with those created by the tool. The protocols, for this step are available online too (`https://github.com/HelgeCPH/contribution-complexity/blob/master/data/cas_evaluation_tab.md`, `https://github.com/HelgeCPH/contribution-complexity/blob/master/data/gaf_evaluation_tab.md`).

Note, the entire experiment setup with dataset creation and data reproduction is automatically reproducible via a Shell script in the `experiment` directory in the CC tool's repository. We provide a replicable environment specification for a virtual machine on DigitalOcean via a Vagrant file.

4.1 Manual Evaluation Results

In 36 cases (75%) our manual classification is equal to the tool's CC score. For the remaining 25% of the cases, most often (8 cases $\approx$ 17%) our assessment is one level higher than the score computed by the tool and in 4 cases ($\approx$ 8%) the tool's assessment is one level higher than our classification. Discrepancies between our classification and tool's score is actually most frequent around the two scores *moderate* and *medium*. In six cases (12.5%) we assigned a *moderate* CC and the tool a *medium* score or vice versa. The second most frequent discrepancy is between *medium* and *elevated* scores (in 3 cases $\approx$ 6%).

That is, our classification differs from the tool's assessment always only by one level and in most cases on those centered around *medium* contribution complexity. Our experience during manual classification, was also that we found it hardest to distinguish complexities on closely related levels. That is, we found coarse-grained assessment into three levels (*low*, *medium*, and *high*) more easy than deciding between more fine-grained five levels of CC.

Threats to Validity: The manual classification of the CC score would likely have been more accurate if performed by developers from the `Gaffer` and `Cassandra` projects. Since we do neither know the design or architecture of the systems our assessments are prone to be too high. We try to mitigate this risk by carefully

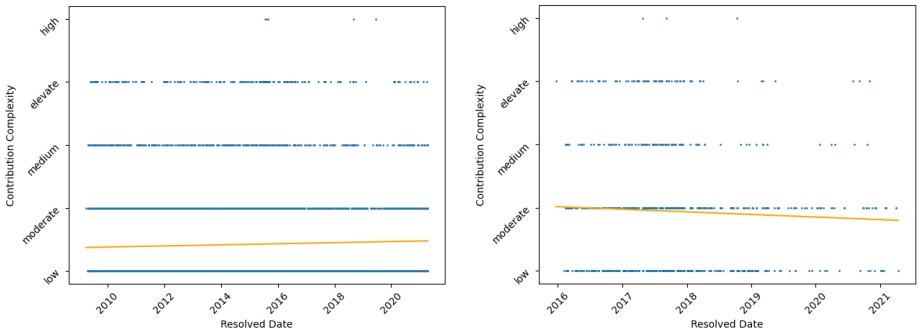

Fig. 2. Development of complexities (CC) of all contributions of Apache `Cassandra` (left) and `Gaffer` (right), with linear regression model (yellow line).

examining each contribution and the corresponding modifications. Our classification might be biased since the author performing it also developed and implemented the CC score. We tried to minimize this risk by running the manual classification first 10 days after the last modification to the CC score, which should be sufficient with regards to memory retention [15].

Here, we evaluate if CC scores correspond to human assessment of complexity of contributions. We do not evaluate to which degree the CC metric assesses the actual complexity of contributions. Even though not a software quality metric, a thorough and more rigid evaluation of the CC metric along the lines of the validation criteria of IEEE 1061 Standard for a Software Quality Metrics Methodology [1], should complement the provided initial evaluation in future.

4.2 Contribution Complexities of Two Open-source Systems

Fig. 2 illustrates the distribution of CC scores over time. Each blue dot is a resolved ticket with contribution (ticket closing times on x-axis). Obviously, the different complexity levels are not equally distributed. Tab. 1 shows the frequencies of the various complexity levels per system. For `Cassandra`, the amount of contributions with a *low* CC is highest. 67.0% of all contributions possess that complexity and frequencies decrease for higher CC scores. Only five contributions (0.1%) are of *high* CC. For `Gaffer` *moderately* complex contributions are most frequent (44.3%) followed by *low* complexity contributions (35.5%) and from *medium* to *high* frequencies decrease, though with higher ratios compared to `Cassandra`.

The yellow lines in Fig. 2 are linear regression models that should provide an impression of development of CC over time, i.e., if software *"becomes more difficult to evolve"*. For `Cassandra` it suggests that contributions become more complex, whereas for `Gaffer` it suggest the opposite. To accommodate for the impact of location of the contributions with *high* CC on the regression models, we also compare the first full year of contributions with the last full year of

	low	moderate	medium	elevated	high
`Gaffer`	291 35.5%	363 44.3%	91 11.1%	72 8.8%	3 0.4%
`Cassandra`	5 273 67.0%	2 055 26.1%	363 4.6%	170 2.2%	5 0.1%

Table 1. Absolute and relative (rounded) frequencies of contribution complexity scores.

contributions per system. In 2016, `Gaffer` has 194 contributions with avg. CC 1.91 (*std* 0.91) and median CC *moderate* versus 37 contributions with avg. CC 2.08 (*std* 0.83) and median CC *moderate* in 2020. In 2010, `Cassandra` has 606 contributions with average CC 1.51 (*std* 0.74) and median CC *low* versus 461 contributions with avg. CC 1.59 (*std* 0.73) and median CC *low* in 2020. These numbers suggest, that there is actually a slight increase in CC for less work in both systems.

5 Related Work & Discussion

There exists a plethora of software complexity metrics, see e.g., Zuse's overview [23] over many of them. Usually, these assess internal structural complexity of *programs*, e.g., how many branch points there are [16], how difficult programs are to understand [19], how well structured they are [12], their size [9], etc. Based on these metrics, higher level models are developed, such as, the SIG Maintainability Model (SIG-MM) [8], which combines multiple such complexity metrics to compute a maintainability score for software.

The Contribution Complexity score described in this paper is different to, e.g., the SIG-MM in that it is not constrained to analyzing complexity of source code artifacts. In essence, it is an aggregate of multiple basic size- and frequency-based metrics (number of changed lines per modification/commits, number of hunks per modifications, number of changed files per commits, etc.), which yields useful results for any textual artifacts including configuration files, build scripts, schema files, documentation, etc. The contributions to `Gaffer` and `Cassandra` that we studied (Sec. 4.1) contain modifications of such files. However, computation of CC (Sec. 3) is inspired by SIG-MM [8, 22] in that results of basic metrics are aggregated and mapped to a single score.

By counting frequencies of hunks, of changed methods per modification, and number of changed files per commit, our CC includes a measure of entropy. This is similar to Hassan's Code Change Models [7] that consider scattered changes across files to be highly entropic. Hassan equates high entropy changes with *complexity* and our work follows the same reasoning. However, Hassan operates only on the level of file changes. Our approach is more fine-grained since it includes, e.g., number of hunks and number of changed methods. Unlike Hassan, who aims to predict faults from patterns of frequently changing files, we are interested in the complexity of contributions to enable research of software evolution. Also Hindle et al. [11] compute complexity of changes, i.e., of commits instead of entire systems or modules. They compute the complexity of source code patches

(modifications in our terminology) based on indentation-levels of code, which they demonstrate to be similarly expressive as, e.g., McCabe complexity. Our work is similar to Hindle et al. in that we provide a language agnostic and simple –in the sense of underlying basic metrics– solution that operates mostly on syntactic properties of modifications. However, in case a modification's source code is in a supported programming language, our solution incorporates the number of changed methods too.

The Delta Maintainability Model (DMM) by di Biase et al. [5] assesses how much a commit in-/decreases the *maintainability* of a software system, which is based on multiple structural complexity metrics, such as, McCabe complexity, coupling, size, etc., The DMM can be considered an adaptation of SIG-MM [8] to commit level instead of system level. Our CC score is different than both whitespace-complexity [11] and DMM [5] since both of them are concerned about assessing the internal structural complexity of commits. By including entropy measures (number of hunks, changed method/files) our CC score captures complexity of integration work too. Also, Hassan and di Biase et al. study only certain kinds of changes *"Feature Introduction modifications"* and bug fixes, requests for enhancement, and improvements respectively. We consider our CC score more universal since it is applicable to any kind of change including work on documentation, tests, etc., which all are part of the initial experiment in Sec. 4.2.

Unfortunately, CC scores of different systems are currently not directly comparable since our algorithm consumes absolute numbers, see Sec. 3. The main reason for not relying on normalized values yet, is that the absolute numbers of files/methods that would serve as denominator in normalization are not fixed per contribution. They can change with every single modification. Hassan suggests to resort on the number of recently changed files as denominator instead. We consider such time-/period-based normalization future work. Another concern about our CC tool might be that the thresholds of the complexity classification models (c_*) that map input values to discrete scores (Sec. 3) appear arbitrary and do not fit across domains. To mitigate this risk, all these functions are user configurable in the tool and we present in this paper the default models. Similarly, weights of certain aggregation functions may be adapted in the tool.

Implications for practitioners: Next to this paper, we provide a readily installable open-source tool (usable as Python library and CLI tool), which can be integrated into development processes, e.g., in CI/CD chains, to automatically assess and report on development of CC of contributions to `Git` repositories. That would not only allow for more accurate assessment of which work tasks (tickets) are most difficult to work on and thereby guide potential refactorings but it would also allow to gradually adjust time and effort estimations when planning new tasks that are similar to resolved ones.

Note however, that our tool analyses local `Git` repositories only. It does not have any dependency to platforms like Github of Gitlab and can therefore not assess richer knowledge that might be present there. For example, Github tracks

related commits of remote repositories, which cannot be assessed by our tool unless explicitly merged with the respective repository.

Implications for researchers: With our work it is now possible to study for example Lehman's second law of software evolution [14] or implications of TD [3] *directly*. Previous work, see e.g., [10], studied development of certain internal complexity metrics on successive versions of entire systems or modules and thereby invalidated Lehman's second law of software evolution. But our results for `Gaffer` and `Cassandra` (Sec. 4.2) suggest a slight increase in difficulty of evolving software, i.e., they support Lehman's second law. It would be interesting to replicate previous studies and compare indirect complexity metrics with CC over time to understand if previous results are only due to indirect assessment of complexity.

6 Conclusions

Our goal with this work is to create, implement, and evaluate an algorithm and tool to automatically assesses complexity of contributing a change into an existing software system. We present the Contribution Complexity (CC) algorithm (Sec. 3) and we provide a readily installable tool for it. To evaluate our CC algorithm and tool, we compare the tool computed CC scores of 48 randomly sampled contributions from two open-source systems (`Gaffer` and `Cassandra`) with manually assessed CC scores of the same contributions (Sec. 4). Our results show that in 75% of the cases the automatic assessment matches the human assessment and we interpret the remaining cases to be due to the tool's superiority when assessing finer-grained complexity differences.

To illustrate applicability of our solution, we present an initial empirical analysis of 8 686 contributions from two open-source systems. Our results show that the average CC scores of both systems are slightly increasing with decreasing contribution frequency, which might hint at, that Lehman's second law of software evolution is not invalid when complexity of evolution tasks is directly assessed instead of indirectly as in previous work.

In future work we plan to extend the study of in-/decrease of difficulty of evolution tasks with the help of our CC to identify root causes of TD. Furthermore, we plan to conceptually extend CC to better distinguish inherent complexity of a contribution versus complexity of integration work.

References

1. Ieee standard for a software quality metrics methodology. Tech. rep. (1998)
2. Amanatidis, T., Chatzigeorgiou, A.: Studying the evolution of php web applications. Information and Software Technology **72**, 48–67 (2016)
3. Avgeriou, P., Kruchten, P., Ozkaya, I., Seaman, C.: Managing technical debt in software engineering (dagstuhl seminar 16162). In: Dagstuhl Reports. vol. 6. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2016)
4. Basili, V.R.: Qualitative software complexity models: A summary. Tutorial on models and methods for software management and engineering (1980)

5. di Biase, M., Rastogi, A., Bruntink, M., van Deursen, A.: The delta maintainability model: measuring maintainability of fine-grained code changes. In: 2019 IEEE/ACM International Conference on Technical Debt (TechDebt). pp. 113–122. IEEE (2019)
6. Dörner, D.: The Logic Of Failure: Recognizing And Avoiding Error In Complex Situations. Merloyd Lawrence Book, Basic Books (1997)
7. Hassan, A.E.: Predicting faults using the complexity of code changes. In: 2009 IEEE 31st international conference on software engineering. pp. 78–88. IEEE (2009)
8. Heitlager, I., Kuipers, T., Visser, J.: A practical model for measuring maintainability. In: 6th international conference on the quality of information and communications technology (QUATIC 2007). pp. 30–39. IEEE (2007)
9. Herraiz, I., Hassan, A.E.: Beyond lines of code: Do we need more complexity metrics? Making software: what really works, and why we believe it pp. 125–141 (2010)
10. Herraiz, I., Rodriguez, D., Robles, G., Gonzalez-Barahona, J.M.: The evolution of the laws of software evolution: A discussion based on a systematic literature review. ACM Computing Surveys (CSUR) **46**(2), 1–28 (2013)
11. Hindle, A., Godfrey, M.W., Holt, R.C.: Reading beside the lines: Indentation as a proxy for complexity metric. In: 2008 16th IEEE International Conference on Program Comprehension. pp. 133–142. IEEE (2008)
12. Hitz, M., Montazeri, B.: Measuring coupling and cohesion in object-oriented systems. Citeseer (1995)
13. Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. ACM SIGOPS Operating Systems Review **44**(2), 35–40 (2010)
14. Lehman, M.M., Fernández-Ramil, J.C.: Rules and tools for software evolution planning and management. Software Evolution And Feedback pp. 539–560 (2006)
15. Loftus, G.R.: Evaluating forgetting curves. Journal of Experimental Psychology: Learning, Memory, and Cognition **11**(2), 397 (1985)
16. McCabe, T.J.: A complexity measure. IEEE Transactions on software Engineering (4), 308–320 (1976)
17. Miller, G.A.: The magical number seven, plus or minus two: Some limits on our capacity for processing information. Psychological review **63**(2), 81 (1956)
18. Rios, N., de Mendonça Neto, M.G., Spínola, R.O.: A tertiary study on technical debt: Types, management strategies, research trends, and base information for practitioners. Information and Software Technology **102**, 117–145 (2018)
19. Shao, J., Wang, Y.: A new measure of software complexity based on cognitive weights. Canadian Journal of Electrical and Computer Engineering **28**(2), 69–74 (2003)
20. Spadini, D., Aniche, M., Bacchelli, A.: Pydriller: Python framework for mining software repositories. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 908–911 (2018)
21. Tamrakar, R., Jørgensen, M.: Does the use of fibonacci numbers in planning poker affect effort estimates? (2012)
22. Visser, J., Rigal, S., van der Leek, R., van Eck, P., Wijnholds, G.: Building Maintainable Software, Java Edition: Ten Guidelines for Future-Proof Code. O'Reilly Media, Inc., 1st edn. (2016)
23. Zuse, H.: Software Complexity: Measures and Methods, vol. 4. Walter de Gruyter GmbH & Co KG (1991)