

Achieving Accurate Clone Detection Results

Elmar Juergens
Technische Universität München
Munich, Germany
juergens@in.tum.de

Nils Göde
University of Bremen
Bremen, Germany
nils@informatik.uni-bremen.de

ABSTRACT

Many existing clone detection approaches produce substantial amounts of clones that are irrelevant to software engineers. These false positives significantly hinder adoption of clone detection in practice and can lead to inaccurate research conclusions. In this paper, we propose clone coupling as an explicit criterion for the relevance of clones and outline a method for clone detection tailoring. The results of a large industrial case study indicate that it can significantly increase clone detection accuracy.

Categories and Subject Descriptors

D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement—*restructuring, reverse engineering, and reengineering*

General Terms

Experimentation, Measurement

Keywords

Software maintenance, clone detection, assessment, tailoring

1. CLONE DETECTION ACCURACY

A wide variety of methods and tools exist to detect duplicated code within a software system [12, 14]. However, these tools detect a significant amount of clones that are irrelevant to software engineers. Kapsner and Godfrey report between 27% and 65% of false positives in case studies investigating cloning in open source software [9]. A recent study [16] has shown that up to 75% of clones detected by state-of-the-art tools are false positives. False positives dilute or even falsify research conclusions drawn from empirical investigations of code clones. Furthermore, they hinder the adoption of clone detection techniques by software developers—inspection of false positives is a waste of developer time.

Unfortunately, there is no common, agreed-upon understanding of the criteria that determine the relevance of clones

for software maintenance. This is reflected in the multitude of different definitions of software clones in the literature [12, 14]. This lack of relevance criteria introduces subjectivity into clone judgement [10, 17], making objective conclusions difficult. The negative consequences become obvious in the study done by Walenstein et al. [17]: three judges independently performed manual assessments of clone relevance; since no explicit relevance criteria were given, judges applied subjective criteria, rating only 5 out of 317 candidates identically. Obviously, such low agreement is unsuited as a basis for improvement of clone detection result quality.

Problem.

Clone detection tools produce substantial amounts of false positives—threatening the correctness of research conclusions and the adoption of clone detection by industry. However, we lack explicit criteria that are fundamental to make unbiased assessments of detection result accuracy; consequently, we lack methods for its improvement.

Contribution.

This paper introduces clone coupling as an explicit criterion for the relevance of code clones for software maintenance. We outline a method for clone detection tailoring that employs clone coupling to improve result accuracy. The results of two industrial case studies indicate that developers can estimate clone coupling consistently and correctly and show the importance of tailoring for result accuracy.

2. TERMS AND DEFINITIONS

This section introduces terms used in this paper.

A *clone group* is a set of clones. Its clones are code regions that are similar according to some definition of similarity—we deliberately chose a very general definition to subsume more specialized ones, such as type 1–3 clones [12]. A clone group with two clones is a *clone pair*. Clones in a single group are *siblings*. The cardinality of a clone group is the number of clones it contains; its length is the length of its longest clone. The set of clone groups detected by a tool is referred to as *candidate* clone groups. Its subset that is relevant for a certain task are the *relevant* clone groups, the remaining ones *false positives*.

A *change* is an alteration of a software system on the conceptual level. A *modification* is an alteration on the source code level. A single change comprises multiple modifications, if its implementation affects several code locations.

Detection result *accuracy* refers to a combination of *both* precision and recall. *Precision* denotes the probability that

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

IWSC2010 May 8, 2010, Cape Town, South Africa
Copyright 2010 ACM 978-1-60558-980-0/10/05 ...\$10.00.

a randomly chosen candidate clone group is relevant. *Recall* denotes the probability that a relevant clone group, chosen from the hypothetical set of all relevant clone groups, is contained in a detection result.

Clone coverage denotes the probability that an arbitrarily chosen source code statement is covered by at least one clone. It thus estimates the probability that a change to a statement is affected by cloning. *Clone blow-up* denotes the ratio of the size of the current system w.r.t. the size of a hypothetical system without cloning. It thus estimates the size-increase due to cloning.

3. CLONE RELEVANCE

This section introduces clone coupling as an explicit criterion to evaluate relevance of clones for software maintenance.

3.1 Clone Coupling

The fundamental characteristic of relevant clones causing problems for software maintenance is their change coupling, i. e., the fact that changes to one clone may also need to be performed to its siblings. This change coupling is the root cause for increased modification effort and for the risk of introducing bugs due to inconsistent changes to cloned code during software maintenance.

The coupling between clones has a direct impact on software maintenance efforts. If clones are coupled, each change to one clone also needs to be performed to its siblings. Each time one clone is changed, effort is required for location, consistent modification and testing of the other clone(s). In case the other clones are not modified, an inconsistency is introduced into the system. If the change was a bug fix, the unchanged clones still contains the bug. If, on the other hand, clones are not coupled, a change to one clone never affects its siblings, requiring no additional effort for location, modification and testing.

This impact of cloning on modification effort is largely independent of other characteristics of clones such as, e. g., their removability. Consequently, due to its implications for maintenance efforts, we propose to employ clone coupling as a criterion to evaluate the relevance of clones for software maintenance.

3.2 Determining Clone Coupling

In order to use clone coupling as a relevance criterion, we need a procedure to determine it on real-world software systems. To be useful in practice, this procedure needs to be broadly applicable. We propose to employ developer assessments of clone groups to estimate coupling, since they are not restricted to a specific system type, programming language, or analysis infrastructure. More specifically, assessors have to answer the following question:

RELEVANCE QUESTION 1. *If you modify a clone during maintenance, do you want to be informed about its siblings to be able to modify them accordingly?*

This way, developers estimate whether they get a positive return on their effort to inspect the siblings when performing a modification to a clone. The question partitions assessed clone groups into two classes—*relevant* clone groups whose expected coupling is high enough to impede software maintenance, and clone groups whose expected coupling is so low that they are *irrelevant* to software maintenance.

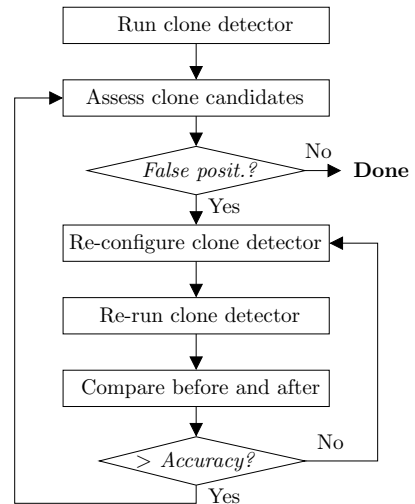


Figure 1: Steps of the tailoring method

4. CLONE DETECTION TAILORING

This section outlines a method for clone detection tailoring. Its goal is to remove false positives—clones that are irrelevant to software maintenance due to a very low coupling—from the detection results, while keeping relevant clones, to improve accuracy.

4.1 Tailoring Method

The steps of the tailoring method are depicted in Figure 1. First, the clone detector is run with a tolerant initial configuration that aims to maximize recall. Second, clone coupling is assessed on a sample of the candidate clone groups—assessment of all clones is typically too expensive. All candidate clone groups classified as uncoupled are treated as false positives. If no false positives are found, clone detection tailoring is complete.

If false positives are found, the clone detector configuration needs to be adapted to reduce the amount of false positives in the detection results. Different strategies can be used for this, as outlined in Section 4.3. Which strategy is used typically depends on the detected false positives. The clone detector is then executed with the adapted configuration. To determine the effect of the re-configuration on result quality, results before and after re-configuration are compared. More specifically, it is inspected whether the clone groups considered relevant are still contained, and whether the irrelevant clone groups are removed from the results. If the improvement of result accuracy is not satisfying, re-configuration and result evaluation is repeated. In case tailoring does not succeed to achieve both perfect precision and recall on the sampled candidate clones, one may be forced to make trade-offs on either precision or recall. From our experience however, precision can substantially be increased without damaging recall.

If the majority of the candidate clone groups in the assessed sample are false positives, it can contain too few relevant clone groups to conclusively estimate precision after tailoring. In this case, the method continues with another assessment (and possibly re-configuration, ...) step.

4.2 Characterizing False Positives

Clone detector reconfiguration determines the success of clone detection tailoring—accuracy is only increased, if reconfigurations are well conceived. Although automation is desirable, reconfiguration is currently a manual process.

Successful tailoring requires the identification of features that are characteristic for (a certain category of) false positives. Once they are known, the clone detector can be configured to handle code that exhibits these features specially. Any attributes of source code can, in principle, be candidates for such features: the location in the namespace or directory structure; filename or file extension patterns; implemented interfaces or super types; occurrence of specific patterns in the source code, e.g., *This code was generated by a tool*. Characteristic ways of structuring, e.g., sequences of constant declarations; identifiers of methods or types; location or role in the architecture.

There is no single, canonic way to determine characteristic features. However, we found that the reasons why developers consider candidate clones irrelevant often yield clues:

Code is unused—it will not be maintained. How can such dead code be recognized? Does it carry, e.g., *Obsolete* annotations as commonly encountered for .NET systems, or do affected types reside in a special namespace? If not, can developers produce a list of files, directories, types or namespaces that contain unused code?

Code is not maintained by hand since it is generated and regenerated upon change. Is generated code in a special folder or does it use a special file name or extension? Does it contain a signature string of the generator? If not, can it be made to do so?

Code has no conceptual relationship—maintenance is independent. This is typically encountered if the clone detector performs overly aggressive normalization, effectively removing all traces of the implemented concepts. Code then appears similar to the detector, despite the lack of a conceptual relationship that causes change coupling. Typical examples are regions of Java getters and setters or C# properties. Which language or system specific patterns can be used to recognize such code regions?

Compiler prevents inconsistent modifications. Examples are interfaces and `NullObject`¹ pattern implementations of the interfaces. Both interface and `NullObject` contain the same methods, down to identifiers and types. However, a developer is notified by the compiler that a change to the interface must be performed to the `NullObject` as well. The fact that the `NullObject` implements the interface can be a suitable characteristic.

4.3 Clone Detector Configuration

This section outlines strategies for clone detector configuration that can be used to remove characterized false positives. Their applicability depends on the applied clone detector. We outline configuration strategies applicable to our clone detector ConQAT²[6], since ConQAT provides great flexibility in detector re-configuration due to its pipes & filters style of analysis specification.

Minimum clone length prevents the detection of clones that are too short to be meaningful. It has a strong impact

¹`NullObjects` are empty interface implementations that reduce the number of required null checks in client code.

²Available as open source at <http://www.conqat.org>

on the results. While one-token clones are not very useful, too large values can significantly threaten recall. Still, excluding very short clones is an effective strategy to increase precision without damaging recall.

Code exclusion removes source code from the detection, and thus prevents detection of clones for certain code areas. ConQAT supports file exclusion based on name or content patterns. It also supports exclusion of code regions, which is crucial in environments where some regions of files are generated, whereas the remainder is hand maintained. This is, e.g., found in .NET development, where the GUI builder generated code is contained in a specific method in otherwise hand-maintained files.

Context sensitive normalization allows to apply different notions of similarity to different code regions. This way, equal identifier names and literal values can, e.g., be required for clones in stereotype or repetitive code such as variable declaration sequences, getters and setters, or select/case cascades, while at the same time differences in literals and identifiers are tolerated for clones in other code. Different heuristics and patterns for context sensitive normalization are available.

Clone Shaping allows to trim clones to syntactic structures such as methods or basic blocks. Clones that are shorter than the minimal clone length after shaping are removed from the results. This can, e.g., be used to remove short clones that contain the end of one and the beginning of another method without conveying meaning.

Post-detection clone filtering removes clones from the detection results. ConQAT supports content-based filtering, removal of overlapping clone groups, gap-ratio based filtering for gapped clones and black listing for filtering based on location-independent fingerprints that are robust during system evolution. Blacklisting can be used to exclude individual clones—it can thus be applied even if no suitable characteristics of false positives are known.

Re-configuration of any clone detection phase—preprocessing, detection, or post-processing—can improve accuracy.

4.4 Assessment Tool Support

Besides a configurable clone detector, further tool support is required to perform clone detection tailoring, namely:

Clone assessment: ConQAT provides tool support to improve the productivity of clone assessment. More specifically, it supports the generation of a random sample and stores the assessment results for each clone group. Most importantly, it offers a clone inspection viewer that displays two sibling clones side-by-side, providing syntax highlighting and coloring of differences between clones. This tool support has also been used for clone assessments in earlier studies [7]. We consider it crucial for productive clone assessment.

Comparison of clone reports: Tool support is provided to inspect the differences between two clone reports. This is helpful to investigate the impact of re-configuration on precision and recall.

Both support for clone assessment and comparison of clone reports, is available in ConQAT.

4.5 Experiences

We have employed the tailoring method (or one of its earlier stages) for our previous work on clone detection research [4, 7] and for our industrial applications of clone detection we performed on systems from our industrial part-

ners (written in C#, Java, ABAP, C/C++, PL/I and Matlab/Simulink from the domains of administration, automation, automotive, finance, retail and transportation). In all systems, tailoring succeeded to improve result accuracy. In most systems, result accuracy was below the developer acceptance threshold before tailoring; in many cases, it was sufficiently high for adoption after tailoring.

From our experience, many systems exhibit some idiosyncrasy that requires project specific tailoring. Examples are custom code generators whose generates are never manually modified—thus ruling out the necessity to perform coupled changes; or the use of some third party API that enforces contracts that give rise to syntactically similar but semantically different code.

In many systems, the biggest improvements in result accuracy could be achieved by exclusion of generated code and through application of context sensitive normalization. On average, we performed four tailoring iterations.

In principle, tailoring has to be performed for each system individually. However, from our experience, tailored configurations can often be reused in part for similar systems, e. g., written in the same programming language, since often the same code generators, third-party APIs or stereotype language constructs are encountered.

4.6 Assumptions

The tailoring method employs developer assessments of clone coupling on a clone sample to determine result accuracy. This is based on three assumptions:

Assessment Consistency. We assume that different developers evaluate the coupling of clones consistently.

Assessment Correctness. We assume that the evaluation of clone coupling is correct regarding how changes will affect clones in reality.

Assessment Generalizability. We assume that assessment results for a sample of the detected clones can be generalized to all clones.

While a certain amount of error can be tolerated, the assumptions must hold on a general level for the use of developer assessments on a sample to make sense.

5. STUDY

This section presents industrial case studies we performed to validate the assumptions of the method for clone detection tailoring and to evaluate its impact on detection results.

5.1 Research Questions

We use a study design with two study objects and four research questions:

RQ 1. *Do developers estimate clone coupling consistently?*

The application of developer assessments to estimate clone coupling is based on the assumption that different developers estimate clone coupling consistently. Experiments by Walenstein et al. [17] have demonstrated that clone assessments require an explicit criterion of clone relevance to produce consistent results. This research question validates whether the estimation of coupling represents such.

RQ 2. *Do developers estimate clone coupling correctly?*

Consistency alone is no sufficient indicator for correctness. Prediction of change, which is part of assessing the coupling

Table 1: Study Objects

	Lang.	Age (years)	Size (kLOC)	Developers (max)
A	ABAP	13	442	10 (40)
B	C#	8	360	4 (12)

between clones, inherently contains uncertainty. To assess how useful developer assessments of clone coupling are for tailoring, we need to understand their correctness.

RQ 3. *Can coupling be generalized from a sample?*

Rating is performed on a sample of the candidate clones, since real-world software systems contain too many clones to feasibly rate them all. The sample must be representative for the system, else sampling makes no sense.

RQ 4. *How large is the impact of tailoring on clone detection results?*

Tailoring changes the results of clone detection. The size of the change in terms of accuracy and amount of detected clones determines the importance of clone detection tailoring for both research and practice.

5.2 Estimation Consistency (RQ1)

Study object.

We use an industrial software system from the Munich Re Group as study object. The Munich Re Group is the largest re-insurance company in the world and employs more than 47,000 people in over 50 locations. For their insurance business, they develop a variety of individual supporting software systems. For non-disclosure reasons, we named the system *A*. An overview is shown in Table 1. Code size refers to the hand maintained code that was subject to analysis. The system implements billing, time and employee management functionality and supports about 3700 users.

Design.

We determine inter-rater agreement between different developers to answer RQ1. For this, developers independently estimate coupling for a sample of candidate clone pairs from the study object by answering assessment question 1 for each pair. Inter-rater agreement is then determined by computing Cohen’s Kappa.

Procedure and Execution.

Clone detection was performed with an untailored configuration on study object *A*. From the results, a random sample of clone pairs was generated. If a sampled candidate clone group contained more than two clones, its first two clones were chosen. Each developer assessed coupling for each clone pair individually. Assessment was guided by a researcher. The researcher explained the assessment tool and asked the assessment question for each clone pair, but took care not to influence assessment results. Developers could provide three answers, namely *accept*, *reject* and *undecided*. Individual rating meetings were limited to 90 minutes since experiences with developer clone assessments from earlier experiments [7] indicated that after 90 minutes, concentration and motivation decrease and threaten result accuracy.

Table 2: Estimation Consistency Results

Developers	Agreement
1 & 2	87.5%
1 & 3	85.4%
2 & 3	89.6%
1 & 2 & 3	81.3%
1 & 2 & 3 (w/o unrated)	88.1%

Results and Discussion.

Clone coupling was estimated for 48 clone pairs by three developers. Three clone pairs were rated as *undecided* by one developer, one clone pair was rated as *undecided* by two developers. Furthermore, five clone pairs received at least one *accept* and one *reject* assessment. The remaining 39 clone pairs all received the same ratings by all three developers. Table 2 shows the results of the assessment.

Agreement between pairs of developers ranges between 85.4% and 89.1%. Overall agreement is 81.3%. In rows 1–4, all clone pairs are taken into account, including clone pairs that were estimated as *undecided* by one developer. For the last row, the four clone pairs for which at least one developer rated *undecided* were removed from the result. On the remaining 44 clone pairs, 88.1% are rated consistently between three developers, indicating substantial agreement. Cohen’s Kappa for the three categories *accepted*, *rejected* and *undecided* and the three raters is 0.87 for the 48 rated clone groups. This is considered as almost perfect agreement.

For the analyzed clone pairs, developers did have a consistent estimation of the coupling of clones. After the assessments were complete, results were discussed with the developers. Developers could agree on a single assessment for four out of the five clone pairs that were assessed contradictorily. Only for a single clone pair developers remained of different opinion. Based on the results of the study, we consider it feasible to achieve consistent estimations of clone coupling through developer assessments.

5.3 Estimation Correctness (RQ2 & RQ3)

Study Object.

We use a second industrial software system from the Munich Re Group as study object. For non-disclosure reasons, we named the system *B*. An overview is shown in Table 1. The system implements damage prediction functionality and supports about 100 expert users.

Design.

Clone detection tailoring partitions the results of untailored clone detection into two sets—the set of *accepted* clone groups that are still detected after tailoring, and the set of *rejected* clone groups that are not detected anymore. If developer assessments of clone coupling are correct and results can be generalized from the sample (and no errors have been made during clone detection tailoring), accepted clone groups must exhibit a higher ratio of coupled changes during their evolution than rejected clone groups.

DEFINITION 1. *Change Coupling Ratio (CCR): Probability that a change to one clone of a clone group should also be performed to at least one of its siblings.*

We state this as a hypothesis:

HYPOTHESIS 1. *CCR for accepted clone groups is higher than for rejected clone groups.*

We determine CCR on the evolution history of the study object for both accepted and rejected clone groups as described below. We then use a paired t-test to test Hypothesis 1 against the null hypothesis that CCR for accepted clone groups is equal or smaller than for rejected clone groups.

CCR is determined by investigating the set of changes that are performed to clone groups during system evolution. CCR is simply the expected probability that a randomly chosen change to a clone group is coupled, which is equal to the ratio of the number of coupled changes to the number of all changes, including uncoupled ones.

In practice, developers do not have perfect change impact knowledge. The modifications developers perform to cloned code can deviate from the intentional nature of the change: developers can miss a clone when implementing a coupled change. The modification of the cloned code gets thus *unintentionally uncoupled*³. The three ways how a change can affect cloned code are: 1) *Consistent modifications* are intentionally coupled modifications to cloned code. 2) *Independent modifications* are intentionally uncoupled modifications to cloned code. 3) *Inconsistent modifications* are unintentionally uncoupled modifications to cloned code.

Information about the intentionality of a modification is, in general, not contained in the evolution history of a system⁴. It is thus manually assessed by the system developers.

We determine CCR for a system by inspecting changes between pairs of consecutive system versions as follows: first, clones are tracked between the two system versions to identify clone groups that were modified; second, all modified clone groups are inspected manually—based on their underlying change, they are classified into sets of consistently, independently or inconsistently changed clone groups; CCR can now be computed as:

$$CCR = \frac{|consistent| + |inconsistent|}{|consistent| + |inconsistent| + |independent|}$$

This procedure does not require accurate and complete evolution histories or genealogies of individual clone groups. To improve accuracy, it can be performed on multiple pairs of consecutive system versions—CCR is then determined on a larger sample of changes.

Procedure and Execution.

The system versions between which code modifications were analyzed were chosen using a convenience sampling strategy. Weekly snapshots of the source code were extracted from the version control system for the year 2006⁵. Between each snapshots, code churn was determined as the

³In principle, developers could also erroneously modify clones in a coupled fashion, although the change should only affect one clone, thus affecting an *unintentionally coupled* modification. However, since this case was not observed on the study object, we ignore it here.

⁴Based on history analysis alone, it is undecidable whether two differently modified sibling clones represent an independent or inconsistent modification and thus whether the underlying change is coupled or not.

⁵The developers employ our clone detection tool ConQAT during development since 2008. We thus analyzed an earlier evolution history fragment to avoid unwanted side effects on the data caused by the use of the clone detector.

number of changed files as an estimate of development activity in that week. Four weekly intervals were chosen for measurement. Their choice aimed at maximizing the covered part of the system evolution, to measure different times of year and to capture different levels of development activity in order to reduce the probability to only cover an unrepresentative part of the system’s evolution.

For each measurement interval, coupling was determined for both accepted and rejected clone groups as follows. First, modifications to cloned code were computed using a clone tracking approach similar to the one described in [5]. Second, all modifications to clone groups were manually classified as *consistent*, *inconsistent* or *independent*. Required effort to individually rate all clone groups for all intervals and both detection configurations would be too high to be feasible. Three measures were taken to reduce review effort:

Clone clustering: Due to the nature of clone groups, long clone groups often overlap with shorter clone groups of higher cardinality. We call such overlapping clone groups a *cluster*. A pre-study we performed to validate the tool setup showed that modifications are often rated equally for all clone groups in a cluster. Although all clone groups in a cluster were rated individually, rating productivity was substantially improved by sorting clone groups according to clone clusters.

Two-phase review: In the first phase, a researcher inspected all modified clone groups and classified those for which no common concept between clones could be identified as *independent*. Typical examples include getter and setter clones that are only considered similar due to overly aggressive normalization. In the second phase, the remaining clone groups were pair-reviewed by a researcher and a developer. The researcher operated the clone inspection tool, the developer took the rating decisions.

Single classification: Rated clone groups were partitioned into *accepted* and *rejected* sets. This was done by matching the rated clone groups against the results of clone tracking using a *tailored* detection configuration. Matching was performed in a semi-automated fashion: clone groups with identical positions were matched automatically, remaining clone groups were matched manually by a researcher based on their location and content⁶. Five out of 91 (5.5%) of the detected clusters could not be matched and were excluded from the study.

Clone detection was performed with ConQAT using a minimal clone length of 10 statements. Tailored detection was performed using an existing tailoring from an earlier collaboration that was created using the method from Section 4. It excludes clone groups with overlapping clones, employs context sensitive normalization of repetitive code regions and excludes *C# using* statements and generated code.

Results and Discussion.

Tables 3 and 4 show the results of the manual change classification and the resulting coupling for the set of accepted and rejected clone groups, respectively. In total, changes to 211 clone groups (containing 1279 clones) were manually classified during the experiment.

In intervals 1 and 2, modifications for one accepted clone group were rated as *don’t know*. For computation of cou-

⁶Tailoring can result in shorter clones that are thus not in identical locations as their untailored correspondents

Table 3: Evolution of accepted clone groups

Int.	Consistent	Inconsistent	Independent	Coupling
1	15	3	3	0.857
2	11	1	10	0.545
3	31	6	13	0.740
4	1	0	0	1.000
1-4	58	10	26	0.723

pling, they were conservatively counted as *independent*. This conservative strategy only makes it harder to answer the research question positively—it does not threaten the validity of a positive answer.

Table 4: Evolution of rejected clone groups

Int.	Consistent	Inconsistent	Independent	Coupling
1	2	0	10	0.167
2	0	1	42	0.023
3	0	0	23	0.000
4	1	0	38	0.026
1-4	3	1	102	0.034

The paired t-test for a confidence level of 95% yields a *p-value* of 0.002162 (< 0.05). This strongly indicates that the greater clone coupling for accepted than for rejected clone groups is statistically significant and thus confirms Hypothesis 1. Developer estimation of clone coupling thus aligns well with the evolution of clones during the system’s evolution history.

5.4 Clone Tailoring Impact (RQ3)

Study Object.

We use the same system from Munich Re as for RQ2.

Design.

We compute several cloning measures for the clone detection results before and after tailoring, namely: count of clones and clone groups, clone coverage and clone blow-up. We then calculate their delta to evaluate the quantitative impact of tailoring on the detection results.

Procedure and Execution.

We performed tailored and untailored clone detection on two versions of the source code of the study object. Untailored clone detection simply returns all type 1 and 2 clones (according to the definition from [12]). All measures were computed automatically by ConQAT. The first version is the one from the first measurement interval. The second version is from mid 2008 (before ConQAT was introduced for continuous clone management). Between these versions, the developers replaced hand-written data-access code with generated code that is never modified manually—if the data-access layer changes, it is fully re-generated—unintentionally uncoupled changes thus cannot occur. We included this second version to investigate the effect of generated code on untailored detection results.

Table 5: Impact of tailoring on detection results

	2006			2008		
	Untail.	Tail.	Δ	Untail.	Tail.	Δ
Clone Grps	598	332	-44%	2,558	1,028	-60%
Clones	2,118	1,005	-53%	12,675	3,558	-72%
Coverage	29.3%	18.3%	-38%	36.2%	19.4%	-46%
Blow-Up	27.8%	14.2%	-49%	41.2%	16.1%	-61%

Results and Discussion.

The results are displayed in Table 5. In both versions, tailoring substantially reduced the number of detected clones and thus clone coverage and blow-up. However, substantial amounts of cloning are still detected after tailoring. Tailoring affects results even more strongly if generated code is present—all measures are reduced by a larger factor.

The mere observation that the introduction of filters during tailoring reduces the number of detected clones is little surprising. However, for the analyzed system, recall was largely preserved—of the 72 clone groups to which coupled changes occurred, 68 were still detected by the tailored clone detection, indicating a recall of the tailored compared to the untailored detection of 94.4%. Consequently, changes in clone (group) count mostly denote changes in precision. More specifically, for the analyzed system, about every second clone group in the untailored result is considered irrelevant by developers. For the analyzed system, adoption of clone detection techniques for continuous clone management failed until tailoring was performed—even though the systems contained substantial amounts of relevant clones, false positive rates were considered too high for productive use.

6. THREATS TO VALIDITY

Internal.

The choice of the measurement intervals for RQ2 can affect result validity. We chose measurement intervals covering a year of development history, with different intervals between them and with different churn in order to reduce the probability of only selecting unrepresentative intervals.

We assume that all consistent changes are intentional, on the basis that a developer does not inadvertently invest effort into changing different clones consistently, if only a single clone needs to be changed. While this simplification can in principle introduce inaccuracy, we expect it to be negligible—of the 43 consistently modified clone groups manually investigated during the case study, not a single one was unintentionally modified consistently.

Our approach to measure clone coupling is unable to detect late propagations, because clones are tracked between two consecutive system versions only. This does not affect the quality of our results however, since manual classification of uncoupled changes by developers recognizes changes that are part of late propagations as unintentional inconsistencies, and thus as coupled changes.

Overeager tailoring can filter out clones that are relevant. This also leads to a substantial change in clone measures, but is not desirable in practice. However, in the analyzed system, 94.4% (68 out of 72) of the clone groups that evolved in a coupled fashion are still contained in the detection results after tailoring—indicating high recall of tailored in relation

to untailored detection results.

Manual classification of clone groups—as done to answer RQ2—entails the risk of misclassification due to human errors. We took several measures to reduce this risk: pair-classification was employed to reduce the probability of individual errors. The participating developer had been working on the project, without break, for several years, covering all measurement intervals—he was thus well familiar with the system. Furthermore, uncertain cases were rated as *don't know* to avoid guesswork and were handled conservatively.

In case clone classes from the untailored and the tailored detection results could not be mapped unambiguously, they were excluded from the study. Since this affected only 5.5% (five out of 91) of the detected clusters, we expect the potential impact of this simplification to be negligible.

External.

Each research question has been evaluated on a single system only. The systems have not been chosen randomly but were selected based on an existing cooperation and the availability and willingness of developers to contribute. Furthermore, only a single clone detector—and hence only a single clone detection approach—was employed. Thus, from the study results, we cannot tell how results are transferable to systems written in different languages, by other developer teams, or to other clone detectors or detection approaches. Although the results from the studies align well with experiences we have gathered applying clone detection tailoring in various other contexts, further studies are required to gain a better understanding of result transferability.

7. RELATED WORK

Clone detection is an active field of research. A general overview is given by Roy and Cordy [14]. We restrict ourselves to related research regarding clone detection tailoring, clone evolution and manual assessment of clones.

7.1 Clone Detection Tailoring

Kapser and Godfrey propose to filter clones based on the code regions they occur in. They report that such filters are successful to remove false positives in regions of stereotype code without substantially affecting recall. [9]. Their approach can be regarded as one tool for clone detector adaptation during clone detection tailoring.

All clone detection tools expose parameters whose valuations influence result accuracy. For some individual tools and systems, their effect on the quantity of detected tools has been reported [8]. However, we are not aware of systematic methods on how result accuracy can be improved.

To the best of our knowledge, this work is the first to propose an explicit method for clone detection tailoring that is furthermore based on an objective and measurable clone relevance criterion.

7.2 Clone Assessment

Burd and Bailey [3] compared three clone detection and two plagiarism detection tools using a single small system as study object. Through subjective assessments, 38.5% of the detected clones were rejected as false positives. None of the tools was able to achieve high precision and high recall at the same time. A more comprehensive study was conducted by Bellon et al. [2]. Six clone detectors were compared using eight different subject systems. A sample of the de-

tected clones was judged manually by Bellon. It was found that—depending on the detection technique—either precision or recall were insufficient to reach a high level of accuracy. Tiarks et al. [16] categorized type-3 clones detected by different state-of-the-art clone detectors according to their differences. Before categorization, they manually excluded false positives. They found that up to 75% of the clones were false positives. In contrast to the above studies, our approach employs clone coupling as a relevance criterion for clone assessment.

Walenstein et al. [17] reveal caveats involved in manual clone assessment. Lack of explicit clone relevance criteria results in low inter-rater reliability. However, they also confirm that once common relevance criteria are established, high inter-rater reliability can be achieved. Similar results are reported by Kapser et al. [10]. Their work emphasizes the need for measurement of inter-rater reliability to make sure common clone relevance criteria are used. It motivated RQ1 and inspired its case study design.

7.3 Changes to Clones

Kim et al. [11] analyze clone genealogies to identify different change patterns. Krinke [13] analyzes changes to clones between consecutive system versions in five open-source systems to quantify consistent and inconsistent changes to clone groups. Aversano et al. [1] investigate inconsistent changes to clones and differentiate between clones that are continuously maintained independently and those that are made consistent in a later version. Thummalapenta et al. [15] analyze different patterns in the evolution of code clones. They report most clone groups are changed consistently or intentionally evolve independently. Still, they also found unintentionally inconsistent changes that had to be propagated later. In previous work, we employ clone tracking to quantify consistent and inconsistent changes to clones in nine open-source systems [5].

This paper employs clone evolution analysis to validate developer assessments of clone coupling. While the above work provided important inspiration, this paper focuses on clone detection tailoring and not on clone evolution.

8. CONCLUSION

Accurate clone detection results are important for both research and practice. Tailoring clone detection results is thus an essential part of the clone detection process. Using clone coupling as criterion for clone relevance, and employing a combination of different tailoring techniques, we were able to substantially improve accuracy by discarding a considerable amount of clones irrelevant to software maintenance.

Obviously, the accuracy of untailored detection results varies between systems. However, the values from the studies coincide well with experiences we have made applying clone detection to systems from other domains. Without tailoring, the result accuracy is frequently too low for continuous application of clone detection during development. Furthermore, low accuracy dilutes conclusions drawn from empirical studies on properties of clones during system evolution.

Our study further indicates that developers can assess clone coupling consistently and correctly. Clone detection tailoring based on developer assessments of clone coupling is thus a promising approach to achieve accurate results.

Future work includes replication of the presented studies

to gain a better understanding of result transferability to systems written in other languages and other domains, and to cloning in other software artifacts, such as requirements specifications and models.

Acknowledgements

We would like to thank Benjamin Hummel, Stefan Wagner and the anonymous reviewers for helpful comments on initial versions of the paper.

9. REFERENCES

- [1] L. Aversano, L. Cerulo, and M. Di Penta. How clones are maintained: An empirical study. In *CSMR'07*, 2007.
- [2] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *IEEE TSE*, 2007.
- [3] E. Burd and J. Bailey. Evaluating clone detection tools for use during preventative maintenance. In *SCAM'02*, 2002.
- [4] F. Deissenboeck, B. Hummel, E. Juergens, B. Schaetz, S. Wagner, J.-F. Girard, and S. Teuchert. Clone detection in automotive model-based development. In *ICSE'08*, 2008.
- [5] N. Göde. Evolution of type-1 clones. In *SCAM'09*, 2009.
- [6] E. Juergens, F. Deissenboeck, and B. Hummel. CloneDetective – a workbench for clone detection research. In *ICSE'09*, 2009.
- [7] E. Jürgens, F. Deißeböck, B. Hummel, and S. Wagner. Do code clones matter? In *ICSE'09*, 2009.
- [8] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE TSE*, 2002.
- [9] C. Kapser and M. W. Godfrey. Aiding comprehension of cloning through categorization. In *IWPSE'04*, 2004.
- [10] C. J. Kapser, P. Anderson, M. Godfrey, R. Koschke, M. Rieger, F. van Rysselberghe, and P. Weißerger. Subjectivity in clone judgment: Can we ever agree? In *Duplication, Redundancy, and Similarity in Software*, Dagstuhl Seminar Proceedings, 2007.
- [11] M. Kim, V. Sazawal, D. Notkin, and G. C. Murphy. An empirical study of code clone genealogies. In *ESEC/FSE'05*, 2005.
- [12] R. Koschke. Survey of research on software clones. In *Duplication, Redundancy, and Similarity in Software*. Dagstuhl Seminar Proceedings, 2007.
- [13] J. Krinke. A study of consistent and inconsistent changes to code clones. In *WCRE'07*, 2007.
- [14] C. K. Roy and J. R. Cordy. A survey on software clone detection research. Technical report, Queens University at Kingston, 2007.
- [15] S. Thummalapenta, L. Cerulo, L. Aversano, and M. Di Penta. An empirical study on the maintenance of source code clones. *Empirical Softw. Eng.*, 2009.
- [16] R. Tiarks, R. Koschke, and R. Falke. An assessment of type-3 clones as detected by state-of-the-art tools. In *SCAM'09*, 2009.
- [17] A. Walenstein, N. Jyoti, J. Li, Y. Yang, and A. Lakhota. Problems creating task-relevant clone detection reference data. In *WCRE'03*, 2003.