Empir Software Eng (2008) 13:601–643
DOI 10.1007/s10664-008-9073-9

Empirical evaluation of clone detection using syntax suffix trees

Raimar Falke · Pierre Frenzel · Rainer Koschke

Published online: 22 July 2008

Editors: Massimiliano Di Penta and Susan Sim

Abstract Reusing software through copying and pasting is a continuous plague in software development despite the fact that it creates serious maintenance problems. Various techniques have been proposed to find duplicated redundant code (also known as software clones). A recent study has compared these techniques and shown that token-based clone detection based on suffix trees is fast but yields clone candidates that are often not syntactic units. Current techniques based on abstract syntax trees—on the other hand—find syntactic clones but are considerably less efficient. This paper describes how we can make use of suffix trees to find syntactic clones in abstract syntax trees. This new approach is able to find syntactic clones in linear time and space. The paper reports the results of a large case study in which we empirically compare the new technique to other techniques using the Bellon benchmark for clone detectors. The Bellon benchmark consists of clone pairs validated by humans for eight software systems written in C or Java from different application domains. The new contributions of this paper over the conference paper are the additional analysis of Java programs, the exploration of an alternative path that uses parse trees instead of abstract syntax trees, and the investigation of the impact on recall and precision when clone analyses insist on consistent parameter renaming.

Keywords Software clone detection · Redundancy · Duplication · Software maintenance · Software evolution · Program analysis

R. Falke · P. Frenzel · R. Koschke (✉)
University of Bremen, Bremen, Germany
e-mail: koschke@informatik.uni-bremen.de

R. Falke
e-mail: rfalke@informatik.uni-bremen.de

P. Frenzel
e-mail: saint@informatik.uni-bremen.de

1 Introduction

It is still a common habit of programmers to reuse code through copy and paste. Even though copy and paste is an obvious strategy of reuse and avoidance of unwanted side effects, this strategy is only a short-term win. The interests of these strategies must be paid back later by increased maintenance. Changes in all copies must be replicated if the original code must be corrected or adapted.

Although some researchers report that programmers argue not to remove clones because of the associated risks (Cordy 2003), there is a consensus that clones need to be detected at least. Detection is necessary to find the place where a change must be replicated and is also useful to monitor development in order to stop the increase of redundancy before it is too late.

Detecting duplicated code (also known as software clones) is an active research area. A recent study has compared these techniques and shown that token-based clone detection based on suffix trees is fast but yields clone candidates that are often not syntactic units (Bellon 2002; Bellon et al. 2007). Current techniques based on abstract syntax trees (AST; Baxter et al. 1998)—on the other hand—find syntactic clones but are considerably less efficient.

There are additional reasons to use AST-based clone detection beyond better precision. Because most refactoring tools are based on ASTs, they need to access clones in terms of nodes in the AST if they support clone removal. Furthermore, ASTs offer syntactic knowledge which can be leveraged to filter certain types of clones. For instance, one could exclude clones in declarative code or strictly sequential assignments as in constructors, which are often unavoidable. From a research point of view, it would also be interesting to categorize and see where redundancy occurs most frequently in syntactic terms (Kapser and Godfrey 2003a). Such empirical studies could also help to identify programming language deficiencies.

Contributions. This paper describes how we can make use of suffix trees to find duplicates in abstract syntax trees. This new approach is able to find syntactic duplicates in linear time and space. The paper reports the results of a case study in which we empirically and quantitatively compare the new technique to many other techniques using the Bellon benchmark for clone detectors. As a side effect of our case study, we extend the Bellon benchmark by additional reference clones.

The additional contributions over the conference paper (Koschke et al. 2006) are as follows. Whereas we were able to analyze only C systems for the previous paper, we are now in the position to handle Java as well. Hence, we can take full advantage of the Bellon benchmark, which consists of C and Java systems. Furthermore, our earlier tool is based on an abstract syntax tree using a full fledged front end for C with the overhead of semantic analysis, that is, name resolution, type binding etc. Because this kind of information is not needed for our analysis and because parsing is relatively cheap compared to semantic analysis, we implemented another tool that generates a parse tree by syntactic analysis. This tool allows us to compare efficiency regarding space and time consumption of our analysis for ASTs and parse trees. To take full advantage of the availability of syntactic information, we implemented syntactic filters for patterns of spurious clones, such as long array initializations, in order to improve the results. Last but not least, we investigate the impact on recall and precision when clone analyses insist on consistent parameter renaming

and extended the section on related research considerably, where we summarize empirical studies in clone detection.

Overview. The remainder of this paper is organized as follows. Section 2 delves into the notion of duplication, redundancy, and clones in software. Section 3 summarizes related empirical research. Techniques for automated clone detection are described in Section 4. In particular, this section describes clone detection based on suffix trees and abstract syntax trees in detail as they form the foundation of our new technique. Section 5 introduces the new technique. In Section 7, we compare the new technique to other techniques based on the Bellon benchmark for clone detectors. Section 8 concludes.

2 Code Cloning

There are different forms of redundancy in software. Software comprises both programs and data. In the database community, there is a clear notion of redundancy that has lead to various levels of normal forms. A similar theory does not yet exist for computer programs.

In computer programs, we can also have different types of redundancy. N-version programming and redundancy in programming languages to allow for type checking are examples for purposeful redundancy. Cloning is another type of redundancy that is often due to copy and paste programming. Unlike in data base theory, however, there is no agreement in the research community on the exact notion of redundancy in general and cloning in particular. Ira Baxter's definition of clones expresses this vagueness:

> Clones are segments of code that are similar according to some definition of similarity. Ira Baxter, 2002

According to this definition, there can be different notions of similarity. They can be based on text, lexical or syntactic structure, or semantics. They can even be similar if they follow the same pattern, that is, the same building plan. Instances of design patterns and idioms are similar in that they follow a similar structure to implement a solution to a similar problem.

Semantic similarity relates to the observable behavior. A piece of code, A, is semantically similar to another piece of code, B, if B subsumes the functionality of A, in other words, they have "similar" pre and post conditions. Unfortunately, detecting such semantic redundancy is undecidable in general. Because of that, automatic clone detectors focus on program-text similarity.

Program-text similarity is most often the result of *copy and paste*; that is, the programmer selects a code fragment and copies it to another location. Sometimes, these programmers are forced to copy because of limitations of the programming language. In other cases, they intend to reuse code. Sometimes these clones are modified slightly to adapt them to their new environment or purpose. Several authors report on 7–23% code duplication (Baker 1995; Kontogiannis et al. 1996; Lague et al. 1997); in one extreme case even 59% (Ducasse et al. 1999).

In this study, we adopt the following notion of a clone. Two segments of code are considered clones if they follow a similar code pattern and if one segment is

changed, the other one might need a similar change as well. In the absence of an agreed precise definition, we adopt this still somewhat vague definition. Yet, in the case study reported in Section 7, we will compare clones as proposed by automated tools to those accepted by human intuition.

Program-text clones can be compared on the basis of the program text that has been copied. We can distinguish the following types of clones accordingly:

- **Type 1 (exact clone)** is an exact copy without modifications (except for white-space and comments when insignificant from the perspective of the language definition).
- **Type 2 (parameter-substituted clone)** is a copy where only parameters (variable, type, function identifiers or literals) have been changed; given a suitable parameter substitution, the transformed copy is a type-1 clone
- **Type 3 (near-miss clone)** is a copy with further modifications, additions, or removals.

The clone detectors we evaluate in this paper are comparing tokens rather than text directly. A lexical analyzer processes indivisible sequences of characters (a *token*) to categorize them according to type. The token type tells whether the characters form a particular keyword of the programming language, an identifier, a punctuation, or a certain type of literal (e.g., integer, float, character, string literals). A token has an *image*, that is, a textual appearance (e.g., `0x10` in C) and an associated *value* in the domain of its meaning (e.g., the integer 16 in the case of `0x10`).

For our tools, we decided to consider token values rather than images for the comparison because the meaning can be the same even though the images are different for two different tokens (e.g., `16` versus `0x10` in C). Type-1 clones in the following are, hence, identical token sequences where both types and values are the same. Type-2 clones are token sequences with the same types, where both identifiers and literals are summarized in a common token type *parameter*. Type-3 clones finally are token sequences where one sequence could be transformed into the other by adding or removing tokens to obtain a type-2 clone (the number of such additions and removing is of course limited so that two sequences can be considered sufficiently similar).

Baker further distinguishes so called parameterized clones (Baker 1992), which are a subset of type-2 clones. Two code fragments A and B are a parameterized clone pair if there is a bijective mapping from A's identifiers onto B's identifiers that allows an identifier substitution in A resulting in A' and A' is a type-1 clone to B (and vice versa).

While type-1 and type-2 clones are precisely defined, the definition of type-3 clones is inherently vague. Some researchers consider two consecutive type-1 or type-2 clones together forming a type-3 clone if the gap in between is below a certain threshold of lines (Baker 1995; Li et al. 2006). Another precise definition could be based on a threshold for the Levenshtein Distance, that is, the number of deletions, insertions, or substitutions required to transform one string into another. There is no consensus on a suitable similarity measure for type-3 clones yet.

The above category of clone types is sufficient for our evaluation. We note that other authors have elaborated more distinguished categorizations useful to identify

the type of refactoring to remove a clone (Balazinska et al. 1999, 2000) or to characterize the scope and syntactic structure of clones (Kapser and Godfrey 2003a, b).

3 Related Empirical Research

This section summarizes related research in empirical studies in root causes, consequences, evolution of clones, and comparisons of clone detectors.

3.1 The Root Causes for Code Clones

A recent ethnographic study by Kim et al. (2004) has shed some light on why programmers copy and paste code. By observing programmers in their daily practice, they identified the following reasons:

Language Limitations Sometimes programmers are simply forced to duplicate code because of limitations of the programming language being used. Analyzing these root causes in more detail could help to improve the language design.

Explorative Programming Furthermore, programmers often delay code restructuring until they have copied and pasted several times. Only then, they are able to identify the variabilities of their code to be factored out. Creating abstract generic solutions in advance often leads to unnecessarily flexible and hence needlessly complicated solutions. Moreover, the exact variabilities may be difficult to foresee. Hence, programmers tend to follow the idea of extreme programming in the small by not investing too much effort in speculative planning and anticipation. This is supported by modern IDEs which have rich methods to support the programmers in these tasks.

Cross-cutting concerns Systems are modularized based on design principles such as information hiding, minimizing coupling, and maximizing cohesion. In the end—at least for systems written in ordinary programming languages—the system is composed of a fixed set of modules. Ideally, if the system needs to be changed, only a very small number of modules need to be adjusted. Yet, there are very different change scenarios and it is not unlikely that the chosen modularization forces a change to be repeated for many modules. The triggers for such changes are called *cross-cutting concerns*. For instance, logging is typically a feature that must be implemented by most modules. Another example is parameter checking in defensive programming where every function must check its parameters before it fulfills its purpose (Bruntink et al. 2004). Then copy and paste dependencies reflect important underlying design decisions, namely, cross-cutting concerns. Aspect-oriented language help to factor out such cross-cutting concerns, but in ordinary languages a programmer is forced to inline the concern at many places in the program.

Templating Another important root cause is that programmers often reuse the copied text as a template and then customize the template in the pasted context.

Kapser et al. have investigated clones in large systems (Kapser and Godfrey 2006). They found what they call *patterns of cloning* where cloning is consciously used as

an implementation strategy. In their case study, they found the following cloning patterns:

Forking is cloning used to bootstrap development of similar solutions, with the expectation that the evolution of the code will occur somewhat independently, at least in the short term. The assumption is that the copied code takes a separate evolution path independent of the original. In such a case, changes in the copy may be made that have no side effect on the original code.

Templating is used as a method to directly copy behavior of existing code where appropriate abstraction mechanisms are unavailable. It was also identified as a main driver for cloning in Kim and Notkin's case study (Kim et al. 2004). Templating is often found when a reused library has a relatively fixed protocol (that is, a required order of using its interface items) which manifests as laying out the control flow of the interface items as a fixed pattern.

Customization occurs when currently existing code does not adequately meet a new set of requirements. The existing code is cloned and tailored to solve this new problem.

Very likely other more organizational aspects play a role, too. Time pressure, for instance, does not leave much time to search for the best long-term solution. Unavailable information on the impact of code changes leads programmers to create copies in which they make the required enhancement; such changes then are less likely to affect the original code negatively. Inadequate performance measures of programmers' productivity in the number of lines of code they produce neither invite programmers to avoid duplicates.

3.2 Consequences of Cloning

There are plausible arguments that code cloning increases maintenance effort. Changes must be made consistently multiple times if the code is redundant. Often it is not documented where code has been copied. Manual search for copied code is infeasible for large systems. Furthermore, during analysis, the same code must be read over and over again, then compared to the other code just to find out that this code has already been analyzed. Only if you make a detailed comparison, which can be difficult if there are subtle differences in the code or its environment, you can be sure that the code is indeed the same. This comparison can be fairly time-consuming. If the code would have been implemented only once in a function, this effort could have been avoided completely.

For these reasons, code cloning is number one on the stink parade of bad smells by Fowler (1999). But there are also counter arguments. In Kapser and Godfrey's study (Kapser and Godfrey 2006), code cloning is a purposeful implementation strategy which may make sense under certain circumstances. Moreover, it is not clear when you have type-3 clones whether the unifying solution would be easier to maintain than several copies with small changes. Generic solutions can become overly complicated. Maintainability can only be defined in a certain context with controlled parameters. That is, a less sophisticated programmer may be better off maintaining copied code than a highly parameterized piece of code. Moreover, there is a risk associated with removing code clones (Cordy 2003). The removal

requires deep semantic analyses, and it is difficult to make any guarantees that the removal does not introduce errors. There may be even organizational reasons to copy code. Code cloning could, for instance, be used to disentangle development units (Cordy 2003).

The current debate lacks empirical studies of the costs and benefits of code cloning. There are very few empirical studies that explore the interrelationship of code cloning and maintainability. Monden et al. (2002) analyzed a large system consisting of about 2,000 modules written in 1 MLOC lines of COBOL code over a period of 20 years. They used a token-based clone detector to find clones that were at least 30 lines long. They searched for correlations of maximal clone length with change frequency and number of errors. They found that most errors were reported for modules with clones of at least 200 lines. They also found many errors in modules with shorter clones up to 50 lines. Yet, interestingly enough, they found the lowest error rate for modules with clones of 50 to 100 lines. Monden et al. have not further analyzed why these maintainability factors correlate in such a way with code cloning.

Chou et al. (2001) investigated the hypothesis that if a function, file, or directory has one error, it is more likely that is has others. They found in their analysis of the Linux and OpenBSD kernels that this phenomenon can be observed most often where programmer's ignorance of interface or system rules combines with copy and paste. They explain the correlation of bugs and copy and paste primarily by programmer ignorance, but they also note that—in addition to ignorance—the prevalence of copy and paste error clustering among different device drivers and versions suggests that programmers believe that "working" code is correct code. They note that if the copied code is incorrect, or it is placed into a context it was not intended for, the assumption of goodness is violated.

Li et al. (2006) use clone detection to find bugs when programmers copy code but rename identifiers in the pasted code inconsistently. On average, 13% of the clones flagged as copy and paste bugs by their technique turned out to be real errors for the systems *Linux kernel*, *FreeBSD*, *Apache*, and *PostgreSQL*. The false positive rate is 73% on average, where on average 14% of the potential problems are still under analysis by the developers of the analyzed systems.

3.3 Clone Evolution

There are a few empirical studies on the evolution of clones. Antoniol et al. propose time series derived from clones over several releases of a system to monitor and predict the evolution of clones fairly reliably (Antoniol et al. 2001). In another case study for the Linux kernel, they found that the scope of cloning is limited to certain subsystems (Antoniol et al. 2002). Only few clones can be found across subsystems; most clones are completely contained within a subsystem. The explanation for this phenomenon is that newer modules are often derived from existing similar ones. The relative number of clones, on the other hand, seems to be rather stable, that is, cloning does not occur in peaks. This last result was also reported by Godfrey and Tu who noticed that cloning is common and steady practice in the Linux kernel (Godfrey and Tu 2001). However, the cloning rate does increase steadily over time as observed by Li et al. (2006) for the Linux kernel and FreeBSD, where again only certain subsystems closer to the hardware show a higher rate of copy and paste code.

Kim et al. analyzed the clone genealogy for two open-source Java systems using historical data from a version control system (Kim et al. 2005). A clone genealogy forms a graph that shows how clones derive in time over multiple versions of a program from common ancestors. Beyond that, the genealogy contains information about the differences among siblings. Their study showed that many code clones exist for only a short time. Kim et al. conclude that extensive refactoring of such short-lived clones may not be worthwhile if they likely diverge from one another very soon. Moreover, many clones, in particular those with a long lifetime that have changed consistently with other elements in the same group cannot easily be avoided because of limitations of the programming language.

3.4 Comparison of Clone Detection Algorithms

The abundance of clone detection techniques calls for a thorough comparison so that we know the strengths and weaknesses of these techniques in order to make an informed decision if we need to select a clone detection technique for a particular purpose.

Clone detectors can be compared in terms of recall and precision of their findings as well as suitability for a particular purpose. There are several evaluations along these lines based on qualitative and quantitative data.

Bailey and Burd compared three clone and two plagiarism detectors (Bailey and Burd 2002). Among the clone detectors were three of the techniques later evaluated by a subsequent study by Bellon et al. (2007), namely, the techniques by Kamiya et al. (2002), Baxter et al. (1998), and Mayrand et al. (1996). For the last technique, Bailey used an own re-implementation; the other tools were the original ones. The plagiarism detectors were JPlag (Prechelt et al. 2000) and Moss (Schleimer et al. 2003).

The clone candidates of the techniques were validated by Bailey, and the accepted clone pairs formed an oracle against which the clone candidates were compared. Several metrics were proposed to measure various aspects of the found clones, such as scope (i.e., within the same file or across file boundaries), and the findings in terms of recall and precision.

The syntax-based technique by Baxter had the highest precision (100%) and the lowest recall (9%) in this experiment. Kamiya's technique had the highest recall and a precision comparable to the other techniques (72%). The re-implementation of Merlo's metric-based technique showed the least precision (63%).

Fig. 1 Participating scientists

Participant	Tool	Comparison
Brenda S. Baker (1995)	Dup	Token
Ira D. Baxter (1998)	CloneDR	AST
Toshihiro Kamiya (2002)	CCFinder	Token
Jens Krinke (2001)	Duplix	PDG
Ettore Merlo (1996)	CLAN	Function Metrics
Matthias Rieger (1999)	Duploc	Text

Fig. 2 Results from the Bellon and Koschke study (a *question mark* indicates that data were not reported from the participant)

	Baker	Baxter	Kamiya	Krinke	Merlo	Rieger
Clone type	1, 2	1, 2	1, 2, 3	3	1, 2, 3	1, 2, 3
Runtime	+ +	-	+	- -	+ +	?
Space	+	-	+	+	+ +	?
Recall	+	-	+	-	-	+
Precision	-	+	-	-	+	-

Although the case study by Bailey and Burd showed interesting initial results, it was conducted on only one relatively small system (16 KSLOC). However, because the size was limited, Bailey was able to validate all clone candidates.

A subsequent larger study was conducted by Bellon (2002) and Bellon et al. (2007). Their quantitative comparison of clone detectors was conducted for 4 Java and 4 C systems in the range of totalling almost 850 KSLOC. The participants and their clone detectors evaluated are listed in Fig. 1.

Figure 2 summarizes the findings of Bellon and Koschke's study. Row *clone type* lists the type of clones the respective clone detector finds (for clone types, see Section 2). The next two rows qualify the tools in terms of their time and space consumption. The data is reported at an ordinal scale −, −, +, + + where − is worst (the exact measures can be found in the paper of this study (Bellon 2002; Bellon et al. 2007)). Recall and precision are determined as in Bailey and Burd's study by comparing the clone detectors' findings to a human oracle. The same ordinal scale is used to qualify the results.

Interestingly, Merlo's tool performed much better in this experiment. However, the difference in precision of Merlo's approach in this comparison to the study by Bailey and Burd can be explained by the fact that Merlo compared not only metrics but also the tokens and their textual images to identify type-1 and type-2 clones in the study by Bellon and Koschke.

While the Bailey/Burd and Bellon/Koschke studies focus on quantitative evaluation of clone detectors, other authors have evaluated clone detectors for the fitness for a particular maintenance task. Van Rysselberghe and Demeyer (2004) compared text-based (Ducasse et al. 1999; Rieger 2005), token-based (Baker 1995), and metric-based (Mayrand et al. 1996) clone detectors for refactoring. They compare these techniques in terms of suitability (can a candidate be manipulated by a refactoring tool?), relevance (is there a priority which of the matches should be refactored first?), confidence (can one solely rely on the results of the code cloning tool, or is manual inspection necessary?), and focus (does one have to concentrate on a single class or is it also possible to assess an entire project?). They assess these criteria qualitatively based on the clone candidates produced by the tools. Figure 3 summarizes their conclusions.

Bruntink et al. use clone detection to find cross-cutting concerns in C programs with homogeneous implementations (Bruntink et al. 2005). In their case study, they used CCFinder—Kamiya's (Kamiya et al. 2002) tool evaluated in other case studies, too—one of the Bauhaus[1] clone detectors, namely ccdiml, which is a variation of

[1] http://www.axivion.com.

Fig. 3 Assessment by Rysselberghe and Demeyer

criterion	most suitable technique
suitability	metric-based
relevance	no difference
confidence	text-based
focus	no difference

Baxter's technique (Baxter et al. 1998), and the PDG-based detector PDG-DUP by Komondoor and Horwitz (2001). The cross-cutting concerns they looked for were error handling, tracing, pre and post condition checking, and memory error handling like range and null-pointer checking. The study showed that the clone classes obtained by Bauhaus' CCDIML can provide the best match with the range checking, null-pointer checking, and error handling concerns. Null-pointer checking and error handling can be found by CCFINDER almost equally well. Tracing and memory error handling can best be found by PDG-DUP.

4 Automated Clone Detection

Several techniques have been proposed to find clones. This section summarizes proposed techniques according to the type of information they are based on.

Textual comparison: whole lines are compared to each other textually (Johnson 1993). The result may be visualized as a dotplot, where each dot indicates a pair of cloned lines. Ducasse et al. detect clones as consecutive duplicated lines as uninterrupted diagonals or displaced diagonals in the dotplot for textual similarity (Ducasse et al. 1999).

Marcus and Maletic use latent semantic indexing (an information retrieval technique) to identify fragments in which similar names occur (Marcus and Maletic 2001).

Token comparison: Baker's technique is also a line-based comparison where the token sequences of lines are compared efficiently through a suffix tree (Baker 1995). First, each token sequence for whole lines is summarized by a so called *functor* and its parameters. The functor characterizes the token sequence uniquely. Identifiers and literals therein are the functor's parameter. An encoding of these parameters abstracts from their concrete values but not from their order so that code fragments may be detected that differ only in systematic renaming of parameters. Two lines are clones if they match in their functors and parameter encoding.

The functors and their parameters are summarized in a trie[2] that represents all suffixes of the program in a compact fashion. Every branch in this trie represents program suffixes with common beginnings, hence, cloned sequences. A more detailed description follows in Section 4.1.

Kamiya et al. increase recall for superficially different, yet equivalent sequences by normalizing the token sequences (Kamiya et al. 2002).

[2] A trie is an ordered tree data structure that is used to store an associative array where the keys are strings.

Because syntax is not taken into account, the reported clones may overlap different syntactic units, which cannot be replaced through functional abstraction. Either in a preprocessing (Cordy et al. 2004; Gitchell and Tran 1999) or post-processing (Higo et al. 2002; Kamiya et al. 2002) step, clones that completely fall in syntactic blocks can be found if block delimiters are known.

Metric comparison: Merlo et al. gather different metrics for code fragments and compare these metric vectors instead of comparing code directly (Laguë et al. 1997; Kontogiannis et al. 1996; Di Lucca et al. 2002; Lanubile and Mallardo 2003). A low distance (for instance, Euclidean distance) between these metric vectors can be used as a hint for similar code.

Comparison of abstract syntax trees (AST): Baxter et al. partition subtrees of the abstract syntax tree of a program based on a hash function and then compare subtrees in the same partition through tree matching (allowing for some divergences) (Baxter et al. 1998). A similar approach was proposed earlier by Yang (1991) using dynamic programming to find differences between two versions of the same file.

Comparison of program dependency graphs (PDG): control and data flow dependencies of a function may be represented by a program dependency graph; clones may be identified as isomorphic subgraphs (Krinke 2001; Komondoor and Horwitz 2001).

Other techniques: Leitao (2003) combines syntactic and semantic techniques through a combination of specialized comparison functions that compare various aspects (similar call subgraphs, commutative operators, user-defined equivalences, transformations into canonical syntactic forms). Each comparison function yields an evidence that is summarized in an evidence-factor model resulting in a clone likelihood. Walter et al. (2004) and Li et al. (2004) cast the search for similar fragments as a data mining problem. Statement sequences are summarized to item sets. An adapted data mining algorithm searches for frequent item sets.

In the following section, we will go into details of token-based clone detection and AST-based clone detection as they build the foundation for our own algorithm. In Section 4.3, we will compare them.

4.1 Token-Suffix-Tree Based Detection

Efficient token-based clone detection is based on suffix trees[3], originally used for efficient string search (McCreight 1976). Brenda Baker has extended the original algorithm to parameterized strings for clone detection (Baker 1995). Baker's approach offers the advantage of finding cloned token sequences with consistent renaming of parameters (variables and literals can be treated as parameters). We will first describe the original string-based approach and then Baker's extension for parameterized strings. We will use the program in Fig. 4 as a simple example. We would not consider such simple statements clones in practice; but we want to keep the example simple.

[3] An alternative to suffix trees are suffix arrays, which offer the advantage of less space consumption (Manber and Myers 1991) but at the cost of more runtime.

Fig. 4 Program fragment

```
1  x = a
2  break
3  x = x
4  y = a
```

For computer programs, we apply the suffix tree construction to tokens of the program. The tokens for the program in Fig. 4 are as shown in Fig. 5 where the unique character $\perp$ denotes the end token.

A suffix tree is a representation of all suffixes of a string, $S = s_1 s_2 \ldots s_n \perp$, over an alphabet Σ of characters including $\perp$ ($s_i \in \Sigma$) as a trie, where every suffix $S_i = s_i s_{i+1} \ldots s_n \perp$, is presented through a path from the root to a leaf. The edges are labeled with non-empty substrings. Paths with common prefixes share edges. Suffix trees are linear in space with respect to the string length (the edge labels are stored as indexes of start and end token of a substring) and there are linear-time algorithms to compute them (McCreight 1976; Ukkonen 1995).

The suffix tree for our running example is shown in Fig. 6. Each path from the root to a leaf represents a suffix S_i. We label the leaves with the corresponding suffix index i.

A clone can be identified in the suffix tree as an inner node of the trie (see Fig. 6). The length is the number of characters from the root to this inner node. We use the notation (a, b, l) to denote a matching clone pair starting at token index a and b, respectively, with length l. The number of occurrences of the clone is the number of leaves that can be reached from it. For instance, `= id` occurs three times and has length 2, denoted by (2, 6, 2), (2, 9, 2) and (6, 9, 2). The sequence `id = id` three times, too, with length 3, denoted by (1, 5, 3), (1, 8, 3) and (5, 8, 3).

As shown in the suffix tree, there are twelve cloned token sequences in the example program, but we notice, too, that many of them (namely, `id` and `= id`) are subsumed by the longest ones, namely, `id = id`. We are not interested in all those clones, but only in the maximally long ones. A maximally long clone can be defined as follows (Baker 1996) ($S(a, b)$ denotes string $s_a s_{a+1} \ldots s_b$):

Definition 1 Suppose $S(i, i+k)$ and $S(j, j+k)$ are a match, i.e., $S(i, i+k) = S(j, j+k)$, where $1 \le i \le i+k \le n$ and $1 \le j \le j+k \le n$ and $i \ne j$. We say this match is *left-extensible* if $S(i-1, i+k)$ and $S(j-1, j+k)$ are a match, and *right-extensible* if $S(i, i+k+1)$ and $S(j, j+k+1)$ are a match. If the match is neither left-extensible nor right-extensible and is not the trivial match $(1, 1, n)$, we say it is a *maximal match*.

We have three maximal clones of the token sequence `id = id` in Fig. 6 corresponding to line 1, 3, and 4 in Fig. 4, although line 3 is an inconsistent renaming of the variable names. For this reason, Baker's extension to suffix trees excludes

Fig. 5 Token table for Fig. 4

token	id	=	id	**break**	id	=	id	id	=	id	$\perp$
index	1	2	3	4	5	6	7	8	9	10	11

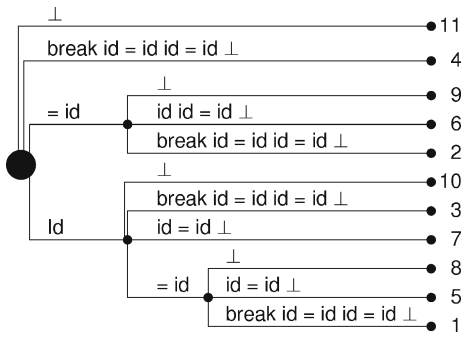


Fig. 6 Suffix tree for Fig. 4 treated as string

inconsistent renaming. In this extension, tokens are divided into two alphabets, Π of constant symbols for parameter tokens and all other tokens, Σ. Parameter tokens can be identifiers and literals and even operators such as + or -. In order to abstract from the concrete textual appearance of the parameter but not from its order of occurrence, parameters are represented by indexes in $\mathbb{N}_0$ (the set of non-negative integers including 0), where the index 0 denotes the first occurrence of a parameter, and any later occurrence is represented by the number of tokens to the closest left occurrence. Hence, strings of the earlier suffix tree approach are generalized to so called parameterized strings, short *p-strings*, this way. For instance, the p-strings for lines 1 and 4 in Fig. 4 are both $0 = 0$ whereas the p-string for line 3 is $0 = 2$.

Maximal matches (including parameterized maximal matches; see below) can be identified efficiently by an algorithm proposed by Baker (1996).

Two p-strings are clones with consistent renaming if they are a p-match (Baker 1996):

Definition 2 A sequence of symbols S' in $(\Sigma \cup \mathbb{N}_0)*$ is called a *parameterized string* or *p-string* for a string S constructed as follows: for each symbol s at position $i \in \mathbb{N}$ of S, there is a corresponding symbol s' at position i in S' where:

$$s' = \begin{cases} s & s \in \Sigma \\ 0 & s \in \Pi \wedge i \text{ is the first left occurrence of } s \\ i - j & s \in \Pi \wedge j \text{ is the left-most previous occurrence of } s \end{cases}$$

Two p-strings are a *parameterized match*, or *p-match*, if one p-string can be transformed into the other by renaming the parameters via a one-to-one function whose domain is the set of parameter symbols occurring in one p-string and whose range is the set of parameter symbols occurring in the other p-string.

The p-suffix tree contains all p-strings for all suffixes of the program. It is shown in Fig. 7 for Fig. 4. Note that it differs structurally from the one in Fig. 6. The difference comes from the fact that, unlike normal strings, p-strings change from one suffix to the other not only in their first character. For instance, the p-string for suffix S_1 is `0 = 0 break 4 = 2 0 = 7` $\perp$ whereas the p-string for suffix S_2 is `= 0 break 0 = 2 0 = 7` $\perp$; once the first occurrence of a parameter is removed, the second one becomes the first one and, hence, receives reference 0. That is why the algorithms

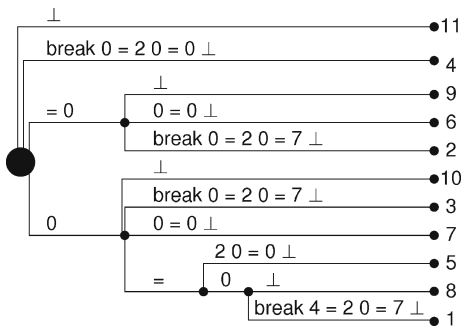


Fig. 7 Suffix tree for Fig. 4 treated as p-string

for constructing suffix trees for normal strings must be adjusted. Baker (1996) found a linear-time extension to the algorithm by McCreight (1976).

Beyond the extended algorithm to construct a p-suffix tree, she describes a linear-time algorithm to retrieve all clone pairs from the suffix tree that are neither both left-extensible nor both right-extensible. The clones are reported as pairs because the maximal match relation is not an equivalence relation, because it is not transitive (Baker 1996). This can be observed in our example of Fig. 5. According to the corresponding p-suffix tree in Fig. 7, (2, 6, 2)[4] and (6, 9, 2) are maximal matches, but (2, 9, 2) is not maximal because it is subsumed by the larger clone (1, 8, 3). Because p-matches do not form an equivalence class, pairs of p-substrings rather than equivalence classes of p-strings must be reported.

A filter on minimal length can be used to exclude short clones, where we can measure length in terms of lines of code or tokens.

Baker has an additional preprocessing step in her technique that summarizes all tokens of the same line as a so called functor. The functor is a one-to-one mapping from all non-parameter tokens of the same line onto a representing symbol. It can be viewed as a perfect hashing function for token subsequences onto symbols of alphabet Σ. The advantage of this summary is that the number of tokens for the suffix tree is reduced. The disadvantage is that the layout influences the results. If a programmer copies code and adds lines breaks, the clones can no longer be found. To overcome sensitivity to layout, one could pretty-print the code or simply not summarize several tokens to one functor.

4.2 AST-Based Detection

Because our algorithm is more similar to the token-suffix-tree approach, this section is briefer. Baxter et al. have proposed a clone detection technique based on abstract syntax trees (AST). An AST is a more compact representation of a parse tree. The abstract syntax abstracts from chain rules and peculiarities of the concrete grammar (often due to the underlying parsing technology). Left and right recursive grammar rules are typically represented explicitly as sequences in the AST.

[4]Please be reminded that (a, b, l) denotes a clone pair starting at token index a and b, respectively, with length l.

To find clones in the AST, we need—in principle—to compare each subtree to each other subtree in the AST. Because this approach would not scale, Baxter et al. use a hash function that first partitions the AST into similar subtrees. Because such a hash function cannot be perfect (there is an infinite number of possible combinations of AST nodes), it is necessary to compare all subtrees within the same partition in a second step. This comparison is a tree match, where Baxter et al. use an inexact match based on a similarity metric. The similarity metric measures the fraction of common nodes of two trees. Cloned subtrees that are themselves part of a complete cloned subtree are combined to larger clones. Special care is taken of chained nodes that represent sequences in order to find cloned subsequences.

4.3 Token Based Versus AST Based

Clone detection methods based on token-suffix trees offers several advantages over other techniques. It scales very well because of its linear complexity in both time and space, which makes it attractive for large systems. Moreover, no parsing is necessary and, hence, the code may be even incomplete and syntactically incorrect. Another advantage for a tool builder is that a token-based clone detector can be adjusted to a new language in short time (Rieger 2005). A scanner for a programming language is typically developed in one or two days. As opposed to text-based techniques, the token-based analysis is independent of layout (this argument is not quite true for Baker's technique, which is line based; however, if one uses the original string-based technique, line breaks do not have any effect). Also, token-based analysis may be more reliable than metrics because the latter are often very coarse-grained abstractions of a piece of code; furthermore, the level of granularity of metrics is typically whole functions rather than individual statements.

Two independent quantitative studies by Bellon/Bellon et al. (Bellon 2002; Bellon et al. 2007) and Bailey/Burd (Bailey and Burd 2002) have shown that token-based techniques have a high recall but suffer from many false positives, whereas Baxter's technique has a higher precision at the cost of a lower recall.

In both studies, a human analyst judged the clone candidates produced by various techniques. One of the criteria of the analysts was that the clone candidate should be something that is relatively complete syntactically, which is often not true for token-based candidates as they do not always form syntactic units. For instance, the two program fragments left and right in Fig. 8 are considered a clone by a token-based analysis because their token sequence is identical:

return id ; } int id ( ) { int id ;

Although from a lexical point of view, these are in fact clones, a maintenance programmer would hardly consider this finding useful.

Syntactic clones can be found to some extent by token-based techniques if the candidate sequences are split by a postprocessing step into ranges where opening and their corresponding closing tokens are completely contained in a sequence. For instance, by counting matching opening and closing brackets, we could exclude many spurious clones such as the one in Fig. 8. However, programming languages may have many types of delimiting tokens beyond brackets. The `if`, `then`, `else`, and `end if` all constitute syntax delimiters in Ada. In particular, `end if` is an interesting example as two consecutive tokens form one delimiter, of which both

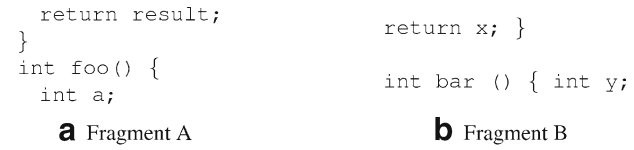


Fig. 8 Spurious clones

a Fragment A **b** Fragment B

can be each individual delimiters in other syntactic contexts. If one wants to handle these delimiters reliably, one is about to start imitating a parser by a lexer.

The AST-based technique, on the other hand, yields syntactic clones. And it was Baxter's AST-based technique with the highest precision in the cited experiment. Moreover, the AST-based clone detection offers many additional advantages. Most refactoring tools are based on ASTs. Hence, they need to access clones in terms of nodes in the AST if they support clone removal. Furthermore, ASTs offer syntactic knowledge which can be leveraged to filter certain types of clones, such as long array initializers.

Unfortunately, Baxter's technique did not match up with the speed of token-based analysis. Even though partitioning the subtrees in the first stage helps a lot, the comparison of subtrees in the same partition is still pairwise and hence requires quadratic time. Moreover, the AST nodes are visited many times both in the comparison within a partition and across partitions because the same node could occur in a subtree subsumed by a larger clone contained in a different partition.

Another point is that developing a syntactic analyzer requires considerable more effort than writing a lexical analyzer. There are complicated languages that need to be parsed using back-tracking which leads to expensive parsing. And with a post processing step, the token-based approach may lead to similar results in the area of eliminating syntactic incomplete clones. On the other hand, in many cases, the syntax tree is already available, so no extra costs incur. Such is the case when a syntactic clone detection is to be integrated into a modern development environment with refactoring support (for instance, Eclipse).

It would be valuable to have an AST-based technique at the speed of token-based techniques. In the next section, we show how a linear-time analysis can be achieved.

5 New Approach Using Suffix Trees for Syntax Trees

Our new algorithm to detect clones in ASTs or parse trees (further on referred to as syntax tree) consists of the following steps:

1. Parse program and generate syntax tree
2. Serialize syntax tree
3. Apply suffix-tree detection
4. Decompose resulting type-1/type-2 token sequences into complete syntactic units

Subsequent type-1 and type-2 clones can be combined to larger type-3 clones using a threshold of maximal gap in between. Baker describes a technique based on dynamic programming (Baker and Giancarlo 2002) to combine type-1 and type-2 clones. (Our implementation currently does not support that.) Step (1) is a standard

```
1 if (x + y) a = i; else foo();
2 if (p)     a = j; else foo();
3 if (q)     z = k; else bar();
```

Fig. 9 Sequence of if statements in C

procedure which will not be discussed further. Step (3) has been described in Section 4.1. We will primarily explain the details of step (4). We will first explain the serialization of the syntax tree of step (2) and then present the algorithm to decompose the cloned token sequences into syntactic units.

5.1 Serializing the Syntax Tree

We will use the example in Fig. 9 as an example to illustrate the algorithm. The syntax tree corresponding to Fig. 9 is shown in Fig. 10.

Because the suffix-tree based clone detection is based on a token stream, we need to transform the syntax tree into such a stream. We serialize the syntax tree by a preorder traversal. The nodes will then form the token stream. For each visited syntax tree node N, we emit N as root and associate the number of arguments (number of syntax tree nodes transitively derived from N) with it (in the following presented as subscript).

Note that we assume that we traverse the children of a node from left to right according to their corresponding source locations so that their order corresponds to the textual order.

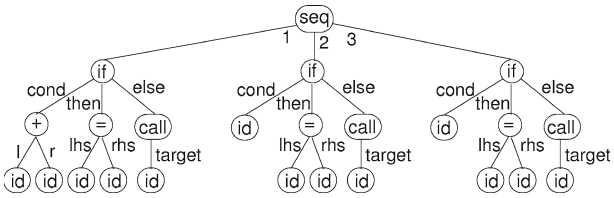
The serialized syntax tree nodes produced in step (2) for the example are shown in Fig. 11.

The serialized form is isomorphic to the original syntax tree. Hence, no clones are lost and no artificial syntactic clones are introduced.

5.2 Suffix Tree Detection

The original suffix tree clone detection is based on tokens. In our application of suffix trees, the syntax tree node type plays the role of a token (we will continue to use *token* instead of *syntax tree node* in the following as *token* was used in Section 4.1). Because we use the syntax tree node type as distinguishing criterion, the actual values of identifiers and literals (their string representation) do not matter because they are treated as syntax tree node attributes and hence are ignored. The actual values of identifiers and literals become relevant in a postprocessing step where we make the distinction between type-1 and type-2 clones by iterating over the parameters in two

Fig. 10 Example AST



```
1   seq23
2       if8 +2 id0 id0 =2 id0 id0 call1 id0
3       if6 id0 =2 id0 id0 call1 id0
4       if6 id0 =2 id0 id0 call1 id0
```

Fig. 11 Serialized syntax tree nodes

sequences and compare whether their values are the same. We omit the details for this simple step.

We can use both Ukkonen's algorithm for strings (Ukkonen 1995) or Baker's algorithms for parameterized strings (Baker 1992) to construct the suffix tree. Because the suffix tree does not contain the information of token values, the retrieval of clones on the suffix tree returns sets of clones that contain both type-1 and type-2 clones.

The results of the step described in this section are returned as a set of fragment sets. A fragment identifies a piece of code. Every fragment in a fragment set is alike to every other fragment in the fragment set. For instance, in the example of Fig. 11, the resulting two fragment sets are Fig. 12 (with two elements corresponding to line 3 and to line 4 in Fig. 11) and Fig. 13 (with three elements corresponding to line 2 column 4–9 and line 3 and line 4 in Fig. 11).

The token sequence in Fig. 12 is a complete syntactic unit whereas the sequence in Fig. 13 is not a single syntactic unit and, hence, needs to be decomposed into three syntactic subsequences as follows: <id0>, <=2 id0 id0>, and <call1 id0>.

5.3 Decomposing into Syntactic Clones

The previous step has produced a set of clone sets of maximally long equivalent syntax tree node sequences. These sequences may or may not be syntactic clones. In the next step—described in this section—these sequences will be decomposed into syntactic clones. This step is the main difference between our algorithm and purely token-based approaches.

The main algorithm is shown in Fig. 14 where `inset` is the input set of cloned fragment sets as determined in the previous step. For each set in `inset`, we select an arbitrary fragment. Because all fragments of a set are alike—since they are clones—we call the selected fragment the representative. We decompose the representative into syntactic sequences. The algorithm `make_pattern` does this job for the representative. It creates the indices at which the clones in `set` are to be cut into syntactic subsequences. The resulting pattern, hence, is a list of pairs; each pair consists of a start and end index identifying a subsequence of the representative of `set`. For instance, the pattern for the example in Fig. 13 is {(1, 1), (2, 4), (5, 6)}

```
1   if6 id0 =2 id0 id0 call1 id0
2 -- in line 3 and 4
```

Fig. 12 Cloned token sequence of Fig. 11

```
1 id0 =2 id0 id0 call1 id0
2 -- in line 2 (token pos. 4-9), 3, and 4
```

Fig. 13 Cloned token sequence of Fig. 11

denoting the subsequences <id$_0$>, <=$_2$ id$_0$ id$_0$>, and <call$_1$ id$_0$>. We will shortly describe `make_pattern` in more detail.

Then this pattern is applied to each element of the set in order to decompose each clone fragment into syntactic subclones. The output of `decompose_all` is denoted by `outset` and incrementally produced by `decompose`. The result is again a set of fragment sets, but the difference here is that each element in `outset` is a syntactic unit. Hence, `outset` is a refinement of `inset`.

Function `decompose` receives the list of pairs and successively decomposes a token sequence into subsequences described by the pairs. More precisely, let $s = s_1 \dots s_n$ be a token sequence and $p = \{(l_1, r_1), \dots, (l_m, r_m)\}$ a pattern; then the result of `decompose` is $\{[s_{l_1} \dots s_{r_1}], \dots, [s_{l_m} \dots s_{r_m}]\}$.

Procedure `emit` is used to report clones based on the representative. It filters clones based on various additional criteria such as length, type of clone, syntactic type (e.g., it may ignore clones in declarative code), it differentiates the clone set elements into type-1 and type-2 clones, and finally it reports all clones of a set to the user. We omit the details of `emit` here.

To ease the presentation, we will first ignore series of consecutive syntactic units that could be combined into one clone subsequence. We will come back to this issue after the presentation of the basic algorithm.

Finding Syntactic Token Sequences (Basic): The underlying observation for our basic algorithm is as follows. Let `ts` be the clone token sequence returned by the token-based clone detection that we use as a representative. A syntax subtree is a complete clone if all its tokens are completely contained in a cloned token sequence. The test whether the tokens of a syntax subtree, rooted by N, are contained in the cloned token sequence `ts` is simple: its root N must be contained and the number of its arguments `tokens(N)` (number of transitive successor syntax tree nodes reachable from N excluding N itself) must not exceed the end of `ts`. More precisely, let 1 be the first index and `ts'last` denote the last index in this sequence,then the

```
1 procedure decompose_all (inset) is
2   outset := ∅
3   for each set in inset loop
4     pattern := make_pattern (representative (set))
5     for each clone in set loop
6       outset := outset ∪ decompose(clone, pattern);
7     end loop;
8   end loop;
9   emit (outset);
```

Fig. 14 Decomposing into syntactic clones

following condition must hold for a complete syntactic unit: `n + tokens (N)` $\leq$ `ts'last` where n is the index of N in `ts`.

Figure 15 shows the basic algorithm. It traverses the whole cloned token sequence `ts`. As an example, let `ts` be as follows:

token type	$\texttt{call}_1$	$\texttt{id}_0$	$\texttt{while}_{14}$	$=_2$	$\texttt{id}_0$	$\texttt{id}_0$
index	1	2	3	4	5	6

Variables `le` and `ri` indicate the currently handled range of tokens for the current syntactic clone subsequence within the representative. If a root, indexed by `le`, is found to be complete (lines 8–10), the search continues after the last token of the rooted tree. The tokens up to the last token are part of the cloned rooted tree and we are interested only in maximal clones. For this reason, they may be skipped. In our example, we find that the token at index 1 is the root of a tree with one ancestor. This subtree is completely contained in the cloned token sequence. Hence, the search continues with the token at index 3.

If the current rooted tree is not completely contained in the cloned token sequence, we continue with its left-most child, which is the next token in the sequence (line 6 in Fig. 15). In our example, the `while` token at index 3 has 14 ancestors but only three of them are part of `ts`. So we skip token 3 and continue with token 4. This way, we descend into subtrees of a root to identify additional clones contained in an incomplete rooted tree. In our example, we would descend into the condition of the `while` loop. Descending into subtrees is necessary only at the end of a cloned token sequence, but it may be necessary to descend recursively. The recursion shows up in the traversal by increasingly smaller steps in the traversal.

Handling Sequence Tree Nodes: The basic algorithm finds syntactic units in the cloned token sequence. Yet, it misses subsequences of subtrees that together form a maximal clone. As an example, consider Fig. 16a and its corresponding syntax tree in Fig. 16b. The example consists of a statement sequence represented by a node type

```
1  function make_pattern (ts) is
2    pattern := ∅;
3    le := 1;
4    while le ≤ ts'last loop
5      if le + tokens(le) > ts'last then -- incomplete?
6        le := le + 1;
7      else -- complete subsequence
8        ri := le + tokens (le);
9        pattern := pattern ∪ {(le, ri)};
10       le := ri + 1;
11     end if;
12   end loop
13   return pattern;
```

Fig. 15 Pattern for syntactic clones; the first index is assumed to be 1

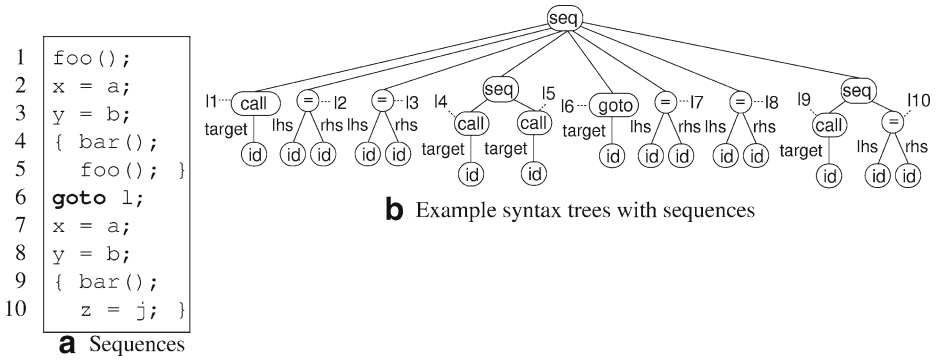


Fig. 16 Example with nested sequences in C; clones are in range 2–4 and 7–9; nodes in the tree are labeled *ln* if they represent a subtree corresponding to line *n*

seq with another nested sequence. The cloned token sequence representative for the example clone in lines 2–4 and 7–9 is as follows:

token type	$=_2$	id_0	id_0	$=_2$	id_0	id_0	seq_5	$call_1$	id_0
index	1	2	3	4	5	6	7	8	9

As you can see, the sequence runs into the nested sequence without covering it completely. The basic algorithm would, hence, generate a pattern $\{(1,3),(4,6),(8,9)\}$ corresponding to syntactic clone subsequences as follows: $<=_2\ id_0\ id_0>$ (x=a), once more $<=_2\ id_0\ id_0>$ (y=b), and $<call_1\ id_0>$ (bar()). However, the two consecutive assignments together form a maximal clone sequence. The basic algorithm misses this maximal clone sequence because only parts of the outer sequence are part of the cloned token sequence. Whereas other syntax tree node types require that all parts are present to form a complete clone, consecutive successors of a sequence node may together form a maximal clone.

The extended algorithm in Fig. 17 considers sequence nodes. The extension is found in lines 8–16. Predicate `is_seq` is true if a node represents a sequence in the syntax tree. In that case, we collect all syntactic cloned token subsequences (line 12) as long as they are completely contained in the cloned token sequence (line 13) and have the same parent (lines 14). Function `parent(N)` returns the parent syntax tree node of `N`.

The result of applying this step to the example in Fig. 16a is the pattern $\{(1,6),(8,9)\}$ corresponding to the two cloned code fragments in line range 2–3 and 7–8 and in line 4 and 9 in Fig. 16a.

6 Tools

In order to compare the approaches not only in terms of precision and recall but also in runtime, we implemented several variations of the approaches (cf. Fig. 18). Because these tools are built on a common infrastructure of ours and were executed on the same hardware, the runtime comparison is more meaningful than the reports in the Bellon study (Bellon 2002).

```
1  function make_pattern (ts) is
2    pattern := ∅;
3    le := 1;
4    while le ≤ ts'last loop
5      if le + tokens(le) > ts'last then -- incomplete?
6        le := le + 1;
7      else -- complete sequence
8        if is_seq (parent (ts (le))) then -- part of a sequence?
9          ri := le;
10         -- assert: ri + tokens(ri) ≤ ts'last
11         loop
12           ri := ri + tokens (ri) + 1;
13           exit when ri + tokens (ri) > ts'last
14                or else parent (ts(le)) ≠ parent (ts(ri));
15         end loop;
16       else
17         ri := le + tokens (le) + 1;
18       end if;
19       ri := ri - 1; -- ri is one off
20       pattern := pattern ∪ {(le, ri)};
21       le := ri + 1;
22     end if;
23   end loop;
24   return pattern;
```

Fig. 17 Pattern considering sequence nodes

This section describes our infrastructure and how the tools we will evaluate in our case study are composed of the infrastructure. We will give a more detailed description of the infrastructure so that one can understand the detailed performance measurements in the evaluation.

6.1 Infrastructure

Figure 19 shows the common infrastructure of the tools we implemented. A tokenizer (T) transforms the source text into a stream of tokens. Our tokenizers for Java and

intermediate rep.	comparison	parameterization	abbreviation	real-life name	author
AST	Tree **M**atch	**N**on-parameterized	AMN	CCDIML	S. Bellon
AST	**S**uffix tree	**N**on-parameterized	ASN	CPDETECTOR	R. Koschke
Parse tree	**S**uffix tree	**P**arameterized	PSP	CLAST	R. Falke
Parse tree	**S**uffix tree	**N**on-parameterized	PSN	CLAST-ba	R. Falke
Token	**S**uffix tree	**N**on-parameterized	TSN	CLONES-uk	R. Koschke
Token	**S**uffix tree	**P**arameterized	TSP	CLONES-ba	R. Koschke
Token	**S**uffix tree	**P**arameterized	TSNS	CSCOPE	P. Frenzel

Fig. 18 Feature combinations (tools above line are syntactic, those below are lexical); the abbreviation is a mnemonic combination of the three distinguishing features. The right-most column lists the real-life names of the tools that were used in previous papers

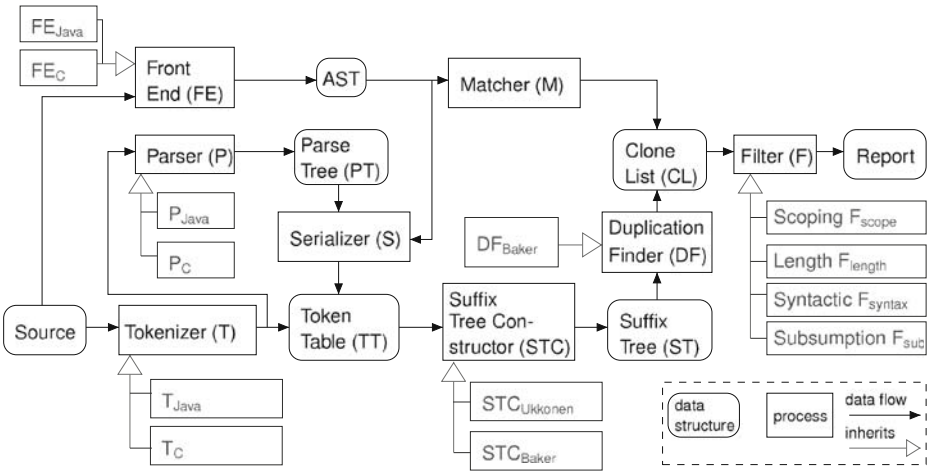


Fig. 19 Tools Architecture

C are generated by a scanner generator based on the token definitions specified by regular expressions. Tokens are stored in a token table (TT). Because different tools use different types of tokens (scanner tokens, parse tree nodes, abstract syntax tree nodes), the token table is implemented as a generic unit and instantiated for each type of token sets. The suffix tree constructor (STC) creates the suffix tree (ST). Our infrastructure provides two types of tree constructors, one for parameterized strings using Baker's algorithm and one for non-parameterized strings using Ukkonen's algorithm. Once constructed, the suffix tree is traversed and clones are identified. There are different possible criteria to retrieve clones from the suffix tree. We adopt Baker's search for maximally long token sequences using her linear-time algorithm (Baker 1996) (denoted as duplication finder $\mathrm{DF}_{\mathrm{Baker}}$ in Fig. 19).

The clone list (CL) contains all found clone pairs. Various filters can be applied to it such as the length of the clone pair (measured by either lines of code or number of tokens), $\mathrm{F}_{\mathrm{length}}$. This filter removes also sparse clones, that is, clones which have a low number of tokens per line where a line-based threshold would not help. Sparse clones are typically not relevant because they consist of very few tokens. A sparse clone is one that fulfills the following criterion (where T is the number of tokens and L is the number of lines of the clone):

$$\left(\frac{T}{L} < 1 \wedge T \leq 5\right) \vee \frac{T}{L} \leq 0.5$$

The definition of a sparse clone is due to the following reasoning. A clone with only five tokens is sparse from our point of view, because, normally, a line has about 10 tokens on average. Our threshold in the experiment for clone length is six lines. The value of six comes from the earlier study by Bellon et al. (2007) and was an agreement among all participating researchers. We reused six lines for consistency in our experiment. This number includes empty lines. However, we found several instances of clones that fulfill the line criterion but have only very few tokens because they contain blank lines. These clone candidates were all spurious. That is, we are not completely convinced that the six-line limit is a good criterion if programs contain

blank lines or lines with only one delimiter such as a bracket. If we have six lines, but not all lines contain tokens (i.e., $T/L < 1$), we would likely have at least five tokens resembling the idea of six with respect to lines with some tolerance. If we do have more than five tokens, code that contains only one token on every second line on average is typically spurious, too, according what we have seen. The threshold of 0.5 in $T/L \leq 0.5$ comes from our experience.

The algorithm we presented in Section 5.3 can be considered a filter, too, that decomposes token sequences into sequences completely contained in a syntactic unit (denoted by F_{syntax} in Fig. 19). In order to do so, the tokens must be annotated with the number of their transitive syntactic successors as described in Section 5.1. If such an annotation is not available, a heuristic approach can be taken based on counting opening and closing scope delimiters such as curly brackets in C and Java. This filter is denoted by F_{scope} in Fig. 19.

Decomposing into syntactically complete subsequences by filter F_{syntax} requires a second test for maximal p-matches even if the original sequences were maximal. Out of two maximal p-matches p_1 and p_2 consisting of sequences $\alpha\beta\gamma$ and $\delta\epsilon\zeta$, respectively, i.e., $\alpha\beta\gamma = \delta\epsilon\zeta$, we could decompose into β and ϵ where β subsumes ϵ. So the clone pair consisting of ϵ would not be maximal. As a consequence, we need to test subsumption once again. The problem is that we have to compare all clones for subsumption. We have not yet invested much time to optimize this step, as our main concern was the suffix-tree related issues. Our subsumption problem is related to selection problems for which so called sweep-line algorithms can be used. Sweep-line algorithms solve the following problem: given a set S of n closed segments in the plane, report all intersection points among the segments in S. A first sweep-line algorithm was proposed by Bentley and Ottmann (1979). The algorithm requires $O(n \times log(n))$ time in the worst case where n is the number of clone sets.

The filter denoted by F_{sub} removes all non-maximal clones. We note that although Baker's algorithm to retrieve maximal p-matches from the suffix tree checks whether a p-match is left or right extensible, it does not check whether one fragment can be extended to the left and the other one to the right. For this reason, we use filter F_{sub} immediately after the duplication finder DF and once more at the end on clone list CL if necessary.

The tool CCDIML, which implements an Baxter-like AST-based clone detection, uses an additional filter relevant only to trees, which can be filtered by depth of the tree and number of nodes.

We note that filters can be combined to a filter pipeline where the order matters only for performance and that we use some filters also at earlier stages in order to decrease the volume of data as soon as possible.

Other paths of clone detection can be taken for syntax trees using either a tree matcher (denoted by M in Fig. 19) as described in Section 4.2 or the suffix-tree detection for serialized tree nodes (our new approach described in Section 5). The abstract syntax trees are built by language-specific front ends. In our case, we use the commercial front end by *Edison Design Group*[5] for C and C++ and the IBM open-source compiler `jikes` for Java. We extended both front ends in order to generate

[5] http://www.edg.com.

our own unified abstract syntax tree (Koschke et al. 1998), which represents both languages among other languages such as Ada.

There is a lot of overhead in full-fledged language front ends to compute semantic information on the program that we do not need actually. For our use of the abstract syntax tree, we need only the syntactic structure but no semantic information such as name resolution and type binding. Moreover, our extensions to these front ends map a language-specific syntax tree onto a generalized abstract syntax tree for many languages. Although this step allows us to reuse our syntax-tree based clone detection across different languages, this extra step requires additional runtime resources.

In order to avoid this extra overhead, we implemented another tool based on our infrastructure that performs only lexical and syntactic analysis and then outputs a parse tree (PT) which contains every token of the program unlike the abstract syntax tree. The parsers for C and Java are generated using the parser generator `yacc`. The parser (P) creates a tree that resembles the parse tree but has certain additional transformations applied to it. The transformations have different purposes. One kind of transformations replaces subtrees derived through left or right recursive grammar rules by explicit sequence nodes, which are a necessity for our algorithm to work. Another type of transformations replaces chained-rule derivations by lifting the leaf node. These transformations are optional but reduce the number of tokens.

One other optional type of transformations is used to implement alternative views on the parse tree from a user's perspective. The advantage of syntactic tools is that it is relatively simple to define patterns for spurious clones that should be ignored or viewed differently for the analysis. Kapser and Godfrey (2005) use this type of filter to improve results, but are basically mimicking a syntactic analysis through a combination of lexical patterns. To take full advantage of the availability of syntactic information, we implemented syntactic transformations for patterns of spurious clones in order to improve the results. Using syntactic patterns, the serializer can ignore very long array initializations, which occur frequently in systems with graphical user interfaces as XPM files. Additionally, another transformation refactors structured `switch` statements into if-then-else cascades. A structured switch statement is one whose labels are all at top-level and whose case branches are all ended by a `break`. Then we can model a user's logical point of view. A user might want to see only clones that completely fall into such a structured `case-break` region, for instance. To achieve this view, we can transform the tree such that the statements subsequent to the `case` label become syntactic children of the `case` node. Because the original grammar is much more liberal for `switch` statements and allows `case` labels and `breaks` at any nested level and also `case` branches with no corresponding break, this transformation is not always applicable. Only through syntactic analysis, we can distinguish structured switch statements from unstructured ones. A clone detector based only on tokens cannot handle this case reliably. We use this transformation as a prototype for the use of user-defined syntactic views.

Both syntax trees and parse trees can be serialized as described in Section 5.1 in order to use the suffix-tree based approach. The serializer component is denoted by *S* in Fig. 19. The main difference between parse trees and abstract syntax trees here is that parse trees contain many more nodes in general. Figure 21 gives an overview on the number of nodes for each type of syntax tree for the systems we studied.

Fig. 20 Tools and components; the abbreviations relate to Fig. 19; components CL and F_{length} are used for all tools

tool	components
AMN	FE, AST, M
ASP	FE, AST, S, TT, STC_{Baker}, ST, DF_{Baker}, F_{syntax}, F_{sub}
ASN	FE, AST, S, TT, $STC_{Ukkonen}$, ST, DF_{Baker}, F_{syntax}, F_{sub}
PSP	T, P, PT, S, TT, STC_{Baker}, ST, DF_{Baker}, F_{syntax}, F_{sub}
PSN	T, P, PT, S, TT, $STC_{Ukkonen}$, ST, $DF_{Ukkonen}$, F_{syntax}, F_{sub}
TSP	T, TT, STC_{Baker}, ST, DF_{Baker}, F_{sub}
TSN	T, TT, $STC_{Ukkonen}$, ST, DF_{Baker}, F_{sub}
TSNS	T, TT, STC_{Baker}, ST, DF_{Baker}, F_{scope}, F_{sub}

Figure 20 shows how the tools are composed of the components of our infrastructure. ASP, ASN, PSP, and PSN implement the technique that we describe in this paper. The closest techniques to our approach are the token-based and syntax-tree based techniques. That is why we chose these techniques as a point of comparison.

6.2 Syntax-based Tools

The tool AMN is a variation of Baxter's CloneDR also based on ASTs. The two tools ASP/ASN and AMN share the same AST as intermediate representation, whereas PSP/PSN uses parse trees. The main differences between AMN and CloneDR are as follows:

- Baxter et al. have not published their hashing function, hence, AMN is likely using a different one
- AMN does not use a similarity metric to compare trees but requires that trees are isomorphic (abstracting from parameter values to detect type-2 clones)
- CloneDR checks for consistent renaming while AMN currently does not do this
- AMN is based on our AST unified for different programming languages; Baxter et al.'s infrastructure is targeting at program transformations where typically less abstraction in the syntax trees is feasible (for instance, the position of every bracket needs to kept here, whereas we do not represent such minor tokens in our AST)
- furthermore, our AST has explicit representations for sequences; Baxter et al.'s infrastructure is grammar-driven, that is, the abstract syntax tree is determined by the grammar of the language to parse; iterative constructs are specified as left or right recursive grammar rules leading to left or right leaning syntax trees; such parts of the tree need special care in the algorithm by Baxter et al.; our explicit sequence representations eases the detection
- AMN hashes only nodes of interest and omits AST nodes that do not directly map onto a source element (such nodes are introduced because we normalize our ASTs), thus reducing the amount of subtrees in the hash buckets

The tools AMN and CloneDR have in common the partitioning by way of the hash function and the pairwise comparison of subtrees in the same partition. Because the pairwise comparison leads to a complexity of $O(n^2)$ where n is the size of the largest partition, AMN offers a threshold that allows to ignore all partitions whose number of entries is greater than the threshold, which bounds the run time at the expense of a potential loss of clones. Bellon, the author of AMN, states that such

large partitions typically contain trivial AST subtrees and the likelihood that more interesting subtrees are in the same partition is low. He has not found any practical indication to the contrary testing this hypothesis on various systems[6].

To generate the candidates of AMN evaluated in this paper, we used 400 as the threshold to ignore very large buckets. In order to explore the effect of the threshold on runtime, we will report runtime measurements for the selected threshold of 400 and for a larger threshold.

6.3 Token-based Tools

TSP is a variation of Baker's technique with the difference that it is not based on lines but solely on tokens. TSN varies from TSP in that it uses non-parameterized suffixes. The advantage of non-parameterized suffixes is that TSN can find clones where a programmer has changed parameters inconsistently. These inconsistent changes are often hints for errors (Li et al. 2006). The disadvantage is that more spurious clones are reported.

As already discussed in Section 4.3, token-based techniques can be extended so that they attempt to find syntactic clones to a certain extent as well by splitting cloned token sequences into subsequences with a balanced set of opening and closing scope delimiters in a postprocessing step. For this reason, we compare our new techniques also to a token-based technique applying this strategy. TSNS implements this postprocessing step and searches for balanced brackets of type {…}, (…), and […]. As a matter of fact, TSNS is not a new tool but just an additional feature built into TSN that can be turned on via a command line switch.

It is worth to note that our suffix tree implementation is generic and is used in ASP/ASN, PSP/PSN, TSP/TSN, and TSNS identically.

We must also note that the token-based techniques work on the non-preprocessed code, that is, preprocessor directives are also considered tokens. Another relevant technical detail is that we considered only identifiers as parameters for the parameterized clone detection. In practice, one could also use literals or even operators as parameters.

All our tools find only type-1 and type-2 clones except for AMN, which also finds type-3 clones; but type-3 detection has been disabled in this experiment.

7 Empirical Evaluation

In this section, we evaluate our new technique empirically by comparing it to alternative techniques. We first describe the experimental layout.

7.1 Bellon Benchmark

The basis for this comparison is the Bellon benchmark that has been developed and used for the most comprehensive quantitative comparison of software clone detectors to date (Bellon 2002; Bellon et al. 2007). In that study, six different tools

[6]Personal communication; March 2007.

(cf. Fig. 1) have been compared based on several Java and C systems. This section describes the oracle, the systems analyzed, and the metrics to measure recall and precision.

The Oracle: From the Bellon benchmark the oracling process and tools for validation and evaluation were re-used.

The results of the tools are evaluated by a human oracle. We rely on human intuition because there is no accepted operational definition of a clone yet. One author of this paper (Pierre Frenzel) played the role of the oracle in our study. To avoid bias, the human oracle did not know which tool produced the clone candidates. They were presented to him by a selection process that was automated, blind, fair, and random:

- *Automated*: an algorithm selected the clone candidates to be presented to the human oracle
- *Blind*: the human oracle had no idea from which tool's candidates the presented clone candidate was drawn
- *Fair*: the algorithm made sure that the relative number of selected presented candidates was even across all tools; i.e., the same percentage of each tool's candidates was selected for presentation
- *Random*: the candidates were selected randomly from the tool's pool of candidates; the order of presentation of candidates was random, in other words, the human oracle could not infer from the order of presentation from which tool the candidate stemmed

The tools report their findings as *clone pairs* uniformly; clone pairs are two code fragments identified by their filename, starting and ending line. Both code fragments need to be at least 6 lines long to be considered.

The human oracle validated the clone pairs of the mentioned tools. Each clone pair suggested by a tool will be called a *candidate* and each clone pair validated and accepted by the human analyst will be called a *reference* in the following. The candidates examined but rejected are *false positives* and are used to measure precision. Those accepted (possibly with slight modification in range) form the reference corpus and are used to measure recall.

In the earlier experiment by Bellon, Stefan Bellon has taken the role of the human oracle. At least two percent of each tool's clone pairs have been judged by him. Although 2% sounds like a small fraction, in absolute terms 2% sums up to a large number for large systems because of the many clones reported. The validation of these fractions of candidates for all systems took him 77 h in total. Anticipating this problem in the design of the experiment, one evaluation was done after 1% of the candidates had been "oracled". Then another percent was "oracled". The interesting observation was that the relative quantitative results are almost the same.

Because we wanted to evaluate the candidates of the tools using the framework of the benchmark, we had to validate the new candidates. We developed a shared notion of what constitutes a clone, which we wrote down as guidelines for our oracling. Then one of the authors, Pierre Frenzel, validated the candidates of all tools. We selected only one human analyst to limit the chances of inconsistency of the human oracle (that could arise despite our common upfront guidelines) as much as possible.

Program	Domain	Language	SLOC	PTN	ASTN	T
`bison` 1.32	parser generator	C	19	165	39	68
`wget` 1.5.3	network downloader	C	16	130	33	83
`SNNS` 4.2	neural network simulator	C	105	1208	304	462
`PostgreSQL` 7.2	database	C	235	2434	497	809
`netbeans-javadoc` release 331	documentation system	Java	19	121	161	88
`eclipse-ant` date 15.2.02	build tool	Java	35	192	127	135
`eclipse-jdtcore` date 15.2.02	compiler	Java	148	1147	651	815
`j2sdk1.4.0-javax-swing`	GUI framework	Java	204	1193	1918	846

Fig. 21 Analyzed programs with different measures of size; *SLOC* source lines of code, *PTN* number of parse tree nodes, *ASTN* abstract syntax tree nodes, and *T* number of tokens; all figures are reported in the unit of *thousand*

Because Bellon validated only candidates of the tools of the earlier experiment and Frenzel validated only candidates from the new tools, we refrain from comparing our tools against the candidates validated by Bellon. As a consequence, we do not report the results of the other tools. The interested reader is referred to the paper by Bellon et al. (2007).

Subject Systems: The subject systems of this benchmark are listed in Fig. 21. The size of these systems is described by different measures in this table.

Some files of the Benchmark systems are generated files (like parsers) that we excluded from the benchmark because such code tends to be regular and appears as spurious clone candidates.

Frenzel spent about eleven working days on the validation to obtain the coverage shown in Fig. 22 for the evaluated tools. Despite considerable effort, we were able to validate only one C and one Java system, namely, `SNNS` and `eclipse-jdtcore`. These are the two second largest systems. We note that the earlier experiment by Bellon did not show any significant differences in terms of recall and precision of the tools among the systems; that is, they behaved consistently across different subject systems.

Although we are not able to present the charts for recall and precision for all systems of the Bellon benchmark due to lack of space, we will at least give runtime measurements for all systems demonstrating that our new tools ran successfully for all systems in Fig. 21.

Metrics: The Bellon benchmark comes with a set of tools to oracle and evaluate the clone detectors, which we reused. In order to compare candidates to references, a two-step process is used. First, the evaluation tool attempts to find a matching reference for each candidate. There are two types of matches. A *good match* is one in which reference and candidate overlap to at least 70% of their fragments.

	TSNS	AMN	ASN	PSN	PSP
`eclipse-jdtcore`	4477/2.0 %	322/2.0 %	446/2.0 %	652/2.0 %	698/2.1 %
`SNNS`	1767/2.0 %	234/2.0 %	358/2.0 %	205/2.0 %	256/2.1 %

Fig. 22 The absolute and relative numbers of seen candidates

The fragments need not be exactly the same because there were some off-by-one differences in the way code lines are reported by the tools. An *OK match* is one in which a candidate is contained to at least 70% of its lines in a reference or vice versa. In this evaluation, we will focus on only good matches due to reasons of space.

The match classifies candidates and references as follows. *Detected references* are those for which a good match exists. *Rejected candidates* are candidates for which no good match exists with any reference.

After matches are found, percentages as well as recall and precision are measured as follows where T is a variable denoting one of the participating tools, P is a variable denoting one of the analyzed programs, and τ is a variable denoting the clone type.

$$\text{Recall}(P, T, \tau) = \frac{|\text{DetectedRefs}(P, T, \tau)|}{|\text{Refs}(P, \tau)|}$$

$$\text{Rejected}(P, T, \tau) = \frac{|\text{RejectedCands}(P, T, \tau)|}{|\text{SeenCands}(P, T, \tau)|}$$

Yet, because we have only seen about 2% of all submitted clones for each tool and because there could be clones that no tool found, we do not know the complete set of clones to be found. As a consequence, these figures are to be interpreted accordingly.

Selected Features: Our infrastructure allows various compositions of clone detectors with different features. For our experiment, we are mostly interested in comparisons with respect to the following features:

1. use of AST suffix tree versus AST matching
2. parse tree versus AST
3. syntax versus lexical
4. parameterized versus non-parameterized

To address all these questions economically, we investigate only the combinations listed in Fig. 18. As already mentioned, TSNS has a postprocessing step to decompose clones that are not completely contained between corresponding brackets. A comparison of AMN and ASP addresses question (1) and one for PSN and ASN addresses question (2). Comparing TSNS to the other tools investigates question (3). The effect of parameterized versus non-parameterized, that is, question (4), can be seen by comparing PSP to PSN.

A comparison of TSNS and TSN would allow us to assess the difference in pure token-based analysis and token-based analysis with some attempt to find syntactic completeness. Yet, we do not evaluate TSP here because this question is not in the focus of this paper and it would have been too much effort. Our experimentation with TSNS and TSN showed that the latter produces about 2–3.5 times more candidates than TSNS. Nevertheless, we will report runtime measures for TSP as well. Moreover, Baker has commented on this difference in her assessment of the Bellon study (Baker 2007).

7.2 Results

This section discusses the above mentioned feature differences in terms of number of candidates generated by the tools (cf. Fig. 23a and 24a), recall (cf. Figs. 23c and

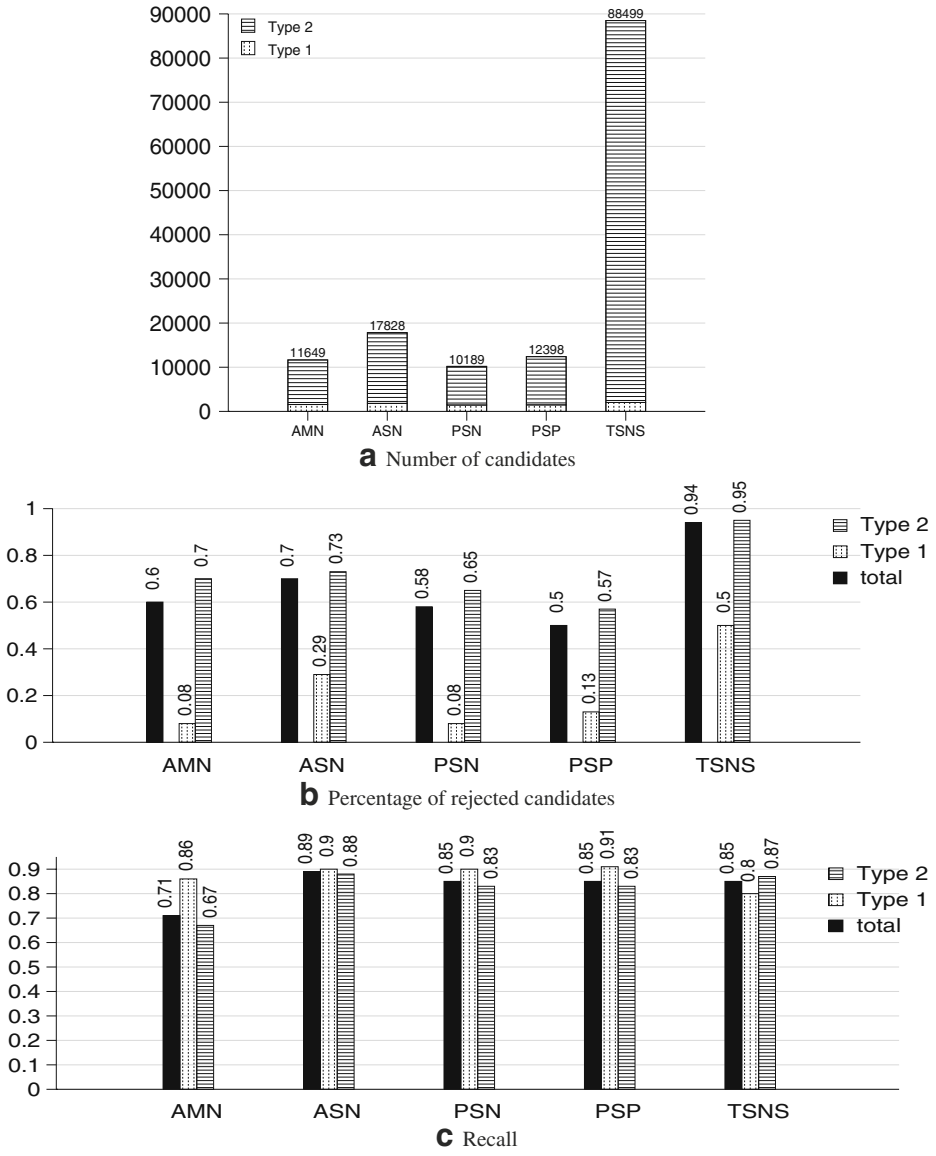


Fig. 23 Results for SNNS

24c), percentage of seen and rejected candidates (cf. Fig. 23b and 24b), and runtime measurements (cf. Fig. 26). We will briefly mention runtime performance in the course of the discussion of the above questions; a detailed discussion of runtime follows at the end of this section.

Number of Candidates. Figures 23a and 24a show the number of candidates generated by the tools. These two graphs confirm results from earlier experiments

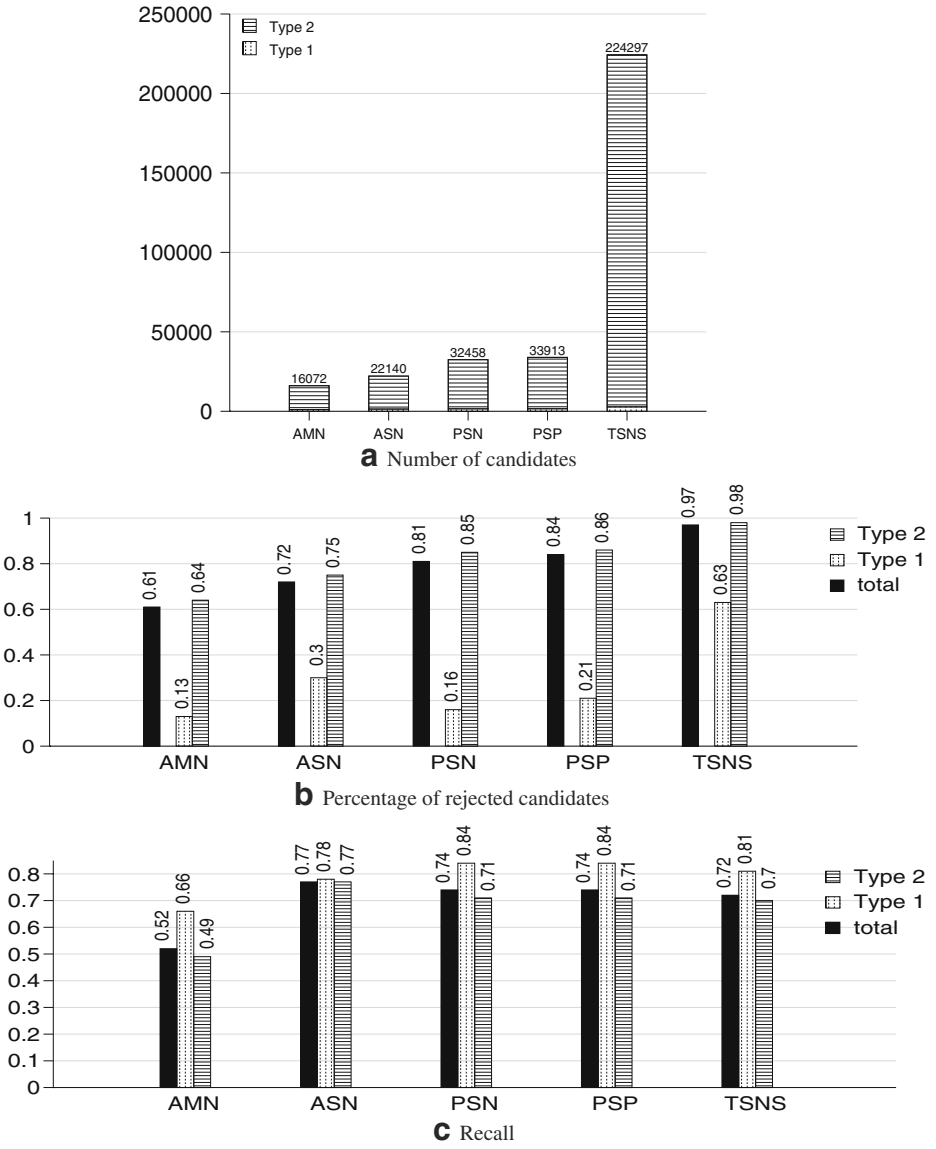


Fig. 24 Results for `eclipse-jdtcore`

that syntax-based tools yield substantially less clone candidates than token-based techniques, and the vast majority of candidates is reported as type-2 clones.

PSP finds more clones than PSN. On one hand, parameterized clone detection uses a stricter constraint on the token sequences and, hence, tends to exclude more sequences. On the other hand, longer sequences with inconsistent renaming that non-parameterized clone detection finds may contain shorter sequences with consistent renaming detected by parameterized detection. As a consequence, the number of found candidates may increase.

The two AST-based tools AMN and ASN yield fewer candidates for `eclipse-jdt-core` than for `SNNS` in relation to the other tools. The reason here is that declarative information is stored in symbol tables for the AST and not subject to clone detection. The Java system has many more declarations than the C system as it is based on many other libraries where only class declarations and not executable code is available.

The reason for the discrepancy in the number of clone candidates between PSN and ASN is a combination of our AST design and the programming style in which the two analyzed systems are written. The idea of our AST is to represent everything exectuable. The declarative code is not contained in the AST but in symbol tables, whereas the parse tree contains both declarative and executable statements. For this reason, ASN cannot find clones consisting of only parameter declarations spread over several lines, which are found by PSN in the Java system. On the other hand, variable declarations containing initializations are executable and are contained in the AST. If a fragment starts and ends with an initialization and in between contains only simple declarations and if this fragment is long enough, ASN may propose it as a clone even though the simple declarations in the midst of the fragment are different. We observed this phenomenon in the C system.

Use of AST Suffix Tree Versus AST Matching. AMN has a lower rejection rate than ASN for both subject systems; on the other hand, the recall of ASN is better. Moreover, ASN requires only half the time of AMN on average.

Parse Tree Versus AST. PSP/PSN have a lower rejection rate than the AST-based tools for the C system but a substantial higher rate for the Java system. This may have to do with characteristics particular to this system but is also because of `import` sequences in Java. Imports lead to type-2 clones, which were always rejected by our human analyst. The AST-based tools are not affected because import statements are represented only as symbolic information and are not part of the AST; that is, the AST-based tools do not find import sequences as clones.

Import sequences have less effect on rejection rate for token-based tools even though token-based tools should find the same import sequences as parse-tree based tools. The reason is as follows. The number of import sequences is limited by the number of files, and token-based tools find many more additional spurious clones. As a consequence, the likelihood that the random selection process picks an import sequence from the candidate set of token-based tools is lower than for parse-tree based tools. Consequently, the likelihood that an import sequence is among the rejected candidates decreases.

The counterpart of `import` in C is the preprocessor directive `include`. Yet, all syntax-based tools analyze the preprocessed code where the `include` statements are already resolved.

In terms of runtime performance, PSP/PSN lie in between ASN and AMN for the C system, but in case of the Java system, PSP/PSN is faster than ASN and AMN by a factor of 2 and 3.8, respectively.

Syntax Versus Lexical. Overall we see a much higher rejection rate for token-based techniques even though TSNS attempts to find syntactically complete clones, which indicates that the simple heuristic of using brackets as syntactic delimiters is not sufficient. The high rejection rate for TSNS questions its practical use where high precision is required.

Often, one needs to make a trade-off between recall and precision; less precision yields often better recall in many applications. Interestingly, the recall of lexical clone detection is not higher than those of syntactic analyses even though the precision is lower.

Parameterized Versus Non-parameterized. The difference of parameterized and non-parameterized detection is visible by comparing PSP with PSN for the rejection rate. Small differences may exist because of the incomplete and random candidate selection. In only one case, there is a difference between the rejection rates above 5% at the advantage of parameterized detection.

The computational effort of parameterized clone detection is lower than for non-parameterized clone detection as can be seen by comparing phase *construct* in Fig. 25, the phase that constructs the suffix tree. Although both algorithms have linear asymptotic time complexity, McCreight's algorithm (upon which Baker's algorithm is based) is slightly faster than Ukkonen's algorithm in practice as is generally known. Yet, this part of the detection process is negligible. The overall difference in performance is due to different numbers of detected clones.

Runtime Comparison. The runtime for each tool is given in Fig. 25, determined on a 64bit Intel architecture with four processors (3.0 GHz) and 16 GB RAM running Linux (Suse), where only one CPU was used. The runtime for the AST-based tools contains loading the AST from disk; i.e., the time excludes parsing. The time to parse the sources and to generate our unified AST took 31 minutes for `eclipse-jdtcore` and 5 minutes for `SNNS`. The time for the token-based tools contains reading and tokenizing source text.

The AST generation of `eclipse-jdtcore` took so much longer because of the many libraries upon which `eclipse-jdtcore` depends. Even though these libraries are precompiled to byte code, their declarations at least are represented in the AST.

The runtime of AMN is dominated by comparing the abstract syntax subtrees, removing trees that are subsumed by larger clones, and to some extent the combination of found ASTs into sequences. As already mentioned, AMN provides a parameter to limit runtime by ignoring overly large buckets. The chosen threshold for bucket size in the experiment was 400. To see the effect on different values on runtime, we measured the number of necessary comparisons depending upon different thresholds. Figure 26 lists these values. If the cutoff threshold were increased to 1,000, phase *comparing* would need about three times longer. If it were infinite, it would need 16 times longer for `SNNS` and 102 times longer for `eclipse-jdtcore`.

The techniques based on suffix tree are dominated by the phases *find*, F_{sub}, and *decompose*. Phase *find* detects clones in the suffix tree (which requires only linear

Fig. 25 Runtime by phases in seconds. The phases are: "token" for token generation (includes lexical analysis for token-based, and both lexical and syntactic analysis for syntax-based techniques), "construct" the construction of the suffix-tree, "find" finding clones in the suffix-tree, subsumption filter F_{sub}, decomposing into syntactic clones (based on scopes or on syntactic information), "loading" the tree, "hashing" the nodes, "comparing" subtrees in the buckets, "removing" subclones, "splitting" into clone lists, "finding clone sequences", "others" operations like printing, "all" the total runtime ▶

Tool	Phase	bison	wget	snns	postgresql	javadoc	ant	jdtcore	swing
TSP	token	0.2	0.2	1.8	3.2	0.3	0.5	2.8	3.7
	construct	0.2	0.2	1.2	2.3	0.2	0.4	2.3	2.4
	find	1.3	1.2	23.6	64.0	1.5	3.0	42.9	48.0
	F_{sub}	0.0	0.0	3.2	8.3	0.0	0.4	4.5	37.9
	others	0.0	0.0	3.2	7.8	0.1	0.7	5.3	25.3
	all	**1.7**	**1.6**	**32.9**	**85.7**	**2.2**	**5.1**	**57.9**	**117.3**
TSNS	token	0.2	0.3	2.1	3.4	0.4	0.6	3.2	3.6
	construct	0.2	0.2	1.2	2.3	0.2	0.4	2.7	2.3
	find	1.2	1.2	21.1	57.7	1.5	3.4	38.4	45.7
	F_{sub}	0.0	0.0	3.4	10.5	0.1	0.5	5.0	39.9
	decompose	0.1	0.1	34.9	105.0	0.2	1.8	38.0	187.0
	others	0.0	0.0	2.0	5.8	0.1	0.5	4.0	11.8
	all	**1.8**	**1.7**	**64.7**	**184.8**	**2.5**	**7.0**	**91.2**	**290.4**
TSPS	token	0.3	0.3	2.1	3.4	0.3	0.6	3.1	3.9
	construct	0.2	0.2	1.2	2.2	0.2	0.3	2.3	2.4
	find	1.2	1.2	23.2	64.4	1.5	3.0	42.6	47.7
	F_{sub}	0.0	0.0	3.2	8.3	0.0	0.4	4.5	37.7
	decompose	0.1	0.0	32.6	78.4	0.2	1.6	31.2	175.4
	others	0.0	0.0	1.8	4.5	0.1	0.4	4.2	10.5
	all	**1.8**	**1.8**	**64.1**	**161.3**	**2.3**	**6.4**	**87.9**	**277.5**
PSN	token	1.6	1.3	13.4	29.8	0.8	1.3	9.7	10.3
	construct	0.4	0.4	7.4	9.2	0.3	0.6	3.4	3.6
	find	1.6	1.2	24.0	66.4	1.6	2.8	36.0	42.9
	F_{sub}	0.0	0.0	24.0	49.6	0.1	0.7	9.7	90.0
	decompose	0.0	0.0	4.1	16.0	0.1	0.3	4.8	16.7
	others	0.0	0.0	0.4	0.7	0.0	0.1	0.7	0.7
	all	**3.7**	**2.8**	**73.4**	**171.7**	**3.0**	**5.7**	**64.3**	**164.2**
PSP	token	1.6	1.2	13.4	29.5	0.8	1.3	9.5	10.8
	construct	0.4	0.4	6.2	8.1	0.3	0.6	3.6	3.8
	find	1.6	1.1	24.0	64.4	1.6	2.7	34.4	42.4
	F_{sub}	0.0	0.0	23.8	49.0	0.0	0.6	8.5	88.2
	decompose	0.0	0.0	3.8	1167.2	0.0	0.2	4.0	15.0
	others	0.0	0.0	0.5	12.5	0.0	0.0	0.6	0.5
	all	**3.7**	**2.8**	**71.6**	**1330.8**	**2.8**	**5.5**	**60.6**	**160.8**
ASN	token	1.2	1.0	7.9	14.8	9.6	7.8	36.8	64.7
	construct	0.1	0.1	0.8	1.3	0.4	0.3	1.9	6.0
	find	1.2	1.0	25.6	56.4	4.8	4.5	65.2	267.3
	F_{sub}	0.0	0.0	0.7	2.5	1.6	1.1	14.2	797.8
	decompose	0.0	0.0	2.2	1.5	7.6	0.6	7.4	92.4
	others	0.0	0.0	0.7	0.3	1.7	0.2	1.4	7.9
	all	**2.6**	**2.2**	**38.0**	**76.8**	**25.7**	**14.4**	**126.9**	**1236.0**
AMN	loading	0.3	0.3	2.4	5.0	6.2	5.8	25.9	32.0
	hashing	0.7	0.7	5.5	8.7	2.3	1.8	9.2	25.2
	comparing	3.1	3.9	49.8	49.9	28.9	31.7	106.4	340.1
	removing	0.2	0.6	10.0	14.1	41.6	52.4	56.2	572.8
	splitting	1.3	1.7	13.6	11.4	6.0	12.2	19.0	55.6
	finding	1.0	2.0	12.4	14.9	1.1	2.2	9.4	6.1
	others	0.2	0.0	0.6	2.7	0.0	0.0	3.1	0.8
	all	**6.8**	**9.2**	**94.4**	**106.8**	**86.2**	**106.1**	**229.1**	**1032.6**

Fig. 26 Number of comparisons (in thousands) in relation to bucket size limit

SNNS		eclipse-jdtcore	
cutoff	# comparisons	cutoff	# comparisons
100	485	100	1,321
400	2,335	400	5,225
1,000	8,773	1,000	17,534
∞	38,494	∞	534,742

time), checks them for length using filter F_{length} (see Section 6.1), and then puts them in the clone pair list.

The filter F_{length} that checks whether a clone is long enough is quite expensive. The most expensive operation for this check is the determination of the line number for a token. Although this operation is trivial at first sight, the number of corresponding assembler instructions is 40 and relatively high. Because the token table is a very large array but the accesses to this array are not local, many cache misses can be expected. What makes it worse is the fact that this operation is called very often.

The line test in ASN is even more demanding. For all other tools it is sufficient to retrieve the line information for the first and last token in the sequence because the tokens appear in the original order for these tools. Because we use various normalizations in our AST, the AST node sequence is not necessarily monotonic so that we need to check each element of the sequence, which handicaps ASN.

A simpler test can be made based on number of tokens rather than lines because the length of a sequence is stored and can be accessed without further indirections. To check whether this alternative test makes a difference in runtime, we ran the analysis once using the token length test for all subject systems. We used 60 tokens as the lower threshold based on the rule of thumb that one line typically has about 10 tokens and the line-based criterion was minimal 6 lines. The improvement for all tools but ASN was about 30% saved effort only for the filtering step. In case of ASN, the savings in effort was a factor between 2 and 3.5.

Filter F_{sub} checks whether clones subsume each other. Only the larger clones should be reported. This check is performed right after the clones are retrieved from the suffix tree as early as possible. It is repeated after clone sequences are decomposed into syntactically complete sequences, but the costs of this call to F_{sub} are accounted in phase *decompose*. The exceptional high costs for filter F_{sub} that tests for clone subsumption for the tool ASN in the case of `j2sdk1.4.0-javax-swing` is a result of the many clones it finds. The number of clones reaches about 18 millions at this stage and is extraordinarily high.

Phase *decompose* decomposes the clone sequences into syntactically complete subsequences. At first sight, the costs of this step contradicts its theoretical linear time complexity. We note that the runtime for this step subsumes other necessary activities. Two other activities filter by subsumption and length once more. Another one is imposed by the layout of the experiment. The benchmark requires us to present an exhaustive list of clone pairs (i.e., also transitive clones need to be reported because the oracle is based on clone pairs). To achieve this goal, the additional activity forms the equivalence classes for clone pairs after clone sequences have been decomposed. It is necessary because decomposed clones become shorter and may fall in the same equivalence with other clones that were decomposed from

completely different clones. Consequently, an expensive pairwise comparison is used to collect all clones for all equivalence classes.

Threats to Validity: There are potential threats to validity in our study that we will discuss in this section. One major threat is the subjectivity of the human validation of clones. The content of the benchmark is the result of the judgement of one person, and, hence, the results depend upon his judgement. At least, we have developed common guidelines for the oracle. Moreover, because the benchmark is publicly available, his judgement can be inspected.

Because one of the authors served as human oracle evaluating our own tools, one could suspect a bias for a particular tool. However, our experimental design is using a selection process which is automated, blind, fair, and random (see Section 7.1). These characteristics should exclude the potential bias. Moreover, we reused this experimental design from an earlier experiment by Bellon et al. (2007) that was conducted at a time when we did not develop clone detection tools ourselves. Hence, the experimental design was not geared towards favoring one of our own tools.

A further threat to validity relates to the selection of sample systems. Although we chose two systems of different languages, size, and application domain, we do not know whether these systems are a representative sample. In particular, all systems are open source. Closed-source systems or systems written in older programming languages such as COBOL might have totally different characteristics.

Because we did not validate all candidates, we rely on whether those candidates we have seen are representative for a tool. Yet, we have chosen the candidates by an automated random selection process and evaluated many candidates at least in absolute terms so that the threat of a non-representative selection should be limited. Moreover, not the absolute numbers of precision and recall should be considered, but the relative differences among the tools.

We must also note that there is some room for accidental differences in this evaluation. Because we were not able to validate all candidates and the candidates were selected randomly, there is a chance that this random selection picked the "best" clones from one tool and the "worst" clones from another tool. The chances decrease by the number of candidates picked of course, but a few percentages might be accounted on this random selection.

We should also be aware that we are comparing primarily tools. The same approach can be implemented differently affecting in particular required runtime resources. We note that we have put in considerable engineering effort to optimize our tools so that our measurements become meaningful. Some of the details of the original methods are not specified, such as thresholds or what to consider a parameter (every type of identifier, literals, or even operators?). Fixing these degrees of freedom in a concrete implementation may affect the results.

8 Conclusions

This paper has described a way to use suffix-tree based clone detection for syntax trees. We compared different techniques empirically addressing the different choices of features in clone detection:

Use of AST Suffix Tree Versus AST Matching: Detecting cloned subtrees in ASTs via suffix trees has the same detection quality as AST matching by and large. AST

matching has a somewhat lower rejection rate, but also a lower recall. Detection based on suffix trees turned out to be 60–80% faster. We note that the suffix tree performs a complete search whereas our AST-based technique uses a heuristic to limit the inherent quadratic complexity at the cost of potential loss of clones; if a full search is performed, the AST match is drastically slower.

Parse Tree Versus AST: Parse trees are easier to obtain than ASTs but have typically more tokens. In terms of recall, precision, and runtime for detection, they are comparable to detection based on ASTs.

Syntax Versus Lexical Analysis: Our experiment has confirmed the earlier result that token-based techniques tend to have lower precision. Unlike earlier experiments, however, they did not have higher recall in our study. Also in terms of runtime, there is no longer an argument for token-based techniques as the same linear-time search algorithm in those techniques can be used for tree-based techniques as well. The real advantage of a token-based technique is that it is easier to implement a lexer than a parser in general, that token-based program representation requires less space than ASTs, that preprocessor directives are no burden, and that the code does not even need to be complete or correct to conduct an analysis. The advantage of syntactic techniques, on the other hand, is its ability to find syntactic clones. The lightweight approach—checking for balanced brackets as syntactic delimiters integrated in the token-based analysis—could not compete with the syntactic approach in this regard. Another advantage of the syntactic approach is the ability to write syntactic filters to ignore syntactic structures of little interest. We used this ability prototypically for long array initializations, for instance. During oracling we detected many more opportunities for such filters, as for instance ignoring long import statement sequences.

Parameterized Versus Non-parameterized: Although in some cases, checking for consistent renaming helped to improve precision, the results were not as good as expected. We speculate that the longer the clones, the less important this test for parameterized clones becomes. Longer equal statement sequences are very often actual clones. It is unlikely that somebody has written a syntactically identical long fragment independently. If someone has copied and pasted a fragment, then the parameters are consistent in most cases. For error detection, it would even be a disadvantage to insist on parameterized clones because inconsistencies in parameter renaming would go unnoticed.

Our experiment showed that overall the acceptance for type-1 clones is much higher than those for type-2 clones. By definition, type-2 clones are more dissimilar than type-1 clones. Often we find common structurally equivalent statements (e.g., sequences of assignments) which refer to totally different concepts and, hence, are just spurious clones (e.g., the assignments assign very different variables which have nothing in common with the variables in the other fragment). Because most clones are of type 2, it would be helpful if we had a metric to measure the similarity between two type-2 segments. The metric could be used to rank the candidates. This metric could be based on the length, the diversity of identifiers between the two code segments, and the general frequency of the other constructs that occur. Shannon information content (Shannon 1984) would be a candidate to measure how significant syntactic structures are. Type information would also be helpful, which is currently

not used by clone detectors. For instance, a short sequence of assignments occurs frequently in a program and is less likely a clone if the identifiers and their types differ a lot.

Because no tool offers both high recall and precision, one must make a choice. The choice for a tool with either high recall or high precision depends upon a concrete task. In case of automated clone removal, one would select a precise clone detector. If one searches for copies of a particular piece of code that needs to be changed, one would likely prefer a tool with high recall instead.

Another point of improvement that relates to the benchmark is to use token counts instead of lines as a measure of clone size. We often found clones that contained two statements separated by several blank or commented lines. Last but not least, the benchmark should be extended to capture who rated a clone (at varying levels of confidence, not just binary) and to allow for inter-rater comparisons. Jim Cordy proposed an experiment in which different raters validate the same candidates[7]. If a tool's discrepancy of proposed and accepted candidates with respect to every rater is not greater than the inter-rater difference, it can be considered useful. We plan to conduct such multi-rater experiments.

Acknowledgements We would like to thank Stefan Bellon for providing us with his benchmark and CCDIML, his support in the evaluation, and for comments on this paper. We also like to thank Felix Beckwermert for his support in the evaluation and Thilo Mende for comments on this paper. We would also like to thank the anonymous reviewers for their valuable comments.

References

Antoniol G, Casazza G, Penta MD, Merlo E (2001) Modeling clones evolution through time series. In: International conference on software maintenance. IEEE CS Press, pp 273–280

Antoniol G, Villano U, Merlo E, Penta M (2002) Analyzing cloning evolution in the linux kernel. Inf Softw Technol 44(13):755–765

Bailey J, Burd E (2002) Evaluating clone detection tools for use during preventative maintenance. In: SCAM

Baker BS (1992) A program for identifying duplicated code. In: Computer science and statistics 24: Proceedings of the 24th symposium on the interface

Baker BS (1995) On finding duplication and near-duplication in large software systems. In: Working conference on reverse engineering. IEEE CS Press

Baker BS (1996) Parameterized pattern matching: algorithms and applications. JCSS

Baker BS (2007) Finding clones with Dup: analysis of an experiment. IEEE Trans Softw Eng 33(9):608–621 (September)

Baker BS, Giancarlo R (2002) Sparse dynamic programming for longest common subsequence from fragments. J Algorithms 42(2):231–254 (February)

Balazinska M, Merlo E, Dagenais M, Lague B, Kontogiannis K (1999) Measuring clone based reengineering opportunities. In: IEEE symposium on software metrics. IEEE CS Press, pp 292–303 (November)

Balazinska M, Merlo E, Dagenais M, Lague B, Kontogiannis K (2000) Advanced clone-analysis to support object-oriented system refactoring. In: Working conference on reverse engineering, IEEE CS Press, pp 98–107 (October)

Baxter ID, Yahin A, Moura L, Sant'Anna M, Bier L (1998) Clone detection using abstract syntax trees. In: ICSM

Bellon S (2002) Vergleich von Techniken zur Erkennung duplizierten Quellcodes. Master's thesis, University of Stuttgart, Germany

[7]Personal communication at Dagstuhl, July 2006.

Bellon S, Koschke R, Antoniol G, Krinke J, Merlo E (2007) Comparison and evaluation of clone detection tools. IEEE Trans Softw Eng 33(9):577–591 (September)
Bentley JL, Ottmann TA (1979) Algorithms for reporting and counting geometric intersections. IEEE Trans Comput C-28:643–647
Bruntink M, van Deursen A, Tourwe T, van Engelen R (2004) An evaluation of clone detection techniques for crosscutting concerns. In: International conference on software maintenance, pp 200–209
Bruntink M, van Engelen R, Tourwe T (2005) On the use of clone detection for identifying crosscutting concern code. IEEE Trans Softw Eng 31(10):804–818
Chou A, Yang J, Chelf B, Hallem S, Engler DR (2001) An empirical study of operating system errors. In: Symposium on operating systems principles, pp 73–88
Cordy JR (2003) Comprehending reality—practical barriers to industrial adoption of software maintenance automation. In: International workshop on program comprehension, IEEE CS Press
Cordy JR, Dean TR, Synytskyy N (2004) Practical language-independent detection of near-miss clones. In: CASCON, IBM Press
Di Lucca G, Di Penta M, Fasolino, A (2002) An approach to identify duplicated web pages. In: COMPSAC
Ducasse S, Rieger M, Demeyer S (1999) A language independent approach for detecting duplicated code. In: ICSM
Fowler M (1999) Refactoring: improving the design of existing code. Addison Wesley, Boston, MA, USA
Gitchell D, Tran N (1999) Sim: a utility for detecting similarity in computer programs. In: SIGCSE, ACM Press
Godfrey M, Tu Q (2001) Growth, evolution and structural change in open source software. In: Workshop on principles of software evolution (September)
Higo Y, Ueda Y, Kamiya T, Kusumoto S, Inoue K (2002) On software maintenance process improvement based on code clone analysis. In: International conference on product focused software process improvement. Lecture notes in computer science, vol 2559. Springer
Johnson JH (1993) Identifying redundancy in source code using fingerprints. In: CASCON, IBM Press
Kamiya T, Kusumoto S, Inoue K (2002) CCFinder: a multi-linguistic token-based code clone detection system for large scale source code. IEEE Trans Softw Eng 28(7):654–670
Kapser C, Godfrey M (2003a) A taxonomy of clones in source code: the reengineers most wanted list. In: Working conference on reverse engineering. IEEE CS Press
Kapser C, Godfrey MW (2003b) Toward a taxonomy of clones in source code: a case study. In: Evolution of large scale industrial software architectures
Kapser C, Godfrey M (2005) Improved tool support for the investigation of duplication in software. Proceedings of the 21st IEEE international conference on software maintenance
Kapser C, Godfrey MW (2006) "Clones considered harmful" considered harmful. In: Working conference on reverse engineering
Kim M, Bergman L, Lau T, Notkin D (2004) An ethnographic study of copy and paste programming practices in OOPL. In: International symposium on empirical software engineering. IEEE CS Press, pp 83–92
Kim M, Sazawal V, Notkin D, Murphy GC (2005) An empirical study of code clone genealogies. In: European software engineering conference and foundations of software engineering (ESEC/FSE)
Komondoor R, Horwitz S (2001) Using slicing to identify duplication in source code. In: Proc. int. symposium on static analysis
Kontogiannis K, Mori RD, Merlo E, Galler M, Bernstein M (1996) Pattern matching for clone and concept detection. Autom Softw Eng 3(1/2):79–108
Koschke R, Girard JF, Würthner M (1998) Intermediate representations for reverse engineering. In: Working conference on reverse engineering. IEEE CS Press, pp 241–250
Koschke R, Falke R, Frenzel P (2006) Clone detection using abstract syntax suffix trees. In: Working conference on reverse engineering. IEEE CS Press, pp 253–262
Krinke J (2001) Identifying similar code with program dependence graphs. In: WCRE
Lague B, Proulx D, Mayrand J, Merlo E, Hudepohl J (1997) Assessing the benefits of incorporating function clone detection in a development process. In: International conference on software maintenance, pp 314–321
Laguë B, Proulx D, Mayrand J, Merlo EM, Hudepohl J (1997) Assessing the benefits of incorporating function clone detection in a development process. In: ICSM

Lanubile F, Mallardo T (2003) Finding function clones in web applications. In: Conference on software maintenance and reengineering
Leitao AM (2003) Detection of redundant code using R2D2. In: Workshop source code analysis and manipulation. IEEE CS Press
Li Z, Lu S, Myagmar S, Zhou Y (2004) Cp-miner: a tool for finding copy-paste and related bugs in operating system code. In: Operating system design and implementation, pp 289–302
Li Z, Lu S, Myagmar S, Zhou Y (2006) Copy-paste and related bugs in large-scale software code. IEEE Trans Softw Eng 32(3):176–192 (March)
Manber U, Myers G (1991) Suffix arrays: a new method for on-line string searches. SIAM J Comput 22(5):935–948 (October)
Marcus A, Maletic J (2001) Identification of high-level concept clones in source code. In: Conference on automated software engineering
Mayrand J, Leblanc C, Merlo EM (1996) Experiment on the automatic detection of function clones in a software system using metrics. In: ICSM. IEEE Computer Society Press
McCreight E (1976) A space-economical suffix tree construction algorithm. J ACM 32(2): 262–272
Monden A, Nakae D, Kamiya T, Sato S, Matsumoto K (2002) Software quality analysis by code clones in industrial legacy software. In: IEEE symposium on software metrics, pp 87–94
Prechelt L, Malpohl G, Philippsen M (2000) Jplag: finding plagiarisms among a set of programs. Technical report, University of Karlsruhe, Department of Informatics
Rieger M (2005) Effective clone detection without language barriers. Dissertation, University of Bern, Switzerland
Schleimer S, Wilkerson DS, Aiken A (2003) Winnowing: local algorithms for document fingerprinting. In: Proceedings of the SIGMOD, pp 76–85
Shannon CE (1984) A mathematical theory of communication. Bell Syst Tech J 27(379–423): 623–656
Ukkonen E (1995) On-line construction of suffix trees. Algorithmica 14(3):249–260
Van Rysselberghe F, Demeyer S (2004) Evaluating clone detection techniques from a refactoring perspective. In: Conference on automated software engineering
Walter V, Seipel D, von Gudenberg JW, Fischer G (2004) Clone detection in source code by frequent itemset techniques. In: Workshop source code analysis and manipulation
Yang W (1991) Identifying syntactic differences between two programs. Software Pract Ex 21(7): 739–755

Raimar Falke received his degree in computer science from the Technische Universität Dresden in Germany. He is currently working at the University of Bremen in Germany. His research includes clone detection and application, program analysis and binary reverse engineering. His Ph.D. thesis will be about clone detection.

Pierre Frenzel received his degree in computer science from the University of Bremen in Germany. His current research includes architecture recovery, feature location and clone detection for product lines. His Ph.D. will be about the consolidation of different variants to a product line.

Rainer Koschke is a professor for software engineering at the University of Bremen in Germany. His research interests are primarily in the fields of software engineering and program analyses. His current research includes architecture recovery, feature location, program analyses, clone detection, and reverse engineering. Rainer Koschke received his degree in computer science from the University of Stuttgart and holds a doctoral degree in computer science from the University of Stuttgart, Germany.