Clone Detection Using Abstract Syntax Suffix Trees

Raimar Falke, Pierre Frenzel, Rainer Koschke
University of Bremen, Germany
http://www.informatik.uni-bremen.de/st/
{rfalke,saint,koschke}@informatik.uni-bremen.de

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 extremely fast but yields clone candidates that are often no syntactic units [26]. 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 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 several large case studies in which we empirically compare the new technique to other techniques using the Bellon benchmark for clone detectors.

1 Introduction

It is still a common habit of programmers to reuse code through copy&paste. Albeit copy&paste is an obvious strategy of reuse and avoidance of unwanted side effects if a programmer does not oversee the consequences of a change to an existing piece of code, this strategy is only a short-term win. The interests of these strategies must be paid back later by increased maintenance through replicated changes in all copies if the original code must be corrected or adapted.

Although some researchers argue not to remove clones because of the associated risks [6], 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 also useful to monitor development in order to stop the increase of redundancy before it is too late.

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 extremely fast but yields clone candidates that are often no syntactic units. Current techniques based on abstract syntax trees (AST)—on the other hand—find syntactic clones but are considerably less efficient.

There are additional reasons for 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 can often not be avoided. From a research point of view, it would be also interesting to categorize and see where the redundancy occurs mostly in syntactic terms [14]. 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 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 several large case studies in which we empirically and quantitatively compare the new technique to nine 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.

Overview. The remainder of this paper is organized as follows. Section 2 summarizes related research. In particular, this section describes clone detection based on suffix trees and ASTs in detail as they form the foundation of our new technique. Section 3 introduces the new technique. In Section 4, we compare the new technique to other techniques based on the Bellon benchmark for clone detectors.

2 Related Research

Software clone detection and removal is an active field of research. This section summarizes research in clone detection.

The foremost question to answer is "What is a clone?" Generally speaking, two code fragments form a clone pair if they are similar enough according to a given definition of similarity. Different definitions of *similarity* and associated levels of tolerance allow for different kinds and degrees of clones.

Ideally, code is free of redundancy. A piece of code, A, is redundant if there is another piece of code, B, that subsumes the functionality of A, in other words, they have "similar" pre and post conditions. We call such a pair (A, B) a *semantic* clone. Unfortunately, detecting semantic clones is undecidable in general.

Another definition of similarity considers the program text: Two code fragments form a clone pair if their program text is similar. The two code fragments may or may not be equivalent semantically. These kinds of clones are most often the result of *copy&paste*; that is, the programmer selects a code fragment and copies it to another location.

Clones of this nature may be compared on the basis of the program text that has been copied. We can distinguish the following types of clones:

- **Type 1** is an exact copy without modifications (except for whitespace and comments).
- **Type 2** is a syntactically identical copy; only variable, type, or function identifiers have been changed.
- **Type 3** is a copy with further modifications; statements have been changed, added, or removed.

Several techniques have been proposed to find these types of clones.

Textual comparison: the approach by Rieger et al. compares whole lines to each other textually [9]. To increase performance, lines are partitioned using a hash function for strings. Only lines in the same partition are compared. The result is visualized as a dotplot, where each dot indicates a pair of cloned lines. Clones may be found as certain patterns in those dotplots visually. Consecutive lines can be summarized to larger cloned sequences automatically as uninterrupted diagonals or displaced diagonals in the dotplot.

Johnson [12] uses the efficient string matching by Karp and Rabin [15] based on fingerprints.

Token comparison: Baker's technique is also a line-based comparison. Instead of a string comparison, the token sequences of lines are compared efficiently through a suffix tree. First, each token sequence for whole lines is summarized by a so called *functor* that abstracts of concrete values of identifiers and literals.

The functor characterizes this token sequence uniquely. Assigning functors can be viewed as a perfect hash function. Concrete values of identifiers and literals are captured as parameters to this functor. 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[1] 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 2.1.

Kamiya et al. increase recall for superfluous different, yet equivalent sequences by normalizing the token sequences [13].

Because syntax is not taken into account, the found clones may overlap different syntactic units, which cannot be replaced through functional abstraction. Either in a preprocessing [7, 10] or post-processing [11] 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 [20, 18, 24, 17]. An allowable distance (for instance, Euclidean distance) for these metric vectors can be used as a hint for similar code. Specific metric-based techniques were also proposed for clones in web sites [8, 21].

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) [4]. A similar approach was proposed earlier by Yang [30] using dynamic programming to find differences between two versions of the same file.

Comparison of program dependency graphs: control and data flow dependencies of a function may be represented by a program dependency graph; clones may be identified as isomorphic subgraphs [19, 16]; because this problem is NP hard, Krinke uses approximative solutions.

Other techniques: Marcus and Maletic use latent semantic indexing (an information retrieval technique) to identify fragments in which similar names occur [23].

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

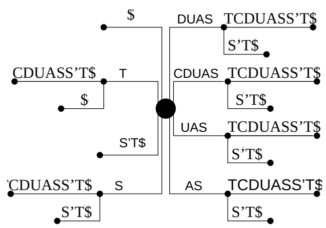

Figure 1. Suffix Tree for *CDUASTCDUASS'T$*; the large dot is the root

Leitao [22] 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 functions yields an evidence that is summarized in an evidence-factor model yielding a clone likelihood. Walter et al. [29] 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.

2.1 Token-Suffix-Tree based Clone Detection

Very efficient token-based clone detection is based on suffix trees. Suffix trees have been originally used for efficient string search [25]. Later, Brenda Baker has extended the original algorithm to parameterized strings for clone detection [2]. Baker's approach offers the advantage of finding cloned token sequences with consisting renaming of parameters (variables and literals can be treated as parameters). To simplify the description, however, we prefer to describe the original string-based approach as follows.

We will use the following string as an example (two concatenated titles of research papers on clone detection):

Clone Detection Using Abstract Syntax Trees Clone Detection Using Abstract Syntax Suffix Trees

A suffix tree is a representation of a string as a trie where every suffix is presented through a path from the root to a leaf. The edges are labeled with the substrings. Paths with common prefixes share an edge. Suffix trees are linear in space with respect to the string length (the edge labels are stored as start and end token of a substring in an efficient implementation) and there are linear algorithms to compute them [25, 28].

The suffix tree for our running example is shown in Figure 1 where we use the first letter as an abbreviation for the words in the text (e.g., *A* for *Abstract*) and *S'* denotes *Suffix*. That is, we construct the suffix tree for the string *CDUASTCDUASS'T$*. The unique character *$* denotes the end of the string.

A clone can be identified in the suffix tree as an inner node. The length is the number of characters from the root to this inner node. The number of occurrences of the clone is the number of the leaves that can be reached from it. For instance, *CDUAS* occurs twice and has length 5 and *AS* occurs twice, too, but has length 2.

As shown in the suffix tree, there are six clones in the text, but we notice, too, that all of them except *T* are suffixes of the longest one, namely, *CDUAS*. Baker describes an algorithm to determine the maximal clone sequences in the tree efficiently [3] .

A filter on minimal length can be used to exclude irrelevant clones such as *T*. In summary, we detect that the string "Clone Detection Using Abstract Syntax" occurs twice.

For computer programs, we apply this kind of clone detection to the tokens of the program.

2.2 AST-based Clone Detection

Baxter et al. have proposed a clone detection technique based on AST. To find clones in the AST, we need to compare each subtree to each other subtree in the AST in principal. 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.

2.3 Token based versus AST based

The suffix-tree-based analysis offers several advantages over other techniques. It scales very well because of its linear complexity in both time and space, which makes it very 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 very short time [27]. A scanner for a programming language is typically developed in one or two days. As opposed to text-based techniques, this 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 is 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/Koschke [5, 26] and Bailey/Burd [1] 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, which is not true for token-based candidates as they often do not form syntactic units. For instance, the two program snippets left and right in Listing 1 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 rightful clones, a maintenance programmer would hardly consider this finding useful.

```
 return result;       return x; }
}
int foo() {          int bar () { int y;
  int a;
```

Listing 1. Spurious clones

Syntactic clones can be found to some extent by token-based techniques if the candidate sequences are split in 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 Listing 1. However, programming languages do have many types of delimiting tokens beyond brackets. The `if`, `then`, `else`, and `end if` all constitute syntax delimiters. In particular, `end if` is an interesting example as two consecutive tokens form one delimiter, of which both 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, which we already mentioned in the introduction.

Unfortunately, Baxter's technique did not match up with the speed of token-based analysis, although inherent parallelism was leveraged. 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.

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.

3 Approach

The algorithm consists of the following steps:

1. parse program and generate AST
2. serialize AST
3. apply suffix tree detection
4. decompose resulting cloned token sequence into complete syntactic units

Step (1) is a standard procedure which will not be discussed further. Step (3) has been described in Section 2.1. We will primarily explain step (4) step-by-step. We will first explain the serialization of the AST and then present the algorithm to cut the cloned token sequence into syntactic units.

3.1 Serializing the AST

We will use the example in Listing 2 as an example to illustrate the algorithm. The AST corresponding to Listing 2 is shown in Figure 2.

```
if x + y then a := i; else foo; end if;
if p     then a := j; else foo; end if;
if q     then z := k; else bar; end if;
```

Listing 2. Sequence of if statements in Ada

Because the token-based clone detection is based on token sequences, we need to serialize the AST nodes. We serialize the AST by a preorder traversal. For each visited AST node N, we emit N as root and associate the number of arguments (number of AST nodes transitively derived from N) with it (in the following, presented as subscript).

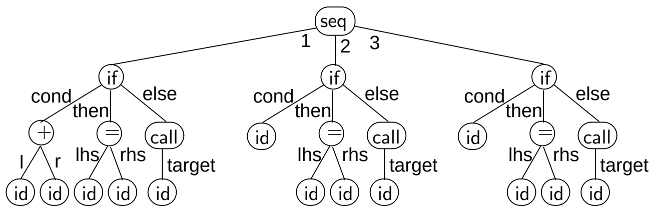

Figure 2. Example AST

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 AST nodes produced in step (2) for the example are shown in Listing 3.

```
1   if_8 +_2 id_0 id_0 =_2 id_0 id_0 call_1 id_0
2   if_6 id_0 =_2 id_0 id_0 call_1 id_0
3   if_6 id_0 =_2 id_0 id_0 call_1 id_0
```

Listing 3. Serialized AST nodes

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

3.2 Suffix Tree Detection

The original suffix tree clone detection is based on tokens. In our application of suffix trees, the AST node type plays the role of a token. Because we use the AST node type as distinguishing criterion, the actual value of identifiers and literals (their string representation) does not matter as they are treated as AST node attributes and hence are ignored. The actual value of identifiers and literals becomes relevant in a postprocessing step where we make the distinction between type-1 and type-2 clones.

Rather than Baker's algorithms for parameterized strings, we are using the simpler string-based algorithm by Ukkonen [28]. Consequently, the token-based clone detection returns equivalence classes of type-1 and type-2 clones as we do not distinguish type-1 and type-2 clones at this stage.

For our running example, the two representative cloned token sequences in Listing 4 and Listing 5, respectively, would be considered.

The token sequence in Listing 4 is a complete syntactic unit whereas the sequence in Listing 5 is not a single syntactic unit and, hence, needs to be decomposed into three syntactic subsequences as follows: <$\texttt{id}_0$>, <$\texttt{=}_2$ $\texttt{id}_0$ $\texttt{id}_0$>, and <$\texttt{call}_1$ $\texttt{id}_0$>.

```
   if_6 id_0 =_2 id_0 id_0 call_1 id_0
-- in line 2 and 3
```

Listing 4. Cloned token sequence

```
id_0 id_0 =_2 id_0 id_0 call_1 id_0
-- in line 1 (token pos. 4-9), 2, and 3
```

Listing 5. Cloned token sequence

3.3 Decomposing into Syntactic Clones

The previous step has produced a set of clone classes of maximally long equivalent AST 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. The main algorithm is shown in Listing 6 where `il` is the input set of clone sequence partitions as determined in the previous step. For each class in `is`, we select a representative and decompose it with algorithm `cut`, which we will explain shortly. The output is denoted by `os` and incrementally produced by `cut`. The result is again a set of token sequence partitions, but the difference here is that each sequence in `os` is a syntactic unit. Hence, `os` is a refinement of `is`.

Procedure *emit* is used to report clones based on the representative. It may filter clones based on various additional criteria such as length, type of clone, syntactic type (e.g., it may ignore clones in declarative code), differentiates the clone class elements into type-1 and type-2 clones, and finally reports all clones of a class 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 representative. An AST subtree is a complete clone if all its tokens are completely contained in a cloned token sequence. The test whether the tokens of an AST 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 AST nodes reachable from N excluding N itself) must not exceed the end of *ts*. More precisely, let *ts'first* and *ts'last* denote the first and last index in this sequence, respectively; then the following condition must hold for a complete syntactic unit: n + *tokens (N)* $\leq$ *ts'last* where n is the index of N in *ts*.

Listing 7 shows the basic algorithm. It traverses the whole cloned token sequence *ts*. If a root is found to

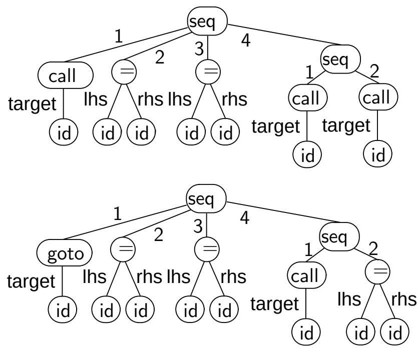

Figure 3. Example ASTs with sequences

be complete (lines 7–9), the search continues after the last token of the rooted tree. The tokens in between are part of the cloned rooted tree and we are interested only in maximal clones. For this reason, they may be skipped.

Variables `le` and `ri` indicate the range of tokens for the current syntactic clone (the representative). If the current rooted tree, indexed by *le*, is not completely contained in the cloned token sequence, we continue with its next child, which is the next token in the sequence (line 5 in Listing 7). This way, we descend into subtrees of a root to identify additional clones contained in an incomplete rooted tree. 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 Sequences 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 Listing 8 and its corresponding AST in Figure 3. The example consists of a statement sequence represented by a node type *seq* with another nested sequence. The cloned token sequence representative for this example is shown in Listing 9.

As you can see, the sequence runs into the nested sequence without covering it completely. The basic algorithm would, hence, find syntactic clone sequences as follows: $<=_2 \mathrm{id}_0\ \mathrm{id}_0>$ (`x=1`), once more $<=_2 \mathrm{id}_0\ \mathrm{id}_0>$ (`y=2`), and $<\mathrm{call}_1\ \mathrm{id}_0>$ (`bar()`). However, the two consecutive assignments together form a maximal clone sequence. The basic algorithm misses this maximal clone

```
procedure cut_all (is) is
os := ∅
for each class in is loop
  cut (representative (class), os)
end loop;
emit (os);
```

Listing 6. Cutting out syntactic clones

```
1  procedure cut (ts, os) is
2  le := ts'first;
3  while le ≤ ts'last loop
4    if le + tokens(le) > ts'last then
5      le := le + 1;
6    else
7      ri := le + tokens (le) + 1;
8      os := os ∪ {ts (le..ri −1)};
9      le := ri;
10   end if;
11 end loop
```

Listing 7. Cutting out syntactic clones

sequence because only parts of the outer sequence are part of the cloned token sequence. Whereas other AST node types require that all parts are present to form a complete clone, consecutive parts of a sequence may together form a maximal clone.

The extended algorithm in Listing 10 considers sequences. The extension is found in lines 8–16. Predicate `if_seq` is true if an AST node represents a sequence. 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 14–15). Function `parent(N)` returns the parent AST node of `N`.

4 Empirical Case Studies

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

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 [26]. In that study, six different research tools (cf. Fig. 4 above double line) have been compared based on several Java and C systems. For our study, we limit ourselves to the systems in Figure 5, as the Bellon study has not shown any significant difference in the performance of these tools for C and Java.

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

```
foo();                  goto l;
x = a;                  x = a;
y = b;                  y = b;
{ bar();                { bar();
  foo(); }                z = j; }
```

Listing 8. Example sequences in C

the mentioned tools blindly and evenly; that is, an algorithm presented him clone pairs in about the same fraction of submitted clones without telling him which tool has proposed the candidate.

Each clone pair suggested by a tool will be called *candidate* and each clone pair of the reference corpus will be called *reference* in the following. The candidates seen but rejected by Bellon are used to measure precision. Those accepted form the reference corpus are used to measure recall.

At least two percents of each tool's clone pairs have been judged by him. Although 2% sounds like a small fraction, one should be aware that it took him 77 hours 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.

We oracled the candidates of our tools and added them to the benchmark. All three authors of this paper firstly oracled jointly to develop a comparable notion of what constitutes a clone and then split to oracle in parallel. We spent about 36 hours to oracle at least 2% of each tool.

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 tools attempt 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 snippets. The snippets 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 line in a reference or vice versa. In this evaluation, we will focus on good matches only due to reasons of space.

The match classifies candidates and references as follows. *True negatives* are references where no candidate of a particular tool has a good match. *Detected references* are those for which a good match exists.*Rejected candidates* are candidates for which no good match exists for 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 that is observed. All three variables have a special value "all" referring to all tools, programs, and clone types, respectively. τ furthermore has a special value "unknown" as some tools cannot categorize clone types.

$=_2$ id_0 id_0 $=_2$ id_0 id_0 seq_5 $call_1$ id_0

Listing 9. Cloned sequence

```
procedure cut (ts, os) is
le := ts'first;
while le ≤ ts'last loop
  if le + tokens(le) > ts'last
  then
    le := le + 1;
  else
    if is_seq (parent (ts (le))) then
      ri := le;
      -- assert: ri + tokens(ri) ≤ ts'last
      loop
        ri := ri + tokens (ri) + 1;
        exit when ri + tokens (ri) > ts'last
                or else parent (ts(le))
                        ≠ parent (ts(ri));
      end loop;
    else
      ri := le + tokens (le) + 1;
    end if;
    os := os ∪ {ts (le..ri-1)};
    le := ri;
  end if;
end loop;
```

Listing 10. Cutting out syntactic sequence clones

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

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

$$\mathrm{TrueNegatives}(P,T,\tau) = \frac{|\mathrm{TrueNegativeRefs}(P,T,\tau)|}{|\mathrm{Refs}(P,\tau)|}$$

Additional Tools In order to compare the tools not only in terms of precision and recall but also in runtime, we implemented three additional variations of the evaluated tools (cf. Fig. 4 below double line). Because these tools are built on a common infrastructure of ours, written in the same programming language and executed on the same hardware, the runtime comparison is more meaningful than the reports in the Bellon study.

`cpdetector` implements the technique that we describe in this paper. The closest techniques to our approach are the token-based and AST-based techniques. That is why we chose these techniques as a point of comparison. `ccdiml` is a variation of Baxter's `CloneDr` also based on ASTs. The main differences are the avoidance of the similarity metric, the handling of sequences, the hashing, and the fact that `CloneDr` works concurrently. Yet, the overall approach is the same. cpdetector and ccdiml share the same AST as intermediate representation.

clones is a variation of Baker's technique with the difference that it is not based on lines but solely on tokens and that it uses nonparameterized suffixes. The advantage of nonparameterized suffixes is that clones does not depend upon layout; the disadvantage is that the distinction between type 1 and 2 must be made in a postprocessing step. Another main difference is that clones does currently not check whether identifiers in type-2 clones are renamed consistently.

As already discussed in Section 2.3, token-based techniques can be extended so that they attempt to find syntactic clones 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. cscope implements this postprocessing step. As a matter of fact, cscope is not a new tool but just an additional feature built into clones 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 cpdetector, clones and cscope identically.

All our tools find type-1 and type-2 clones except for ccdiml, which also finds type-3 clones.

Author	Tool	Comparison
Brenda S. Baker	Dup	Tokens
Ira D. Baxter	CloneDr	AST
Toshihiro Kamiya	CCFinder	Tokens
Jens Krinke	Duplix	PDG
Ettore Merlo	CLAN	Function Metrics /Tokens
Matthias Rieger	Duploc	Text
Rainer Koschke	cpdetector	AST/Suffix
Stefan Bellon	ccdiml	AST
Rainer Koschke	clones	Tokens
Pierre Frenzel	cscope	Tokens

Figure 4. Compared tools

Program	Domain	Size
bison 1.32	parser generator	19K
wget 1.5.3	network downloader	16K
SNNS 4.2	neural network simulator	105K
postgreSQL 7.2	database	235K

Figure 5. Analyzed programs (size in SLOC)

Results Because of limited space, we present only one system. We choose SNNS because some tools of the earlier experiment had problems with the larger postgreSQL system. The results for the other systems are comparable, however.

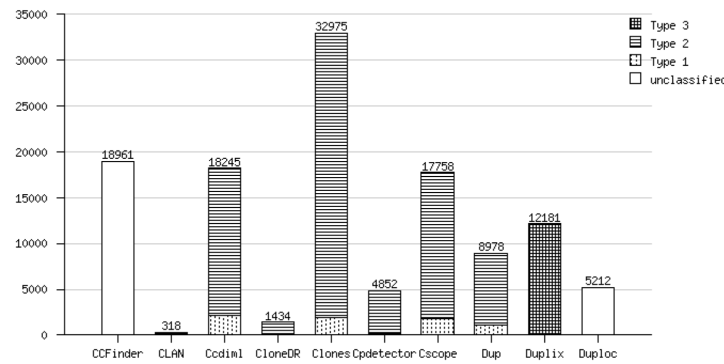

Figure 6. Number of candidates

Figure 6 shows the number of candidates found by the various tools for SNNS. Here, clones found 71% more clones than CCFinder (the tool with the highest number in the earlier experiment), while the number for cscope is comparable to the numbers of the other token-based tools. cpdetector yields a low number of candidates and ccdiml compares to cscope. The number of candidates of token-based approaches almost doubles those found by AST-based tools.

The rejected candidates are shown in Figure 7. These are very high for clones and cscope. Other token-based methods have a reject rate of 50% and 54% while clones and cscope have 90% and 82%. The reason is that clones and cscope do not check for consistent renaming of identifiers and literals.

Rejects for ccdiml and cpdetector are comparable to CCFinder, Dup, and Duploc with respect to type-2 clones. The reason is that ccdiml and cpdetector neither check for consistent renaming. Given the fact that ccdiml and cpdetector have much lesser rejected type-1 clones, we conclude that they could perform better than token-based techniques if they checked for consistent renaming.

Figure 8 contains the true negatives. The tools clones, cscope, and cpdetector find an average percentage of references of 30%, 33% and 26%, respectively. ccdiml has the second best result (53%) after CCFinder (61%). There is no substantial difference in the overall percentage of true negatives for token-based versus AST-based approaches.

In Figure 9, recall is shown. The recall of clones, cscope, and cpdetector is with 30%, 33% and 26% at average. The recall of ccdiml, however, is with 53% the second best with a quite big advantage. The average recall of the token-based tools (clones, cscope, Dup, CCFinder) is higher (54%) than the AST-based tools (30%).

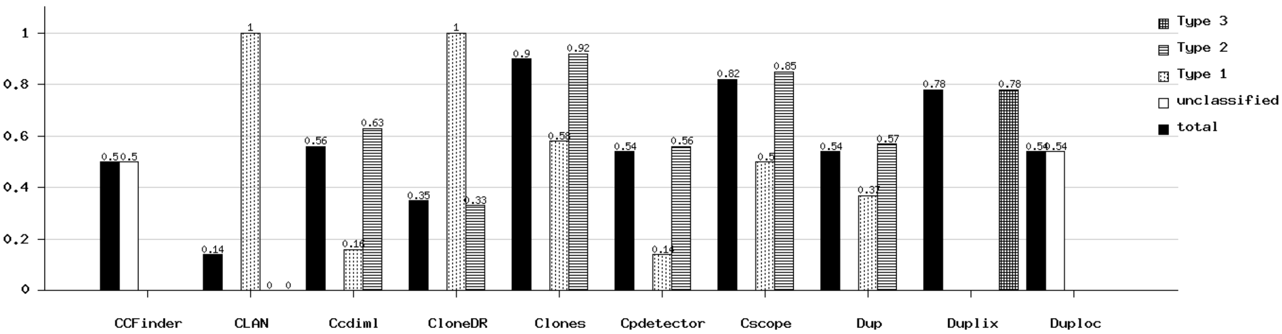

Figure 7. Percentage of rejected candidates

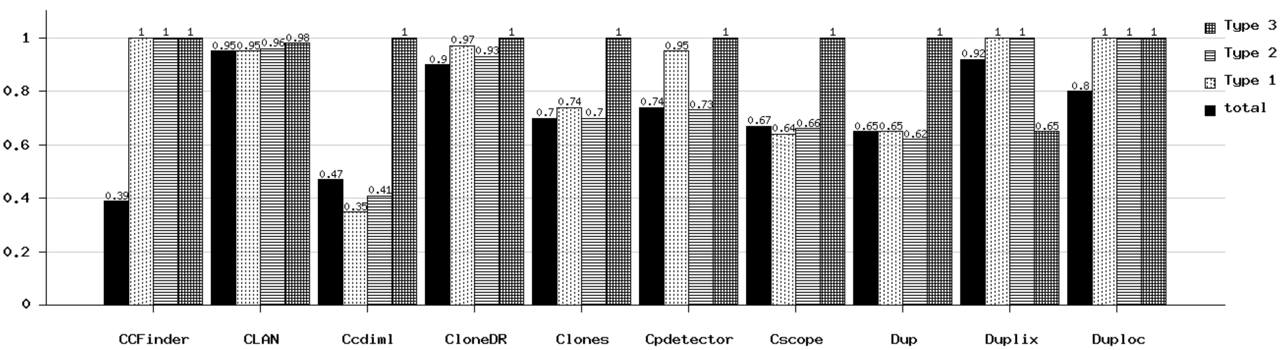

Figure 8. Percentage of true negatives

Runtime Comparision The runtime for each tool is given in Figure 10, determined on a 64bit Intel dual-processor architecture (3.0 GHz) with 16 GB RAM running Linux (Fedora Core 5), where only one CPU was used. The runtime for the AST-based tools contains loading the AST from disk; that is, the time excludes parsing. The runtime for the token-based tools contains reading and tokenizing source text.

Comparing `cpdetector` to `ccdiml`, we notice a dramatic performance difference. The non-linear behavior of `ccdiml` is mainly caused by a quadratic tree comparison necessary to compare each pair of AST subtrees in the same bucket.

We note that `ccdiml` can be adjusted in various ways which affect both precision and scalability. We chose the setting of parameters which give the most precise results, but also require most resources. The figures for `ccdiml` are comparable to those of `CloneDR`, which required 10800 seconds for `SNNS` and 9780 seconds for `postgreSQL` but on a weaker hardware (Pentium Pro/200 MHz running Windows NT 4.0) in the earlier experiment.

The extra effort of `cscope` compared to `clones` lies in the postprocessing step to split clones into syntactic units. As one can notice, this step is a considerable factor. The effort of `cscope`, which also finds syntactic clones, versus `cpdetector` is similar, in case of `postgreSQL` even much lower. As the data for `postgreSQL` suggest, splitting sequences into syntactic clones using ASTs is more efficient for larger systems. `cpdetector` skips all tokens subsumed by a rooted AST tree in one step, whereas `cscope` traverses through a sequence token by token because it does not have a notion of syntactic containment.

System	ccdiml	cpdetector	clones	cscope
bison	68,20	3.76	1.82	3.72
wget	81.12	3.46	2.83	4.55
snns	7009.45	29.66	11.63	25.73
postgres	16942.38	62.61	74.02	113.08

Figure 10. Runtime comparison [seconds]

5 Conclusion

Our experiment has confirmed the earlier result that token-based techniques tend to have higher recall but also lower precision for type-1 clones. We noticed that checking for consistent renaming is an important feature of a clone detector. The results for token-based techniques that perform this check are much better than those that do not. As a consequence, we could

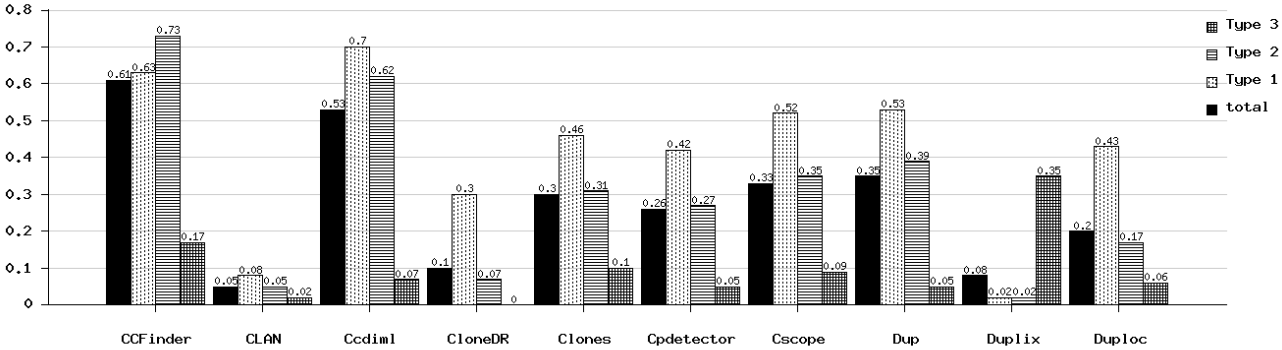

Figure 9. Recall

also have better precision for type-2 clones for our AST-based tools if they made this check, too.

	`CloneDR`	`cpdetector`	`ccdiml`
Rejection	+	∘	∘
Recall	−	∘	+
True negatives	−	∘	+
Runtime	−	+	−

Figure 11. Comparision of AST based tools

Figure 11 shows a summarizing comparison for the AST-based tools. `cpdetector` is a compromise between recall and precision, but offers better scalability that compares to token-based techniques.

During oracling, we made additional observations. Commonly, token-based tools proposed the end of one function plus the beginning of the lexically next function as a clone. Here our `cscope` tool helped to increase the quality of the results compared to `clones`. A second observation is that tools report frequently occurring patterns such as sequences of assignments, very large initializer lists for arrays, and structurally equivalent code with totally different identifiers of record components. Such spurious clones could be filtered out by syntactic property checks. Such a filtering in syntactic terms, however, is possible only with an AST-based approach.

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. In addition, generated files (like parsers) should be excluded from the benchmark since such code tends to be regular and appear as spurious clone candidates.

Acknowledgement We would like to thank Stefan Bellon for providing us with his benchmark, support in the evaluation, and for comments on drafts of this paper.

References

[1] J. Bailey and E. Burd. Evaluating Clone Detection Tools for Use during Preventative Maintenance. In *Second IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'02)*, page 36ff, Oct. 2002.

[2] B. S. Baker. On Finding Duplication and Near-Duplication in Large Software Systems. In *WCRE*, pages 86–95. IEEE CS Press, July 1995.

[3] B. S. Baker. Parameterized Pattern Matching: Algorithms and Applications. *Journal Computer System Science*, 52(1):28–42, Feb. 1996.

[4] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone Detection Using Abstract Syntax Trees. In *Proceedings; International Conference on Software Maintenance*, 1998.

[5] S. Bellon. Vergleich von Techniken zur Erkennung duplizierten Quellcodes. Master's thesis, University of Stuttgart, Germany, Sept. 2002.

[6] J. R. Cordy. Comprehending reality—practical barriers to industrial adoption of software maintenance automation. In *IWPC*, pages 196–205. IEEE CS Press, 2003.

[7] J. R. Cordy, T. R. Dean, and N. Synytskyy. Practical language-independent detection of near-miss clones. In *CASCON*, pages 1–12. IBM Press, 2004.

[8] G. Di Lucca, M. Di Penta, and A. Fasolino. An approach to identify duplicated web pages. In *COMPSAC*, pages 481–486, 2002.

[9] S. Ducasse, M. Rieger, and S. Demeyer. A Language Independent Approach for Detecting Duplicated Code. In *Proceedings of the International Conference on Software Maintenance (ICSM99)*, 1999.

[10] D. Gitchell and N. Tran. Sim: a utility for detecting similarity in computer programs. In *SIGCSE '99: The proceedings of the thirtieth SIGCSE technical symposium on Computer science education*, pages 266–270. ACM Press, 1999.

[11] Y. Higo, Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue. On software maintenance process improvement based on code clone analysis. In *International Conference on Product Focused Software Process Improvement*, volume 2559 of *Lecture Notes In Computer Science*, pages 185–197. Springer, 2002.

[12] J. H. Johnson. Identifying redundancy in source code using fingerprints. In *CASCON*, pages 171–183. IBM Press, 1993.

[13] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A Multi-Linguistic Token-based Code Clone Detection System for Large Scale Source Code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[14] C. Kapser and M. Godfrey. A taxonomy of clones in source code: The reengineers most wanted list. In *WCRE*. IEEE CS Press, 2003.

[15] R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal Research and Development*, 31(2):249–260, Mar. 1987.

[16] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *Proc. Int. Symposium on Static Analysis*, pages 40–56, July 2001.

[17] K. Kontogiannis, R. DeMori, M. Bernstein, M. Galler, and E. Merlo. Pattern matching for design concept localization. In *WCRE*, pages 96–103. IEEE Computer Society Press (Order Number PR07111), July 1995.

[18] K. Kontogiannis, R. D. Mori, E. Merlo, M. Galler, and M. Bernstein. Pattern matching for clone and concept detection. *Automated Software Engineering*, 3(1/2):79–108, June 1996.

[19] J. Krinke. Identifying Similar Code with Program Dependence Graphs. In *Proceedings of the Eighth Working Conference On Reverse Engineering (WCRE'01)*, 2001.

[20] B. Laguë, D. Proulx, J. Mayrand, E. M. Merlo, and J. Hudepohl. Assessing the benefits of incorporating function clone detection in a development process. In *International Conference on Software Maintenance*, pages 314–321, 1997.

[21] F. Lanubile and T. Mallardo. Finding function clones in web applications. In *CSMR*, pages 379–386, 2003.

[22] A. M. Leitao. Detection of redundant code using r2d2. In *SCAM*, pages 183–192. IEEE CS Press, 2003.

[23] A. Marcus and J. Maletic. Identification of high-level concept clones in source code. In *ASE*, pages 107–114, 2001.

[24] J. Mayrand, C. Leblanc, and E. M. Merlo. Experiment on the Automatic Detection of Function Clones in a Software System using Metrics. In *Proceedings of the International Conference on Software Maintenance*, pages 244–254, Washington, Nov. 4–8 1996. IEEE Computer Society Press.

[25] E. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 32(2):262–272, 1976.

[26] C. of Clone Detection Techniques. Stefan bellon and rainer koschke. submitted for publication, 2006.

[27] M. Rieger. *Effective Clone Detection Without Language Barriers*. Dissertation, University of Bern, Switzerland, 2005.

[28] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[29] V. Wahler, D. Seipel, J. W. von Gudenberg, and G. Fischer. Clone detection in source code by frequent itemset techniques. In *SCAM*, pages 128–135, 2004.

[30] W. Yang. Identifying syntactic differences between two programs. *Software–Practice and Experience*, 21(7):739–755, July 1991.