Scalable Detection of Semantic Clones*

Mark Gabel Lingxiao Jiang Zhendong Su
Department of Computer Science
University of California, Davis
{mggabel,lxjiang,su}@ucdavis.edu

ABSTRACT

Several techniques have been developed for identifying similar code fragments in programs. These similar fragments, referred to as code clones, can be used to identify redundant code, locate bugs, or gain insight into program design. Existing scalable approaches to clone detection are limited to finding program fragments that are similar only in their contiguous syntax. Other, semantics-based approaches are more resilient to differences in syntax, such as reordered statements, related statements interleaved with other unrelated statements, or the use of semantically equivalent control structures. However, none of these techniques have scaled to real world code bases. These approaches capture semantic information from Program Dependence Graphs (PDGs), program representations that encode data and control dependencies between statements and predicates. Our definition of a code clone is also based on this representation: we consider program fragments with isomorphic PDGs to be clones.

In this paper, we present the first scalable clone detection algorithm based on this definition of semantic clones. Our insight is the reduction of the difficult graph similarity problem to a simpler tree similarity problem by mapping carefully selected PDG subgraphs to their related structured syntax. We efficiently solve the tree similarity problem to create a scalable analysis. We have implemented this algorithm in a practical tool and performed evaluations on several million-line open source projects, including the Linux kernel. Compared with previous approaches, our tool locates significantly more clones, which are often more semantically interesting than simple copied and pasted code fragments.

Categories and Subject Descriptors

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

General Terms

Languages, Algorithms, Experimentation

Keywords

Clone detection, program dependence graph, software maintenance, refactoring

*This research was supported in part by NSF CAREER Grant No. 0546844, NSF CyberTrust Grant No. 0627749, NSF CCF Grant No. 0702622, US Air Force under grant FA9550-07-1-0532, and a generous gift from Intel. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

ICSE'08, May 10–18, 2008, Leipzig, Germany.

1. INTRODUCTION

Considerable research has been dedicated to methods for the detection of similar code fragments in programs. Once located, these fragments, or *clones*, can be used in many ways. Clones have been used to gain insight into program design, to identify redundant code to use as candidates for refactoring, and to be analyzed for consistent usage for the purpose of bug detection.

DECKARD [9], CP-Miner [14], CCFinder [10], and CloneDR [3] represent the most mature clone detection techniques. These tools share several common characteristics. Each tool locates *syntactic* clones, and each has been shown to scale to millions of lines of code. Under empirical evaluation, each tool has been shown to locate comparable numbers of clones.

By operating on token streams and syntax trees, these techniques locate clones that are resilient to minor code modifications, such as the changing of types or constant values. This resilience gives these tools some modicum of *semantic awareness*: two program fragments may differ in their concrete syntax, but the normalizing effects of the respective clone tools allow the detection of their semantic similarity.

The sets of clones located by each of these tools are fundamentally limited by the working definition of a code clone. Each tool is capable of finding clones solely within a program's contiguous, structured syntax. Certain interesting clones can elude detection: these tools are sensitive to even the most simple structural differences in otherwise semantically similar code. These structural differences can include reordered statements, related statements interleaved with other unrelated statements, or the use of semantically equivalent control structures.

As a motivating example, consider the code snippet in Figure 1. When compared with the listing in Figure 2, the code is similar: both perform the same overall computation, but the latter snippet contains extra statements to time the loop. Current scalable clone detection techniques are unable to detect these interleaved clones.

While detecting true semantic similarity is undecidable in general, some clone detection techniques have attempted to locate clones with a less strict, semantics preserving definition of similar code. Rather than scanning token sequences or similar subtrees, these techniques have operated on *program dependence graphs* [6], or PDGs. A PDG is a representation of a procedure in which the nodes represent simple statements and control flow predicates, and edges

```
1  int func(int i, int j) {
2    int k = 10;
3
4    while (i < k) {
5      i++;
6    }
7
8    j = 2 * k;
9
10   printf("i=%d, j=%d\n", i, j);
11   return k;
12 }
```

Figure 1: Example code listing.

```
1  int func_timed(int i, int j) {
2    int k = 10;
3
4    long start = get_time_millis();
5    long finish;
6
7    while (i < k) {
8      i++;
9    }
10
11   finish = get_time_millis();
12   printf("loop took %dms\n", finish − start);
13
14   j = 2 * k;
15
16   printf("i=%d, j=%d\n", i, j);
17   return k;
18 }
```

Figure 2: A similar example code listing.

encode data and control dependencies. PDG-based similarity detection tools have all used some variant of subgraph isomorphism to detect either similar procedures or code fragments [12, 16]. These computations are particularly expensive, and each technique has not been shown to scale to even moderately-sized code bases.

In this paper, we introduce an extended definition of code clones, based on PDG similarity, that captures more semantic information than previous approaches. We then provide a scalable, approximate algorithm for detecting these clones. We reduce the difficult graph similarity problem to a simpler tree similarity problem by creating a mapping between PDG subgraphs and their related structured syntax. Specifically, we make the following technical contributions:

1. We extend the definition of a code clone to include semantically (but not necessarily syntactically) related code fragments. Our definition is a generalization of previous syntactic clone definitions, and it thus defines a superset of previously defined clones.

2. We introduce an approximate algorithm for detecting these clones that scales to millions of lines of code. Our algorithm is based on a reduction of deliberately selected PDG subgraphs to abstract syntax tree forests. We then utilize an existing, tree-based detection technique [9] to locate clones.

3. We implement a practical tool based on our algorithm and perform an extensive empirical evaluation. Our tool is capable of scanning large, real-world C and C++ projects. Compared with previous approaches, our tool locates significantly more clones, which are often more semantically interesting than simple copied and pasted code fragments.

The rest of this paper is structured as follows. We begin with a discussion of background information on components of our analysis (Section 2). The body of our work continues with the presentation of our definitions and algorithm (Section 3). We then discuss our implementation (Section 4) and present the results of our empirical evaluation (Section 5). Finally, we discuss related work (Section 6) and conclude with ideas for future work (Section 7).

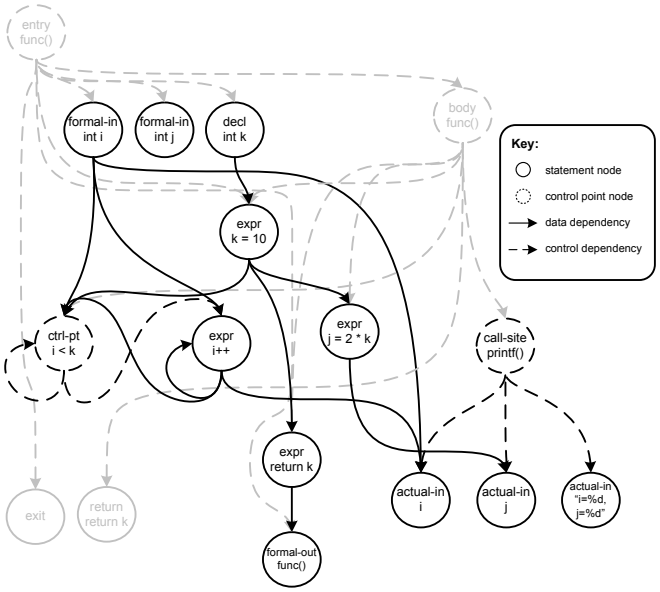

Figure 3: The PDG for Figure 1.

2. BACKGROUND

Our algorithm augments an existing clone detection technique, DECKARD [9], with semantic information derived from *program dependence graphs* (PDGs). This section provides the necessary background on both program dependence graphs and DECKARD's vector based clone detection.

2.1 Program Dependence Graphs

A *program dependence graph* [6] (PDG) is a static representation of the flow of data through a procedure. It is commonly used to implement *program slicing* [20]. The nodes of a PDG consist of program points constructed from the source code: declarations, simple statements, expressions, and *control points*. A control point represents a point at which a program branches, loops, or enters or exits a procedure and is labeled by its associated predicate.

A PDG models the flow of data through a procedure. In effect, the PDG abstracts away many arbitrary syntactic decisions a programmer made while constructing a function. For example, any possible arbitrary interleaving of unrelated statements within a procedure yields precisely the same PDG.

The edges of a PDG encode the data and control dependencies between program points. Given two program points p_1 and p_2, there exists a directed *data dependency* edge from p_1 to p_2 if and only if the execution of p_2 depends on data calculated directly by p_1. For example, consider the statements on lines 2 and 8 of the listing in Figure 1. The second statement calculates a value that is initialized in the first. This dependency is illustrated by a directed edge between the two nodes in Figure 3.

Note that the node corresponding to the formal parameter **j** does not have any outgoing edges. This accurately reflects the fact that **j** is redefined without ever being used at line 8 in the listing.

The incrementing of **i** on line 5 also presents an interesting case. Because an increment constitutes both a use and a definition, the node in the PDG corresponding to **i++** has both a self data dependency loop and outgoing data dependency edges.

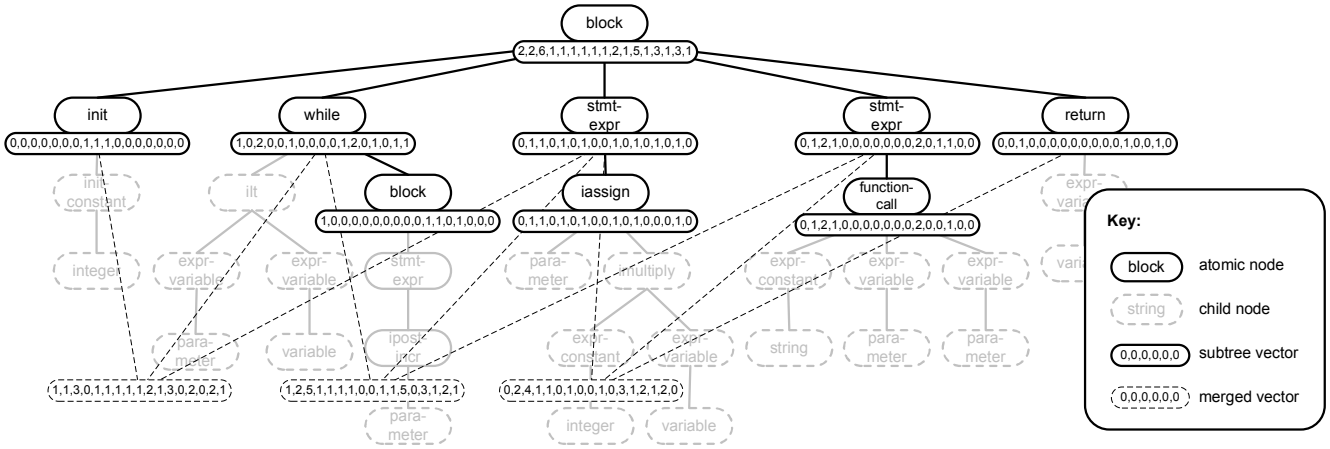

Figure 4: The AST for Figure 1 with characteristic vectors.

Similarly, there exists a directed *control dependency* edge from p_1 to p_2 if and only if the choice to execute p_2 depends on the test in p_1. The **while** loop on line 4 of the listing illustrates the use of control dependency edges flowing from a control point node. The corresponding PDG node in Figure 3 is labeled with the guard expression, and there is a control dependency edge to the enclosed increment statement. Note that this node also has a control dependency self loop. This is indicative of a looping structure: if we replace **while** with **if**, we need only remove this one self looping control edge to yield the new PDG.

Callsites are modeled as control points that control the execution of expressions corresponding to the calculation of the actual parameters and the assignment of the return value (or assignments to out parameters). The call to **printf** on line 10 is modeled in this way: there are three outgoing control edges that connect to the three parameters.

PDGs may also contain implicit nodes that do not have a direct source correspondence. These include entry, exit, and function body control points, and are represented by a light shade in Figure 3. These nodes are used to connect PDGs and form a larger, interprocedural structure that is sometimes referred to as a *system dependence graph* (SDG) [8]. Because our work focuses on intraprocedural dependencies only, we simplify our graphs by omitting these nodes.

2.2 Scalable Tree-based Clone Detection

The foundation of our analysis is the DECKARD clone detection tool. DECKARD implements a tree-similarity based technique that uses the idea of *characteristic vectors* to efficiently match clone pairs. This section provides a brief review of this technique. A more detailed discussion can be found in the full paper [9].

A characteristic vector is a numerical approximation of a particular subtree. The dimension of the vectors is uniform and is determined by the total number of possible types of q-*level atomic patterns* deemed relevant to approximate a given tree, where a q-level atomic pattern is a complete binary tree of height q with tree node labels for identification. In our context, these node labels correspond to terminals and non-terminals in the grammar for a language. The maximum number of q-level atomic patterns is L^{2^q-1} if the number of possible labels (including the empty label) is L.

Each characteristic vector is a point $\langle c_1, ..., c_n \rangle$ in n-dimensional Euclidean space, where n is the number of distinct q-level atomic patterns. Each c_i counts the number of occurrences of the atomic pattern represented by index i. One important property of characteristic vectors is that given two subtrees, T_1 and T_2, and their respective q-level vectors, v_1 and v_2, if the edit distance between the trees is k, then the Euclidean distance between the vectors is no more than $(4q-3)k$ [21]. This characterization allows the tree-based clone detection problem to be reformulated as a nearest neighbor problem on numerical vectors.

For the application of tree-based clone detection, we use 1-level atomic patterns. The actual domain of atomic patterns is given by the various node types defined in the grammar for a language that the user deems "significant." For parse trees, insignificant nodes might include semicolons and brackets, and significant nodes might include expressions, operators, and statements.

DECKARD focuses on parse trees, but we have adapted the algorithm to function on abstract syntax trees. The algorithm is essentially the same, but a few changes were necessary. This will be discussed in Section 4.

DECKARD first generates vectors that effectively cover an entire tree. This is done in two distinct phases. The first phase traverses the tree in postorder and generates vectors for each "significant" subtree, where relevance is a heuristic setting that marks a node type as being a suitable parent node. In the original implementation, "significant" nodes included statements, expressions, and declarations. In our current, AST-based implementation, we have the luxury of more semantic information from the AST class hierarchy. We define "significant" nodes to be those that descend from the parent "statement" class.

Figure 4 depicts a simplified abstract syntax tree for the code listing in Figure 1. The subtree vectors appear below each significant node. For each node, DECKARD first creates a vector that consists of the sum of the node's children's vectors. Then, it increments the value in the current node's index position if the current node is significant.

The second phase consists of moving a sliding window along adjacent subtrees and merging the subtree vectors. This allows groups of contiguous statements or expressions to be grouped into a single vector for matching. Configuration options prevent merges that are not likely to be useful, like the merging of the tail end of a block with the head of an adjacent block. Figure 4 contains merged vectors that were generated with a sliding window size of three.

At this point, the tree has been reduced to a set of points in the Euclidean space. To efficiently cluster large numbers of vectors, DECKARD uses *Locality Sensitive Hashing* [7], an efficient approximate near-neighbor solver. When combined with a lossless partitioning of the vectors based on their size [9], the LSH engine is capable of enumerating clone groups from millions of vectors in a few minutes.

Using these approximations, DECKARD is able to enumerate a comprehensive set of clone groups over millions of lines of code in

tens of minutes. In practice, the use of vector approximations and an approximate nearest neighbor solver does not affect the quality of the results; the false positive rate is extremely low.

3. ALGORITHM DESCRIPTION

The vectors generated during the first phase of DECKARD's execution provide a high degree of coverage of the syntactic structure of a given program. Our approach involves augmenting DECKARD's vector generation phase with a third pass: the generation of vectors for *semantic* clones. We then use the same LSH-based clustering technique to solve the near-neighbor problem and generate clone reports.

3.1 Definitions

Considering once again the motivating example in Figure 2, we notice that the computation of data is identical to that of the code in Figure 1. However, purely syntactic definitions of code clones do not capture this relationship. Syntactic definitions of code clones are defined similarly:

Definition 3.1 (Syntactic Code Clone) *Two disjoint, contiguous sequences of program syntax S_1 and S_2 are* code clones *if and only if $\delta(S_1, S_2)$.*

In this general definition, the precise form of the syntax is not described, and δ refers to a similarity function. CP-Miner uses a distance metric on token streams called *gap*, CloneDR uses a size-sensitive definition on trees referred to as *Similarity*, and DECKARD uses tree edit distance.

We expand this definition to include non-contiguous but related code. Assume we have a mapping function, ρ, that maps a sequence of syntax (of arbitrary type) to a PDG subgraph.

Definition 3.2 (Semantic Code Clone) *Two disjoint, possibly non-contiguous sequences of program syntax S_1 and S_2 are* semantic code clones *if and only if S_1 and S_2 are syntactic code clones or $\rho(S_1)$ is isomorphic to $\rho(S_2)$.*

Applying this relaxed definition to our example allows us to consider a subset of the code in Figure 2 as a candidate for clone detection. In this instance, we disregard the timing code. The remaining subset is a syntactic clone with the body of Figure 1, and their associated PDGs are identical.

3.2 High-Level Algorithm

There are difficulties with locating these semantic clones in a scalable manner. First, there is a combinatorial explosion of possible clones. Second, although graph isomorphism testing may be feasible for small, simple PDGs, it is computationally expensive in general. Any method that would require pairwise comparisons would not scale.

Thus far, no scalable algorithm exists for detecting semantic clones. We present a scalable, approximate technique for locating semantic clones based on the fact that both structured syntax trees and dependence graphs are derived from the original source code. Because of this relationship, we are able to construct a mapping function that locates the associated syntax for a given PDG subgraph. We refer to this associated syntax as the *syntactic image*. For compatibility with DECKARD's tree-based clone detection, we map to AST forests.

Definition 3.3 (Syntactic Image) *The* syntactic image *of a PDG subgraph G, $\mu(G)$, is the maximal set of AST subtrees that correspond to the concrete syntax of the nodes in G. The set is a* dominating set, i.e., *for all pairs of trees $T, T' \in \mu(G)$, $T \not\subseteq T'$.*

For example, consider the code snippet in Figure 1. Suppose we have the PDG subgraph corresponding to all lines that reference the variable **i**, *i.e.*, the graph consisting of the nodes (from Figure 3) "**int i**," "**i < k**," "**i++**," and the actual parameter to the **printf** call.

We map each of these nodes to their structured syntax. The actual parameter maps to the subtree corresponding to the call of **printf**, which is the "**function-call**" subtree in Figure 4 (we only consider subtrees that can be syntactically separated).

The incrementing of **i** corresponds to the "**ipost-incr**" subtree, and the control point (**i < k**) corresponds to the "**while**" subtree. Because the "**ipost-incr**" subtree is subsumed by the "**while**" subtree, we include only the latter in the syntactic image.

Mapping a PDG subgraph to an AST forest effectively reduces the graph similarity problem to an easier tree similarity problem that we can solve efficiently using DECKARD.

yields something that we can match very efficiently, both partially and fully, using DECKARD's vector generation. This relationship to syntax effectively reduces the graph similarity problem to an easier tree similarity problem.

The overall architecture is shown in Figure 5. At a high level, our algorithm functions as follows:

1. We run DECKARD's primary vector generation. Subtree and sliding window vectors efficiently provide contiguous syntactic clone candidates for the entire program.

2. For each procedure, we enumerate a finite set of *significant subgraphs*; that is, we enumerate subgraphs that hold semantic relevance and are likely to be good semantic clone candidates. These algorithms are discussed in Section 3.3. In short, we produce subgraphs of maximal size that are likely to represent distinct computations.

3. For each subgraph G, we compute $\mu(G)$ to generate an AST forest.

4. We use DECKARD's sliding window vector merging to generate a complete set of characteristic vectors for each AST forest.

5. We use LSH to quickly solve the near-neighbor problem and enumerate the clone groups. As before, we apply a set of post-processing filters to remove spurious clone groups and clone group members.

If a semantic vector is a member of a clone group, then the flow of data represented by its syntactic members is duplicated by the other members of the clone group—each of which can come from any phase of vector generation. That is, a given semantic vector can match either:

- a complete AST subtree,
- a sequence of contiguous statements, or
- another semantic vector: a slice of another procedure.

3.3 PDG Subgraph Selection

Our algorithm generates vectors over a finite set of subgraphs of a given procedure's PDG. This section details our definitions of interesting subgraph sets and our algorithms for enumerating them.

3.3.1 Weakly Connected Components

Consider two statements, s_1 and s_2, that are contained in a single given procedure with PDG P. As discussed in Section 2.1, s_1 and s_2 have an implied relationship (a data or control dependency) if there exists a path between them. However, the absence of a path does not imply that the statements are completely unrelated: the two statements could both influence a third, common statement.

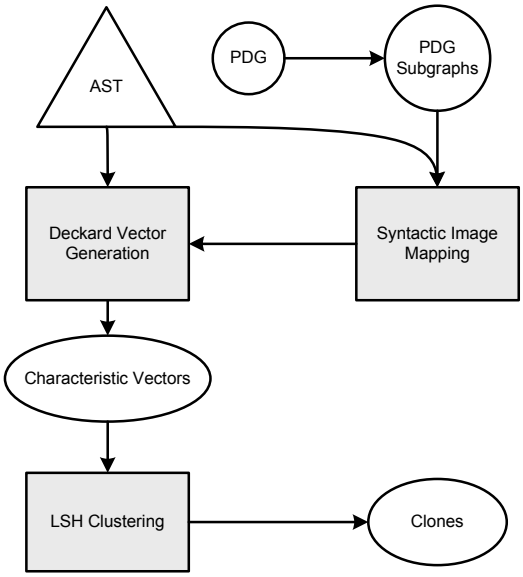

Figure 5: Semantic clone detection algorithm.

Consider our earlier example in Figure 1. Statements 5 (increment) and 8 (simple assignment) do not have a connecting path in the PDG, but they are still tied together by their common use at the call site on line 10.

Note, though, that these indirect relationships are characterized by *undirected* paths in the PDG. We say that two statements are *unrelated* if there does not exist any undirected path between them in the PDG, *i.e.*, each resides in separate *weakly connected components*.

A natural but conservative choice for a set of interesting PDG subgraphs is to cluster the graph into weakly connected components. Statements that are separated are definitely unrelated, and statements that are clustered have a semantic relationship. These subgraphs are especially interesting if their respective sets of statements are interleaved when viewed in the context of the concrete syntax of the procedure.

Practically, this can be implemented with a series of linear-time graph searches. As previously noted, we operate over our simplified PDGs: implicit entry and exit nodes are omitted. Without breaking the implicit control dependence of function entry and exit, every procedure would have exactly one weakly connected component.

3.3.2 Semantic Threads

Although analyzing weakly connected components for clones does yield interesting results, many semantic clones may not be detected this way. Thus, for better coverage, we need a more fine-grained partitioning of the statements in a procedure body. First, to illustrate the problem, let us consider the following hypothetical example in Figure 6.

It is clear that there are two distinct flows of data throughout the function, which only merge at the end. However, the aggregation of the two calculations (through "return result") causes the entire function body to be grouped as a single component. The PDG subgraphs corresponding to these computations overlap, but only at the end when the results are returned.

One way of modeling this particular flow of data is through the concept of a *forward slice* [8], which is a specific variant of program slicing [20]. A forward slice from a program point s with respect to a variable v consists of all program points that may be directly or indirectly affected by the execution of s with the value

```
struct file_stat *compute_statistics() {
    struct file_stat *result = malloc(sizeof(struct
        file_stat));
    int avg_temp_file_size = 0;
    int avg_data_file_size = 0;

    /* iterate the temp files */
    ...
    /* iterate the data files */
    ...
    /* avg results and store in avg_temp_file_size */
    ...
    /* avg results and store in avg_data_file_size */
    ...

    result->temp_size = avg_temp_file_size;
    result->data_size = avg_data_file_size;
    return result;
}
```

Figure 6: Semantic thread example.

of v. In our work, we adopt a simplified definition, which assumes that all slices from point s are computed with respect to all variables used or defined in s.

This type of static intraprocedural slicing is straightforward when given a dependence graph. Forward slices are defined by graph connectivity in the PDG: each forward edge encodes a specific data or control dependency.

Definition 3.4 (Forward Slice) *Let G be the PDG of procedure P, and let s be a statement in P. The static intraprocedural* forward slice *from s over P, $f(s)$, is defined as the set of all nodes reachable from s in G.*

In the above example, two forward slices from the declarations at the second and third lines yield two distinct but interleaving threads of computation that intersect at one statement.

We wish to enumerate these potentially overlapping flows of data for each procedure, which we refer to as *semantic threads*.

Definition 3.5 (Semantic Thread) *A* semantic thread *of a procedure P is either a forward slice $f(s)$ or the union of one or more forward slices.*

Simply enumerating the set of forward slices of a procedure yields redundant data. For example, some slices may fully subsume others. It is clear from our definition that

$$\forall s_1, s_2 \in P.\ s_1 \in f(s_2) \Rightarrow f(s_1) \subseteq f(s_2)$$

In addition, although some overlapping among different slices should be allowed (and even desired), a significant amount of overlapping may imply that these slices might be part of the same higher-level computation. This is especially evident in forward slices from consecutive, related declarations:

```
int count_list_nodes(struct list_node *head) {
    int i = 0;
    struct list_node *tail = head->prev;

    while (head != tail && i < MAX) {
        i++;
        head = head->next;
    }

    return i;
}
```

The forward slices from the declarations on the first and second lines differ only in their respective first nodes. Considering these to be separate computations would be a mistake: we not only create

Algorithm 1 Construct Semantic Threads

```
1: function BST(P : PDG, γ : int): STs
2:     IST, seen ← ∅
3:     Sort nodes in P in asc. order w.r.t. their locs. in source code.
4:     for all node n in P do
5:         if n ∉ seen then
6:             slice ← DepthFirstSearch(n, P)
7:             seen ← seen ∪ slice
8:             IST ← AddSlice(IST, slice, γ)
9:         end if
10:    end for
11:    return IST
12: end function
13:
14: function AddSlice(IST : STs, slice : ST, γ : int): STs
15:    conflicts ← ∅
16:    for all thread T in IST do
17:        if |slice ∩ T| > γ then
18:            conflicts ← conflicts ∪ {T}
19:        end if
20:    end for
21:    if conflicts = ∅ then
22:        return IST ∪ {slice}
23:    else
24:        slice ← slice ∪ ⋃_{T∈conflicts} T
25:        return AddSlice(IST \ conflicts, slice, γ)
26:    end if
27: end function
```

redundant data, but we also fail to recognize the larger semantic thread and may miss important clones.

We define a set, $IST(P, \gamma)$, that consists of our set of *interesting γ-overlapping semantic threads*. These subgraphs represent our candidates for possible semantic clones.

Definition 3.6 (Interesting Semantic Threads) *The set of* interesting γ-overlapping semantic threads *is a finite set of semantic threads with the following properties:*

1. *The set is complete; its union represents the entire PDG.*
2. *The set must not contain any fully subsumed threads:*
$$\nexists\, sl, sl' \in IST(P,\gamma).sl' \subseteq sl$$
3. *Any two threads in the set share at most γ nodes.*
$$\forall sl, sl' \in IST(P,\gamma).|sl \cap sl'| \leq \gamma$$
4. *$IST(P, \gamma)$ is maximal,* i.e.*, it has the maximal size of all sets that meet properties 1-3.*

With γ set to one, the first code example has two semantic threads. Note that setting γ to zero is precisely equivalent to computing weakly connected components.

Algorithm 1 is a simple greedy algorithm for computing this set. The function `AddSlice` ensures that the final set contains no threads that overlap by more than γ nodes, and the enumeration of each node in the PDG implies the completeness of the returned set. The following argues that Algorithm 1 produces a maximal set.

Lemma 3.7 *If* `AddSlice` *combines two slices, then they must be combined in any set that satisfies the definition of interesting semantic threads.*

PROOF. Consider a procedure P and two statements, s_1 and s_2. Assume that $|f(s_1) \cap f(s_2)| > \gamma$. Their individual presence in the final set would clearly violate property 3.

Let $IST(P, \gamma)$ be an arbitrary set that meets our requirements. Because every node must be included in at least one semantic thread and the domain of semantic threads consists of non-empty unions of forward slices, it follows that:

$$|f(s_1) \cap f(s_2)| > \gamma \Rightarrow \exists\, T \in IST(P,\gamma).f(s_1) \cup f(s_2) \subseteq T$$

Similarly, if two slices must be combined, then any thread that conflicts with this combined thread must also be combined.

`AddSlice` implements this process of recursive greedy combination exactly. □

Theorem 3.8 (Maximality) *The set of interesting semantic threads returned by BST is maximal.*

PROOF. Define $BST(P, \gamma)$ to be the set returned by Algorithm 1. Assume that there exists another set, IST, that meets all requirements and is strictly larger than this set.

$$BST(P,\gamma) = \{T_1, \ldots, T_n\}$$
$$IST(P,\gamma) = \{T'_1, \ldots, T'_n, \ldots, T'_m\}$$

Because each set contains no fully subsumed threads, it follows that there exists at least one "head node," $h_1, \ldots, h_n$ and $h'_1, \ldots, h'_m$ for each semantic thread. By the pigeonhole principle,

$$\exists\, i, j, k.\, h'_i \in T_k \wedge h'_j \in T_k$$

h'_i and h'_j are associated with unique forward slices that do not fully subsume each other. By Lemma 3.7, because `AddSlice` combined these two slices ($f(h'_i) \cup f(h'_j) \subseteq T_k$), they must be combined in every set that meets our requirements. Because these slices are separated in IST, IST does not meet our requirements—a contradiction. □

In the worst case, the algorithm's execution time is cubic in the number of nodes of a given procedure's PDG. In practice, this is not a problem. The size of a given PDG is usually small, in the tens of nodes, and the number of non-subsumed forward slices is considerably less.

The problem also scales gracefully in the sense that procedure sizes are generally bounded: larger code bases have more procedures, not necessarily larger procedures. Finally, our empirical results show that the execution time of this algorithm is inconsequential (Section 5).

3.3.3 Empirical Study

This section contains a brief evaluation of the occurrence of weakly connected components and semantic threads in real systems. We evaluated five open source projects: The GIMP, GTK+, MySQL, PostgreSQL, and the Linux kernel (these same projects are analyzed for clones in Section 5). Figure 7 contains the numbers of weakly connected components per project.

	Procedures	Procedures with n WCCs				
		1	2	3	4	5+
GIMP	13337	7255	3498	1255	627	702
GTK	13284	8773	2967	763	348	433
MySQL	14408	5419	6134	1450	616	789
PostgreSQL	9276	4105	3290	1033	335	513
Linux	136480	60533	52771	13273	5094	4809

Figure 7: Number of weakly connected components.

We noted that each project contains a significant number of procedures with more than one weakly connected component. This suggests that there are functions in real systems that do in fact perform separate computations. Not all of these computations are necessarily interleaved; they could be sequential, and may not represent new targets for clone detection.

Figure 8 counts the number of procedures that contain *non-trivial* weakly connected components ($\gamma = 0$) and $\gamma = 3$ semantic threads.

	Procedures	Procs w/non-triv. $\gamma = 0$ STs	Procs w/non-triv. $\gamma = 3$ STs
GIMP	13337	903	3008
GTK	13284	697	2380
MySQL	14408	1618	2441
PostgreSQL	9276	1221	2267
Linux	136480	10609	22514

Figure 8: Number of procedures with *non-trivial* semantic threads.

Non-trivial semantic threads include interleaved sequences of related code that cannot be detected by current scalable clone detection techniques.

Overall, we are able to consider a significant number of new clone candidates. In addition, the concept of γ-overlapping semantic threads allows us to extend our search to a far greater number of potential clone candidates.

4. IMPLEMENTATION

This section describes the implementation of our tool. It consists of three primary components: AST and PDG generation, vector generation, and LSH clustering. To generate syntax trees and dependence graphs, we use Grammatech's CodeSurfer[1], which allows us to analyze both C and C++ code bases. We output this data to a proprietary format using a Scheme script that utilizes Grammatech's API.

This raw data is read and used by a Java implementation of DECKARD's vector generation engine. This component also performs the syntactic image mapping and semantic vector generation. The LSH clustering back-end of DECKARD is used without modification.

4.1 Implementation Details

DECKARD's vector generation engine previously operated over parse trees. We reimplemented the algorithm to generate vectors for abstract syntax trees. In the process, we made several core improvements.

In order to efficiently utilize our dual core machines, we made the vector generation phase parallel using Java's concurrency API. At present, we use a procedure as a single unit of work. The tasks are inherently independent: generating the vectors for a procedure does not require any data outside of the procedure. Our parallel Java implementation generated vectors faster than we expected (Section 5).

The move to ASTs also posed a challenge. Unlike token-based parse trees, setting the minimum size for a vector was not intuitive. While 30 tokens (DECKARD's default) usually map to about three statements, 30 AST nodes could map to either fewer (less than one) or many more. Instead of judging a vector's size on its magnitude, we utilize the additional semantic information from the AST type hierarchy to judge vectors based on the number of contained statement nodes. We modified both the subtree and sliding window phases to use this new measure.[2]

One challenge the original DECKARD faced was the coverage of all interesting combinations of statements. The coverage was affected by three parameters: the minimum vector size, the size of the sliding window, and the sliding window's *stride*, or how often it outputs vectors. Because the sliding window is now sized on statement nodes, we can permanently set the stride to one and output all interesting vectors: each new vector has at least one new statement.

Instead of operating in a single pass over a fixed vector size, we scan several times, starting at the minimum vector size. Each pass increases the sliding window size by a multiplicative factor, which we have set at 1.5. The sliding window phase terminates when the minimum vector size exceeds the size of the procedure. We apply this exponential sliding window when generating vectors over semantic threads as well.

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

[2] As a usability improvement, we can also set the measure to use lines of code. This can cause issues with multiline statements, though.

4.2 Other Implementation Considerations

Our greatest limitation is that we must have compilable code to retrieve ASTs and PDGs, and only the compiled code is reflected in these structures. At present, we do not have a way to scan code that is deleted by the preprocessor before compilation (other than running multiple builds with different settings). To mitigate this problem for our evaluated projects, we set the configuration to maximize the amount of compiled code whenever possible. For example, our Linux configuration builds every possible kernel option and builds modules for every driver.

The construction of PDGs is not a trivial task, and it presents scaling issues in its own right. CodeSurfer facilitates this process by offering numerous options that control the precision of the PDG build. In order to build PDGs for projects on the million line scale, we were forced to disable precise alias analysis on all builds. This undoubtedly leads to some imprecision in the final graphs, and we could potentially produce multiple semantic threads where only one truly exists. This may cause our tool to miss certain semantic clones, but it does not cause false positives.

With the addition of the semantic vector phase, we have the potential to generate many duplicate vectors. This is not a problem in practice. First, we take advantage of the intraprocedural model and buffer all vectors before printing them, conservatively removing the likely duplicates as they are added. The comparatively small size of a single procedure lets these linear algorithms run quickly.

Second, the LSH back-end is robust against these extra vectors: they merely show up as duplicates in true clone groups or as spurious clone groups. Our post-processing engine quickly removes these.

Third, we exploit the fact that there is a correlation between the number of PDG nodes and the number of AST statement nodes. In the semantic vector phase, we size γ (the overlap constant) to be strictly smaller than our minimum vector size.

5. EMPIRICAL EVALUATION

We evaluated the effectiveness of our tool on five open source projects: The GIMP, GTK+, MySQL, PostgreSQL, and the Linux kernel. The evaluation was performed against DECKARD, the state-of-the-art tool for detecting syntactic clones. In this section, we also present examples of new classes of detectable clones.

5.1 Experimental Setup

We performed our evaluation on a Fedora 6/64bit workstation with a 2.66GHz Core 2 Duo and 4GB of RAM. We used CodeSurfer 2.1p1 and the Sun Java 1.6.0u1 64-bit server VM. To set up the data for analysis, we first maximized the build configuration of each project. We then allowed CodeSurfer to build the PDGs and dump the information to a file. Figure 9 lists the approximate project sizes and build times for our test targets. The size metric is approximate; all whitespace is counted.

	Size (MLoc)	PDG Build Time	PDG Dump Time
GIMP	0.78	25m 57s	20m 40s
GTK	0.88	12m 50s	16m 54s
MySQL	1.13	16m 56s	12m 36s
PostgreSQL	0.74	9m 12s	21m 48s
Linux	7.31	296m 1s	241m 4s

Figure 9: Project sizes and AST/PDG build times.

The PDG builds—especially for the Linux kernel—are particularly expensive. When viewed in the context of other PDG based detection approaches that use subgraph isomorphism testing, though, the build times are quite reasonable. In addition, this cost is incurred once per project: repeated runs of our tool reuse the same input data.

5.2 Performance

Through our testing, we observed that requiring a minimum statement node count of 8 produces clones similar in size to DECKARD's minimum token count of 50. Figure 10 shows the execution times for both our semantic clone detection algorithm and our tree-based algorithm.

	AST Only		AST/PDG	
	VGen	Cluster	VGen	Cluster
GIMP	0m37s	1m11s	0m44s	1m45s
GTK	0m31s	0m57s	0m34s	0m53s
MySQL	0m27s	1m16s	0m29s	1m34s
PostgreSQL	0m40s	1m50s	0m51s	2m30s
Linux	8m42s	6m1s	9m48s	7m24s

Figure 10: Clone detection times.

In this table, the VGen phase performs all vector generation. For both the tree-only and the tree/PDG modes, this includes the subtree and sliding window phases. In addition, the AST/PDG mode enumerates both the weakly connected components and the $\gamma = 3$ semantic threads (Algorithm 1) and enumerates their respective vectors using the sliding window.

Semantic vector generation adds surprisingly little to the execution overhead. We can attribute this to several factors:

- PDGs are in general significantly (about an order of magnitude) smaller than their equivalent ASTs;
- There are relatively few semantic threads per procedure; and
- Our parallel Java implementation allows the utilization of spare CPU cycles that sat idle during the previously IO-bound tree-only phase.

Coverage wise, our tool locates more clones than its tree-only predecessor. This is expected: we produce exactly the same set of vectors, then augment it with vectors for semantic clones. In many cases, we observed that the average number of cloned lines of code per clone group differs significantly between the tree-only and semantic versions of the analysis. As we increase the minimum number of statement nodes for a given clone group, the clones reported by the semantic analysis tend to cover more lines of code than those reported by the tree-only analysis.

We believe this is because when the minimum vector size is set to smaller values, the larger semantic clones are detected simultaneously with their smaller, contiguous constituent components. While the semantics-based analysis is able to tie these disparate components together, it does not necessarily increase the coverage. When the minimum is raised, these smaller components are no longer detected as clones.

Figure 11 contains our coverage results for the Linux kernel. Line counts are conservative: we count the precise set of lines covered by each clone group. Whitespace is ignored, and multi-line statements are usually counted as a single line.

After each of our experiments, we sampled thirty clone groups at random and verified their contents as clones. When the minimum number of statement nodes was set to 4, we experienced a false positive rate of 2 in 30. These false positives took the form of small (two to three lines) snippets of code that incidentally mapped to identical characteristic vectors. When the minimum was set to 8 or more, we found no false positives in these random samples.

```
1  static void zc0301_release_resources(struct zc0301_device* cam)
2  {
3    DBG(2, "V4L2 device /dev/video%d deregistered"
           , cam->v4ldev->minor);
4    video_set_drvdata(cam->v4ldev, NULL);
5    video_unregister_device(cam->v4ldev);
6    kfree(cam->control_buffer);
7  }
```

```
1  static void sn9c102_release_resources(struct sn9c102_device* cam)
2  {
3    mutex_lock(&sn9c102_sysfs_lock);

5    DBG(2, "V4L2 device /dev/video%d deregistered"
           , cam->v4ldev->minor);
6    video_set_drvdata(cam->v4ldev, NULL);
7    video_unregister_device(cam->v4ldev);

9    mutex_unlock(&sn9c102_sysfs_lock);
10   kfree(cam->control_buffer);
11 }
```

Figure 12: Two semantic clones differing only by global locking (Linux).

This low false positive rate is possibly due to the relatively large magnitude of AST-based vectors: the Linux kernel code contained (after macro expansion) an average of 30 AST nodes per line. These larger vectors create a more unique signature for each line of code that is less likely to incidentally match a non-identical line.

5.3 Qualitative Analysis

The quantitative results show that this technique finds more clones with a larger average size. However, this new class of analysis deserves a closer, qualitative look at the results. Semantic clones are more interesting than simple copied and pasted or otherwise structurally identical code. We have observed programming idioms that are pervasive throughout the results.

On a general level, our tool was able to locate semantic clones that were slightly to somewhat larger than their syntactic equivalents, which were also found. The semantic clone often contained the syntactic clone coupled with a limited number of declarations, initializations, or return statements that were otherwise separated from the syntactic clone by unrelated statements. In addition, many semantic clones were subsumed by larger syntactic clones.

We observed cases where our tool was able to locate clone groups that differed only in their use of global locking (Figures 12 and 13). In each case, the tool generated semantic threads for the intrinsic calculation as well as the locking. While the locking pattern itself was too small to be considered a clone candidate, the calculations themselves were matched. In Figure 12, we omitted the third member of the clone group due to space restrictions. This third member also had the locking code in place, and each came from very similar drivers. This lack of locking in one of the three could possibly be indicative of a bug.

Our tool also found clones that differed only by debugging statements. One example appears in Figure 14. While we found several examples of this behavior, we do suspect that we missed other cases due to the fact that logging code often displays current state information. This places a data dependency on the logging code and causes its inclusion in a larger semantic thread.

We were able to discover specific data access patterns. One example appears in Figure 15. The pattern consists of the semantic thread created by the union of the forward slices of the underlined variables. Note that unrelated (data-wise, but perhaps temporally) statements are interleaved through the pattern-forming code. These frequency and complexity of these "patterns" implies that they are possibly prescriptive and not just coincidental. They could then be used as a specification for bug finding.

Min. Nds.	AST Only	PDG/AST
4	935203	940497
8	350804	354079
14	150694	152484
22	65275	66489
32	30039	30367

(a) Cloned LOC

Min. Nds.	AST Only	PDG/AST
4	160934	170544
8	49003	54761
14	16114	18918
22	5692	7439
32	2295	3446

(b) Num. of Clone Groups

Min. Nds.	AST Only	PDG/AST
4	13.9	14.1
8	15.5	16.2
14	20.8	22.5
22	26.5	30.1
32	31.9	38.9

(c) Avg. Cloned LOC / Group

Figure 11: Coverage results for the Linux Kernel.

```
1  static void os_event_free_internal(os_event_t event)
2  {
3     ut_a(event);
4     /* This is to avoid freeing the mutex twice */
5     os_fast_mutex_free(&(event->os_mutex));
6     ut_a(0 == pthread_cond_destroy(&(event->cond_var)));
7     /* Remove from the list of events */
8     UT_LIST_REMOVE(os_event_list, os_event_list, event);
9     os_event_count--;
10    ut_free(event);
11 }
```

```
1  void os_event_free(os_event_t event)
2  {
3     ut_a(event);
4     os_fast_mutex_free(&(event->os_mutex));
5     ut_a(0 == pthread_cond_destroy(&(event->cond_var)));
6           /* Remove from the list of events */
7     os_mutex_enter(os_sync_mutex);
8     UT_LIST_REMOVE(os_event_list, os_event_list, event);
9     os_event_count--;
10    os_mutex_exit(os_sync_mutex);
11    ut_free(event);
12 }
```

Figure 13: Another example of semantic clones differing only by global locking (MySQL).

6. DISCUSSION AND RELATED WORK

This work presents the first scalable clone analysis that incorporates semantic information. Komondoor and Horwitz [12] use the dependence graph to find semantically identical code fragments. They also successfully use this technique [11] to identify candidates for automatic procedure extraction. Our work also has the potential to be used in this way: our semantic threads are similar to the subgraphs that they discover and extract. Their work relies heavily on expensive graph algorithms and pairwise comparisons and does not scale like ours: they report analysis times of more than an hour [12] (not including the PDG build) for a 10,000 line program. Our algorithm's scalability would allow us to analyze larger projects that may have a greater number of duplicate code fragments. This scalability also makes possible a more detailed and direct comparison with different techniques and tools, similar to the experiments performed by Bellon *et al.* [4].

We use a scalable, approximate technique for solving the tree similarity problem. Wahler *et al.* [19] use frequent itemset mining on serialized representations of ASTs to detect clones. Other techniques [13, 17] generate fingerprints of subtrees and report code with similar fingerprints as clones. Compared with our vector-based clone detection, these techniques are less scalable and more coarse grained.

Most potential applications for purely syntactic clone detection are also feasible for semantics-assisted clone detection, and other, new applications exist as well. In the previous section, we identified code patterns that our tool was able to find with the aid of dependency information. Bruntink *et al.* studied the capabilities of token-based and AST-based clone detection tools for detecting crosscutting concerns [5]. Our PDG-based clone definition may further facilitate such a detection since crosscutting concerns may form large semantic threads. Li and Zhou [15] use frequent itemset mining to identify similar code patterns. Their technique is highly scalable as well, but the mined properties lack temporal information–only association. The patterns we inferred are specific and precise, reflecting direct data flow relationships. However, we found fewer total patterns. We leave for future work the study of our tool's efficacy in mining true specifications and the evaluation of these pattern and data-based specifications against those found by automaton-learning techniques [1].

Clone detection has also been used to detect design level similarities. Basit and Jarzabek [2] use CCFinder to detect syntactic clone fragments and later correlate them using data mining techniques. Our semantics-based technique could be used in this way as well, and the ability to detect interleaving patterns might increase the scope of the analysis.

Another potential interesting application of this work is software plagiarism detection. Current, well-used tools include Moss [18] and JPlag[3], but these are too coarse grained to find general sets of code clones. Liu *et al.* [16] have recently developed the GPLAG tool, which applies subgraph isomorphism testing to PDGs to identify plagiarized code. They note that PDGs are resilient to semantics-preserving modifications like (unrelated) statement insertion, statement reordering, and control replacement. Our technique can easily handle interleaved clones, which are characteristic of code with purposefully inserted garbage statements. We expect that our technique can be straightforwardly extended to handle control replacement as well.

We also handle statement reordering: we eliminate ordering information as a result of our transformation of trees to characteristic vectors. Our scalability provides additional opportunities. For example, our tool could be used to perform open source license compliance checks for proprietary software. In this mode, we could generate a large body of vectors representing common or related open source projects and include them in the clustering phase. We leave for future work the evaluation of our tool's applicability to plagiarism detection.

Aside from the scale issues of performing both pairwise comparisons and subgraph isomorphism testing, GPLAG considers only top level procedures as candidates for clones. We are able to consider a much larger set that includes smaller code fragments. Their definition of code similarity as general subgraph isomorphism is also less refined than ours: two isomorphic subgraphs that cross logical flows of data are not likely to be interesting clones. We more carefully enumerate these flows as semantic threads.

7. CONCLUSIONS AND FUTURE WORK

This paper presents the first scalable algorithm for semantic clone detection based on dependence graphs. We have extended the definition of a code clone to include semantically related code and provided an approximate algorithm for locating these clone pairs. We reduced the difficult graph similarity problem to a tree similarity problem by mapping interesting semantic fragments to their related syntax. We then solved this tree-based problem using a highly scalable technique. We have implemented a practical tool based on our algorithm that scales to millions of lines of code. It finds

[3]http://www.jplag.de

```
1   struct nfs_server *server = NFS_SB(sb);
5   struct inode *inode;
6   int error;

8   /* create a dummy root dentry with dummy inode for this superblock */
9   if (!sb->s_root) {
10    struct nfs_fh dummyfh;
16    nfs_fattr_init(&fattr);
17    fattr.valid = NFS_ATTR_FATTR;
18    fattr.type = NFDIR;
```

```
1   struct nfs_server *server = NFS_SB(sb);
4   struct inode *inode;
5   int error;

7   dprintk("--> nfs4_get_root()\n");

9   /* create a dummy root dentry with dummy inode for this superblock */
10  if (!sb->s_root) {
17    nfs_fattr_init(&fattr);
18    fattr.valid = NFS_ATTR_FATTR;
19    fattr.type = NFDIR;
```

Figure 14: Partial semantic clones differing only by a debugging statement (Linux).

```
1   pg_index = heap_open(IndexRelationId, RowExclusiveLock);
2   indexTuple = SearchSysCacheCopy(INDEXRELID,
            ObjectIdGetDatum(indexRelationId), 0, 0, 0);

4   if (!HeapTupleIsValid(indexTuple))
5     elog(ERROR, "cache lookup failed for index %u",
              indexRelationId);

7   indexForm = (Form_pg_index) GETSTRUCT(indexTuple);

9   Assert(indexForm->indexrelid = indexRelationId);
10  Assert(!indexForm->indisvalid);
11  indexForm->indisvalid = true;

13  simple_heap_update(pg_index, &indexTuple->t_self, indexTuple);
14  CatalogUpdateIndexes(pg_index, indexTuple);

16  heap_close(pg_index, RowExclusiveLock)
```

```
1   rel = heap_open(TypeRelationId, RowExclusiveLock);
2   tup = SearchSysCacheCopy(TYPEOID, ObjectIdGetDatum(typeOid),
            0, 0, 0);

4   if (!HeapTupleIsValid(tup))
5     elog(ERROR, "cache lookup failed for type %u",
              typeOid);

7   typTup = (Form_pg_type) GETSTRUCT(tup);

9   typTup->typowner = newOwnerId;

11  simple_heap_update(rel, &tup->t_self, tup);
12  CatalogUpdateIndexes(rel, tup);

14  /* Update owner dependency reference */
15  changeDependencyOnOwner(TypeRelationId, typeOid, newOwnerId);

17  heap_close(rel, RowExclusiveLock); /* Clean up */
```

Figure 15: An example of semantic clones revealing a pattern (PostgreSQL).

strictly more clones than previous syntax-only techniques, and it is capable of producing interesting sets of semantically similar code fragments.

For future work, we are interested in developing an intraprocedural analysis framework that could aid us in generating PDGs more quickly and for other languages. We also plan to explore applications of this technique.

8. REFERENCES

[1] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. In *Proceedings of POPL*, 2002.

[2] H. A. Basit and S. Jarzabek. Detecting higher-level similarity patterns in programs. In *ESEC/FSE*, 2005.

[3] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *ICSM*, 1998.

[4] S. Bellon. Comparison and evaluation of clone detection tools. *IEEE Trans. Softw. Eng.*, 33(9), 2007. Member-Rainer Koschke and Member-Giulio Antoniol and Member-Jens Krinke and Member-Ettore Merlo.

[5] M. Bruntink, A. van Deursen, T. Tourwe, and R. van Engelen. An evaluation of clone detection techniques for identifying crosscutting concerns. In *Proceedings of ICSM*, 2004.

[6] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3), 1987.

[7] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *VLDB*, 1999.

[8] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.*, 12(1), 1990.

[9] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of ICSE*, 2007.

[10] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *TSE*, 28(7), 2002.

[11] R. Komondoor and S. Horwitz. Semantics-preserving procedure extraction. In *POPL*, 2000.

[12] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *SAS*, 2001.

[13] K. Kontogiannis, R. de Mori, E. Merlo, M. Galler, and M. Bernstein. Pattern matching for clone and concept detection. *Automated Soft. Eng.*, 3(1/2), 1996.

[14] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: A tool for finding copy-paste and related bugs in operating system code. In *OSDI*, 2004.

[15] Z. Li and Y. Zhou. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *ESEC/FSE*, 2005.

[16] C. Liu, C. Chen, J. Han, and P. S. Yu. GPLAG: detection of software plagiarism by program dependence graph analysis. In *KDD '06: Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2006.

[17] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *ICSM*, 1996.

[18] S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: local algorithms for document fingerprinting. In *SIGMOD*, 2003.

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

[20] M. Weiser. Program slicing. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, 1981.

[21] R. Yang, P. Kalnis, and A. K. H. Tung. Similarity evaluation on tree-structured data. In *SIGMOD*, 2005.