Hierarchical Program Representation for Program Element Matching

Fernando Berzal, Juan-Carlos Cubero, and Aída Jiménez

Dept. Computer Science and Artificial Intelligence,
ETSIIT - University of Granada, 18071, Granada, Spain
{fberzal|jc.cubero|aidajm}@decsai.ugr.es

Abstract. Many intermediate program representations are used by compilers and other software development tools. In this paper, we propose a novel representation technique that, unlike those commonly used by compilers, has been explicitly designed for facilitating program element matching, a task at the heart of many software mining problems.

1 Introduction

Program element matching is a common problem that must be addressed in many software mining applications. It is required for maintaining several versions of the same program (a.k.a. multi-version program analysis [1]), merging, regression testing automation, understanding the evolution of software code and the nature of software changes [2], detecting duplicated code (or near duplicates) for refactoring (or even bug fixing), and also for concept analysis [3], reverse engineering, and re-engineering.

Quite often, matching is approximated by comparing the textual similarity of program elements, at their source code level [4] [5]. As an alternative approach, some techniques match elements at their syntactic level [6] [7] [8] [9] [10] [11]. This enables them to detect some simple transformations that might go unnoticed at the source code level.

Even though no automatic tool can be perfectly accurate in determining the semantic equivalence of two programs (because of the inherent undecidability of the semantic program equivalence problem), this paper introduces a novel hierarchical program representation that can be useful for matching program elements in situations where existing techniques fall short.

2 Beyond Syntax Trees

A program textual description (i.e. its source code), or its equivalent syntactic representations (e.g. syntax trees and control flow graphs), describes the processes that the program has been designed to perform, at least in imperative programming languages. Substituting a state description for a process description, however, can help simplify its description, provided that we find the right representation for the program structure.

The program dependence graph (PDG) [12], for instance, is an intermediate program representation that makes explicit both data and control dependences. Control dependences are derived from the control flow graph and represent the essential control flow relationships in a program. Data dependences (hazards in the hardware jargon) refer to situations where instructions use data involved in the execution of preceding instructions. True data dependences, also known as RAW (read-after-write) hazards, refer to situations where a statement needs, as an operand, the result computed by a preceding statement. Dependence analysis, thus, is used to discover execution-order constraints in a software program. These constraints help determine if it is safe or not to reorder or parallelize statements, hence their importance in optimizing compilers [13] [14].

Program dependence graphs have also been used for program element matching. Unfortunately, the proposed algorithms are not directly applicable to current programming languages: they work only on limited languages without global variables, pointers, arrays, or procedures [1]; or they are just too inefficient to be of practical use.

- Horwitz [15] determines which program elements have changed by comparing two versions of a program. She builds a program representation graph (PRG) that combines features of program dependence graphs (PDG) and static single assignment forms (SSA). Then, program elements are partitioned into sets of equivalent behavior using an efficient graph partitioning algorithm. The proposed technique is able to flag *semantic* changes that might go unnoticed if we used a text-based program comparator (e.g. direct or indirect uses of changed variable values), but only in a limited language with scalar variables, assignment statements, structured control statements, and output statements.
- Krinke's approach [16] identifies similar code in programs by finding maximal isomorphic subgraphs in program dependence graphs. His algorithm detects subgraphs with identical k-length paths in attributed directed graphs. Unfortunately, high amounts of duplicated code cause exploding running times since testing graph isomorphism is NP-complete. Moreover, large duplicated code sections cause many overlapping duplicates to be reported. This is a common problem with many existing techniques that have grown out of work on compiler optimization, which requires semantic-preserving transformations (i.e. they always err on the side of flagging spurious differences and never miss a real difference).
- Jackson and Ladd's *semantic diff* [17] takes two versions of a procedure and generates a report summarizing the differences between them in terms of the observable input-output behavior of the procedure. Unlike the aforementioned approach, this tool does not check for dependence graph isomorphism. It also sacrifices soundness in the presence of aliasing and pointers. Common semantic-preserving transformations will be correctly interpreted as such (e.g. local variable renaming, using temporary variables for common subexpressions...), but some real differences will be missed (e.g. off-by-one errors). Surprisingly, the lack of alias analysis in this tool turned out not

to be a serious problem in the reported experiments since "aliases occur relatively infrequently" (sic) in practice.
- *Dex (Difference Extractor)* [2] does not use PDGs but abstract semantic graphs (ASGs) in order to compare different versions of a C program. ASGs are no more than standard abstract syntax trees with additional edges indicating type information. As a matter of fact, Dex works on ordered, rooted trees it extracts from the ASGs. Since Dex matches trees, and not graphs, it has a polynomial worst-case time complexity (it avoids working on arbitrary graphs, which would lead to NP complexity). Dex has been used for the analysis of bug fixes, but it should be extended to detect changes that involve dependences between non-contiguous program elements.

By taking dependences into account, three of the aforementioned techniques consider not only the syntactic structure of programs, but also the data flow within them. This makes them robust with respect to the relative ordering of the independent statements in a program, a feature they share with the novel program representation we now introduce.

3 Program Dependence Higraphs

In Nature, complexity frequently takes the form of hierarchic systems – the complex system being composed of subsystems that in turn have their own subsystems –, maybe because hierarchic systems can evolve far more quickly than non-hierarchic systems of comparable size [18]. Hierarchies also appear in human problem solving and, as we are all familiar with, in Software Engineering (a domain with no obvious connections with natural evolution).

In the form of trees, hierarchies provide relatively simple descriptions for complex systems when they are decomposable (or near decomposable). The novel representation model we propose, the program dependence higraph or PDH for short, lends itself to the hierarchical interpretation of any software system.

Figure 1 includes a simple code snippet whose dependence higraph is also shown. We will use this example to illustrate the construction of dependence higraphs. But before we formally present them, we must introduce some preliminary definitions.

3.1 Reduced Blocks

As in any dependence-based representation model, we first perform some preliminary control-flow analysis. However, instead of partitioning the intermediate code into *basic blocks* (i.e. maximal sequences of consecutive instructions), as any self-respecting compiler back end would do, we use *reduced blocks*, which are defined as follows:

A **reduced block** is a minimal, stand-alone, consecutive sequence of instructions whose execution *(a)* has an observable effect in the program state, or *(b)* may affect the program control flow.

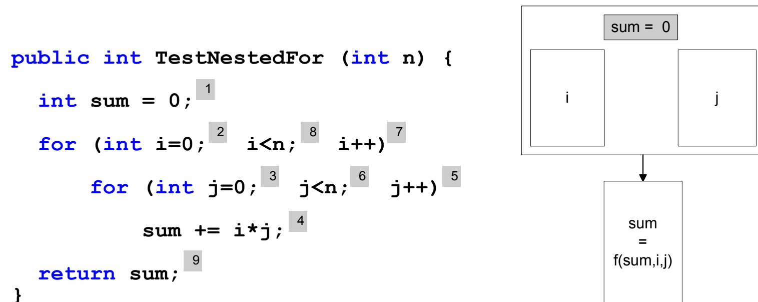

Fig. 1. Simple code snippet (left) and its corresponding dependence higraph (right).

The first case corresponds to sequences of instructions used to evaluate expressions whose results are stored into program variables (including local variables and globally accessible data). This case also applies to procedure calls with visible side effects.

The second case matches those code fragments that appear at the end of basic blocks and determine control flow. Therefore, every basic block will contain, at least, one reduced block (but may contain many of them).

If we view a basic block as a directed acyclic graph (DAG), where common subexpressions are shared among different expressions, then a reduced block is obtained from each tree derived from the DAG representation of the basic block (i.e. we explicitly introduce redundant expressions). This unusual transformation, from the compiler point of view, is intended to convert reduced blocks into small independent black boxes. In some sense, this is similar to the input-output dependence tracking found in *semantic diff* [17]).

In the example shown in Figure 1, we can identify nine reduced blocks. The numbers annotating the source code in Figure 1 mark these blocks. A control flow graph can then be derived from the set of reduced blocks. Figure 2(left) represents the control flow graph derived from our code snippet. It should be noted that, for instance, reduced blocks 4 and 5 would have been merged into a single node if we had used basic blocks instead of reduced blocks for our control-flow analysis.

3.2 The Dominance DAG

Once we have analyzed the program control flow, we must perform some data-flow analysis before we can construct the dependence higraph.

For each reduced block B_i, we define $def(B_i)$ as the set of variables whose values might be modified by the execution of B_i (this includes any variable within the current program element scope: local variables, globally-accessible data, procedure parameters, and function return values). Similarly, we define $use(B_i)$ as the set of variables whose values might be used during the execution of B_i.

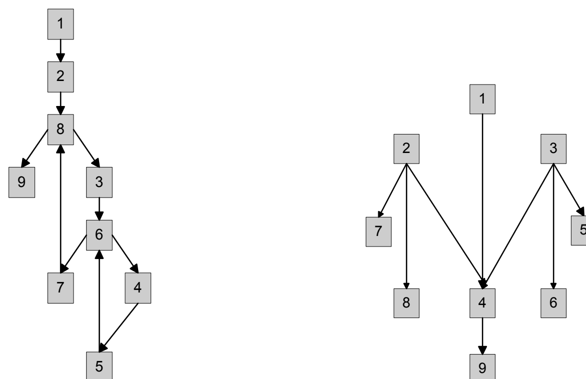

Fig. 2. The control flow graph (left) and the resulting dominance DAG (right) derived from the code snippet in Figure 1.

We say that there is a *def-use chain* from B_i to B_j if a variable v defined by B_i is used by B_j (i.e. $def(B_i) \cap use(B_j) \neq \varnothing$) and there exists a path from B_i to B_j in the control flow graph where the value of v is not changed by any node in the path preceding B_j.

Now, we can define a **strong dominance relationship** as follows: A block B_i strongly dominates a block B_j if, and only if, there is at least one def-use chain from B_i to B_j *and* every path in the control flow graph from the entry node to B_j includes B_i

The dominance relationship computation can be expressed as a data-flow problem and solved using standard data-flow analysis techniques [19]. As an antisymmetric relationship defined among the reduced blocks in a program, it defines a directed acyclic graph that will serve as the basis for the construction of the program dependence higraph.

Figure 2(right) shows the dominance DAG corresponding to the code snippet in Figure 1. It should be noted that, unlike dominator trees [19], which are exclusively defined in terms of the control flow graph, dominance DAGs also incorporate data-flow information. This important difference explains why the strong dominance relationship yields directed acyclic graphs and not just trees.

3.3 Dependence Higraph Definition

Higraphs, as a general kind of diagram, are useful for displaying topological structures [20]. They have applications in databases, knowledge representation, and the behavioral specification of complex concurrent systems (e.g. Harel's statecharts and their descendants, including UML state machine diagrams).

A higraph is a graph whose nodes may contain higraphs within them. Given a higraph node n, the children of n are the nodes in the graph directly within n.

In order to define the program dependence higraph, we extend the strong dominance relationship we defined in the previous section in terms of blocks.

Given two children of a node in the higraph, n_i and n_j, we will say that n_i strongly dominates n_j if two reduced blocks B_i and B_j exist so that B_i is contained within n_i, B_j is contained within n_j, and B_i strongly dominates B_j.

A program dependence higraph (PDH), if we ignore the control and data dependences among the reduced blocks in the program, can then be defined as a tree with two kinds of nodes: P-nodes and S-nodes.

- **P-nodes** have children that are not related by the strong dominance relationship. In a very limited sense, the nodes within a P-node might be *parallelized*.
- On the other hand, **S-nodes**' children are *sequentially* connected by the strong dominance relationship. In other words, if an S-node contains children n_1, n_2... n_k, then n_i strongly dominates n_{i+1} for all the S-node children but n_k (i.e. the last node in the S-node does not dominate any other node in the S-node).

With the program dependence higraph defined as above, the reduced blocks in the program control flow graph will be, therefore, the leaves in the tree defined by the S-node and P-node containment hierarchy.

Figure 1 included the program dependence higraph corresponding to the code snippet shown at its left. The sample program is represented by an S-node whose second child, which corresponds to the `sum` value computation, strongly depends on its first child. This, in turn, is a P-node containing three independent higraphs, which roughly correspond to the `sum` value initialization and the counters that control the execution of the two nested (but independent) loops.

In the next section, we will show how we can obtain such an *intuitive* higraph from the program dependence DAG in Figure 2.

3.4 Dependence Higraph Construction

Program higraph construction can be performed by an iterative bottom-up algorithm that traverses the edges in the program dependence DAG. The program higraph construction algorithm proceeds by merging nodes until there is only one node left, which will be the root of the resulting higraph.

The traversal of the dependence DAG is done backwards in order to lump together nodes that share the same set of predecessors in the DAG:

- We create *S-nodes* containing sequences of nodes (n_1, n_2... n_k) whenever n_i is the only predecessor of n_{i+1} in the current program dependence DAG *and* either *(a)* n_i has no other successors in the current dependence DAG, or *(b)* n_{i+1} has no successors and does not share its set of predecessors with any other node in the DAG. From an intuitive point of view, S-nodes cluster sequences that are tightly related by the strong dominance relationship.
- Next, we use a greedy algorithm to discover *P-nodes* containing sets of nodes $\{n_1, n_2...n_k\}$ that share their sets of predecessors and successors in the current program dependence DAG. Each of this sets will lead to a new P-node in the program dependence higraph.

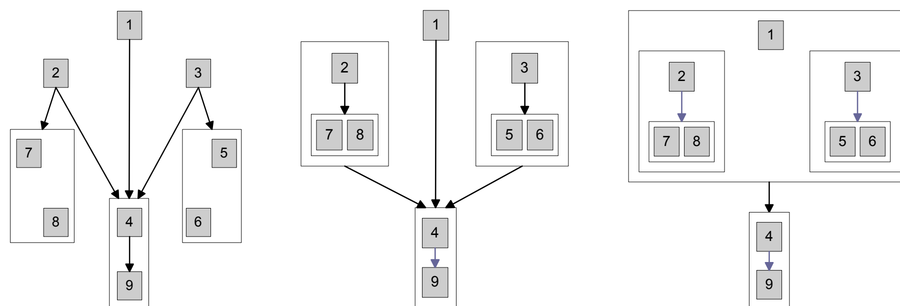

Fig. 3. Dependence higraph construction (from left to right): *(a)* initial S-node and P-nodes; *(b)* identification of S-nodes; *(c)* P-node identification.

Once we create an S-node or a P-node, we replace the enclosed nodes in the dependence DAG by the newly-created node (and its corresponding strong dependences). The process is repeated until the dependence DAG is reduced to a single node, which will become the root of the program dependence higraph. Obviously, this root might be either an S-node or a P-node, depending on the particular situation.

This polynomial-time iterative algorithm eventually terminates because, starting from the finite dependence DAG derived from an imperative program, each algorithm iteration reduces the number of nodes in the DAG.

It is important to note that the two steps above must be performed in the specified order (S-nodes before P-nodes) in order to facilitate the semantic interpretation of the resulting higraph.

Let us now return to the code snippet from Figure 1, whose dependence DAG was shown in Figure 2.

In the first iteration of the PDH construction algorithm, we detect that blocks 4 and 9 can be lumped together into an S-node. Once these blocks are merged into the S-node 49, we also detect that we can create two new P-nodes, merging blocks 7 and 8 on the left side, and blocks 5 and 6 on the right side. We can do this because each pair of blocks share the same set of predecessors in the DAG (i.e. block 2 for the P-node 78, block 3 for the P-node 56). The resulting DAG is shown in Figure 3(a).

Now, the newly-created P-blocks 78 and 56 have only one predecessor, have no successors, and they do not share their sets of predecessors with any other nodes in the current higraph. Therefore, two new S-nodes are created as shown in Figure 3(b).

Next, by a backwards traversal of the current DAG, we find that nodes 278, 1, and 356 share their predecessors (the empty set, since none of them has any predecessor) and they also share their only successor. Hence, we lump them into a new P-node and our original DAG has been reduced to a simple two node higraph, as shown in Figure 3(c).

Therefore, our program dependence higraph consists of an S-node with two children: an initial P-node and a nested S-node following the initial P-node. The resulting higraph matches the one we introduced in Figure 1 and the reader can now check the semantic interpretation we provided when we introduced the idea of dependence higraphs in the previous section.

4 Dependence Higraph Matching Abilities

The main limitation of text- and syntax-based matching algorithms is that relatively simple modifications can hinder the detection of code duplication. A matching tool based on dependences is not so easily confused. We can analyze the potential matching abilities each approach provides by studying some common change scenarios:

- **Verbatim copying** (as in copy & paste programming), for which any technique should properly work.
- **Text insertion/deletion** (adding or removing source code comments, changing I/O messages, and other minor formatting modifications) might reveal some limitations of naive string matching algorithms, although every clone detector should still handle this kind of changes.
- **Renaming** (changing the names of program elements: variables; functions, procedures, or methods; classes, modules, or packages): A clone detector provided with a string tokenizer can handle such changes.
- **Altering assignments and expressions** (adding, removing, or changing variables and expressions in assignment statements): Most clone detectors would also detect this change scenario.
- **Control flow modifications** (changing the circumstances the code will be executed under; e.g. insertion of new conditional statements or changing existing conditions in conditional expressions): A cleverly-devised text-based matching algorithm would detect them, although some syntax-based clone detectors can be misled.
- **Replacement** (changing expressions and control statements for equivalent ones, e.g. `for` → `while`): Most text-based and some syntax-based matching algorithms will fail. Dependence-based algorithm will properly work.
- **Reordering** (changing the order of independent statements while preserving the program semantics, e.g. exchanging `for` loops in the example from Figure 1): In this situation, only dependence-based techniques are reliable.
- **Splitting** (splitting a procedure or module into several ones, as in the 'method extraction' refactoring) and **Merging** (merging procedure bodies, as in procedure inlining): Only dependence higraphs can be truly useful here, since they can benefit from existing embedded tree mining algorithms [21].

As Table 1 shows, syntax-based techniques generally improve token-based matching, even though they might fail to properly match code blocks within new conditional statements. Dependence-based techniques are robust with respect to this kind of modifications and they are also better at the detection of equivalent

	String matching	**Lexical analysis**	**Syntactic analysis**	**Dependence graphs**	**Dependence higraphs**
Verbatim copying	■	■	■	■	■
Text insertion/deletion	⊠	■	■	■	■
Renaming	□	⊠	■	■	■
Assignment modifications	□	■	■	■	■
Control flow alteration	□	■	⊠	■	■
Replacement	□	⊠	⊠	■	■
Reordering	□	□	⊠	■	■
Splitting	□	□	□	⊠	■
Merging	□	□	□	⊠	■

Table 1. The potential matching abilities of alternative program representation techniques (■ excellent, ⊠ partial, □ poor).

control structures. Finally, they are not easily confounded by semantic-preserving statement reordering.

5 Conclusions and Future Work

Text matching tools are usually line-based, so even lexical changes with no syntactic effects might spuriously appear as modifications. Syntactic matching tools are able to properly detect some code transformations, but they are only able to show the syntactic scope of the detected change. This could be a severe drawback if we are interested in understanding the effects of a given change (as maintainers must do not to introduce new bugs when modifying an existing code base).

Semantic matching techniques, and the use of dependence graphs in particular, eliminate this limitation, since the repercussions of a change can be determined by following dependence edges in the dependence graph. Unfortunately, traditional semantic-based techniques are computationally expensive (global analysis is often unfeasible in large systems).

Dependence higraphs, however, are amenable to some optimizations due to their hierarchical structure. S-nodes, due to their sequential structure, can be matched as ordered, rooted trees to find perfect matches (dynamic programming can be used for approximate matches). Unordered P-nodes matching can be sped up with the help of heuristics.

Another interesting (and differentiating) feature of dependence higraphs is that, due to their hierarchical nature, they are well suited for detecting some change scenarios beyond the scope of previous techniques (procedure splitting and merging, for instance). This fact opens up new possibilities in the study of refactorings, aspect mining, and software modularization in general.

References

1. Kim, M., Notkin, D.: Program element matching for multi-version program analyses. In: MSR '06: Proceedings of the 2006 International Workshop on Mining Software Repositories. (2006) 58–64
2. Raghavan, S., Rohana, R., Leon, D., Podgurski, A., Augustine, V.: Dex: A semantic-graph differencing tool for studying changes in large code bases. In: ICSM'2004. (2004) 188–197
3. Tonella, P.: Formal concept analysis in software engineering. In: ICSE'2004. (2004) 743–744
4. Hunt, J.W., Szymanski, T.G.: A fast algorithm for computing longest common subsequences. Communications of the ACM **20**(5) (1977) 350–353
5. Tichy, W.F.: The string-to-string correction problem with block moves. ACM Trans. Comput. Syst. **2**(4) (1984) 309–321
6. Yang, W.: Identifying syntactic differences between two programs. Software – Practice & Experience **21**(7) (1991) 739–755
7. Hunt, J.J., Tichy, W.F.: Extensible language-aware merging. In: ICSM'2002. (2002) 511–520
8. Neamtiu, I., Foster, J.S., Hicks, M.: Understanding source code evolution using abstract syntax tree matching. In: MSR'2005: Proceedings of the 2005 International Workshop on Mining Software Repositories. (2005) 282–290
9. Laski, J., Szermer, W.: Identification of program modifications and its applications insoftware maintenance. In: ICSM'1992. (1992) 1–5
10. Apiwattanapong, T., Orso, A., Harrold, M.J.: A differencing algorithm for object-oriented programs. In: ASE'2004. (2004) 2–13
11. Wang, Z., Pierce, K., McFarling, S.: Bmat – a binary matching tools for stale profile propagation. Journal of Instruction-Level Parallelism **2** (2000)
12. Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst. **9**(3) (1987) 319–349
13. Muchnick, S.: Advanced Compiler Design and Implementation. Morgan Kaufmann (1997)
14. Allen, R., Kennedy, K.: Optimizing Compilers for Modern Architectures: A Dependence-based Approach. Morgan Kaufmann (2001)
15. Horwitz, S.: Identifying the semantic and textual differences between two versions of a program. In: PLDI'1990. (1990) 234–245
16. Krinke, J.: Identifying similar code with program dependence graphs. In: WCRE'2001. (2001) 301–309
17. Jackson, D., Ladd, D.A.: Semantic diff: A tool for summarizing the effects of modifications. In: ICSM'1994. (1994) 243–252
18. Simon, H.A.: The Sciences of the Artificial. 3rd edn. The MIT Press (1996)
19. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. 2nd edn. Addison Wesley (2006)
20. Harel, D.: On visual formalisms. Communications of the ACM **31**(5) (1988) 514–530
21. Zaki, M.J.: Efficiently mining frequent embedded unordered trees. Fundamenta Informaticae **66**(1-2) (2005) 33–52