A Semantic Difference Algorithm for Structured Visual Dataflow Programs

Philip T. Cox, Tuan Ahn Dang

Faculty of Computer Science, Dalhousie University, 6050 University Avenue
Halifax, Nova Scotia, Canada B3H 1W5
pcox@cs.dal.ca, amh@cs.dal.ca

Received (November 10, 2010)

The diff utility is an important basic tool, providing a foundation for many of the fundamental practices of software development, such as source code management. While there are many file differencing tools for textual programming languages, including some that look at more than simple textual variations, there are few for visual programming languages. We present an algorithm for semantic comparison of programs in controlled visual dataflow languages; that is, languages in which dataflow diagrams are embedded in control structures. This algorithm performs depth-first search of call structures, comparing embedded diagrams using a graph matching algorithm such as subgraph isomorphism, to determine if two programs are semantically equivalent, and if they are not, discovers the differences. We use the visual language Prograph for illustration; however, the mechanism we propose could be applied to any controlled dataflow language, such as LabVIEW. Results are reported of experiments in which the algorithm is applied to large code samples.

Keywords: Visual programming language; structured dataflow; semantic difference; code management; heuristic search; graph isomorphism.

1. Introduction

Although visual programming languages (VPLs) have been the subject of continuing research for at least the last 25 years, they, unlike their textual counterparts, have made few inroads into the world of industrial software development. While it has become the norm to use visual representations to specify the architecture of software systems, visual representation of algorithms has not caught on as a replacement for or supplement to standard, imperative, textual programming languages (TPLs). This lack of success is at least partly due to the reluctance of professional developers to invest in learning about a new technology [6]. However, as pointed out by participants in a focus group study conducted by Apple to determine the viability of Prograph CPX as a development environment for Windows applications, unless the new technology satisfies certain criteria, the professional developer should be wary of adopting it [28]. To become a viable alternative to textual programming, a VPL should, among other things, include visual counterparts of the many code management and analysis tools available for TPLs, the focus of the work reported here.
One of the mainstays of many code management tools is differencing, exemplified in its simplest form by the Unix `diff` command which finds lexical differences between two text files or source programs. It is used to manage modifications and roll back changes, reveal anomalies during debugging, manage concurrent changes made by several people, and merge changes from different versions of a program. Differencing underpins many source code control systems.

Most VPLs that have achieved some level of industrial success are based on the dataflow model, and are either domain-specific or general purpose, and structured or unstructured, where a structured dataflow VPL (SVPL), is one in which the dataflow diagrams are acyclic, enclosed in control structures of some kind, and have the single-assignment property [9]. Some examples are as follows.

Although Simulink, a dataflow VPL for simulation of physical systems, provides some control structures, it is primarily unstructured, allowing feedback loops appropriate to its application domain [22]. LabVIEW and VEE are structured and domain-specific, designed for data acquisition and virtual instrument control [1,5]. Prograph is structured and general purpose [11]. During its commercial life, Prograph CPX was used in a range of projects where C++ would have been the usual choice [27], including building CD-based multimedia games, a spreadsheet application [30], an online stock trading system, and an automobile configuration system. To our knowledge, it is the only visual programming environment that has been used in this way, as a replacement for traditional text-based tools in industrial software development. Hence, in recent research on tools for supporting software development in VPLs, we focussed on SVPLs, developing an algorithm for comparing programs in such languages, reported in [11], although the underlying principles could also be applied to unstructured dataflow VPLs such as Simulink. This paper expands on that work by reporting on experiments that test the algorithm on a substantial a body of Prograph code.

In the next section, we discuss differencing in textual software development, and briefly review existing differencing tools in VPLs. We then define semantic equivalence, an equivalence relation on program elements in an SVPL, and present an algorithm for determining semantic equivalence or discovering semantic differences.

2. Differencing

File differencing first appeared in Unix in the early 1970s, using an algorithm reported by Hunt and McIlroy [19]. The standard algorithm for lexical differencing was described by Miller and Myers [25,26] and Ekkonen [29]. The Unix `diff` utility compares any text files, so when applied to programs, frequently produces irrelevant results. For example, a minor difference, such as an extra space or line break can contribute significantly to the result of a comparison, since `diff` looks for physical differences rather than syntactic ones.
Syntactic comparison utilities improve on lexical tools by comparing the syntax trees of programs, but they too can produce misleading results [16]. To overcome their limitation, various comparison methods have been proposed that build structural representations of programs, allowing semantics to be taken into account. One such representation is program dependence graphs (PDGs) which include both control and dataflow information [15]; another is program representation graphs, a variant of PDGs in which extra variables are introduced to obtain the single assignment property [18]. Semantic differencing tools based on applying graph isomorphism to such graphs have achieved some level of success [1,20].

From the perspective of structure reflecting semantics, SVPLs have an inherent advantage over TPLs. To apply semantic differencing to programs in standard imperative TPLs, the graphs that represent the semantics must first be constructed. SVPLs are functional, however, so their semantics, like those of a functional language, are closely aligned with their syntax, a dataflow graph at the lowest level [21]. Hence no construction phase is necessary.

While extensive research on comparison of textual programs has resulted in a variety of techniques, briefly summarised above, little work has been done on comparing programs in visual languages. Given the relationship between visual dataflow and textual functional languages, one might expect to find semantic comparison techniques developed for the latter that could be applied to the former. Functional language researchers, however, seem to have concentrated their efforts on transformations that can be applied to functional programs that alter their structure while preserving semantics, such as the work of Burstall and Darlington [7]. Structural transformations could possibly be used to show two dataflow programs to be equivalent by reducing them both to some canonical form, but would not contribute to finding differences.

The Windows version of Prograph CPX provides a simple file comparison tool. However, it detects only superficial syntactic differences, and does not provide any useful information about the structural (and therefore semantic) differences between dataflow diagrams.

LabVIEW includes a tool for comparing two dataflow diagrams by looking for a maximal pair of isomorphic subgraphs [5]. However, it matches only syntactically identical vertices, ignoring semantic equivalence that might be determined by comparing the diagrams that implement the vertices. Hence, although it is more sophisticated than the mechanism provided by Prograph, it is still essentially a syntactic rather than semantic tool.

SimDiff, a model comparison tool for Simulink has functionality similar to that of the LabVIEW utility, providing a single-level syntactic match between dataflow graphs [14].
3. Structured Visual Programming Languages

We use Prograph as the sample language on which to base our discussion of comparison in SVPLs. Although we assume the reader is familiar with Prograph, we will briefly review the example in Fig. 1. to introduce notation and terminology. A detailed description can be found in [11].

A Prograph program is a set of methods together with a set of persistents, which are globally accessible storage locations. A persistent always has an associated value, the initial value of which is called its static value. A method consists of a sequence of cases, each a dataflow diagram of operations connected by datalinks. For example, the two cases of a method quicksort are as in Fig. 1. Every operation has a type, which is one of input bar, output bar, primitive, match, constant, persistent, call or local. Each case has exactly one input bar, usually adorned with little circles denoting roots (data sources), and exactly one output bar, adorned with terminals (data sinks), which respectively pass values into and out of the diagram. In the first case of quicksort there is a match operation, named (), which tests the value flowing into it from the input bar, and a constant () which passes its value to the output bar. In the second case of quicksort, detach-l, attach-l, and (join) are primitive operations, which invoke built-in functions; the two quicksort operations are calls, which initiate executions of the quicksort method; and partition is a local operation which represents a sequence of cases (not shown).

Each operation has a sequence of terminals along the top, where data flows in, and a sequence of roots along the bottom where results flow out. Each terminal and root has a type, which can be simple, list or loop. All terminals and roots in the example

Fig. 1. Prograph method quicksort
are simple, except for the roots and the rightmost terminal of the partition operation, which are of type list. Each operation also has a control associated with it. In Fig. 1., the match ( ) in the first case of quicksort has the control next-case-on-failure $\times$, while all other operations have the control continue-on-success, which has no visual representation. When quicksort is invoked, its first case is attempted. If the incoming value is the empty list, the match succeeds, the empty list is passed to the output bar and execution of quicksort concludes. Otherwise, the first case is abandoned, and the second case tried. The head is removed from the list by the detach-l operation, and the head and tail passed to the partition operation. The list annotation $\epsilon\rightarrow\phi$ on the second terminal of partition indicates that the operation will be applied to each element of the incoming list. If the comparison in the first case of partition succeeds, then the current element from the tail of the list is less than the head, and is passed to the left terminal of the output bar; otherwise the second case is executed, passing this element to the right terminal of the output bar. The list annotations on the roots of partition indicate that the items produced by the executions of partition are assembled into lists, which are respectively, elements of the tail that are less than the head, and those that are not. The two lists are then sorted, and the resulting sorted list assembled and passed to the output bar of the second case of quicksort.

The only operation type not represented in Fig. 1. is persistent. A persistent operation refers to a persistent by name, and may have one at most one terminal, and at most one root for, respectively, setting and getting the value of the associated persistent. The example also does not include a synchro, which is a link of the form $\leftrightarrow\ldots\leftrightarrow$, from one operation to another that enforces order of execution, or a loop-annotated terminal/root pair (depicted $\land$ and $\lor$), which corresponds to a loop variable in procedural languages. An example of a synchro can be found in Fig. 2., and of loop terminal/root pairs in Fig. 3.

![Fig. 2. The dataflow diagram on the right includes a synchro link from the local operation div to the match operation $\theta$, ensuring the former is executed before the latter. The two graphs are isomorphic but violate equivalence conditions (see Definition 1:)](image-url)
To illustrate the common features of SVPLs, Fig. 3 shows two programs implementing the bubblesort algorithm, in Prograph and LabVIEW. The inner loop of bubblesort, which makes one pass through the input list bubbling the largest element to the end, is implemented in Prograph by the local operation inner, and corresponds to the similarly labelled for loop in the LabVIEW program. The outer loop, which executes the bubbling process \( n \) times, where \( n \) is the list length, is implemented in Prograph and LabVIEW by the local operation outer and the for loop labelled “outer”, respectively. The loop-annotated terminal/root pair and the list root on the local operation outer in the Prograph program correspond, respectively, to the shift registers (\( \square \) and \( \square \)) and auto-index tunnel (\( \square \)) on the for loop “outer” in the LabVIEW program. The two cases of inner in Prograph correspond to the two diagrams of the conditional structure in LabVIEW (only the False diagram is shown). We invite the
reader to identify the remaining syntactic and semantic similarities between the two programs. There are, of course, differences too. For example, the comparison operation is outside the LabVIEW conditional construct, but inside the corresponding local operation in Prograph.

4. Equivalence of SVPL Programs

In this section, we provide a definition of equivalence of SVPL programs, based on a comparison of their structure. Since a structural comparison depends on the details of the specific SVPL, we use Prograph as the basis for our definition, but will argue that the definition nevertheless applies to any language of this class.

Note that input and output bars behave like primitives in the sense that they invoke built-in functions that perform common tasks; so, for the purposes of the definition that follows, we assume that all input bars have the same name, and all output bars have the same name. Since a local operation can be replaced by a call X and a corresponding method invoked only by X, we will assume that programs have no local operations.

To streamline the definitions, we introduce some notation as follows. If P is a program, \( \text{opers}(P) \) denotes the set of all operations occurring in P. If C is a case, \( \text{opers}(C) \) denotes the set of operations in the dataflow diagram of C. If X is an operation, terminal or root, \( \text{type}(X) \) denotes the type of X. If X is a method, \( \text{cases}(X) \) denotes its sequence of cases. If X is a call, \( \text{ref}(X) \) denotes the corresponding method, and we use \( \text{cases}(X) \) and \(|X|\) as shorthand for \( \text{cases}(\text{ref}(X)) \) and \( \text{cases}(\text{ref}(X))\) respectively. If X is a persistent operation, \( \text{ref}(X) \) denotes the corresponding persistent. If X is any other operation, \( \text{ref}(X) \) denotes the name of X.

If X is an operation, then \( \text{roots}(X) \), \( \text{terms}(X) \) and \( \text{arity}(X) \) denote, respectively, the sequence of roots of X, the sequence of terminals of X, and the pair of integers \( (|\text{terms}(X)|,|\text{roots}(X)|) \).

If X is a sequence, \( X_i \) denotes its \( i^{th} \) element.

**Definition 1:** If P is a program and \( \equiv \) is an equivalence relation on \( \text{opers}(P) \), then \( \equiv \) is called a semantic equivalence iff \( \forall B,C \in \text{opers}(P) \), if \( B \equiv C \) then

1. \( \text{arity}(B) = \text{arity}(C) \), and
2. \( \forall i \ (1 \leq i \leq |\text{terms}(B)|), \ 	ext{type}(\text{terms}(B)_i) = \text{type}(\text{terms}(C)_i), \ and \)
3. \( \forall i \ (1 \leq i \leq |\text{roots}(B)|), \ 	ext{type}(\text{roots}(B)_i) = \text{type}(\text{roots}(C)_i), \ and \)
4. B and C have the same control, and
5. \( \text{type}(B) = \text{type}(C), \ and \)
6. either
   6.1. \( \text{ref}(B) = \text{ref}(C) \)
   6.2. \(|B| = |C|, \ and \)
   6.3. \( \forall i \ (1 \leq i \leq |B|) \) there is a bijection \( f : \text{opers}(\text{cases}(B)_i) \rightarrow \text{opers}(\text{cases}(C)_i) \)
   such that
6.3.1. \( \forall A \in \text{opers}(\text{cases}(B_i)), A \equiv f(A), \) and

6.3.2. \( \forall D,E \in \text{opers}(\text{cases}(B_i)):
\)

(a) there is a datalink from roots(D) to terms(E)_{j_k} for some \( j \) and \( k \)

\( \iff \) there is a datalink from roots(f(D))_{j_k} to terms(f(E))_{j_k}

(b) there is a synchro from D to E

\( \iff \) there is a synchro from f(D) to f(E).

Two operations B and C in a program P are said to be semantically equivalent iff there exists a semantic equivalence on \( \text{opers}(P) \) such that \( B \equiv C \).

Semantic equivalence classifies operations according to what they compute. For non-call operations, this is easily determined (6.1). Functionality of a call is determined by the structure of the sequence of cases of the corresponding method. The bijection between dataflow diagrams defined in 6.2 is a more constrained form of graph isomorphism. For example, condition 6.3.2(a) requires that datalinks not only connect corresponding operations in two graphs as required for isomorphism, but also connect corresponding terminals and roots on those operations. So although the two graphs in Fig. 2. are isomorphic, they violate several of these extra conditions: specifically, the roots of detach-1 are connected to the terminals of div in a different order, violating 6.3.2(a); the () operations have different controls, violating 4; the operations div and div have different types, violating 5, and different arities, violating 1; the show and display operations have different references, violating 6.1; and the terminals of show and display have different types, violating 2.

As noted above, the semantics of structured dataflow programs, like the semantics of functional programs and unlike those of imperative languages, is closely aligned with the syntax. Hence, although the above definition of semantic equivalence appears to be purely syntactic, it captures the notion of identical input/output behaviour, as does semantic equivalence of textual programs [20]. The following lemma states the relationship between semantic equivalence and the execution functions of Prograph program elements [27].

**Lemma 1:** If \( P \) is a program A, B \( \in \text{opers}(P) \) and \( \text{arity}(A) = \text{arity}(B) = (m,k) \), then A and B are semantically equivalent iff \( f_A(w) = f_B(w) \) for every \( m \)-tuple \( w \) of values, where \( f_A \) and \( f_B \) are the execution functions of A and B, respectively.

We omit the proof of this lemma for two reasons. First, while the reader may be familiar with the Prograph language, the proof relies on a detailed knowledge of its formal semantics [27], and would inject a tedious distraction from the main point of this work, a practical differencing algorithm. Second, while the connection between syntax and semantics that the lemma expresses applies to any SVPL, the proof of this particular version would apply only to Prograph.

While comparing two operations in one program is useful, programmers often want to compare two different versions of a program. Accordingly, we need to extend the definition of semantic equivalence. First, we note that if the two programs we...
wish to compare have disjoint name spaces, and we combine the two programs into one, the lemma will still hold in the absence of persistents. If persistents are involved, however, we need to ensure that there is a one-to-one correspondence between appropriate subsets of the persistents of the two programs. Accordingly, we extend the above definition, obtaining the following one, in which \( \text{pers}(P) \) denotes the set of persistents of a program \( P \), and \( \text{value}(X) \) denotes static value of a persistent \( X \).

**Definition 2:** If \( P_1 \) and \( P_2 \) are two programs, which we can assume without loss of generality to have no names in common, \( A \in \text{opers}(P_1) \) and \( B \in \text{opers}(P_2) \), then \( A \) in \( P_1 \) is semantically equivalent to \( B \) in \( P_2 \), denoted \( A[P_1] \equiv B[P_2] \) iff for some \( V_1 \subseteq \text{pers}(P_1) \) and \( V_2 \subseteq \text{pers}(P_2) \), there is a bijection \( g: V_1 \rightarrow V_2 \) such that \( \forall X \in V_1, \text{value}(X) = \text{value}(g(X)) \) and \( A \equiv B \), where

- \( \equiv \) is a semantic equivalence relation on \( \text{opers}(P') \),
- \( P' \) is the program obtained by combining \( P_1 \) and \( P_2 \), and
- \( P'_2 \) is obtained by renaming persistent operations in \( P_2 \) as follows:
  - if \( C \) is a persistent operation in \( P_2 \), and \( \text{ref}(C) = G \) for some \( G \in \text{pers}(P_2) \) such that \( G \in V_2 \) then modify \( C \) so that \( \text{ref}(G) = g(G) \).

Note that Lemma 1 also holds for this extended definition.

**5. The Comparison Algorithm**

In this section, we present and discuss an algorithm that determines whether two methods in two programs are semantically equivalent, and if not, finds differences between them. Note that, although Definition 1 defines semantic equivalence for operations, as a by-product, it embodies the definition of semantic equivalence for methods.

The algorithm uses depth-first search to traverse the two programs, guided by heuristics based on estimates of the numbers of differences between the items being compared. We say “estimates”, because there may be more than one way to account for the differences between two programs. For example, we might decide that the difference between the two operations in Fig. 4 resulted from changing the types of the second and third terminals. Alternatively, we might conclude that the difference arose from dragging the second terminal to the right of the third. Although this ambiguity might be resolved by, for example, looking to see what roots the terminals are connected to, it is generally not possible to provide a precise account of semantic differences [20].
5.1. Counting differences

To count differences, we define two functions, \textbf{Count} and \textbf{Local}. \textbf{Local} applies to pairs of operations, methods or cases, and to subgraph isomorphisms between cases, producing an estimate of the number differences which can be observed \textit{locally}, that is, by examining only the structure of its argument. \textbf{Count} applies to subgraph isomorphisms, and to pairs of operations or methods, producing a count that includes differences contributed by other parts of the program.

5.1.1. Operations

\textbf{Local}((A,B)), the number of differences between two operations A and B, is computed according to conditions 1, 4, 5 and 6.1 of Definition 1. In view of the bijection required by Definition 2, however, condition 6.1 is not applied to persistents, which are discussed later. To illustrate, consider the two operations in Fig. 5. The numbers of roots of these operations differ by 1, violating 1; their first terminals have different types, as do their second terminals, violating 1; and the operations have different controls and types, violating 4 and 5. Hence, in this example \textbf{Local} is 5. Note that we have chosen not to compare types of roots (or terminals) if the numbers of roots (or terminals) differ.

\begin{figure}
\centering
\includegraphics[width=0.4\textwidth]{Fig5}
\caption{Differences between operations}
\end{figure}

5.1.2. Methods

When two methods \(M_1\) and \(M_2\) are compared, the local difference count is computed as:

\[
\text{Local}((M_1,M_2)) = ||M_1| - |M_2||
\]

and the total difference count as:

\[
\text{Count}((M_1,M_2)) = \text{sum}\{\text{Count}(C_{1i},C_{2j}) | 1 \leq i \leq n, C_{1i} \text{ and } C_{2j} \text{ are the } i^{th} \text{ cases of } M_1 \text{ and } M_2, \text{ and } n = \min(|M_1|,|M_2|)} + \text{Local}(M_1,M_2)
\]

where \textbf{Count} for pairs of cases is computed as discussed below. Note that we have assumed that if one method has more cases than the other, then we should match cases in sequence, starting at the beginning, and treat the extra cases at the end of the longer sequence as “differences”, that is, items that have been added, and do not correspond with anything in the smaller sequence. This is an arbitrary choice, but is cheap to compute compared to alternatives involving a search for the best match.
5.1.3. Cases and subgraph isomorphisms

Comparing two cases $C_1$ and $C_2$ is somewhat more complicated. First, from each case is derived a directed acyclic graph such that the vertices are the operations, and there is an edge from $A$ to $B$ if there is a datalink or a synchro from $A$ to $B$. Second, the set $S(C_1, C_2)$ of subgraph isomorphisms between the two graphs is computed. Note that because of their special status as transmitters of values into and out of a case, the input and output bars of one case must be mapped to the input and output bars of the other. Hence $S(C_1, C_2)$ excludes any function which violates this condition. Finally, for each function $f$ in $S(C_1, C_2)$, several counts are computed, according to conditions in Definition 1, as follows.

Condition 6.2 requires a bijection between the cases, but although $f \in S(C_1, C_2)$ is injective, it is not necessarily surjective. A measure of the extent to which $f$ is not surjective is provided by

$$\text{xoCount}(f) = |\text{opers}(C_2)| - |\text{opers}(C_1)|$$

the number of extra operations in the larger case.

Condition 6.3.2(a) requires that the bijection preserves each datalink in $C_1$. The number of mismatched datalinks, $d\text{Count}$, is computed by counting the terminals in $C_1$ which have datalinks attached that comply with the condition, and subtracting this number from the total number of terminals in $C_1$.

$$d\text{Count}(f) = |\{ T \mid T \text{ is a terminal of some } A \in \text{opers}(C_1) \}|
- |\{ T \mid \text{ for some } A \in \text{opers}(C_1) \text{ and some } i, T = \text{terms}(A)_i, \text{ either there is no datalink attached to } T \text{ and no datalink attached to } \text{terms}(f(A))_i \text{ or for some } B \in \text{opers}(C_1), |\text{roots}(B)| = |\text{roots}(f(B))|, \text{ and for some } j \text{ there are datalinks from } \text{roots}(B)_j \text{ to } T \text{ and from } \text{roots}(f(B))_j \text{ to } \text{terms}(f(A))_i \}|$$

The computation of $d\text{Count}$ considers all the datalinks in $C_1$ and any datalink in $C_2$ attached to a terminal of some operation $B$ such that $B = f(A)$ for some operation $A$ of $C_1$. However, we need to account for the remaining datalinks in $C_2$, which are counted as follows:

$$xd\text{Count}(f) = |\{ T \mid T \text{ is a terminal of some } A \in \text{opers}(C_2) \} - \{ f(B) \mid B \in \text{opers}(C_1) \} \text{ and there is a datalink attached to } A \}|$$

Finally, by condition 6.3.2(b), it is necessary to count mismatched synchros, accomplished as follows:

$$xs\text{Count}(f) = \text{number of synchros in } C_1 + \text{number of synchros in } C_2
- 2|\{ A, B \in \text{opers}(C_1) \text{ and there is a synchro from } A \text{ to } B \text{ and a synchro from } f(A) \text{ to } f(B) \}|$$
Using these functions, a count of the local differences between cases that arise from subgraph isomorphism \( f \) is calculated as follows:

\[
\text{Local}(f) = \sum \{ \text{Count}(A, f(A)) \mid A \in \text{opers}(C_1) \} \\
+ \text{xolCount}(f) + \text{dCount}(f) \\
+ \text{xdCount}(f) + \text{xsCount}(f)
\]

and the total difference count is computed as:

\[
\text{Count}(f) = \text{Local}(f) \\
+ \sum \{ \text{Count}(\text{ref}(A), \text{ref}(f(A))) \mid A \in \text{opers}(C_1) \text{ and both } A \text{ and } f(A) \text{ are calls} \} \\
+ \sum \{ \text{pCount}(\text{ref}(A), \text{ref}(f(A))) \mid A \in \text{opers}(C_1) \text{ and both } A \text{ and } f(A) \text{ are persistent operations} \}
\]

The function \( \text{pCount} \) occurring in the last expression cannot be described in the same neat declarative fashion as the others since it deals with persistents, the non-functional feature of Prograph, similar to non-functional features frequently found in other functional languages. According to Definition 2, two persistent operations are the same if the persistents they refer to are related by a bijection. This bijection, however, is not known in advance, and must be computed on the fly, as discussed in Section 5.4 below.

Finally, the local and total difference counts for the cases \( C_1 \) and \( C_2 \) are computed as follows.

\[
\text{Local}((C_1, C_2)) = \begin{cases} 
1 + \|\text{opers}(C_1)\| - \|\text{opers}(C_2)\| \\
+|\text{no. of datalinks in } C_1 - \text{no. of datalinks in } C_2| \\
+|\text{no. of synchros in } C_1 - \text{no. of synchros in } C_2| \\
\text{if } S(C_1, C_2) = \emptyset \\
\infty \text{ otherwise}
\end{cases}
\]

\[
\text{Count}((C_1, C_2)) = \min(\{ \text{Count}(f) \mid f \in S(C_1, C_2) \} \cup \{ \text{Local}((C_1, C_2)) \})
\]

Note that if there are no subgraph isomorphisms between the cases, there is no reasonable way to compare them in detail, so we have chosen a formula which gives a rough estimate of the difference in size, and is cheap to compute. The 1 in this formula is necessary to ensure correctness (see Lemma 2, Section 6).

### 5.2. The Algorithm

Since the structure of the comparison algorithm is a standard depth-first search, we will describe it informally rather than by providing a listing, concentrating instead on its unique features. For simplicity, we assume that the programs being examined have no persistents, and discuss later how they are dealt with.

The algorithm traverses a search tree, each vertex of which is either a pair of methods, a pair of cases, or a subgraph isomorphism between cases. We refer to these as \textit{method}, \textit{case} and \textit{isomorphism} nodes, respectively, indicated by \( M \), \( C \) and \( I \) in Fig. 6. The pair of methods or cases in a node indicates program elements to be com-
pared, while the structure of the subtree descending from a node results from
propagating this comparison through the calling structure of the program, as required
by the definitions of the counting functions above.

In general, the children of a method node are the case nodes \((C_{11},C_{21}),\ldots,(C_{1n},C_{2n})\)
where \(n=\min(|M_1|,|M_2|)\) and for each \(i\), \(C_{1i}\) and \(C_{2i}\) are the \(i^{th}\) cases of \(M_1\) and \(M_2\).
However, if the method node is a descendent of another method node consisting of
the same pair of methods, then it has no children. The children of a case node \((C_1,C_2)\)
are the nodes consisting of the functions in \(S(C_1,C_2)\), so if there are no subgraph
isomorphisms between the cases, the case node has no children. The children of an
isomorphism node \(f\) are the method nodes of the form \((\text{ref}(A),\text{ref}(f(A)))\) where \(A\) is a
call in the domain of \(f\) and \(\text{ref}(f(A))\) is also a call. The node will have no children if
there is no \(A\) in the domain of \(f\) such that \(\text{ref}(f(A))\) is defined.

Fig. 6. shows the
structure of a search tree.
The algorithm applies depth-first, left-to-right search to the search tree, guided by
heuristics based on estimates of the number of differences between items being com-
pared, to compute the Count value for the root node, and find, for each case node it
visits, the child (isomorphism) node that minimises the number of differences
between the cases.

As search proceeds, the Count value for each node is incrementally computed as
the search tree below it is explored. The Count value of a case node is the minimum
of the Count values of its child nodes, while the Count value of each of the other
nodes is its Local value, plus the sum of the Counts of its children plus their local
differences. Hence the Counts of case nodes can only decrease during search while the
Counts of other nodes can only increase. We exploit this fact to reduce the
number of nodes visited by a technique similar to alpha-beta pruning [4].

When a node \(X\) is visited, three associated values are initialised, as follows.
\[
\begin{align*}
C(X) &= \text{Local}(X) \\
\text{done}(X) &= \begin{cases} 
\text{true} & \text{if } X \text{ has no children} \\
\text{false} & \text{otherwise}
\end{cases}
\end{align*}
\]
As described below, these values change as the search proceeds in such a way that, for each node X that is visited, the value of $C(X)$ tends towards $\text{Count}(X)$. As soon as $\text{done}(X)$ becomes true, further search in the subtree rooted at X is abandoned to avoid exploring parts of the search tree that cannot affect the value of $C$ that will be computed for the root.

If node Y is the parent of a node X, and $\text{done}(X)$ is assigned, or updated to, true, then the values associated with Y are updated as follows.

1. if (Y is a method or isomorphism node) then $C(Y) = C(Y) + C(X)$
2. else $C(Y) = \min(C(Y), C(X))$;
   if ($\text{done}(Z) = \text{true}$ for every child Z of Y)
3. then $\text{done}(Y) = \text{true}$;
   if (Y is a case node and C(Y)=0)
4. then $\text{done}(Y) = \text{true}$;
   if (Y is a case node)
5. then $\alpha(Y) = \min(\alpha(Y), C(Y));$
   if (Y is a method or isomorphism node and $C(Y) \geq \alpha(Y)$)
6. then $\text{done}(Y) = \text{true}$;

Fig. 7. illustrates two conditions under which search terminates. In this and following figures, a node is drawn black, grey or white to indicate, respectively, that it has been visited, will not be visited because of cutoff, or may yet be visited as search proceeds. In this figure, the isomorphism node labelled X has no children, so $\text{done}$ is set to true (line 3), terminating search below this node. Its $C$ value remains 0. The $C$ value of its parent node Y set to 0 (line 2), and its $\text{done}$ value to true (line 4), terminating search below Y, and cutting off the grey-shaded parts of the tree.

The value used for determining when to terminate search beneath a method or isomorphism node Y, is the minimum value of $C(Z)$ among all case-node ancestors Z of Y. This is illustrated in Fig. 8. In this figure, the value of $\alpha(Y)$ is inherited from node Z in steps 2 to 4. When the search below node X terminates, update of the values associated with node Y is triggered, and because the updated value of $C(Y)$ is greater than $\alpha(Y)$, search below Y is terminated (line 6).

5.3. Further optimisation

Clearly, if search below a node will be cut off, the sooner this can be discovered the better. Hence, since search below a case node X will stop as soon as the value of $C(X)$ is reduced to 0, the children of X should be visited in order of ascending value of $\text{Local}$, assuming that nodes with lower $\text{Local}$ values will have lower $\text{Count}$ values.
Fig. 7. (1) Search down a path stops at a node X with no children. (2) Cut-off occurs when C(Y) becomes 0.

Fig. 8. The value alpha(Y) used to cut off search in step 5 is inherited from node Z via steps 2 to 4.
Similarly, the search below an isomorphism or method node \( X \) will be cut off as soon as \( C(X) \) exceeds \( \alpha(X) \). Therefore, the children of \( X \) should be visited in order of decreasing \( \text{Local} \) value, so that the value of \( C(X) \) is increased as quickly as possible.

Note that to achieve the second of these two optimisations, when an isomorphism node \( X \) is visited, \( \text{Local} \) must be calculated for each of its children so they can be visited in the required order. This leads to a third optimisation. A variable \( K \) is initialised to \( \text{Local}(X) \), and incremented by \( \text{Local} \) for each child as it is computed. After each addition, if \( K \) is equal to or greater than \( \alpha(X) \), \( C(X) \) is set to \( K \), and \( \text{done}(X) \) is set to \( \text{true} \), terminating the search below \( X \). For example, regardless of the order in which \( \text{Local} \) values are computed for the children of the isomorphism node in Fig. 9., search below the isomorphism node will terminate when values have been determined for at most two of the method nodes, cutting off search below any of the children.

### 5.4. Practical issues

The algorithm as described simply computes \( \text{Count} \) for the method node \( X \) that it starts with, and identifies the subtree of the search space rooted at \( X \) that corresponds to this computation. In addition, the algorithm as implemented produces a catalogue of the differences, at all levels of the structure, that contribute to the count computed for the root node. This information could be used in an appropriate interface to highlight the places in the code where differences occur.

To avoid repeating searches, two global structures are maintained; a list of pairs of methods that have been found to be not equivalent, and a set of equivalence classes of methods that have been found to be equivalent. During search, if the final value of \( C(X) \) for a method node \( X \) is not 0, and no cut-off occurred in the search tree below \( X \), then the pair of methods is added to the “non-equivalent” list. Similarly, if the final value of \( C(X) \) is 0, the equivalence classes of the two methods are merged. Note that if a \( C \) value of 0 is computed for a method node \( X \), then the two methods are guaranteed to be equivalent, whether or not cut-off has occurred in the part of the tree rooted at \( X \).

![Fig. 9. Summing \( \text{Local} \) for at most 2 of the method nodes will lead to cut-off.](image-url)
As noted earlier, persistents are a non-declarative feature that we need to deal with specially. In particular, the algorithm needs to compute the bijection \( g \) in Definition 2 on the fly. Accordingly, it builds a global list \( G \) of pairs of persistents that represents \( g \), attempting to add a pair to this list when it encounters two operations, \( A \) and \( B \), that are matched by a subgraph isomorphism, but refer to two different persistents. There are three possibilities. First, if \((\text{ref}(A), \text{ref}(B))\) is in \( G \), then \( A \) and \( B \) are considered to be equivalent. Second, if \((\text{ref}(A), P)\) is in \( G \) and \( P \neq \text{ref}(B) \) (or vice versa), then \( A \) and \( B \) are considered not to be equivalent. Finally, if neither \( \text{ref}(A) \) nor \( \text{ref}(B) \) occurs in any pair in \( G \), \((\text{ref}(A), \text{ref}(B))\) is added to \( G \). There is a complication, however. If the isomorphism \( f \) that matches \( A \) and \( B \) turns out not to be the one which minimises the value of \( C(Y) \), where \( Y \) is the parent node of the isomorphism node \( X \) corresponding to \( f \), then the addition of \((\text{ref}(A), \text{ref}(B))\) to \( G \) must be undone. Hence, when a pair is added to \( G \), it is added provisionally, creating a local list \( G(X) \), the scope of which is the search of the subtree rooted at \( X \). When the final value of \( C(Y) \) is determined, \( G \) is updated to \( G(Z) \), where \( Z \) is the selected child of \( Y \).

6. Correctness and Performance

As noted in Section 2, the differencing tools currently available in VPLs perform syntactic comparison, and compare diagrams only at one level. The algorithm we have proposed searches all levels, and is able to determine equivalences that purely syntactic tools cannot. For example, it can determine that the methods `fact-a` and `fact-b` in Fig. 10 are equivalent.

If two programs are not equivalent, there is no precise answer to the question of how they differ semantically [20], so there is little that can be proved about the correctness of an algorithm such as ours with respect to semantic difference. However, it is possible to prove the following.

**Lemma 2:** If \( A \) and \( B \) are operations and \( P_1 \) and \( P_2 \) are programs, the algorithm described above, including the optimisations in Section 5.3, will compute a value of 0 for \((\text{ref}(A), \text{ref}(B))\) iff \( A \) in \( P_1 \) is equivalent to \( B \) in \( P_2 \) (Definition 2).

Since we have not provided a formal listing of the algorithm, we cannot present a formal proof of this lemma here. However, it is important to note that the algorithm is closely related to depth-first search with alpha-beta pruning first described by Hart and Edwards [17], and the proof, which is essentially an induction on search depth, is similar to the proof of correctness of that algorithm, due to Knuth and Moore [23]. In particular, search below a \( C \) node will be cut off once the correct isomorphism is found.

Although the subgraph isomorphism problem is NP-complete, there are subgraph isomorphism algorithms which, in practice, perform well on large graphs, such as VF [8]. Furthermore, the number of operations in a diagram is usually quite small. For example, in the Prograph CPX application framework and associated editors,
consisting of 2800+ methods distributed over 300+ classes, diagrams rarely have more than 6 operations. Hence, subgraph isomorphism is not a bottleneck.

Using subgraph isomorphism to match dataflow diagrams can, however, produce unsatisfactory results. For example, in Fig. 11. the cases $C_1$, $C_2$ have the same numbers of operations, datalinks and synchros, and there are no subgraph isomorphisms between them, so $\text{Local}(C_1,C_2)$ will be set to 1 and search below the node $(C_1,C_2)$ will terminate. Clearly, however, there are two local differences (mismatched datalinks), and search should continue with the method node $(\text{ref}(X),\text{ref}(Y))$. 

![Diagram](image-url)
To deal with such situations, maximal common subgraph (MCS) isomorphism can be used to find bijections between two graphs when subgraph isomorphism fails. There are various definitions and algorithms for MCS in the literature, such as those due to McGregor [24], Balas and Yu [3], and Durand et al. [13]. We have chosen, for experimental purposes, the McGregor definition of MCS, as follows.

**Definition 3:** Suppose, without loss of generality, that graph $G_1$ has more more vertices than graph $G_2$, then if $S_1$ is a subgraph of $G_1$ including all vertices of $G_1$, $S_2$ is a subgraph of $G_2$, $f: S_1 \rightarrow S_2$ is an isomorphism, and there is no subgraph of $G_1$ which has more edges than $S_1$ and satisfies these conditions, then $f$ is a **maximal common subgraph isomorphism**, and $S_1$ and $S_2$ are corresponding maximal common subgraphs of $G_1$ and $G_2$.

Note that this definition includes the obvious bijection in Fig. 11.

Although the functions described in section 5.1 were designed to count differences when subgraph isomorphism is used to match diagrams, they will also work with the McGregor algorithm, since all vertices in the smaller graph in Definition 3 are included in the mapping. However, the time complexity of McGregor’s algorithm in the worst case is factorial.

To gauge the performance of the algorithm and verify its accuracy, we built a simple prototype implementation. Our test examples were created by making changes to the Prograph CPX development framework, which consists of two class hierarchies, the Application Building Classes (ABC) that provide the application framework, and the Application Builder Editor (ABE) classes that implement high-level editors for ABC classes. **Error! Reference source not found.** shows the numbers of methods and cases, and the average number of operations per case in these hierarchies, an important measurement since it determines the “bushiness” of search trees explored by the algorithm. This code, which was built and maintained by a team of professional programmers over a five-year period, is a realistic example of the kind of industrial-size project to which differencing tools are routinely applied.

To test responsiveness, we applied the prototype to several pairs of the form (A,A) where A is a method in ABC or ABE, such that the maximum number of operations per case is 10. For the largest such example, involving 61 methods and locals, with 72

![Diagram](image-url)
cases in total, and a call depth of 20, we found that response time was less than one second, using either VF or McGregor algorithms.

The second experiment tested for accuracy, using four examples ranging in depth from 5 to 20. Each example, (A,A’), was created by starting with some method A from the ABCs or ABEs, making a copy A’ of A and the code on which A relies, and introducing some differences into the copy. To maximise the potential for interference between the differences, in each example, they all lie on a single call path. The differences are summarised in Table 2, which shows the number of violations of each condition in Definition 1 and the depth at which they occur.

Note that condition 6.3 requires the existence of a bijection between the sets of operations of the two cases being compared, so is violated if the sets are different sizes. Hence, instead of putting 1 in an entry in the 6.3 column to signify one violation of the condition, we put the number of excess operations, since these are the differences which cause the violation. Similarly, an entry in column 6.2 indicates the number of extra cases that contribute to violation of condition 6.2. In all other columns, the number of violations of the condition is equal to the number of differences causing the violations. In all examples, there were no more than 6 operations per case on average. The Total column shows the total number of violations of conditions of Definition 1, together with the total number of contributing differences, in parentheses.

On each example, the response time was less than one second for both VF and McGregor, and all differences were found using McGregor. These results are encouraging, given that no attention to efficiency was paid in the implementation of the prototype, which is a simple proof-of-concept that reads and processes XML files containing representations of Prograph code, using off-the-shelf implementations of VF and McGregor. When using VF, the algorithm fails to find all differences in example 4 because at depth 9, there are no subgraph isomorphisms between the cases being compared, so only the differences down to and including depth 9 are found. Similarly, in example 3 there is no subgraph isomorphism at depth 1, so using VF the search goes no deeper.

For the third experiment, we augmented our four examples by adding operations so that several cases had 10 operations. With these examples, the response time using VF was still one second or less, but using McGregor, it was considerably longer.

<table>
<thead>
<tr>
<th>Method Count</th>
<th>Case Count</th>
<th>Average number of operations per case</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABC</td>
<td>2286</td>
<td>6510</td>
</tr>
<tr>
<td>ABE</td>
<td>550</td>
<td>2394</td>
</tr>
</tbody>
</table>

Table 1. Size of ABC and ABE
Table 2. For each example, the number is shown of violations of the conditions of Definition 1 at each call depth. Entries for 6.2 and 6.3 also include, in parentheses, the number of extra cases and operations, respectively, contributing to the violation.

<table>
<thead>
<tr>
<th>Depth</th>
<th>Condition from Definition 1</th>
<th>Total</th>
<th>VF</th>
<th>McGregor</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>Ex 1</td>
<td>(depth 5)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Ex 2</td>
<td>(depth 10)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ex 3</td>
<td>(depth 15)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ex 4</td>
<td>(depth 20)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note, however, that if there is a subgraph isomorphism between two cases, then the set of maximal common subgraph isomorphisms between the two cases is the same as the set of subgraph isomorphisms. Hence, the more expensive McGregor algorithm need not be applied unless VF fails to find any bijections. As noted in Table 1, however, the average number of operations per case in the Prograph CPX application framework, which is representative of industrial-scale code, is less than 6; hence the need to resort to more expensive graph-matching algorithms such as McGregor should arise quite infrequently.
7. Concluding Remarks

Although we have presented and discussed the semantic differencing algorithm in terms of Prograph, all SVPLs have a similar structure, acyclic dataflow diagrams embedded in enclosing control structures, as illustrated in Fig. 3. Therefore the algorithm, is based on this structure, and consists of comparing local features related to operation type, name, and decoration [conditions 4, 5, 6.1 of Definition 1], input and output type, name and decoration [conditions 1, 2, 3], matching up corresponding dataflow graphs [condition 6.2, 6.3], and finding isomorphisms between embedded dataflow graphs [conditions 6.3, 6.3.1, 6.3.2]. Hence, with appropriate modifications, the definition and algorithm can apply to any member of this class of languages. For example, comparing two LabVIEW programs requires exactly these steps, requiring the algorithm to differ only in nomenclature (e.g. virtual instrument = method), details such as the types of operations (e.g. counted loop, sequence), inputs and outputs (e.g. shift register, array), and the way that diagrams are paired off.

A formalisation of the SVPL class of languages, called controlled dataflow, has been proposed in [10]. We intend to use this formalism as a basis for formulating generic versions of Definition 1, the counting functions and our differencing algorithm, which can then be systematically specialised to individual SVPL languages.

In Prograph, and other SVPLs, programming, testing and debugging is done in highly interactive environments that include visualisations of execution state and progress. Clearly, an SVPL differencing tool should integrate into such an environment, providing a visual representation, at various levels of detail, of the spots where differences are found, and helping the user to navigate through the program structure in a methodical way.

While the counting functions we have used to guide the heuristic search seem to do a reasonable job on most examples, as demonstrated by the experiments reported above, we intend to experiment further to pinpoint their weaknesses in order to fine-tune them.

VF and McGregor are general algorithms for matching directed graphs. We are looking into the design of algorithms that take advantage of the unique features of SVPL dataflow diagrams.

8. Acknowledgements

This work was supported by the Natural Sciences and Engineering Research Council of Canada via Discovery Grant OGP0000124.

References


**Philip T. Cox**

Philip Cox is a Professor in the Faculty of Computer Science at Dalhousie University, Halifax, Canada. He received an M.Sc. degree with First Class Honours in Mathematics from the University of Auckland in 1970, and a Ph.D. in Computer Science from the University of Waterloo in 1977. He has held various academic positions at the Universities of Waterloo, Toronto, Auckland, Acadia University, and Dalhousie. His research interests include visual design and programming languages, computational logic, abductive inference, logic and functional programming. He has provided expert advice in patent litigations involving visual programming. Dr. Cox has served as General Chair and Program Chair of major international conferences, and has organized several workshops. He is on the Editorial Board of the Journal of Visual Languages and Computing.

**Tuan Anh Dang**

Anh Dang received a Bachelor of Software Engineering degree from Vietnam National University, Ho Chi Minh City in 2006, and in 2009, a Master of Computer Science from Dalhousie University, Halifax, Canada. He has worked as a software developer for Augen Software Group in Vietnam, and is currently a web application developer in Halifax, Canada.