Visual dataflow languages (VDFLs), which include commercial and research systems, have had a substantial impact on end-user programming. Like any other programming languages, be they visual or textual, VDFLs often contain faults. Providing programmers of these languages with some of the benefits of traditional testing methodologies has been the driving force behind our effort in this work. In this article we introduce, in the context of Prograph, a testing methodology for VDFLs that is based on structural test adequacy criteria and coverage. This article also reports on the results of two empirical studies. The first study was conducted to obtain meaningful information about, in particular, the effectiveness of our all-Dus criteria in detecting a reasonable percentage of faults in VDFLs. The second study was conducted to evaluate, under the same criterion, the effectiveness of our methodology in assisting users to visually localize faults by reducing their search space. Both studies were conducted using a testing system that we have implemented in Prograph’s IDE.

Categories and Subject Descriptors: D.2.5 [Software Engineering]: Testing and Debugging; D.2.6 [Software Engineering]: Programming Environments; D.1.7 [Programming Techniques]: Visual Programming.

General Terms: Algorithms, Languages, Verification.

Additional Key Words and Phrases: Software testing, Visual dataflow languages, Fault detection, Fault localization, Color.

1. INTRODUCTION

Visual dataflow languages (VDFLs) provide meaningful visual representations for the creation, modification, execution and examination of programs. In VDFLs, users “code” by creating icons (operations) and linking them together. In this article, we refer to this activity as visual coding. The icons, or visual constructs, are the source code and not some visual representations of a textual code that lies beneath the icons. In VDFLs, the order of execution, when not explicitly defined by the user, is determined by the visual language’s editing engine. That is, the computation is governed by the dataflow firing rule which states that a computational element can...
execute as soon as sufficient incoming data has arrived to begin the computation. This model of execution is known as the dataflow computational model, and has inspired several visual programming languages (e.g., Bernini and Mosconi [1994], Kimura et al. [1990], and Fisk [2003]). It is, in general, conceptualized as a kind of fluid that flows through linkages between computational elements. The elements can be thought of as filters or processors that use the incoming data to produce a new stream of outgoing data. In a pure dataflow computational model control constructs are not explicitly specified by the programmer; rather, the order of execution is implied by the operations’ data interdependencies. To allow the user to explicitly add control constructs such as those found in imperative languages, VDFLs [Shafer 1994; Marten 2005] extended the pure computational dataflow model to include the necessary control constructs. This extension, as some researchers believe, is necessary for a dataflow language to have any practical use in developing “traditional” software. Thus, a visual program that is based on the dataflow computational model can be characterized by both its data and control dependencies.

VDFLs are widely used by researchers and end-users alike, for a variety of research and development tasks. For example, there is research into steering scientific visualization [Burnett et al. 1994], using VDFLs for writing functional programs [Kelso 2002], writing and evaluating XML and XPath queries [Boulos et al. 2006; Karam et al. 2006], developing general-purpose applications [Shafer 1994], building domain-specific applications such as those used for laboratory instrumentations [Paton 1998], and buffering of intermediate results in dataflow diagrams [Woodruff and Stonebraker 1995]. In general, researchers feel that it is in these directions that visual dataflow programming languages show the most promise.

Despite claims that the use of graphics in VDFLs allows direct and concrete representations of problem-domain entities, and that the direct representations can simplify the programming task of researchers and developers alike, we found that many informal investigations into VDFLs reveal that, like any other languages, they contain faults. A possible factor in the existence of these faults, as argued by [Meyer and Masterson 2000], is most likely related to incorrect control annotations and datalinks. In spite of this evidence, we found no related discussions in the research literature of techniques for testing or evaluating a testing methodology that can reveal faults and assist in fault localization in VDFLs. In fact, there has been some work on testing in other paradigms. For example, in the domain of form-based languages, recent work focused on testing visual form-based languages [Rothermel et al. 2001]. Although the visual dataflow paradigm is similar to the visual form-based paradigm in that they are both visual, several characteristics of the form-based paradigm such as the dependency-driven nature of its evaluation engine and the responsiveness of its editing environment suggest a different approach when testing dataflow languages. There has been also work on specification-based testing for imperative languages [Kuhn and Frank 1997; Ouabdesselam and Parissis 1995]. Since most VDFLs are intended for use by a variety of researchers and professional end-users alike, few of these users are likely to create specifications for their programs [Wing and Zaremski 1991]. Moreover, even when specifications exist, evidence suggests that code-based testing techniques can provide an effective mechanism for detecting faults [Hutchins et al. 1994; Wong et al. 1995] and increasing software reliability [Del Frate 1995]. Other research (e.g., Azem et al. [1993], Belli and Jack [1993], and Luo et al. [1992]) considered problems of testing and reliability determination for logic programs written in Prolog.
Although VDFLs are like logic-based programs in that both are declarative (i.e. declaring data and control dependencies between operations), several features of the logic paradigm, such as the bidirectional nature of unification and backtracking after failure, are so different from VDFLs that the testing techniques developed for Prolog cannot be applied to VDFLs.

On the other hand, there has been extensive research on testing imperative programs (e.g., Clarke et al. [1989], Frankl and Weiss [1993], Frankl and Weyuker [1988], Harrold and Soffa [1988], Hutchins et al. [1994], Korel and Laski [1983], Ntafos [1984], Offutt et al. [1996], Perry and Kaiser [1990], Rapps and Weyuker [1985], Rothermel and Harrold [1997], Weyuker [1986; 1993], and Wong et al. [1995]). In fact, the family of testing criteria we present for VDFLs in this article are rooted in the body of the aforementioned work on imperative testing. There are significant differences between VDFLs and imperative languages, and these differences have ramifications for testing strategies of VDFLs. These differences can be divided into three classes, as described next.

The first class pertains to the evaluation or execution order. The order of execution of non-predicate operations or statements is not predetermined by the programmer, but is simply based on data dependencies. If we were to construct, analogous to imperative languages, a control flow graph (CFG) that represents the flow of control in VDFLs, taking into consideration all possible execution orders, it would be extremely large and complex. It would also most likely be impossible to satisfy any criteria using this graph, since most dataflow language implementations choose a specific execution ordering for non-predicate operations and use it for every execution. This is, in fact, the execution behavior of VDFLs. The execution order is maintained by a topological sort that is performed on all operations during the phases of visual coding and modification of the code. Since we are not considering for the purpose of our current work operations that can have side-effects (object oriented features such as the get and set operations that read and respectively write objects’ data, and global variables), any execution order of a sequence of non-predicate operations will yield the same result. Thus, we need only consider one order — the one determined by the editing engine of the language. In future works we intend to address the side effects issues.

The second class of differences pertains to performing static source code analysis in VDFLs. With imperative languages, a wide range of techniques for computing source code (control-flow and data-flow) analysis in individual procedures and programs are well known [Aho et al. 1986] and have been used in various tools, including data-flow testers (e.g., Korel and Laski [1985], Frankl et al.[1985], and Harrold and Soffa [1988]). With VDFLs however, the variable declaration process and the visual constructs’ characteristics have impact on how precisely and efficiently source code analysis can be done. First, in general, variables in VDFLs cannot be explicitly defined; that is, users write programs by creating icons that are connected via datalinks. In general, a datalink is created between an outgoing port on one icon, and an incoming port on another. In VDFLs such as the one we are considering in this work, outgoing ports are known as roots, and incoming ports are known as terminals. One way to deal with the implicit declaration of variables is to think of roots as variable definitions and terminals connected to those roots as variable uses. Second, although formal grammars for pictures [Zhang and Zhang 1997] and parsing algorithms for pictorially represented programs have been investigated [Chang et al. 1989], none of these techniques is applicable to VDFLs for the purpose of efficiently performing source code analysis. One of the solutions to overcome this
difference is to take advantage of the control-flow and data-flow information that can be obtained from the data structures that is maintained by the editing engine of VDFLs. The third class of differences pertains to the reporting mechanisms of test results. With imperative languages, the use of textual logs as a way to view test results, albeit difficult to read and interpret, is practically impossible in VDFLs. Test results of VDFLs should be reported in a way that complements the visual constructs, their characteristics and properties. For example, some indication should be given to the tester when a datalink connecting a variable definition to its use is not exercised. Therefore, the reporting mechanism should be visual and well incorporated within the Integrated Development Environment (IDE) that supports the implementation of the VDFL.

In Section 2 of this article we present Prograph’s syntax and its formal semantics. In Section 3, we present a family of structural unit-based test adequacy criteria for VDFLs. In Section 4 we examine the applicability of several code-based data-flow applicability criteria for testing VDFLs. In Section 5, we introduce a methodology that takes advantage of the aforementioned classes of differences to achieve efficient and precise control-flow and data-flow analyses in VDFLs, and provide a testing environment that implements, in particular, the all-Dus. The testing environment accommodates the user base of VDFLs with a visual interface that is augmented with a color mapping scheme [Jones et al. 2002] that facilitates the task of localizing faults by reducing their search space without requiring a formal testing theoretical background. In Section 6 we describe the design of our experiments and present the empirical results obtained. Section 7 concludes the article with a summary of our contributions and an outline of planned directions of future work.

2. VISUAL DATAFLOW LANGUAGES

Since much of our work is based on the visual programming environment of Prograph [Shafer 1994], we will next informally introduce its syntax and semantics using the example that is depicted in Figure 1.

2.1 Prograph

Figure 1 shows a Prograph implementation of the well known algorithm “quicksort” for sorting a list into ascending order. The bottom right window entitled Universals of “Quicksort” in this figure depicts two icons for the methods call sort and quicksort. Note that Prograph is an object-oriented language, hence the term “method” or “universal” is used to refer to entities known as “procedures” in imperative programming languages.

The left side window entitled 1:1 call sort in Figure 1 shows the details of the method call sort, a dataflow diagram in which three operations are connected and scheduled to execute sequentially. The first operation in this diagram, ask, is a primitive that calls system-supplied code to produce a dialogue box requesting data input from the user (or a list of numbers typed into the dialogue box). Once the ask primitive has been executed, the data entered by the user flows down the datalink to the operation quicksort, invoking the method quicksort. This method expects to receive a list, which it sorts as explained below, outputting the sorted list, which flows down the datalink to the show operation. The show produces a dialogue displaying the sorted list. The small circle icons on the top and bottom of an operation, representing inputs and outputs are called terminals and roots, respectively. Note
that the “1:1” in the window entitled 1:1 call sort indicates that the method call sort has only one Case associated with it. In general, a method consists of a sequence of Cases. More on what a Case is and how it executes can be found next.

The method quicksort consists of two Cases, each represented by a dataflow diagram as shown in the windows entitled 1:2 quicksort and 2:2 quicksort of Figure 1, respectively. The first Case, 1:2 quicksort, implements the recursive Case of the algorithm, while the second, 2:2 quicksort, implements the base Case. The thin bar-like operation at the top of a Case where parameters are copied into the Case is called an Input-Bar, while the one at the bottom where results are passed out is called an Output-Bar. In the first Case of quicksort, the first operation to be executed is the match operation, \( \frac{0}{\sqrt{\text{True}}} \), which tests to see if the incoming data on the root of the Input-Bar is the empty list. The check mark icon attached to the right end of the match is a Next-Case on success control, which is triggered by success of the match, immediately terminating the execution of the first Case and initiating execution of the second Case. If this occurs, the empty list is simply passed through from the Input-Bar of the second Case to its Output-Bar, and execution of quicksort finishes, producing the empty list.

If the input list is not empty, the control on the match operation in the first Case is not triggered, and the first Case is executed. Here, the Primitive operation detach-l (or detach left) outputs the first element of the list and the remainder of the list on its left and right roots respectively. Next, the operation, \( \frac{\text{ >= } \text{ List-Terminal}}{\text{List-Terminal}} \), is executed. This operation is an example of a Multiplex illustrating several features of the language. First, the three-dimensional representation of the operation indicates that the primitive operation \( \text{ >= } \) will be applied repeatedly. Second, the terminal annotated as \( \text{ List-Terminal } \) is a List-Terminal, indicating that a list is expected as data, one element of which will be consumed by each execution of the operation. In this example, when the Multiplex is executed, the first element of the list input to the Case will be compared with each of the remaining elements. Finally the special roots and indicate that this particular Multiplex is a partition, which divides the list of items arriving on the list annotated terminal into two lists, items for which the comparison is successful and those for which it is not. These two lists appear on the and roots, respectively.
The lists produced by the partition *Multiplex* are sorted by recursive calls to the *quicksort* method. The final sorted list is then assembled using the two primitive operations *attach-l*, which attaches an element to the left end of a list, and *(join)*, which concatenates two lists.

The execution mechanism of Prograph is data-driven dataflow. That is, an operation executes when all its input data is available. In practice, a linear execution order for the operations in a *Case* is predetermined by topologically sorting the directed acyclic graph of operations and subject to certain constraints. For example, an operation with a control should be executed as early as possible.

In our example the method *quicksort* has only one input and one output, and therefore does not illustrate the relationship between the terminals of an operation and the roots of the *Input-Bar* in a *Case* of the method it invokes. These terminals and roots must be of equal number, and are matched from left to right. A similar relationship exists between the roots of an operation and the terminals of the *Output-Bar* in a *Case* of a method invoked by the operation.

One important kind of operation not illustrated in the above example is the *Local* operation. A *Local* operation is one that does not call a separately defined method such as the *quicksort* method shown above. Instead it contains its own sequence of *Cases* and their operations, and is therefore analogous to a parameterized *begin-end* block in a standard procedural language. A *Local* operation, often referred to as *Local* method, can also have roots and terminals attached to it. Roots and terminals on a *Local* operation can be annotated with a loop to create a repeating *Case* or *Cases*. *Terminate on success* and *Terminate on failure*, as the name indicate, are controls that can be applied to operations in a *Local* operation to stop the iterations during execution. More information on the operations and possible control can be found in Section 3.2.

The formal semantics of Prograph are defined by specifying an execution function for each operation in a program. Each execution function maps a list $X$ to a pair $(Y, c)$ where $c$ is a control flag, $Y$ is a list, and the lengths of the lists $X$ and $Y$ are respectively equal to the number of terminals and the number of roots of the operation, and the elements $X$ and $Y$ are from a domain $\Delta$ containing all values of simple types, and instances of classes. Execution functions may produce the special value *error*; for example, if a list terminal receives a value which is not a list. By defining execution functions for operations, the input/output behavior of a program is specified. To find more on the language syntax and semantics, the interested reader is referred to [Shafer 1994].

### 3. A FAMILY OF TEST ADEQUACY CRITERIA FOR VISUAL DATAFLOW LANGUAGES

As previously mentioned, test adequacy criteria have been well researched for imperative languages. In this section we explore the appropriate applicability of several of these criteria (e.g., Laski and Korel [1983], Ntafos [1984], and Rapps and Weyuker [1985]) to *VDFLs*. We argue that an abstract control-flow model which is common to all these test adequacy criteria can be appropriately adapted to *VDFLs*. Moreover, we argue that code-based data-flow test adequacy criteria which relate test adequacy to interactions between definitions and uses of variables in the source code (definition-use associations), can be highly appropriate for *VDFLs*; in
particular that of [Rapps and Weyuker 1985]. There are several reasons for this appropriateness. The first reason involves the types of faults that have been known to occur in VDFLs, the largest percentage of which have been observed to involve errors in incorrect or missing controls, and datalinks [Meyer and Masterson 2000]. Second, the application of [Rapps and Weyuker 1985] criteria, combined with the use of the color mapping scheme of [Jones et al. 2002] on the datalink constructs of VDFLs can be shown to provide significant advantages which includes a relative ease of applicability to individual datalinks (or, as we show later in Section 4.2, definition-use associations) and increased ability to localize faults.

3.1 The Abstract Model For Prograph

Test adequacy criteria are often defined on models of programs rather than on the code itself. We have created such a model for VDFLs [Karam and Smedley 2001] that we call the Operation Case Graph (OCG). In this article we augment the OCG to represent the complete set of Prograph’s operations and their applicable controls. These are: Input-Bar; Output-Bar; Locals; and Universal methods or functions; Constants; Match; and Primitives or system defined methods. These operations, may have the following control annotations: Next-Case, Finish, Terminate, Fail, Continue, or Repeat annotation. Roots and terminals on Locals or Universals may have List or Loop annotations. An operation that is control annotated is referred to as a predicate operation otherwise it is a non-predicate operation.

Given a Prograph procedure \( p \) in a program \( P \), we say that since each procedure consists of one or more Cases, we define an abstract graph model for Prograph by constructing an OCG for each Case \( c \) in \( p \) or OCG\(_{(c)}\). Each OCG\(_{(c)}\) has a unique entry and exit nodes \( n_e \), and \( n_x \) respectively, and will be assigned the name and label of its corresponding Case in Prograph. For example, if a method X has two Cases “1:2 X” and “2:2 X” denoting X “case one of two” and X “case two of two”, respectively, the OCG graphs will be labeled OCG\(_{(1:2 X)}\) and OCG\(_{(2:2 X)}\). A Local operation, as previously mentioned, has its own sequence of Cases, and each Case will also have its own OCG. We next describe the building process of the control-flow abstract model of OCG\(_{(p)}\) = \{OCG\(_{(c_1)}\), OCG\(_{(c_2)}\), ..., OCG\(_{(c_n)}\)\} in the context of the example of Figure 2 which is designed to introduce all the control constructs of Prograph.

3.2 The Building Process of the OCGs

Most test adequacy criteria for imperative languages are defined in terms of abstract models of programs, rather than directly on the code itself. This means that the code is translated into a control-flow graph (CFG) and testing is applied to the graph representing the code. This definition reflects explicitly the code constructs, and program execution follows the flow of control in the CFG. In general, to construct a CFG for a function \( f \) or CFG\(_{(f)}\), \( f \) is decomposed into a set of disjoint blocks of statements. A block is a sequence of consecutive statements in which the flow of control enters at the beginning of the block and leaves at the end without halt or the possibility of branching except at the end. A control transfer from one block to another in CFG\(_{(f)}\) is represented by a directed edge between the nodes such that the condition of the control transfer is associated with it.

With VDFLs, the dataflow computational model (control and data dependency) makes grouping several non-predicate operations in one block not very desirable since the execution order is determined by the editing engine. Therefore, to deal with the first class of differences between imperative languages and VDFLs that we
mentioned in Section 1, we represent each procedure \( p \in P \) with an \( OCG(p) \) that preserves two properties of \( p \)'s operations: data and control dependencies. With regards to preserving the data dependencies, we say that for each procedure \( p \), there is a sequence of operations, \( O = \{o_1, o_2, o_3, \ldots, o_m\} \), whose order corresponds to a valid execution ordering with respect to the data dependencies in the procedure (in particular, this will be the execution order chosen by the editing engine). Thus, to model the data dependencies in \( OCG(p) \) and preserve the execution order we represent each non-control non-Local operation \( o_i \in O \), with a node \( n_i \in OCG(c) \in OCG(p) \) linked together with an edge to \( n_{i+1} \), the node representing the next operation \( o_{i+1} \in O \), provided that \( o_{i+1} \) is a non-Local operation. If \( o_{i+1} \) is a Local operation, then we construct the edge to the entry node \( n_e \) of the Local operation's. For example, as depicted in Figure 2, the operation labeled \( n_2 \) is represented with node \( 2^1 \) in \( OCG(Main) \), and an edge is constructed to node \( n_{3e} \), the entry node of the Local operation represented by \( OCG(1:1 A) \). For each non-control Local operation \( o_i \in O \), we construct the appropriate \( OCG(o_j) \) with \( n_e \) and \( n_x \), the entry and exit nodes of \( OCG(o_j) \), respectively, and construct an edge from \( n_x \) to \( n_{i+1} \), the node representing the next operation \( o_{i+1} \in O \), provided that \( o_{i+1} \) is not a Local operation. If \( o_{i+1} \) is a Local operation, then we construct an edge to its entry node \( n_e \).

To represent the control dependencies between operations in \( O \), we classify the set of operations \( O \) into two subsets \( O_s \) and \( O_x \) where \( O_s \in O \) is the subset of operations that always succeed by default, and \( O_x \in O \) is the subset that does not. This classification is necessary to accurately represent and account for the flow of control in \( OCG(p) \), since control annotated operations \( \in O_s \) (on failure) do not change the control flow of the program. For example, in Figure 2, the primitive operation show that is labeled \( n_{13} \) which is annotated with a Next-Case on failure \( \square \). Since \( n_{13} \) succeeds by default, the control flow in \( OCG(1:2 Main) \) will not be affected. However, had the control been applied (on success) to \( n_{13} \), the flow of control would have been affected, and subsequently when the operation is evaluated, the control will be transferred to the next Case “2:2 Main” or \( OCG(2:2 Main) \). The Output-Bar, among other operations, belongs to \( O_s \).

Therefore, to represent the control dependencies between operations in \( O \), we perform static analysis on each control annotated operation \( o_i \in O \). Next we present each control and discuss how it is represented in the \( OCG \).

- **Next-Case Annotation**: an operation \( o_i \) with a Next-Case annotation will be represented as follows: If \( o_i \) is a non-Local operation \( \in O_x \), then \( o_i \) will be represented by a node \( n_i \) with two edges coming from it (true and false), one connected to \( n_{i+1} \) which is the node representing the next sequential operation \( o_{i+1} \) in the current Case, and the other to entry node \( n_e \) of the next Case. If \( o_i \) is a non-Local operation \( \in O_s \), we check the type of the control annotation, if it is Next-Case on failure \( \square \), we ignore the control; however, if it is a Next-Case on success \( \checkmark \), we construct an edge to the entry node \( n_e \) of the next Case. For example, as depicted in Figure 2,

1 It is assumed that when we say \( n_k \) we make reference to an operation in the prograph code, and when we say node \( k \) we make reference to \( n_k \)'s corresponding node in the OCG.
the operation labeled $n_{23}$ is annotated with a Next-Case on success, so we construct an edge from node 23, its corresponding node in $OCG(1:2 \text{Main})$, to the entry node of the $OCG(2:2 \text{Main})$. If $o_i$ is a Local operation $\in O_x$, we construct the appropriate $OCG(o_i)$, with its $n_e$ and $n_x$, respectively. We recursively process all of $o_i$’s content, and then construct two edges on $n_x$ of $o_i$ to represent the flow of control (true and false): one to $n_{i+1}$, the node representing the next sequential operation $o_{i+1}$ in the current Case; and the other to $n_e$, the entry node of the next Case. For example, as depicted in Figure 2, since the Local operation labeled $n_3$ is annotated with a Next-Case on Failure, we represent it with $OCG(1:1 \text{A})$, and construct two edges on its $n_x$, one to the next operation $n_{13}$, and the other to $n_e$ of $OCG(2:2 \text{Main})$.

- **Finish Annotation:** an operation $o_i$ with a Finish annotation is unique in the sense that, when evaluated/activated in non-repeated or looped Case, the flow of control does not change upon the outcome of the Finish control. This means that if the outcome is either true or false, the next node that gets evaluated or executed is the same. The same situation occurs when the Finish annotated operation happens to be in a repeated or looped Case; however, here the true or false outcome may set a flag that will indicate whether successive iterations will take place after finishing the current iteration. Thus, to handle a Finish annotated operation, be it in a repeated or non-repeated Case, and represent both its true and false outcomes, an operation $o_i$ with a Finish annotation will be represented as follows: if $o_i$ is a non-Local operation $\in O_s$, we simply ignore the evaluation of its control (on success or
failure), and represent \( o_i \) with a node \( n_i \) with one edge connected to \( n_{i+1} \), the node representing the next sequential operation \( o_{i+1} \). If \( o_i \) is a non-Local operation \( \in O_x \), we represent \( o_i \) with a node \( n_i \) with two edges coming out from it (true and false). One edge connected to \( n_{i+1} \), the node representing the next sequential operation \( o_{i+1} \) in the current Case, and the other edge to a dummy node \( d \). From the dummy node \( d \) we also construct an edge to \( n_{i+1} \). The reason for constructing this edge is to allow the flow of control to go to \( n_{i+1} \) when the outcome of the Finish control goes through the dummy node \( d \). When \( d \) exists in a looped-Local and is traversed, it sets a flag value that indicates whether the loop edge can be traversed. For example, as depicted in Figure 2, the operation labeled \( n_{27} \) is annotated with a Finish on success \( \square \). We therefore represent \( n_{27} \) with node 8 in \( OCG(1:1 C) \), and construct two edges from it, one to the next operation \( n_9 \), and the other to \( d \), the dummy node. If \( o_i \) is a Local operation \( \in O_x \), we construct the appropriate \( OCG(o_i) \) with its \( n_x \) and \( n_x' \) respectively. We recursively process all of \( o_i \)'s content, and then construct two edges on \( n_x \) of \( o_i \) to represent the flow of control (true and false): one to \( n_{i+1} \), the node representing the next sequential operation \( o_{i+1} \) in the current Case; and the other to dummy node \( d \).

- **Terminate annotation**: an operation \( o_i \) with a Terminate annotation will be represented as follows: if \( o_i \) is a non-Local operation \( \in O_x \), we check the type of the control annotation, if it is Terminate on failure \( \square \), we ignore the control. For example, as depicted in Figure 2 the operation labeled \( n_{21} \) is annotated with a Terminate on success, so we construct an edge from node 21, its corresponding node in \( OCG(1:2 C) \), to node \( n_{i+1} \) or node 22. However, if \( o_i \) is annotated with a Terminate on success \( \square \), we construct an edge to \( n_x \), the exit node of the current Case. If \( o_i \) is a non-Local operation \( \in O_x \), we represent \( o_i \) with a node \( n_i \) with two edges coming from it (true and false): one edge connected to \( n_{i+1} \), the node representing the next sequential operation \( o_{i+1} \) in the current Case; and the other edge to \( n_x \), the exit node of the current Case. If \( o_i \) is a Local operation \( \in O_x \), we construct the appropriate \( OCG(o_i) \) with its \( n_x \) and \( n_x' \) respectively. We recursively process all of \( o_i \)'s content, and then construct two edges on \( n_x \) of \( o_i \) to represent the flow of control (true and false): one to \( n_{i+1} \), the node representing the next sequential operation \( o_{i+1} \) in the current Case; and the other to \( n_x \), the exit node of the current Case. For example, as depicted in Figure 2, the operation labeled \( n_{14} \) is annotated with a Terminate on failure, so we construct its \( OCG(1:1 C) \), and construct on the \( n_x \) of \( OCG(1:1 C) \) or node \( x \) two edges, one to \( n_{i+1} \) or node 23, and the other to the exit node \( x \) of \( OCG(2:2 \text{Main}) \).

- **Fail Annotation**: the evaluation of an operation \( o_i \) with a Fail annotation is analogous to exception throwing in imperative languages. An operation \( o_i \) with a Fail annotation will be represented as follows: if \( o_i \) is a non-Local operation \( \in O_x \), we check the type of the control and if it is Fail on failure \( \square \) we ignore the control. For example, as depicted in Figure 2 the operation labeled \( n_{27} \) is annotated with a Fail on failure, so we ignore the control, and we just construct an edge from node 27, to the exit node \( x \) of \( OCG(2:2 \text{Main}) \). However, if the control is Fail on success
we construct an edge to the entry node \( n_e \) of the next Case. **If \( o_i \) is a non-Local operation** \( \in O_x \), \( o_i \) will be represented by a node \( n_i \) with two edges coming from it (true and false): one connected to \( n_{i+1} \) which is the node representing the next sequential operation \( o_{i+1} \) in the current Case; and the other to \( n_x \), the exit node of the current Case. **If \( o_i \) is a Local operation** \( \in O_x \), we construct the appropriate \( OCG(o_i) \), with its \( n_e \) and \( n_x \), respectively. We recursively process all of \( o_i \)'s content, and then construct two edges from \( n_x \) of \( o_i \), to represent the flow of control (true and false) one to \( n_{i+1} \), the node representing the next sequential operation \( o_{i+1} \) in the current Case, and the other to \( n_x \), the exit node of the current Case. For example, as depicted in Figure 2, the Local looped operation labeled \( n_{16} \) is annotated with a Fail on Failure. We therefore represent it with \( OCG(1:1) \), and construct two edges from its exit node \( 16_x \), one to the next operation node \( 21 \), and the other to the exit node of \( OCG(1:1) \) or node \( 14_x \).

- **List, Repeat, Partition, or Loop Annotation:** an operation \( o_i \) with a list, repeat, partition, or a loop annotation will be represented according to the type of the operation. **If \( o_i \) is a non-Local operation** \( \in O_p \), we represent it with a node \( n_i \), and construct an edge that goes out of \( n_i \) and back into \( n_i \), and construct another edge to \( n_{i+1} \) the next node in the Case. For example, as depicted in Figure 2, the operation labeled \( n_{26} \) is annotated with a list control, and we therefore represent it with node \( 26 \) and construct one edge that goes out of node \( 26 \) and back into it, and another edge to node \( 27 \). If \( o_i \) is annotated with a second control on success, we construct the appropriate edges, otherwise we simply ignore it because it will not have any effect on the control flow of the program. **If \( o_i \) is a non-Local operation** \( \in O_x \), we represent it with a node \( n_i \), construct an edge that goes out of \( n_i \) and back into \( n_i \), and construct another edge to \( n_{i+1} \), the next node in the Case. We then check to see if \( o_i \) is annotated with a second control. If \( o_i \) is annotated with a control annotation, we construct the appropriate edges that represent the control. For example, as depicted in Figure 3, the operation labeled \( n_4 \) is annotated with a partition control,
and therefore we represent it with node 4 and construct an edge that goes out of node 4 and back into it. We then construct, as previously described for the terminate control, two edges: one to node 5; and the other to the exit node. Another example is the list annotated operation \( n_5 \) in Figure 3. It should be noted that although the operation labeled \( n_7 \) is annotated with a partition control, it does not alter the control flow of the program, and therefore, as depicted in Figure 3, we represent it by constructing node 7, and an edge that comes out of node 7 and back into it. If \( o_i \) is a local operation \( \in O_x \), we construct an edge from the node representing the output-bar of \( OCG_{(oi)} \) to the node representing the input-bar. For example, in \( OCG_{(1:1)} \) of Figure 2, the \( OCG \) representing the local operation labeled \( n_9 \) that is annotated with a loop control, we construct an edge from the output-bar of \( OCG_{(1:1)} \) or node 10 to its input-bar or node 7. If \( o_i \) is control annotated, as it is the case with the local operation labeled \( n_{16} \), we then construct the appropriate edges, as previously described for the fail control: one to node 21; and the other to node 14x.

### 3.3 Control-flow Test Adequacy Criteria For Visual Dataflow Languages

In imperative languages, a test adequacy criterion fails to be applicable if it requires coverage of non-executable code constructs, often referred to as dead code. For such code, it is a common practice to render the applicable criterion and redefine it so that it is applicable to executable code only. In VDFLs a similar approach applies; we next define our applicable node and control test adequacy criteria for VDFLs. Before we do that however, we need to first define a test case and a test suite for VDFLs. Formally, a test suite can be defined as follows:

- **Definition 3.1** - (A test suite \( T \) in VDFLs): We define a test case \( t \) for a Universal method \( p \) to be the tuple \((z, i, o_{v/i}, c_p)\), where: \( z \) is the test case number; \( i \) is a set of input values for \( t \); \( o_{v/i} \) is \( p \)'s output valid/invalid results, and \( c_p \) is the set of exercised nodes in \( OCG_{(p)} \) that is obtained as the result of executing \( t \) on \( p \). We say that \( t \) is valid or \( t = (z, i, o_v, c_p) \) if the actual output for an execution of \( p \) with \( t \) is the same as the expected output for \( t \); otherwise, \( t \) is invalid or \( t = (z, i, o_v, c_p) \). Having defined what a test case is, a test suite \( T \) can then be defined as the tuple \((Z, I, O_{V/I}, C_N)\), where: \( Z = \{z_1, z_2, ..., z_k\} \) is the set of test case numbers; \( I = \{i_1, i_2, ..., i_k\} \); \( O_{V/I} = \{o_{v/i} \in O_{V/I}\} \) is \( p \)'s output valid/invalid set of results of executing \( p \) with all test cases \( t \in T \); and \( C_N \) is the set of covered nodes in \( OCG_{(p)} \) that is obtained as the result of executing \( p \) with all test cases \( t \in T \).

- **Definition 3.2** - (all-Nodes criterion for VDFLs): formally, given a VDFL Universal method \( p \) with an operation case graph \( OCG_{(p)} \), a test \( t \) exercises a node \( n \in OCG_{(p)} \) if \( t \) causes the evaluation of \( p \), and traverses a path through \( OCG_{(p)} \) that includes \( n \). A test suite \( T \) is node-adequate for \( p \) and for each dynamically executable node \( n \) in \( OCG_{(p)} \) if there is at least one \( t \in T \) that exercises \( n \).

- **Definition 3.3** - (all-Edges criterion for VDFLs): formally, given a VDFL Universal method \( p \) with an Operation Case Graph \( OCG_{(p)} \), a test \( t \) exercises an edge \( e = (n_i, n_j) \in OCG_{(p)} \) if it causes the execution of \( p \), and that execution traverses a path through \( OCG_{(p)} \) that includes \( e \). A test suite \( T \) is edge-adequate for a \( p \) if, for each dynamically executable edge \( e \in OCG_{(p)} \), there is at least one \( t \in T \) that exercises \( e \).
As with imperative languages, if all edges in $OCG(p)$ are covered, all nodes are necessarily covered. This observation leads to the conclusion that branch coverage in $VDFLs$ is stronger than node coverage. When a testing criterion $A$ is stronger than another testing criterion $B$, we say that $A$ subsumes $B$. Thus, branch coverage subsumes node coverage, and a test suite $T$ that satisfies branch coverage, must also satisfy node coverage.

3.4 Data-flow Analysis in OCGs

The data interaction model in $VDFLs$, although visual, is somewhat analogous to that of imperative languages. In imperative languages, data interaction between variables is made by explicitly defining and referencing the names of variables in a procedure. For example, the statement $s_1: x = 3$ explicitly defines $x$ and writes to its memory location, whereas the statement $s_2: y = x+3$ explicitly uses or references $x$ by reading from its memory location. In $VDFLs$, variables cannot be explicitly defined, and their interactions are modeled as datalinks connecting operations’ roots to other operations’ terminals. In this interaction model, roots serve as implicit variable definitions and terminals connected to those roots serve as variable uses. In general, when a root $r$, on an operation $o_i$ is connected to a terminal $t$ on an operation $o_j$ ($i$ and $j > 1$), we say that $t$ in $o_j$ references $r$ in $o_i$. In other words, $r$ is used in $o_j$. For example, as depicted in Figure 4, in “1:1 C” root $r_1$ is connected to terminal $t_1$ on the Universal method $D$.

The basic structure of a Prograph Universal method is similar to a procedure in imperative languages. The roots on the Input-Bar of a Universal represent the method’s inputs, and correspond to reference parameters in imperative languages. The terminals on the Output-Bar of a Universal represent the method’s outputs.

Figure 4 — Universal methods $A$, $B$, $C$, and $D$. 

and correspond to variables “returned” in imperative languages. The reader should note that Prograph, unlike imperative languages, allows more than one root on the Output-Bar. When a Universal method has more than one Case, roots on the Input-Bar of each Case are essentially the same “variables”. A similar situation exist for terminals on the Output-Bar. For example, in Figure 4, the roots on the Input-Bars of Cases “1:2 A” and “2:2 A” are the same. Likewise, the terminals on the Output-Bars of Cases “1:2 A” and “2:2 A” are also the same.

Operations in the body of a Universal that are connected to roots on the Universal’s Input-Bar get their values from the roots which are connected to the terminals at the call site. For example, in Figure 4, the reference parameter labeled $i_1$ in “1:1 D” gets its value from the root labeled $r_1$ in “1:1 C”. As with imperative languages, reference parameters and actual parameters in visual dataflow languages are bound at call sites. Thus, we say that the reference parameter or $i_1$ in “1:1 D” is bound to the actual parameter or $r_1$ in “1:1 C”. Since we are only interested in unit-based data-flow analysis in VDFLs, a call site is considered a primitive operation. Work is underway to deal with inter-procedural dataflow analysis and testing for interactive units in VDFLs.

A Local operation is similar in structure to a Universal method; however, the roots on the Local’s Input-Bar are not considered reference parameters; rather, they correspond to the roots to which the Local operation’s terminals are connected to. For example, as depicted in Figure 4, roots $r_2$ and $r_3$ on the Input-Bar of Case “1:1 aLocal” corresponds to the root labeled $r_1$ on the ask Primitive in method “1:1 B”. Therefore, $r_2$, and $r_3$ in “1:1 aLocal” can be considered as references to $r_1$ in “1:1 B”.

Figure 5 — Method E and its looped-Local “bLoop”.
The terminals on a Local's Output-Bar carry the values of the roots to which they are connected to, and through an assignment at the Output-Bar assign those values to the Local operations' roots. To illustrate, consider the Local operation “1:1 alocal” in Figure 4 where, the terminal labeled $t_5$ carries the value of the root labeled $r_4$, and through the “$r_2 = r_4$” assignment at the Output-Bar, assigns the value of $r_4$ to $r_2$ in “1:1 B”. Roots corresponding to actual parameters can be redefined only when they are used as loop roots. To illustrate, consider the looped-Local operation “1:1 bLoop” in Figure 5, where the assignments “$r_2 = r_1$” and “$r_2 = r_4$” occur at the at the loop root in “1:1 E” and Output-Bar of the “1:1 bLoop”, respectively. The first assignment “$r_2 = r_1$” is necessary to carry the flow of data in case the flow of execution breaks before it reaches the Output-Bar of “1:1 bLoop”. The second assignment is necessary to increment the index of the loop or $r_2$. In this work, we make no distinction between atomic data such as a root representing an integer and aggregate data such as a root representing a list or root-list. Thus, a datalink connecting a root-list or partition control list root as it is the case on the left or right root on the operation that is labeled $n_4$ in Figure 3, to a list-terminal (the left or right list-terminal on the operation that is labeled $n_5$ in Figure 3), is regarded as definition-use association on the whole list datum.

4. APPLICABLE DATA-FLOW TEST ADEQUACY CRITERIA FOR VDFLs

Most code-based data-flow test adequacy criteria for imperative programs are defined in terms of paths that define and use variables through the control flow graph of a program. In this section we examine the applicability of several code-based test adequacy criteria to VDFLs.

4.1 Background: Traditional Data-flow Analysis

In imperative languages, data-flow analysis in a control flow graph of a function $f$ or $CFG_{(f)}$ focuses on how variables are bound to values, and how these variables are to be used. Variable occurrences in a $CFG_{(f)}$ can be either definitions or uses, depending on whether those occurrences store values in, or fetch values from the memory, respectively. Two types of uses are of interest to dataflow testing: computational use or $c$-use; and predicate use or $p$-use. Given a definition of a variable $x$ in a block $b_i$ corresponding to a node $n_i \in CFG_{(f)}$, we say that a block $b_j$ corresponding to a node $n_j \in CFG_{(f)}$ contains a computational use or $c$-use of $x$ if there is a statement in $b_j$ that references the value of $x$. We also say that a node $n_k \in CFG_{(f)}$ contains a predicate use or $p$-use of $x$ if the last statement in $b_k$ contains a predicate statement where the value of that variable is used to decide whether a predicate is true for selecting execution paths. A conditional transfer statement in a node $n \in CFG_{(f)}$ has two executional successors: $n_l$ and $n_m$, such that $l$ is different from $m$.

Paths in the $CFG_{(f)}$ that trace a definition to all or some of its uses is commonly known as definition-use chains or du-chains.

Data-flow analysis techniques for computing du-chains for individual procedures [Aho et al. 1986] are well known, and data-flow test adequacy criteria have been well researched for imperative languages, and various criteria have been proposed (e.g., see Clarke et al. [1989], Frankl and Weyuker [1988], Laski and Korel [1983], Ntafos [1984], Perry and Kaiser [1990], and Rapps and Weyuker [1985]). Dataflow test adequacy criteria can be particularly applicable to VDFLs. There are
We make a distinction between flow testing to be realized for precise algorithms to be used for obtaining data-flow analysis, thus allowing data-parts; a fact that shall become more clear in Section 4.3.

4.2 Data-flow Associations for VDFLs

Data-flow test adequacy concentrate on interactions between definitions and uses of variables in the source code. These are called definition-use associations or simply DUs. DUs can be appropriately modeled in VDFLs as datalinks. There are several reasons for this appropriateness. First, as with imperative languages, we recognize two types of variable uses in VDFLs: c-use or computational use; and p-use or predicate use. A c-use occurs when a datalink connects a root r on one operation o_i to a terminal t that exists on a non-control annotated operation o_j (j > i). For example, as depicted in Figure 4, the root labeled r_2 in method “2:2 A” is c-used in the Primitive operation show. A p-use occurs when a datalink connects a root r on an operation o_k to a terminal t that exists on a control annotated operation o_j (k > l). For example, as depicted in Figure 4, the root labeled r_1 in Case “1:2 A”, is p-used in the control annotated Match “2”. In VDFLs we consider two types of DUs: definition-c-use (def-c-use) association; and definition-p-use (def-p-use) association. Given a Universal method p, let O = {o_1, o_2,..., o_n} be the set of operations in p, and N be the set of blocks or nodes in an OCG_{(p)} representing p. Let R = {r_1, r_2,...,r_N} be the set of roots on an operation o_i, where 1 < i < n, and let T = {t_1, t_2,...,t_n} be the set of terminals on an operation o_j, where 1 < j < n, we have the following definitions:

• **Definition 4.1 - (def-c-use association for VDFLs):** a def-c-use association is a triple (n_i, n_j, (r, t)), such that, n_i and n_j are nodes ∈ OCG_{(p)} representing operations o_i ∈ O and o_j ∈ O respectively, r ∈ R in o_i, t ∈ T in o_j, there is a datalink between r and t, o_j is a non-control annotated operation, and there exists an assignment of values to p’s input, in which o_i reaches n_j. For example, the def-c-use with respect to r_1 at n_{12} and its c-use at t_7 in “2:2 E” of Figure 5 is (n_{12}, n_{13}, (r_5, t_7)).

• **Definition 4.2 (def-p-use association):** a definition-p-use association is a triple (n_i, (n_j, n_k), (r, t)), such that, n_i, n_j, and n_k are nodes or blocks in ∈ OCG_{(p)} representing the subset of operations {o_i, o_j, o_k} ∈ O, r ∈ R in o_i, t ∈ T in o_j, there is a datalink between r and t, o_j is a control annotated operation, and there exists an assignment of values to p’s input, in which n_i reaches n_j, and causes the predicate associated with n_j to be evaluated such that n_k is the next node to be reached. For example, the def-p-use with respect to r_1 at n_{2} and its p-use at n_{3} in “1:2 E” of Figure 5 is: (n_{2}, (n_{3}, n_{4e}), (r_1, t_1)), (n_{2}, (n_{3}, e_{(2:2 E)}, (r_1, t_1)).

There are three points to consider about these definitions:

• We make a distinction between p-uses and c-uses, and that lets us track whether a test suite T that exercises all du-associations in OCG_{(p)} also exercises both out-
comes of each control annotated operation that has a datalink connecting the root of some other operation to one of its terminals. This distinction, as we shall see, has consequences in the visual reflection technique we use in our color mapping scheme to show exercised du-associations or datalinks.

- In the absence of Persistents or global variables, the re-definition of a variable or root r does not arise, except on the Output-Bar of a Case that has been annotated with Loop control, and thus will not interfere with any other definition or redefinition of r along any particular path in OCG\(_{(p)}\). To illustrate, consider the example that is depicted in Figure 5, \(r_2\) is defined at the entry node of the loop \(n_{4e}\), and always redefined at the Output-Bar \((n_8)\) of every Case belonging to the Loop operation. Therefore, any du-chains that are applicable to \(r_2\) in \(n_{4e}\) are also applicable to \(r_2\) in \(n_8\). This fact is one factor that facilitates more efficient data-flow analysis in VDFLs. Aside from extracting non Loop-roots related du-associations from the editing engine, accounting for the Loop-root related ones, is a simple exercise of knowing where the definition is (node) and associating that definition with the same uses as those of the original definition since it is not possible to have a redefinition on any path in the OCG\(_{(p)}\).

- Analogous to the definition of du-associations for imperative programs in (e.g., see Clarke et al. [1989], Frankl and Weyuker [1983], Korel and Laski [1983], Ntafos [1984], Perry and Kaiser [1990], and Rapps and Weyuker [1985]), our du-associations, which can be determined statically may not all be executable. There may be no assignment of input values to a program that will cause a definition of a root r to reach a particular use on a terminal t. Determining whether such du-associations are executable is shown to be impossible in general and often infeasible in practice [Frankl and Weyuker 1988; Weyuker 1983]; thus, data-flow test adequacy criteria typically require that test data exercise (cover) only executable du-associations. In this respect, our criterion (as we shall show) is no exception and can indeed contain nonexecutable du-associations. In the rest of this article, to distinguish the subset of the static du-associations in a VDFLs that are executable, we refer to them as executable du-associations.

The second reason for the appropriateness of modeling datalinks as DU\(_s\) in VDFLs can be attributed to the visual reflection of tested or exercised DU\(_s\) that are associated with a datalink during testing or after a test suite has been exercised. In general, there is visually, a one-to-one mapping between every datalink and its associated DU\(_s\). A more detailed description of our color mapping technique to datalinks is found in Section 5.1.

4.3 Applicable Data-flow Testing Criteria

Having introduced the definition and use associations in VDFLs, we next briefly examine the applicability of three major dataflow testing criteria: Ntafos [1984]; Laski and Korel [1983]; and Rapps and Weyuker [1985].

Ntafos [1984] proposed a family of test adequacy criteria called the required k-tuples, where \(k\) is a natural number > 1. The required k-tuples require that a path set \(P = \{p_1, p_2, ..., p_m\}\) covers chains of alternating definitions and uses, or definition-reference interactions called k-dr interactions \((k > 1)\). An example of a k-dr interactions is depicted in Figure 6. In \(n_1\), there is a definition of a variable \(x_1\), that is used to define variable \(x_2\) in \(n_2\) such that \(x_1 \in n_1\) reaches \(n_2\) via path \(p_1\). Therefore, the
information assigned to variable $x_1 \in n_1$ is propagated to variable $x_2 \in n_2$. This information is further propagated to another variable, say, $x_3 \in n_3$ such that $x_2 \in n_2$ reaches $n_3$ via path $p_2$. This information propagation process continues until it reaches $n_k$. Thus, the set of paths \{p_1, p_2, ..., p_{k-1}\} form a k-dr interactions. The required k-tuples requires some subpath propagating each k-dr interaction such that (1) if the last use is a predicate the propagation should consider both outcome (true and false), and (2) if the first definition or the last use is in a loop, the propagation should consider either a minimal or some larger number of loop iterations. The required k-tuple coverage criterion, or k-dr interaction chain coverage criterion, then requires that all feasible k-dr-interaction chains should be tested.

Laski and Korel [1983] defined and studied another kind of testing path selection criteria based on data-flow information. They observed that a given node may contain uses of several variables. Each variable may be defined at more than one node. Different combinations of the definitions constitute different contexts of the computation at the node. Figure 7 depicts one of their strongest criteria; Ordered Context Coverage.

In an OCG($\phi$), it is possible to apply either the Ntafos [1984] or the Laski and Korel [1983] criteria; however, there are two main reasons why these criteria would not be feasible in VDFLs. First, accounting for the data-flow information required by these criteria requires complex and costly data-flow analysis necessitating back-ward propagation of variables and their uses, which, as we shall show, is more expensive then the chosen [Rapps and Weyuker 1985] criteria. Second, required paths in k-dr interactions and Ordered Context paths do not have a direct one-to-one mapping with data-links in VDFLs — a strategy that makes it possible to color an exercised datalink when the path that is associated with it has been traversed — and would be too complex and overcrowding for the tester to have to deal with, both visually and theoretically.

Therefore our approach to defining a dataflow test adequacy criterion for VDFLs adapts the “all-dus” dataflow test adequacy criterion defined for imperative programs in [Rapps and Weyuker 1985]. The all-dus, is concerned with tracing all “definition-clear” subpaths that are cycle-free or simple-cycles from each definition of a variable $x$ to each use reached by that definition and each successor node of the
use. Thus, adapting the all-dus testing criterion of [Rapps and Weyuker 1985] the
all-Dus for VDFLs can be defined as follows:

- **Definition 4.3 - (All-Dus):** given a VDFL Universal method \( p \) with its \( OCG(p) \), the
  set of complete feasible paths \( M \) in \( OCG(p) \), a set of execution paths \( Q \subset M \), and a
test suite \( T \) for \( p \), we say that \( T \) satisfies this criterion iff for each feasible definition-use
association \( = \{(n_i, n_j, (r, t)), (n_i, (n_j, n_k), (r_m, t_n)) \text{ or } (n_i, (n_j, n_l), (r_m, t_n))\} \),
there is some test case \( t \) in \( T \) such that, when \( p \) is executed on \( t \), there exists at
least one path \( q \) in \( Q \) on which every DU in \( OCG(p) \) is exercised.

There are several advantages associated with the applicability of the all-dus
criterion to VDFLs. First, since variables in VDFLs cannot be redefined, accounting
for the all-Dus can be obtained from the editing engine, and accounting for the
redefinition of loop-associated variable definitions (Loop-root) can be easily calcu-
lated since they always occur at the same place (Output-Bar of a Case). A second
advantage is the ease of visually mapping every path corresponding to a du-associ-
ation to a datalink. The du-associations corresponding to the redefinition of a Loop-
root on an Output-Bar of a Case are constructed during a testing session as indirect
datalinks. The construction process of the indirect links will discussed in Task 2 of
Section 5.1, in the context of the example in Figure 13. The third advantage is the
color mapping scheme that can be applied to each exercised du-associations or
datalink. Our color mapping scheme uses a continuous coloring technique to color
datalinks that are exercised only during failed executions (incorrect results), passed
executions (correct results), or both.

5. A METHODOLOGY FOR TESTING VDFLs

Several algorithms for collecting static control-flow and data-flow analysis tech-
niques [Aho et al. 1986] have been developed for imperative languages. All these
algorithms process the program’s textual source code to build the flow of control
and extract static analysis by propagating variable definitions along control flow
paths to conservatively identify executable du-associations. As previously discussed
in Section 1, the second class of differences between VDFLs languages and traditional
ones (the iconic nature of the VDFLs) makes it impossible to use conventional
text scanners to: construct the \( OCG \); preserve the control and data dependencies
among operations; extract all-Dus; or probe the code for dynamic tracking purposes.
To compensate for these fundamental differences, we have augmented a Prograph-
to-Java translator in a way that allowed us to extract the topologically sorted oper-
ations\(^2\), and perform LR parsing on them (before they are translated to Java) to
accurately build the \( OCGs \), as well as integrate support for our testing methodology
into Prograph’s environment at a fine granularity, providing functionalities that
gave us the ability to:

- Determine the source code static control-flow and data-flow analysis.
- Probe the Java code to automatically track execution traces of every test case \( t \)
in \( T \), which provide the information necessary to determine the exercised du-
associations that is involved in producing valid or invalid output values in a test

\(^2\) The topological sorted order of operations is performed automatically by the editing engine
while editing the program. This sorting preserves both the control and data dependencies
in developed programs.
• Visually communicate, through the use of a coloring scheme, the coverage of datalinks, which in turn play an important role in helping users locate potential faults in the visual code.

• Launch a user-accessible test sessions facility — developed through the tools add-on in Prograph — to visually pronounce validations output, and communicate to the user how well exercised a function is.

5.1 The Details of the Methodology

The approach we chose to gain access to the set of operations that is stored and maintained by the editing engine, involved the design and implementation of a component that acts as an interface to the editing engine’s Application Programming Interface (API). This approach has proven to be very useful in the phases of: (1) constructing the OCG and extracting dataflow analysis, using a one-pass LR “parsing” technique, (2) probing of the code as it is translated to Java, and (3) coloring and visual validation of datalinks after testing. We next discuss details related to each phase. The code used in all phases in our prototype was mainly written in Prograph, with the exception of few libraries which were written in C++. We next explain each phase of our methodology in the context of the example that is depicted in Figure 5.

Task 1 — Constructing the OCG and Computing Static Data-flow Analysis

Using the API interface component, we were able to access an indexed set of operations that is stored and maintained by the editing engine of Prograph. This information made it possible to perform a “one-pass” over the set, building the OCGs, and collecting their relevant du-associations. A portion of the design and algorithms of our object oriented one-pass solution is depicted in Figure 8. Figure 8 (a) depicts part of the Operation class hierarchy (implemented in Prograph). Each inherited class in the hierarchy has a method buildOper() — depicted in Figure 8 (c) and Figure 8 (d) for Primitive and InputBar operation, respectively — that determines the type of operation, its control, and then builds the appropriate node, edge(s), and extract relevant du-associations. The algorithm in Figure 8 (a) is invoked on all operations in the set. To illustrate how its main function (buildOper()) works without getting wrapped up in infinite details, we next describe our one-pass technique in the construction context of the Input-Bar (n1), and the primitive operations ask and integer? of Figure 5 that are labeled n2 and n3, respectively.

• When the Input-Bar operation is fetched from the set, the buildOper() algorithm in Figure 8 (c) is invoked. Line 2 constructs n1 in the OCG of Figure 5. Line 3 creates the edge between n1 (already fetched) and n2 the next operation to be fetched. Line 4 records the edge (n1, n2), and assign a false value to it. This false value will be true when this edge is traversed.

• When the operation ask is fetched from the set, the buildOper() algorithm in Figure 8 (c) is invoked. Since the ask Primitive is not annotated with any type of control, the first case of the switch statement in the algorithm of Figure 8 (c) on line 4 is executed. Line 5 determines O1 (or n2) in the OCG in Figure 5. Line 6 creates the edge between n2 (already fetched) and n3 the next operation to be fetched. Line 7 records the edge (n2, n3) in the OCG and assigns a false value to it. Line 8 makes a call to extractDUs (O, null, null, null) algorithm of Figure 8. Since n2 does not have any control annotations, the first switch case on line 2 of
Figure 8 — Algorithms in (b), (c), and (d) for building the OCG and extracting dataflow information.

1. **algorithm** buildOper (Oper $O$) {
    2.  constructNode ($O$);
    3.  switch (Oper) {
        4.  Case !isRepeat($O$) && !isControl($O$):
            5.  $O_1 = \text{determineNextOper} (O)$;  // Determine the next operation in the topologically sorted set.
            6.  $e = \text{constructEdge} (O, O_1)$;  // $e$ is the edge built between $O$ and $O_1$
            7.  OCG($f$).Edges = OCG($f$).Edges $\cup \{ e, \text{false}\}$  // Collect each edge in the OCG and assign it to false.
            8.  extractDUs ($O$, null, null, null);  // A call to extract the Def-c-uses of $O$
        9.  Case isRepeat($O$) && isControl($O$):
            10. $e = \text{constructEdge} (O, O)$;
            11. $O_2 = \text{determineNextOper} (O)$;
            12.  $e_1 = \text{constructEdge} (O, O_2)$;  // $e_1$ is the edge that represents one of the True and False outcomes
            13.  $O_T = \text{determineType} (O)$;  // $O_T$ is the operation type = {Os or Ox}
            14.  $O_C = \text{determineControl} (O)$;  // $O_C$ is the operation control {Fail, Finish, Terminate, or NextCase}
            15.  $O_2 = \text{determineNThTargetOper} (O, O_T, O_C)$;  // $O_2$ is the target Oper according to $O_T$ and $O_C$
            16.  if ($O_2 == \text{null}$)  // The $O_T$ and $O_C$ do not allow for 2 edges out of $O$
                17.  OCG($f$).Edges = OCG($f$).Edges $\cup \{ e, \text{false}\}$, $\{ e_1, \text{false}\}$;
            18.  extractDUs ($O$, $e_1$, null, null);  // The Def-c-uses of $O$
            19.  else  
                20.  $e_2 = \text{constructEdge} (O, O_2)$;
                21.  OCG($f$).Edges = OCG($f$).Edges $\cup \{ e, \text{false}\}$, $\{ e_1, \text{false}\}$, $\{ e_2, \text{false}\}$;
                22.  extractDUs ($O$, $e_1$, $e_2$, null);
            23.  Case isRepeat ($O$):
                24.  $e = \text{constructEdge} (O, O)$;
                25.  $e_1 = \text{constructEdge} (O, \text{determineNextOper} (O))$;
                26.  OCG($f$).Edges = OCG($f$).Edges $\cup \{ e, \text{false}\}$, $\{ e_1, \text{false}\}$;
                27.  OCG($f$).Edges = OCG($f$).Edges $\cup \{ e, \text{false}\}$, $\{ e_1, \text{false}\}$, $\{ e_2, \text{false}\}$;
                28.  extractDUs ($O$, $e_1$, null, null);
                29.  Case isControl ($O$):
                    30.  $O_1 = \text{determineNextOper} (O)$;
                    31.  $e = \text{constructEdge} (O, O_1)$;
                    32.  OCG($f$).Edges = OCG($f$).Edges $\cup \{ e, \text{false}\}$;
                    33.  $O_T = \text{determineType} (O)$;
                    34.  $O_C = \text{determineControl} (O)$;
                    35.  $O_2 = \text{determineNThTargetOper} (O, O_T, O_C)$;
                    36.  if ($O_2 == \text{null}$)  
                        37.  extractDUs ($O$, $e_1$, null, null);
                    38.  else  
                        39.  $e_1 = \text{constructEdge} (O, O_2)$;
                        40.  OCG($f$).Edges = OCG($f$).Edges $\cup \{ e_1, \text{false}\}$;
                        41.  extractDUs ($O$, $e_1$, $e_2$, null);
                        42.  }  // The Def-c-uses of $O$
}  // The Def-c-uses of $O$

the `extractDUs` algorithm is chosen. The for loop however does not execute since `n_2` does not have any terminals. Now consider the integer? `Primitive`. When the operation integer? is fetched from the set, the `buildOper()` algorithm in Figure 8 (c) is invoked. Since the integer? `Primitive` is annotated with Next-Case control, the fourth case of the switch statement in the algorithm of Figure 8 (c) on line 28 is executed. Line 29 determines `O_1` (or `n_4`) in the OCG in Figure 5. Line 30 creates the edge between `n_3` (already fetched) and `n_4` the next operation to be fetched. Line 31 records the edge (`n_3`, `n_4`) in the OCG and assigns a `false` value to it. Line 32 determines and returns `O_T` the type of `n_3` (`O_x` or `O_3`). You may recall from Section 3.2 that `O_x` ∈ `O` is the subset of operations that always succeed by default, and `O_x` ∈ `O` is the subset that does not. Line 33 determines and returns `O_C` the type of control (i.e. `Fail`, `Finish`, Next-Case, or `Terminate`) (in this case, it is `Next-Case` on failure). Line 34 invokes `determineNThTargetOper(O, O_T, O_C)` which determines `O_3` the `n^{th}` operation to which the other edge of the evaluation of the integer? is built to (in this case its the `Input Bar` or `n_15` in the OCG). Line 38 creates the edges between `n_3` and `n_15`; (shown in Figure 5 as an edge from `n_3` to the Entry node “1:2 E”, and from that to `n_11`). Line 39 records the edge (`n_3`, `n_11`), and finally line 40 makes a call to `extractDUs (O, e_1, e_2, null)`. Since `n_3` has a control annotations, the first switch case on line 2 of the `extractDUs` algorithm is chosen.

As previously mentioned, with the presence of loops in Prograph, `roots` associated with a loop or `Loop-roots` are first defined at the local’s `loop-root`, and then implicitly redefined at the `Output-Bar` of each case in the looped-Local. For example, as depicted in Figure 5, the `loop-root` `r_2` is first defined at the looped operation `bLooped`, and subsequently redefined at the operation labeled `n_8` with

```
1. algorithm extractDUs(Oper O, Edge e_1, Edge e_2, Edge e) {
2.     Case (e_1 = e_2 = e = null) : // O does not have any controls
3.         for each terminal t ∈ O {                     // for each terminal on the operation being processed
4.             O.use = O.t_0;
5.             O.p = O.t_0.connectedRoot();         // O_p is a predecessor of O.
6.             O.def-c-use = O.def-c-use ∪ {O_p.def, O.c-use, false} }
7.     Case (e_2 = null) && (e_1 != e = null) :    // O has a Repeat control
8.         for each terminal t ∈ O {                 // for each terminal on the operation being processed
9.             O.use = O.t_0;
10.            O.p = O.t_0.connectedRoot();         // O_p is a predecessor of O.
11.            O.def-p-use = O.def-p-use ∪ {O_p.def, O.p-use, e_1, false}};
12.     Case (e_1 != e_2 != e = null) :            // O has a Repeat and another control
13.         for each terminal t ∈ O {                 // for each terminal on the operation being processed
14.             O.use = O.t_0;
15.            O.p = O.t_0.connectedRoot();         // O_p is a predecessor of O.
16.            O.def-p-use = O.def-p-use ∪ {O_p.def, O.p-use, e_1, false}};
17.     Case (e_2 = null) :
18.         for each terminal t ∈ O {                 // O has a control
19.             O.use = O.t_0;
20.            O.p = O.t_0.connectedRoot();         // O_p is a predecessor of O.
21.            O.def-p-use = O.def-p-use ∪ {O_p.def, O.p-use, e_1, false})
```

Figure 9 — Collecting the dus.
the implicit statement \( r_2 = r_4 \). Thus, the du-associations related to a loop-root \( lr \) are divided into two sets. One set that satisfies the du-associations with regards to the definition of \( lr \) on the looped-Local, and a second set that satisfies the du-associations with regards to the implicit redefinition of \( lr \) on the Output-Bar of the looped-Local. The first set is collected by computing the du-associations with regards to the definition of \( lr \) that is connected or has uses, via wrap-around datalinks, to operations inside the looped-Local. For example, the definition of the loop-root \( r_2 \) on node 4_e in Figure 5 has a c-use on the operation labeled \( n_7 \), and a p-use on the operation labeled \( n_6 \). The second set is collected by computing the du-associations with regards to the implicit redefinition of \( lr \) (at the Output-Bar) that has uses on operations inside the looped-Local. For example, the implicit redefinition of the loop-root \( r_2 \) at \( n_8 \) has a c-use on the operation labeled \( n_7 \), and a p-use on the operation labeled \( n_6 \). Since the implicit redefinition of a loop-root always occur at the Output-Bar of the looped cases, collecting the du-associations associated with a loop-root before the implicit redefinition, can be resolved statically by relying on the automatic collection of the dataflow information provided by the editing engine during the visual coding phase.

Task 2 — Tracking Execution Traces, and Visual Representation of Results

To track the dynamic execution we have simply instrumented the Prograph-to-Java translator in a way that allows us to probe each operation/control annotated operation before it is translated. Once the Java textual code has been compiled, test suites are then run on the Java code. The probes allowed our testing environment to record, under each test case \( t \) in \( T \), the execution traces and maintain the states (true or false) of each operation, predicate operation, or du-associations, as collected by the algorithms in Figure 8 and Figure 9. To apply our color mapping scheme to exercised datalinks, our testing environment also maintained a set of pointers for each operation, root, and terminal to allow us to effectively color these datalinks after the execution of each test case \( t \) in a test suite \( T \). To illustrate the use concept of these pointers, consider the ask and integer? operations in “1:2 E” of Figure 5 and their root \( r_1 \) and terminal \( t_1 \), respectively. Figure 10 depicts in Window (a) and its related windows (b, c, d, e, and f), a series of connected arrows that starts in the highlighted item of (a) and ends in the highlighted item in (e). The series of arrows shows the various highlighted window items’ value, when double clicked in the same order as that of the arrow sequence. The sequence, when followed from (a) to (e), shows the primitive operation integer? in (a); its pointer value in (b) as the first item in the Window; the terminal pointer of \( t_1 \) in (d); its connected operation ask in (e); and finally the pointer value of operation ask in the top item of (f). The various values of operations, terminals, and roots are used in the color mapping scheme during testing.

The visual illustrations and colors we used to represent all-Dus coverage reflect two constraints that we believe to be important for the “integration” of visual testing into VDFLs. We derived these constraints from literature on cognitive aspects of programming (Gren and Petre [1996] and Yang et al. [1997]), and the use of color in fault localization [Agrawal, et al. 1995; Jones et al. 2002]. The constraints we placed on the visual representation and colors of exercised datalinks under test should: (1) be courteous of screen space and maintain consistency of visual constructs; and (2) be accessible and recognizable.

To satisfy the first constraint when reflecting through color the all-Dus coverage of datalinks and their associated du-associations, we introduced only one addi-
ational artifact to existing datalinks; indirect datalinks. Direct datalinks are of course created by the user at the visual coding phase. Indirect datalinks however are constructed, after the user initiates a testing session, from Loop-roots to terminals inside a looped-Local operation. For example, as depicted in Figure 13, four indirect datalinks are constructed from the loop-roots on the looped-Local “factorial” operation to the terminals on operations “0” “-1” and “*” to represent their associated du-associations. While indirect datalinks do introduce additional artifacts to the visual code, their presence are necessary to communicate, through color, their testedness. The indirect datalinks appear and disappear depending on whether the looped-Local window that is associated with the use (c-use or p-use) is opened. That is, if the looped-Local window is closed/minimized the indirect datalinks are made to disappear and the loop-roots that are associated with the indirect datalinks are made to blink to indicate that user attention is needed. Once the user double clicks on a blinking loop-root, the looped-Local window is opened and the indirect datalinks are made to reappear. The blinking, as a metaphor, has been used commercially in many of today’s operating systems and applications to draw a user’s attention to a specific area on the screen. For the sake of simplicity, direct and indirect datalinks will be referred to in the rest of this article as datalinks.

To satisfy the second constraint, and assist the analyst in identifying and minimizing the fault search space, we incorporated a color mapping scheme that uses a continuous level of varying color (hue) to indicate the ways in which the datalinks participate in the passed and failed test cases in a test suite. We selected colors along the red (danger), yellow (cautious), and green (safe), to improve our goal of drawing users’ attention to testing results, and potential fault locations. A similar approach [Jones et. al. 2002; Liblit et al. 2005] that used color to localize faulty statements in imperative languages found these color combinations to be the most natural and the best for viewing. Other studies found that, due to the physiology of

Figure 10 — A fragment of Prograph’s implementation showing the pointers used in tracking and coloring of operations ask and integer, and root r1 and terminal t1 in the example of Figure 5.
the human eye, red stands out while green recedes [Christ 1975; Murch 1984; Shneiderman 1998]. The color mapping scheme measurements and testing results will be described in Study 2 in Section 6.2.2.

The testing system we have implemented in Prograph’s IDE and used to produce Figure 10 and Figure 13 in this article, provide users with the ability to initiate an all-Dus testing session, and use the static and dynamic data collected from Task 1 and Task 2 to present the user with testing results in a debug-like window. This approach provides, in Prograph, an integrating environment of testing and debugging. Creating these debug-like windows were made possible through the API and external library that are available to third part developers of Prograph.

6. EXPERIMENTAL DESIGN AND EMPIRICAL RESULTS

To obtain meaningful information about the effectiveness of our all-Dus adequate testing methodology in: (a) revealing a reasonable percentage of faults in VDFLs; and (b) assisting users in visually localizing faults by reducing their search space, we designed and conducted two studies. We next describe the design setup, measurements analysis, and empirical results for Study 1. Study 2 is discussed in Section 6.2.

6.1 Study 1

In setting up the design of our first study, we used a set \( F = \{f_1, f_2, ..., f_8\} \) of 8 programs/functions\(^3\) we called the “base functions” set, and produced, for each \( f \in F \) a test pool\(^4\) \( tp(f) \). Next, we used \( tp(f) \) to create for each \( f \in F \) (i) \( DUT(f) = \{T_1, T_2, ..., T_k\} \) the set of du-adequate test suites with regards to \( f \), and (ii) \( RST(f) = \{T'_1, T'_2, ..., T'_k\} \) the set of randomly selected test suites for \( f \), such that for each \( j < k \), \( T_j = \{t_1, t_2, ..., t_d\} \) and \( T'_j = \{t'_1, t'_2, ..., t'_d\} \) are of equal size, and \( t \) and \( t' \) are test cases in \( T \) and \( T' \), respectively. We then created \( V(f) = \{v_1(f), v_2(f), ..., v_n(f)\} \) the set of faulty versions for each \( f \in F \), where each faulty version \( v_i(f) \in V(f) \) contained a distinct single fault. Finally, for each faulty version \( v_i(f) \in V(f) \in F \), we ran on \( v_i(f) \): (1) every du-adequate test suite \( T_j \in DUT(f) \) and recorded its fault detection ability; and (2) every randomly selected test suite \( T'_j \in RST(f) \) and recorded its fault detection ability. We say that a fault is detected when: in (1) the output of \( v_i(f) \) when executed on \( t \in T \), produce results that differ from that of \( f \) when executed on \( t \); in (2) the output of \( v_i(f) \) when executed on \( t' \in T' \), produce results that differ from that of \( f \) when executed on \( t' \). Similar studies, design approach, and fault-detecting abilities of several varieties of test suites applied to imperative and form-based programs, have been used in [Frankl and Weiss 1993]; [Hutchins et al. 1994]; and [Rothermel et al. 2001]. We next describe the rest of our study design setup and results.

6.1.1 Base Programs and Test Suites

Our base programs’ specifications for this study were chosen not to be overly complex. The logic behind this was to allow people involved in this study to seed faults in these programs, create their test pools, and examine their code for infeasible du-associations. The subject programs had to however be complex enough to be

---

3 We use the words program and function interchangeably in the rest of this article.
4 Details of creating test pools and other experimental steps will be explained later in this section.
considered realistic, and permit for the seeding of many hard-to-find faults. Each function was chosen to be compilable and executable as a stand-alone unit. Since our testing methodology has been implemented in Prograph’s IDE we obtained 8 Prograph programs from examples that were shipped with Prograph, and were thus considered to be commercially produced. Table 1 lists the numbers of operations, control annotated operations, and du-associations for each base program. These base programs provided the oracle to check the results of test cases executed on the faulty versions.

Table 1: The number of operations, edges, du-associations (Dus), number of faulty versions, and test pool size (number of test cases) for each base program.

<table>
<thead>
<tr>
<th>Programs</th>
<th>Operations</th>
<th>Edges</th>
<th>Dus</th>
<th>Faulty Version</th>
<th>Test pool size for each f</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bubble Sort</td>
<td>54</td>
<td>32</td>
<td>49</td>
<td>11</td>
<td>267</td>
</tr>
<tr>
<td>Factorial</td>
<td>27</td>
<td>6</td>
<td>32</td>
<td>15</td>
<td>179</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>49</td>
<td>8</td>
<td>50</td>
<td>13</td>
<td>192</td>
</tr>
<tr>
<td>Gaussian elimination</td>
<td>88</td>
<td>32</td>
<td>106</td>
<td>19</td>
<td>481</td>
</tr>
<tr>
<td>Matrix multiplication</td>
<td>64</td>
<td>28</td>
<td>56</td>
<td>9</td>
<td>318</td>
</tr>
<tr>
<td>Replace subtree</td>
<td>31</td>
<td>10</td>
<td>29</td>
<td>16</td>
<td>298</td>
</tr>
<tr>
<td>Topological sort</td>
<td>66</td>
<td>28</td>
<td>74</td>
<td>18</td>
<td>143</td>
</tr>
<tr>
<td>Word place: finding the place of a word in a sentence</td>
<td>82</td>
<td>31</td>
<td>107</td>
<td>20</td>
<td>524</td>
</tr>
</tbody>
</table>

6.1.2 Base Programs’ Test Pools and their Test Suites
Since it is easy to show, that for VDFLs, as well as for imperative programs, data-flow adequacy criteria (Definition 4.4) can produce test suites that are stronger than statement (Definition 3.2) or decision coverage criteria (Definition 3.3), we elected to acquire du-adequate test suites for our base programs to evaluate the effectiveness of our testing methodology. To obtain these we began by asking one Prograph expert to generate $TP(F) = \{tp(f_1), tp(f_2), ... , tp(f_8)\}$ the set of test pools containing for each base function $f_i \in F$ its own test pool $tp(f_i)$. To populate each $tp(f_i) \in TP(F)$, the user first created an initial set of test cases based on his understanding and knowledge of exercising $f_i$’s functionality and special values and boundary points that are easily observable in the visual code. The tester then examined the du-coverage achieved by the initial test cases set, and modified/augmented the initial set of test cases, to ensure that each executable du-association was exercised by at least 7 test cases. The resulting test pools for each $f_i \in F$, as depicted in Table 1, ranged in size from 143 to 524 test cases.

To obtain the du-adequate test suite $dut(f_i) = \{t_1, t_2,..., t_m\}$ from each $tp(f_i)$, we first determined, for each test case $t \in tp(f_i)$ it’s exercised du-associations, and then created $dut(f_i)$ by randomly selecting a test case $t$ from $tp(f_i)$, and adding it to $dut(f_i)$ only if it incrementally added to the overall cumulative coverage achieved by test cases added to $dut(f_i)$. We repeated this process until $dut(f_i)$ was du-adequate; yielding, for each $f_i \in F$ a $dut(f_i)$ of size $m$. This process resulted, after discarding dupli-
cate test suites, between 12 and 17 du-adequate test suites for each \( f_i \in F \). Finally, to create a randomly selected test suite \( rst_{(f)} \) for each function \( f_i \in F \), we performed the random function \( t' = \text{ran} (tp_{(f_i)} = \{t_1, t_2, ..., t_m\}) \), \( m \) times, where \( m \) is the size of \( dut_{(f)} = \{t'_1, t'_2, ..., t'_m\} \) to randomly select test cases from \( tp_{(f)} \) of the same sizes as that of the latter.

6.1.3 Faulty Versions

Ideally, the most desirable type of faults to study would be real faults that have been recorded in the course of the development of the software; however, due to inadequate information available to us about faults of that type, we created \( V_{(f)} \) the set of \( n \) faulty versions for each base function \( f_i \in F \). To do that, we asked 8 different individuals with different perception of fault-related production in Prograph programs (ranging in expertise from intermediate to experts), and mostly without knowledge of each other’s work, to introduce in each \( v_{i(f)} \in V_{(f)} \), a single fault that reflects, as realistically as possible, their experience with real Prograph development in an effort to preserve the accuracy of our study. The fault seeding process yielded, as depicted in Table 1, between 9 and 20 faulty versions for each \( f_i \in F \). The seeded faults were mostly changes to a single feature of the visual code, and took the form of creating erroneous or missing: datalink; control; input; or primitive.

6.1.4 Measuring the Effectiveness of Fault Detection

Let \( T \in DUT_{(f)} \) and \( T' \in RST_{(f)} \) be two test suites for \( f \in F \), respectively, and \( V_{(f)} = \{v_{1(f)}, v_{2(f)}, ..., v_{n(f)}\} \) is the set of faulty versions in \( f \) each containing a single known fault. We say that if \( TS = (T \cup T') \) detects \( (e < n) \) of the faults in \( V_{(f)} \), then the effectiveness of \( TS \) can then be measured by percentage of faulty versions whose faults are detected by \( TS \), and is given by \((e/k \times 100)\). A fault is detected by \( TS \) if there exists at least one test case \( t = (z, i, o_{v/i}, c_n) \in TS \) (recall Definition 3.1) that, when applied to \( v_{i(f)} \) and \( f \), causes the production of the tuple \( (z, i, a_{f}, c_n) \) in \( v_{i(f)} \), and the tuple \( (z, i, a_{f}, c_n) \) in \( f \). As mentioned earlier, base programs provided the oracle to check the output \( (o_{v} \) for valid or \( o_{i} \) for invalid) of a test case executed on a faulty versions.

6.1.5 Data Analysis and Overall Results

Figure 11 contains a separate graph for each of the eight base program \( f \in F \). Each graph contains every faulty version \( v_{i(f)} \) of \( f \), and each \( v_{i(f)} \) occupies a vertical bar position along the x-axis and is represented by an overlapping pair of vertical bars. The two overlapping bars depict the percentage of the du-adequate test suites (black bars) and the percentage of the randomly generated test suites (light grey), respectively, that detected the fault in that faulty version. The legend and information on the X-axis and Y-axis are depicted in the lower part of Figure 11. As the overlapping bars of each program and its faulty versions indicate in Figure 11, both \( DUT_{(f)} \) and \( RST_{(f)} \) missed faults in Study 1 which involved 121 faulty versions. Figure 12 indicates three vertical bars on the X-axis: The first bar represents with the Y-axis, the number of faulty versions in which the \( DUT_{(f)} \) was more effective than \( RST_{(f)} \); the second bar indicates, with the Y-axis, the number of faulty versions in which the \( RST_{(f)} \) was more effective than \( DUT_{(f)} \); and the third bar indicates, with the Y-axis, the number of faulty versions in which the \( RST_{(f)} \) was as effective
Figure 11 — Percentages of test suites that revealed faulty versions, per program, per version. Black bars depict results for du-adequate test suites; gray bars depict results for randomly generated test suites.

Figure 12 — This graph captures for each base program and its versions, three vertical bars on the X-axis containing the numbers of: \( \text{DUT}(f) > \text{RST}(f) \); \( \text{DUT}(f) < \text{RST}(f) \); and \( \text{DUT}(f) = \text{RST}(f) \); respectively.
as the $DUT_{(f)}$ in detecting the fault. As depicted in Figure 12, there were across the entire study 80 faulty versions in which the $DUT_{(f)}$ were more effective then $RST_{(f)}$. 20 faulty versions in which the $RST_{(f)}$ were more effective then $DUT_{(f)}$, and 21 faulty versions in which $RST_{(f)}$ were as effective as $DUT_{(f)}$. This results clearly indicate that du-adequate test suites are more effective at revealing faults than their randomly generated counterparts. Two similar studies by Hutchins et al. [1994] for imperative languages, and [Rothermel et al. 2001] for form-based languages also showed that du-adequate test suites were more successful at detecting faults.

6.1.6 Study Conclusion

In this study, we conducted a highly accurate measurements of test suites ($T \in DUT_{(f)}$ and $T' \in RST_{(f)}$) effusiveness in detecting faults. There are however some aspects of this study that would limit our ability to generalize results pertaining to the capabilities of our test suites results. We thus need to make it clear that (i) our base programs may not be large enough, and may not unnecessarily capture the program space in VDFLs; (ii) our faulty versions creation method may not necessary represent the fault space in VDFLs; and (iii) our test pools were generated based on the correct base programs, and it may be worth investigating the effectiveness of our test suites had we included the faulty version in the test pool extraction. Second, we cannot claim any knowledge on different groupings of faults. For example, we were not able to determine, after an exhaustive examination of the faults in our programs, why some faults were easily detected by $DUT_{(f)}$, and others by $RST_{(f)}$, or faults equally detected by both $RST_{(f)}$ and $DUT_{(f)}$. We did conclude that there is no clear strategy to follow to bring about such grouping.

6.2 Study 2

Thus far, we have not demonstrated that our testing methodology, combined with our color mapping scheme and visual feedback, can assist users in visually localizing faults by reducing their search space. In Study 2, we use the $DUT_{(f)}$ of Study 1 (since it was proven to be more effective at detecting faults), to measure how effective is our color mapping scheme in helping to locate faults after a test suite set is executed on a faulty version. The procedure in this study was as follows: for every $f \in F$, we executed each $v_i(f) \in V(f)$ on each $T \in DUT_{(f)}$, and then applied, for to each $(v_i(f), T)$ pair, the continuous color scheme and visual feedback to the datalinks (du-associations: $c$-use and $p$-use). We then calculated for each $v_i(f)$, over all test suites $T \in DUT_{(f)}$, the continuous color of each datalink in $v_i(f)$. The datalinks color were then used in this study to investigate the space faults, and the user’s ability to localize faults. We next describe our color mapping scheme, empirical results, and study conclusion.

6.2.1 Color Mapping Scheme

The fundamental idea of our color mapping scheme revolves around continuously coloring (red to yellow to green$^5$), according to the relevant percentage of test cases that produce valid results when executing the du-associations to the test cases that exercise the du-associations but produce invalid results. As such, the

$^5$ These color combination were found to be the most natural in [Jones et al. 2002], and they also complement the blue colored program constructs in Prograph.
color of a datalink can be anywhere in the continuous spectrum of colors from red to orange to yellow to green. Similar color mapping scheme was used in [Agrawal et al. 1995; Jones et al. 2002] to study the localization of faulty program lines or statements. Other similar studies colored cells for validation in form-based languages [Rothermel et al. 2001].

For every direct and indirect datalink in $v_{i(f)} \in V(f)$, our color mapping scheme is as follows:

- If the percentage of the test cases in $DUT_{i(f)}$ that exercises a direct and indirect datalink produces valid results, when ran on $v_{i(f)}$, is much higher than the percentage of test cases that produce incorrect results, then the hot spot appears more green; otherwise it appears more red. The intuition here is that datalinks that are executed primarily by test cases that produce valid results are not likely to be faulty, and thus are colored green to denote “possible safe neighborhood”; otherwise, datalinks that are executed primarily by test cases that produce invalid results should be highly suspicious as being faulty, and thus are colored red to denote “possible faulty neighborhood”. The notion of neighborhood is necessary in VDFLS since, unlike imperative languages, the code constructs is a graph-like, and distributed over many Cases. Hence our notion of localizing faults through color mapping differ from that introduced for imperative languages [Agrawal et al. 1995; Jones et al. 2002]. In our study the fault space is considered to be found if it is located in the “neighborhood” of one or more datalinks that are colored “suspiciously” or in the reddish-orangish color range in a Case containing the fault. The intuition here is that this can direct a user’s attention to a faulty neighborhood, and minimize the search space of faults.

- If the percentage of the test cases in $DUT_{i(f)}$ that produces valid results, when ran on $v_{i(f)}$, is near-equal to the percentage of test cases that produce incorrect results, then the hot spot is colored yellow. The intuition here is that datalinks that are executed by test cases that produce a mixture of valid and invalid results are consider “a cautious neighborhood, and are thus colored yellow.

- If a 100% of the test cases in $DUT_{i(f)}$, ran on $v_{i(f)}$, do not exercise a particular hot spot, then that hot spot is not colored, and is left blue (its original color). The intuition here is that if a datalink is never exercised, leaving it in its original color could give incentive to users to further investigate its neighborhood in case of failure.

6.2.2 Color Measurements of Datalinks

Recall Definition 3.1 where we defined a test suite $T = (Z, I, O_{V/I}, C_N)$. Using $Z$ the test case number, $I$ the set of inputs values, $O_{V/I}$ $v_{i(f)}$’s output valid/invalid set of results of executing $v_{i(f)}$ using $T$, and $C_N$ the set of covered nodes in $OCG(v_{i(f)})$ that helped us to determine the exercised datalinks in each $t = (z, i, o_{v/i}, c_n) \in T$ we colored a hot spot or datalink representing a c-use as follows: color = low red + (%valid() / (%valid() + %invalid()) * color range). The valid() function returns, as a percentage, the ratio of the number of test cases that executed the hot spot and produced a valid result to the total number of test cases in the test suite. Likewise, the

---

6 The reader is encouraged to read the PDF file of this article or print a colored copy.
invalid() function returns, as a percentage, that executed the hot spot and produced an invalid result to the total number of test cases in the test suite. The value for the *low red* is the lower end of the color spectrum, and has a value of 0. The *color range* corresponds to the high end of the desired color spectrum; green in our case. For example, if a datalink representing a *c-use* is executed by 50% of test cases that produced invalid results, and 100% of the test cases that produced valid results, its color will be 10/15 * 120 = 80 which renders this as a green-yellowish color, as depicted in the color legend of Figure 14. The color of a hot spot or datalink representing a *p-use* is computed, using the above formula, as the average color of the du-associations representing the *true* and *false* exercised outcome.\(^7\) For example, if a datalink representing a *p-use* (true) is executed by 50% of test cases that produced invalid results, and 100% of the test cases that produced valid results, and its *p-use* (false) is executed by 100% of test cases that produced invalid results, and 50% of the test cases that produced valid results, its color will be 1/3 * 120 = 40, and thus the datalink will be rendered at (80 + 40)/2 = 60. The Indirect datalinks representing (*c-use* and *p-use* associations) that are constructed during testing are colored analogously to their direct datalinks counterpart.

### 6.2.3 Data Analysis and Overall Results

To evaluate our color mapping technique, and report how frequently a fault is found in a reddish-orangish neighborhood, we analyzed the color mapping applied to our subject program’s faulty versions. To simplify the presentation of faulty versions and their fault space, we decided to represent, as depicted in Figure 14, a separate graph for each of the eight base program’s faulty versions. Each faulty version \( v_{if} \) of \( f \) occupies a vertical bar (with color spectrum) position along the x-axis and represents the low and high neighborhood color of the fault space where the fault is found. The Y-axis represents the color range (120). The color legend is depicted in the lower part of Figure 11.

Across all versions of the base programs, there were two categories of neighborhood fault space that were found to exceed the “danger zone”. The first category had its low color spectrum in the appropriate zone but its high color spectrum in the inappropriate zone. For example, as depicted in Figure 14, versions 3 and 6 of the Factorial base program have their fault space approximately between 53 and 60. In the second category, both the low and high color spectrum were not found in their appropriate zone. For example, as depicted in Figure 14, version 11 of the Factorial base program which has its fault space approximately between 60 and 70. Across all faulty versions of the base programs, less than 6% was found to be in category 1, and less then 3% was found in category 2. We were not very concerned with category one since the majority of the datalinks in all instances were in the low color spectrum, and were very indicative to the user, as to where the fault space is. As for the second category, we examined their versions, and discovered that the fault was in erroneous datalinks that initialize roots (variables) values which were used in datalinks execution by all or most test cases.

Versions 11 and 15 of Factorial and Topological sort; in particular, were found to have more than one faulty space. This is not considered a threat to our testing.

\(^7\) We initially tried to color datalinks represents def-p-use associations [recall Definition 4.2: \((n_1, (n_2, n_k), (r_x, t_y)), (n_1, (n_j, n), (r_x', t_y'))\) by dividing them into two halves and applying the color mapping scheme to each half separately. This approach was found to be difficult to explain to users with regards to the overall color mapping consistency scheme.
techniques since users can use the process of elimination in examining the faulty spaces or neighborhoods.

6.2.4 Study Conclusion

One significant problem in Study 2 was the presence of infeasible paths and their related du-associations in the faulty versions. There are two different causes to infeasible paths. The first cause is semantic dependencies that always hold. These dependencies in the single fault versions of our subject programs varied in nature from a du-association on a path that is directly related to conditional statements whose either true or false outcome can never be realized to loops-related du-associations paths that could never be traversed. The presence of these du-associations is usually referred to as dead du-associations. The second cause to infeasible paths is due to the limitation or lack of input data to exercise them.

Not being able to “color” datalinks was difficult to explain to users. It is our intention to deal with the first cause by trying to detect, using static analysis similar to that of Clarke [1976], and color identified infeasible datalinks green.

As for the category of the fault space spectrum, we intend to address this situation by using a more advanced visualization technique that could perhaps make use of the dependency analyses and slicing of the code. Computing static output slices can be achieved by either using iterative dataflow equations [Aho et al. 1986; Weiser 1984], or using a dependence graph [Horwitz et al. 1990]. The second

Figure 13 — The Factorial Main under test - showing color reflections on datalinks.
approach is particularly applicable to VDFLs, since OCG is already a graph representing both data and control dependencies.

7. CONCLUSIONS

The recent increasing popularity in the visual paradigm in general and VDFLs in particular, resulted in many languages that are being used to produce many research and commercial software. Further, this popularity is likely to grow due to the user's anticipation of moving into the visual age of programming. VDFLs are prone to faults that can occur at the visual coding phase. To provide users of this paradigm with some of the benefits that are enjoyed by their imperative counterpart, we have developed a testing methodology that is appropriate for VDFLs. Our methodology and algorithms are compatible with and accommodates all visual languages with a dataflow computational engine. Our algorithms are also efficient; especially when given the fact that our static dataflow analysis is derived from probing the same data structures that is used to execute the visual code. The article

Figure 14 — This graph captures for each base program the color range of the neighbourhood where the fault was found.
presented an implementation overview of a tool that we have developed that implements our testing methodology, along with the results of two studies that evaluate the effectiveness of our fault detection and fault localization technique.

Our empirical results from Study 1 suggest that our testing methodology can achieve fault detection results comparable to those achieved by analogous techniques for testing imperative languages. To investigate the importance of these results and evaluate their potential benefits to VDFLs users, we implemented for Study 2 a color mapping technique that is based on our all-DUs testing methodologies. This coloring technique provided a visual mapping of the participation of datalinks in testing to assist users with no formal testing theory in minimizing the fault search space. The datalinks are colored using a continuous spectrum from red to yellow to green. Our empirical results from study 2 suggest that our technique is promising for helping locate suspicious constructs in VDFLs and suggest some directions for future work. We were encouraged that, for our subject programs our technique significantly reduced the search space for the faults in a single fault version. We suspect however that users that have programming experience will perform differently using our testing approach than users who do not have such experience. Thus, we intend to examine the relative effectiveness of our testing to both of these user populations. To build knowledge of testing skills among less experienced users, one approach we could take will be to allow these users to experiment with weaker adequacy criteria such as all-Nodes or all-Edges.

Since Study 2 focused on a single faulty version, we are currently conducting additional studies to further evaluate our technique with multiple faults by applying all-Nodes, all-Edges and all-Dus in one testing session, all at once, and evaluate the faults localization detection ability of our system. The color mapping scheme that we are applying to the operations and control annotated operations (hot spots) is depicted in Table 2.

One approach that can be beneficial to the user is the ability of modify the source code after locating a fault, and have our testing environment rerun the modified program on the test suite and dynamically update the views. One approach to doing this is to incorporate regression test selection technique similar to those found in [Gupta et. al 1996].

Finally, the testing methodology presented in this article addresses only one of the important problems in dealing with VDFLs errors. Other testing problems have

<table>
<thead>
<tr>
<th>Hot Spot</th>
<th>Hot spot color scheme</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
<td>A non control annotated operation's borders are colored to indicate its correspondent OCG's node participations in a test suite.</td>
</tr>
<tr>
<td>Predicate Operation</td>
<td>The borders and the check/cross marks represent the hot spots of interest, and they are associated with the True and False edges in the OCG, respectively. If the operation $o \in O_s$, then both hot spots will be proportionately colored when the edge $e \in OCG$ that ia associated with $o$ is exercised. If $o \in O_x$, then the borders are appropriately colored when the when the true edge $e \in OCG$ is exercised, and the check/cross mark is appropriately colored when the when the true edge $e \in OCG$ is exercised.</td>
</tr>
<tr>
<td>Loop</td>
<td>The loop arrows are appropriately colored to indicate the &quot;loop back edge&quot; participation in the OCG after a test suite.</td>
</tr>
<tr>
<td>Multiplex</td>
<td>The 3-D like lines on the multiplexed operation are colored to indicate the &quot;back edge&quot; participation in the OCG after a test suite.</td>
</tr>
</tbody>
</table>
been examined in the context of imperative languages. These problems include: generating test inputs; validating test outputs; and detecting nonexecutable code. These problems are also important in the context of VDFLs, and in our ongoing work we are investigating them. We are also working on a way to scale up the methodology by taking into account global variables, and dynamic variables.

ACKNOWLEDGMENTS

We thank Pictorius Incorporated for providing us with details and insight on how to integrate our testing methodology into the IDE of Prograph. In particular many thanks to Garth Smedley.

REFERENCES


