Controlled Dataflow Languages

Philip T. Cox Simon Gauvin

Dalhousie University, Canada

{*pcox,gauvins*}*@cs.dal.ca*

Abstract

In an important subclass of visual dataflow languages that includes many developed for industrial use, programs consist of acyclic diagrams embedded in control structures of some form. We present here a formalisation of this class of languages, which we call controlled dataflow. This work was motivated by a previous study of an exceptions mechanism for languages of this type, since to define how the exceptions mechanism would be incorporated into any CDL, we needed a formalism to precisely capture the syntax and semantics of this class, including a protocol for including language-specific control structures. To illustrate the formalism, we provide examples that show how it captures conditional execution, iteration and exception handling.

1. Introduction

It has been reported that the main challenge to visual dataflow language design has not been the difficulty in graphically representing dataflow, but rather, representing control flow semantics within dataflow [11, 10]. Although, formal semantics play a fundamental role in the development, specification, and comprehension of control flow constructs, existing dataflow formalisms are either too language specific, or incomplete, to provide a general solution to this problem [12].

Classical dataflow, developed by Dennis *et al.* [8] as a parallel hardware architecture solution to the so-called "von Neumann Bottleneck" problem and previously by Sutherland as a graphical specification for procedures [23], influenced the design of early visual programming languages such as DDN [6], and GPL [7]. Ackerman [1], and later Abramson [24] listed the following important features of dataflow languages: 1) freedom from side-effects, 2) locality of effect, 3) data dependencies equivalent to scheduling, 4) single assignment of variables, 5) lack of history sensitivity in procedures. However, they also note that items 1 and 4 lead to an awkward notion for conditional execution and iteration based on gate and switch nodes. This design presents challenges to developers as programs increase in size and complexity, and the comprehensibility of the graph decreases.

The difficulty of representing control structures in classical dataflow encouraged language designers to abandon switches, gates, and cycles and adopt high-level control nodes to represent control constructs. In these languages, control is achieved via hierarchically embedded control structures. Examples of this class of *Controlled Dataflow Languages* (CDL) are LabVIEW [13], ShowAndTell [19], and Prograph [5]. Despite these advances work still remains on improving control structures. A recent survey of advances in dataflow programming languages confirms that control flow constructs are not adequately represented in extant languages [14].

The development of imperative textual languages, which constitutes the largest group of languages in professional use today, has benefited from the existence of mature tools built up over many years. These tools provide facilities for the definition of syntax, and development of scanners, parsers, interpreters, compilers, and debuggers. Of particular significance is that these tools are based on sound semantic theory [26]. In contrast, although the commercial CDLs listed above share common features, each was developed "from scratch", and their documented descriptions, ranging from ad hoc to formal, have no common thread. As a result, CDLs do not have a theory to support tools specific to this class of languages.

The goal of the work reported here is to provide a framework for investigating the addition of various kinds of control constructs to CDLs. The research evolved as a result of earlier work on exception handling for CDLs [4], since to define how the exceptions mechanism would be incorporated into any such language, we needed a formalism to precisely capture the common syntax and semantics of CDLs, including a protocol for including language-specific control structures. Such a framework not only serves as a basis for investigating new control flow constructs, but as a kernel for test-bed implementations for new tools and CDLs.

In the next section, we characterise CDLs by defining their common syntactic and semantic features. In section 3 we provide examples illustrating these definitions. The first shows how they capture LabVIEW's iteration and conditional execution, while the second involves a simple case of exception handling. Section 4 discusses related work, and section 5 provides some concluding observations.

2. Characterisation of Controlled Dataflow

In this section we formally define the class of CDLs. Note that these definitions do not describe a language, but capture the common characteristics of languages of this class. In particular, we include the concept of control structure without defining any specific control structures.

Section 3 provides examples that illustrate these definitions. We urge the reader to consult these examples while reading the rest of this section.

2.1. Conventions

In the following, various entities are defined as tuples, the components of which are identified by names. We will use functional notation to refer to components of such entities. For example, if E is an expansion (see Definition 9), then $\mathbf{oper}(E)$ refers to the first component of E.

We will use functional notation to refer to elements of sequences. For example, if X is a sequence $X(2)$ denotes its second element. Expressions in which functional notation occurs more than once are read left to right; for example $x(y)(z)(w)$ means $((x(y))(z))(w)$.

Sequences may be treated as sets when the meaning is clear in context. In particular, a sequence may be used as an operand to a set operation.

If f is a function with domain X, and Y is a sequence over X, $f(Y)$ denotes the sequence $(f(Y(1)), f(Y(2), \ldots)$.

If f is a function with domain X, and Y is a subset of X, $f(Y)$ denotes the set $\{f(y) \mid y \in Y\}$.

If X is a set, we denote the set of all finite sequences of elements of X by X^*.

2.2. Syntax

The following definitions describe the basic characteristics of CDLs. In a specific language, additional structures and functions may be necessary. For example, operations in Prograph have names, necessitating a function that maps the set of operations of Def. 1 to some name set.

We assume the existence of infinite, arbitrary but fixed sets of $variables$, denoted $\mathcal{X}$, and $identifiers$, denoted $\mathcal{N}$.

Definition 1 A *controlled dataflow language* $\mathcal{L}$ is a 5-tuple ($\mathcal{O}$, $\mathcal{D}$, $\mathbf{terms}$, $\mathbf{roots}$, $\mathbf{body}$), where $\mathcal{O}$ and $\mathcal{D}$ are mutually disjoint sets called $\mathbf{operations}$ and $\mathbf{diagrams}$ of $\mathcal{L}$ respectively, and:

(i) $\mathbf{terms}$ and $\mathbf{roots}$ are functions from $\mathcal{O} \cup \mathcal{D} \to X^*$

(ii) $\mathbf{body}$ is a function mapping $\mathcal{O} \to 2^{N \times \mathcal{D}}$, and $\mathcal{D} \to 2^{\mathcal{O}}$.

(iii) if $O \in \mathcal{O}$, then $\mathbf{terms}(O)$ and $\mathbf{roots}(O)$ are disjoint, and no variable occurs more than once in $\mathbf{roots}(O)$.

(iv) if $D \in \mathcal{D}$ and some variable occurs more than once in $\mathbf{vars}(D)$ (see Def. 12), then it occurs exactly once in $\mathbf{roots}(D) \cup \{x \mid x \in \mathbf{roots}(O)$ for some $O \in \mathbf{body}(D)\}$

(v) if $D \in \mathcal{D}$, there exists a partial order $<$ on $\mathbf{body}(D)$ such that if x is a root of O_1 and a terminal of O_2, then $O_1 < O_2$

If O is an operation, identifiers are associated with diagrams in $\mathbf{body}(O)$ in order to distinguish the roles the diagrams play in the execution of O. To streamline our discussion, we will use informal but clear phrases such as "the diagrams of O".

Roots and terminals are data producers and consumers, respectively; so if a variable occurs in a diagram as a root and a terminal, a data flow link is established from the root to the terminal. Condition (iv), the single-assignment property, guarantees that no variable is assigned a value more than once. Condition (v) disallows data flow cycles. As we will see, every operation involved in an execution occurs in a diagram, except for the operation that starts the execution. Condition (iii) ensures that this "ex-diagram" operation does not violate the single assignment and acyclicity conditions.

Definition 2 Given a CDL $\mathcal{L}$, we define a function $\mathbf{vars}$: $\mathcal{O} \cup \mathcal{D} \to 2^{\mathcal{X}}$ as follows:

(i) if $O \in \mathcal{O}, \mathbf{vars}(O) = \mathbf{terms}(O) \cup \mathbf{roots}(O)$.

(ii) if $D \in \mathcal{D}, \mathbf{vars}(D) = \mathbf{terms}(D) \cup \mathbf{roots}(D) \cup \{x \mid x \in \mathbf{vars}(O)$ for some $O \in \mathbf{body}(D)\}$.

Definition 3 The *inarity* and *outarity* of an operation O are, respectively, $|\mathbf{terms}(O)|$ and $|\mathbf{roots}(O)|$.

Definition 4 The *inarity* and *outarity* of a diagram D are, respectively, $|\mathbf{roots}(D)|$ and $|\mathbf{terms}(D)|$.

Definition 5 If the inarity and outarity of an operation or a diagram are m and n respectively, we say that the *arity* of the operation or diagram is (m, n).

Definition 6 Let O_1 and O_2 be operations of L, then O_2 is a $variant$ of O_1 induced by a bijection γ on $\mathcal{X}$ iff

(i) O_1 and O_2 have the same arity.

(ii) $\gamma(\mathbf{terms}(O_1)) = \mathbf{terms}(O_2)$

(iii) $\gamma(\mathbf{roots}(O_1)) = \mathbf{roots}(O_2)$

(iv) there exists a bijection ψ from $\mathbf{body}(O_1)$ to $\mathbf{body}(O_2)$ such that if $(x, D) \in \mathbf{body}(O_1)$ then $\psi((x, D)) = (x, D')$ where D' is a variant of D induced by γ.

Definition 7 Let D_1 and D_2 be diagrams of L, then D_2 is a $variant$ of D_1 induced by a bijection γ on $\mathcal{X}$ iff there exists a bijection ψ from $\mathbf{body}(D_1)$ to $\mathbf{body}(D_2)$ such that

(i) for each $O \in \mathbf{body}(D_1), \psi(O)$ is a variant of O induced by γ;

(ii) for all operations $O_1, O_2 \in \mathbf{body}(D_1)$ and integers i and j such that $1 \leq i \leq |\mathbf{roots}(O_1)|$ and $1 \leq j \leq |\mathbf{terms}(O_2)|$, $\mathbf{roots}(O_1)(i) = \mathbf{terms}(O_2)(j)$ iff $\mathbf{roots}(\psi(O_1))(i) = \mathbf{terms}(\psi(O_2))(j)$.

There is not necessarily a unique variant of an operation or diagram corresponding to a given bijection γ. For example, a diagram may have two operations which have the same terminals, no roots and the same body, in which case there are two possible bijections ψ for mapping the body of a diagram to the body of a variant.

Definition 8 A CDL L is $closed$ iff for every bijection $\gamma : \mathcal{X} \leftrightarrow \mathcal{X}$, if O_1 is an operation of L and D_1 is a diagram of L, then there exists an operation O_2 of L and a diagram D_2 of L such that O_2 and D_2 are, respectively, variants of O_1 and D_1 induced by gamma.

Definition 8 embodies a standard property of programming languages, that consistent renaming of variables does not alter the meaning of a program.

2.3. Semantics

The definitions that follow apply to a closed CDL L. We will define the semantics of L by characterising an execution as a tree which is transformed by repeatedly applying a function to certain nodes. We define two types of structures, *operation expansions* and *diagram expansions*, that form the nodes of this tree.

A diagram expansion captures the state of execution of a dataflow diagram; that is, the usual notion of execution driven by the flow of data through a graph. An operation expansion corresponds to the execution of an operation in a data flow diagram, and embodies the non-dataflow control semantics specific to the language in question.

Definition 9 An $expansion$ of an operation O is a pair $E = (\mathbf{oper}, \mathbf{body})$ where $\mathbf{oper}(E) = O$, $\mathbf{body}(E)$ is a set of diagram expansions such that for each $B \in \mathbf{body}(E)$

(i) B is an expansion of a variant of a diagram in $\mathbf{body}(O)$;

(ii) $\mathbf{vars}(B)$ is disjoint from both $\mathbf{terms}(O)$ and $\mathbf{roots}(O)$;

(iii) if $B' \in \mathbf{body}$ and $B' \neq B$, then $\mathbf{vars}(B)$ and $\mathbf{vars}(B')$ are disjoint.

Definition 10 If E is an operation expansion, $\mathbf{vars}(E) = \mathbf{terms}(\mathbf{oper}(E)) \cup \mathbf{roots}(\mathbf{oper}(E)) \cup \{x \mid x \in \mathbf{vars}(B)$ for some $B \in \mathbf{body}(E)\}$.

Definition 11 An $expansion$ of a diagram D is a 4-tuple $E = (id, \mathbf{roots}, \mathbf{terms}, \mathbf{body})$ where id is an identifier, $\mathbf{roots} = \mathbf{roots}(D)$, $\mathbf{terms} = \mathbf{terms}(D)$ and $\mathbf{body}$ is a set of operation expansions such that for each $B \in \mathbf{body}$

(i) there is exactly one $O \in \mathbf{body}(D)$ such that $O = \mathbf{oper}(B)$;

(ii) $\mathbf{vars}(\mathbf{body}(B))$ is disjoint from both $\mathbf{terms}$ and $\mathbf{roots}$;

(iii) if $B' \in \mathbf{body}$ then $\mathbf{vars}(\mathbf{body}(B))$ is disjoint from $\mathbf{vars}(B')$.

The restriction in these definitions on the occurrences of variables ensure that variables are local, so no data is unexpectedly transmitted between unrelated diagrams.

Definition 12 If E is a diagram expansion, $\mathbf{vars}(E) = \mathbf{terms}(E) \cup \mathbf{roots}(E) \cup \{x \mid x \in \mathbf{vars}(B)$ for some $B \in \mathbf{body}(E)\}$.

Definition 13 An operation expansion E if *basic* iff $\mathbf{body}(E) = \varnothing$. A diagram expansion E is *basic* iff every operation expansion in $\mathbf{body}(E)$ is basic.

Definition 14 If E is an operation expansion, we denote by $\mathbf{inputs}(E)$ and $\mathbf{outputs}(E)$ the set if variables $\mathbf{terms}(\mathbf{oper}(E)) \cup \{x \mid x \in term(B)$ for some $B \in \mathbf{body}(E)\}$ and $\mathbf{roots}(\mathbf{oper}(E)) \cup \{x \mid x \in \mathbf{roots}(B)$ for some $B \in \mathbf{body}(E)\}$, respectively.

Execution of CDLs involves two familiar mechanisms: function calling, in which an operation is executed by invoking the code which it represents, and dataflow triggering, in which execution of an operation is initiated by the presence of data at some (perhaps none) of its terminals. We will define an execution as a tree, the nodes of which are operation expansions and diagram expansions, respectively corresponding to these two mechanisms. In an execution step, some subtree is replaced by a new tree. The structural properties of this replacement are defined as follows.

Definition 15 If E is an expansion of a diagram D, $E_1 \in \mathbf{body}(E)$ and E_2 is an expansion of $\mathbf{oper}(E_1)$ such that $\mathbf{vars}(\mathbf{body}(E_2))$ is disjoint from $\mathbf{vars}(E) - \mathbf{vars}(\mathbf{body}(E_1))$, then $(id(E), \mathbf{roots}(E), \mathbf{terms}(E), (\mathbf{body}(E) - \{E_1\}) \cup \{E_2\}))$ is an expansion of D, called a *direct replacement of* E_1 *by* E_2 *in* E.

Definition 16 If E is an expansion of an operation O, $E_1 \in \mathbf{body}(E)$ and E_2 is an expansion of a variant of a diagram in $\mathbf{body}(O)$ such that $\mathbf{vars}(E_2)$ is disjoint from $\mathbf{vars}(E) - \mathbf{vars}(E_1)$, then $(\mathbf{oper}(E), (\mathbf{body}(E) - \{E_1\}) \cup \{E_2\}))$ is an expansion of O, called a *direct replacement of* E_1 *by* E_2 *in* E.

Definition 17 If E_1 and E_2 are expansions, then E_1 *occurs in* E_2 iff either $E_1 = E_2$ of E_1 occurs in an element or $\mathbf{body}(E_2)$.

Definition 18 If E_1 and E_2 are expansions of the same operation, and E_1 occurs in an expansion E, then a *replacement* of E_1 by E_2 in E is either a direct replacement of E_1 by E_2 in E, or a direct replacement of E_1' by E_2' in E, where E_1 occurs in $E_1' \in \mathbf{body}(E)$ and E_2' is a replacement of E_1 by E_2 in E_1'.

A consequence of conditions (ii) and (iii) of Definition 9 and condition (ii) and (iii) of Definition 11 is that one expansion can occur in another only once. That is if E_1 occurs in E_2 and $E_1 \neq E_2$, then E_1 occurs in exactly one element of $\mathbf{body}(E_2)$. Consequently, if E_1 and E_2 are both replacements of E_3 by E_4 in some expansion E, then $E_1 = E_2$.

Definition 19 A *range* is a set that includes that special value $\otimes$ ("undefined"). If V is a range and $x, y \in V^*$, then x and y are *compatible* iff $|x| = |y|$ and for $1 \leq i \leq |x|$, either $x(i) = y(i)$ or one of $x(i)$ and $y(i)$ is $\otimes$.

Definition 20 If $\mathcal{Y}$ is a subset of $\mathcal{X}$ a *valuation for* Y *over a range* V is a function from Y to V.

The semantics of a particular CDL are embodied in a function which conforms to the conditions in the next definition. Given an operation expansion and the values for the variables that occur in it, this function computes a new expansion for the same operation, and values for its variables. In Prograph, for example, if the given operation expansion is basic and all the terminals of its operation have values other than $\otimes$, the function will produce an expansion, the body of which consists of an expansion of the diagram for the first case of the corresponding method. It will also copy the values from the terminals of the operation to the roots of this diagram expansion.

Definition 21 Let $\mathcal{E}$ be the set of all operation expansions of L, and $\mathcal{V}$ the set of all valuations of subsets of $\mathcal{X}$ over some range V. An *expansion function for* L *over* V is a partial function $\phi : 2^{\mathcal{X}} \times \mathcal{V} \times \mathcal{E} \rightarrow \mathcal{V} \times \mathcal{E}$ such that

(i) $\phi(Y, v, E)$ is undefined iff any of the following hold:

  (a) the domain of v is not $\mathbf{inputs}(E) \cap \mathbf{outputs}(E)$

  (b) $Y \cap \mathbf{vars}(E) \neq \varnothing$

(ii) if $\phi(Y, v, E_1) = (u, E_2)$, then

  (a) $\mathbf{oper}(E_2) = \mathbf{oper}(E_1)$;

  (b) the range of u is $\mathbf{outputs}(E_2)$

  (c) if $B \in \mathbf{body}(E_2)$ then either $B \in \mathbf{body}(E_1)$ or B is a basic expansion of a variant of some diagram in $\mathbf{body}(\mathbf{oper}(E_1))$ such that $\mathbf{vars}(B) \subseteq Y$;

  (d) if x is in the domains of both u and $v|_{output(E_1)}$, then either $v(x) = \otimes$ or $u(x) = v(x)$.

(iii) if $\phi(Y, v_1, E_1) = (u_1', E_1')$, $\phi(Y, v_2, E_2) = (u_2', E_2')$, $\mathbf{oper}(E_1)$ is a variant of $\mathbf{oper}(E_2)$ and $v_1(\mathbf{terms}(\mathbf{oper}(E_1))$ and $v_2(\mathbf{terms}(\mathbf{oper}(E_2))$ are compatible, then $u_1'(\mathbf{roots}(\mathbf{oper}(E_1'))$ and $u_2'(\mathbf{roots}(\mathbf{oper}(E_2'))$ are compatible.

Note that condition (iii) of Definition 21 enforces referential transparency, and would clearly have to be modified in languages with side-effects.

We can now define the semantics of CDLs by describing the execution of an operation. As we remarked above, the "execution" is a tree representing the state of execution of an operation. The root of the tree is an expansion of the operation in question. The execution is advanced by applying the expansion function ϕ to some operation expansion in the tree.

Definition 22 If O is an operation and ϕ is an expansion function for L over some range V, an *execution of* O *with respect to* ϕ is a pair (v, E) where v is a valuation of $\mathcal{X}$ over V and E is an expansion of O, such that

(i) *either* E is basic and $v(x) = \otimes$ for every $x \notin \mathbf{terms}(\mathbf{oper}(E))$,

(ii) *or* there exists an execution (v', E') of O such that

  (a) E is a replacement of E_1 by E_2 in E' where $\phi(\mathcal{X} - \mathbf{vars}(E'), v'|_{\mathbf{inputs}(E_1) \cup \mathbf{outputs}(E_1)}, E_1) = (u, E_2)$

  (b) $v(x) = \begin{cases} u(x) & if x \in \mathbf{outputs}(E_2) \\ \otimes & if x \in \mathcal{X} - \mathbf{vars}(E) \\ v'(x) & otherwise \end{cases}$

Note that conditions (i)(b) and (ii)(c) of Definition 21 together with the way the expansion function is applied in Definition 22, guarantee that the constraints on the occurrences of variables in the definitions of expansions are not violated.

2.4. Discussion

The semantics of a particular language is embodied in the expansion function that drives execution. First, the expansion function defines different kinds of data-driven execution of diagrams, depending on how it reacts to values available at the terminals of an operation. For example, it may initiate execution of one or more of the operation's diagrams, even if every terminal of the operation has the value $\otimes$ ("undefined"), resulting an "eager" execution. At the other extreme, it may wait until every terminal has a value other than $\otimes$, the conservative style of execution implemented in Prograph, for example.

Second, the expansion function embodies any non-dataflow control mechanisms that the language may have. When applied to an operation expansion, the function decides, on the basis of the current input and output values of the operation and of the diagrams involved in its execution, whether execution of some of the diagrams should be terminated and whether new diagram executions should be initiated. It also determines new values for inputs to the diagrams and outputs from the operation, subject to the condition that it cannot overwrite any value except $\otimes$. This condition, together with the syntactic restrictions on variable occurrences, ensures that CDLs have the single-assignment property.

The definition of execution does not include any notion of ordering: the expansion function may be applied anywhere in the tree. Hence it accommodates any kind of implementation, parallel or sequential. Note that a property of all CDLs, embodied in the function body in Def. 1, is the notion that an operation refers to a set of diagrams that will be involved in any execution of the operation. This provides the basis for procedure calling, and therefore recursion.

3. Modeling Control Flow

In this section we illustrate the definitions with examples of conditional branching, iteration and exception handling constructs.

3.1. LabVIEW: Conditional Branching and Iteration

We begin with an example illustrating both conditional branching and iteration. Figure 1 illustrates a LabVIEW operation P_1 with associated dataflow diagram, D_1, that sums the odd elements of an integer array. Comments beside the operations, terminals, and roots name the associated operations and variables. Table 1 provides the correspondence between items in the diagram and the preceding definitions. Note that the identifiers 0 and 1 in the body elements of P_4 define the roles of the two diagrams in the conditional construct, clarified in the execution below.

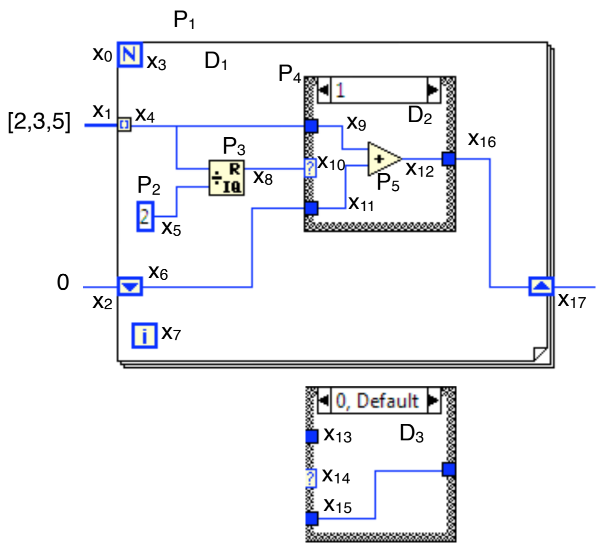

Figure 1. Example: LabVIEW Loop and Conditional

Table 1. The structure underlying the LabVIEW fragment in Figure 1

operation / diagram	body	roots	terms
P_1	$\{(0, D_1)\}$	(x_{17})	(x_0, x_1, x_2)
D_1	$\{P_2, P_3, P_4\}$	(x_3, x_4, x_6, x_7)	(x_{16})
P_2	$\varnothing$	(x_5)	$()$
P_3	$\varnothing$	(x_4)	(x_4, x_5)
P_4	$\{(1, D_2), (not - 1, D_3)\}$	(x_{16})	(x_4, x_8, x_6)
D_2	$\{P_5\}$	(x_9, x_{10}, x_{11})	(x_{12})
D_3	$\varnothing$	(x_{13}, x_{14}, x_{15})	(x_{15})
P_5	$\varnothing$	(x_{12})	(x_9, x_{10}, x_{11})

To illustrate the definition of execution, suppose operation P_1 is provided with input array value $[2, 3, 5]$ on terminal x_1 and 0 on terminal x_2. We construct a sequence $K_0, K_1, \ldots$ of executions, where $K_0 = (v_0, E_0)$, E_0 is the basic expansion $(P_1, \varnothing)$, $v_0(x_1) = [2, 3, 5]$, $v_0(x_2) = 0$ and $v_0(x) = \otimes$ for $x \neq x_1$, $x \neq x_2$. In the following, ϕ_L denotes the expansion function of LabVIEW. A full definition of ϕ_L is beyond the scope of this paper: however, the applications of ϕ_L in the example should give the reader the flavour of this definition.

In the following, to simplify the presentation, each valuation maps to $\otimes$ all variables for which it is not explicitly defined. Iteration, in this example, is controlled by a special *tunnel* terminal, x_1, which limits the iteration count to the length of the array it receives as input. The first iteration starts by replacing the expansion E_0 for P_1 with an expansion E_1 for P_1, the body of which contains an expansion of D_1. The new valuation sets the root x_4 of the diagram expansion to the value of array element 0, as indicated by the identifier of the diagram expansion.

$K_1 = (v_1, E_1)$ where
$\phi_L(\mathcal{X}\text{-}\{x_0, x_1, x_2, x_{17}\}, v_0|_{\{x_0,x_1,x_2,x_{17}\}}, E_0) = (u_1, E_1)$
$E_1 =$
$(P_1, \{(0, (x_3, x_4, x_6, x_7), (x_{16}), \{(P_2, \varnothing), (P_3, \varnothing), (P_4, \varnothing)\})\})$
$v_1(x_4) = u_1(x_4) = 2$ (the array element at index 0)
$v_1(x_6) = u_1(x_6) = v_1(x_2) = 0$
$v_1(x_7) = u_1(x_7) = 0$ (identifier of diagram expansion)

Note that E_1 is a replacement of E_0 by E_1 in E_0, so that K_1 is properly derived from K_0 according to Definition 22. We invite the reader to verify that each of the following steps is a proper application of this definition.

As we remarked earlier, Definition 22 does not prescribe the expansion to be replaced when generating a new execution from an existing one since the choice is specific to each language. In LabVIEW, operation P_2 is executed next. Accordingly, the next execution in the sequence is:

$K_2 = (v_2, E_2)$ where
$\phi_L(\mathcal{X} - \{x_0, \ldots, x_8, x_{16}, x_{17}\}, v_1|_{\{x_5\}}, (P_2, \varnothing)) = (u_2, (P_2, \varnothing))$
$E_2 = E_1$
$v_2(x_5) = u_2(x_5) = 2$
$v_2(x) = v_1(x)$ for $x \in \mathcal{X} - \{x_5\}$

In K_2, the structure of the execution is unchanged, but the valuation is extended to assign the value 2 to x_5. Odd elements are selected using modulo 2 calculated by operation P_3.

$K_3 = (v_3, E_3)$ where
$\phi_L(\mathcal{X} - \{x_0, \ldots, x_8, x_{16}, x_{17}\}, v_2|_{\{x_4,x_5,x_8\}}, (P_3, \varnothing))$
$= (u_3, (P_3, \varnothing))$
$E_3 = E_2$
$v_3(x_8) = u_3(x_8) = 0$
$v_3(x) = v_2(x)$ for $x \in \mathcal{X} - \{x_8\}$

Next, the output of P_3 is used to choose the appropriate diagram for P_4. Accordingly, the expansion $(P_4, \varnothing)$ in E_3 is replaced by an expansion for P_4, the body of which is an expansion of D_3.

$K_4 = (v_4, E_4)$ where
$\phi_L(\mathcal{X} - \{x_0, \ldots, x_8, x_{16}, x_{17}\}, v_3|_{\{x_4,x_6,x_8,x_{16}\}}, (P_4, \varnothing))$
$= (u_4, F)$
$E_4 =$
$(P_1, \{(0, (x_3, x_4, x_6, x_7), (x_{17}), \{(P_2, \varnothing), (P_3, \varnothing), F\})\})$
$F = (P_4, \{(0, (x_{15}), (x_{13}, x_{14}, x_{15}), \varnothing)\})$
$v_4(x_{13}) = u_4(x_{13}) = v_3(x_4) = 2$
$v_4(x_{14}) = u_4(x_{14}) = v_3(x_8) = 0$
$v_4(x_{15}) = u_4(x_{15}) = v_3(x_6) = 0$
$v_4(x) = v_3(x)$ for $x \in \mathcal{X} - \{x_{13}, x_{14}, x_{15}\}$

Since the body of D_3 is empty, the next step simply involves propagating the value of x_{15}, as follows.

$K_5 = (v_5, E_5)$ where
$\phi_L(\mathcal{X} - \{x_0, \ldots, x_8, x_{13}, x_{14}, x_{15}, x_{17}\}, v_4|_{\{x_4,x_6,x_8,x_{13},\ldots,x_{16}\}}, F)$
$= (u_5, F)$
$E_5 = E_4$
$v_5(x_{16}) = u_5(x_{16}) = v_4(x_{15}) = 0$
$v_5(x) = v_4(x)$ for $x \in \mathcal{X} - \{x_{13}, x_{14}, x_{15}\}$

Since no unexecuted operations remain the first iteration is complete.

The second iteration begins with replacing the current expansion of P_1 with a new expansion, the body of which is an expansion of a variant of D_1.

$K_6 = (v_6, E_6)$ where
$\phi_L(\mathcal{X} - \{x_0, \ldots, x_8, x_{13}, \ldots, x_{17}\}, v_0|_{\{x_0,\ldots,x_8,x_{16},x_{17}\}}, E_5)$
$= (u_6, E_6)$
$E_6 =$
$(P_1, \{(1, (x'_3, x'_4, x'_6, x'_7), (x'_{16}), \{(P'_2, \varnothing), (P'_3, \varnothing), (P'_4, \varnothing)\})\})$
$v_6(x'_4) = u_6(x'_4) = 3$ (array element at index 1)
$v_6(x'_6) = u_6(x'_6) = v_5(x_{16}) = 0$ (looped variable)
$v_6(x'_7) = u_6(x'_7) = 1$ (identifier of the diagram expansion)

Execution continues in this way until the identifier of the diagram expansion in the current expansion for P_1 is 3, exceeding the allowable index for the input array.

3.2. Vivid: Exception Handling

The second example illustrates how we can model exception handling. The Vivid Framework [9] implements an exception handling mechanism for CDLs in which an exception is handled by popping the call stack to find a handler that matches the type thrown. In Vivid, an exception handler is realised by a set of dataflow graphs executed in place of the exception-generating dataflow graph, to supply output values to allow execution to continue.

A simple example is shown in Figure 2. Executing the operation **Save** invokes its diagram D_1, causing the execution of **SaveNote** which calls the **AppendFile** primitive to append a string to a file. **AppendFile** throws a **WriteEx** exception upon write failure. This exception is caught by the **WriteEx** *alias* attached to the left side of the **SaveNote** operation. An alias is an operation that invokes an exception

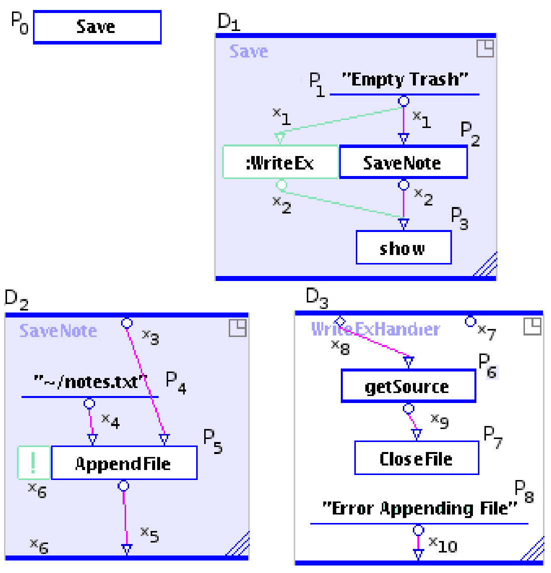

Figure 2. Example: Vivid Exception Handler

Table 2. Part of the structure underlying the Vivid code fragment in Figure 2

operation / diagram	body	roots	terms
P_0	$\{(1, D_1)\}$	$()$	$()$
D_1	$\{P_1, P_2, P_3\}$	$()$	$()$
P_2	$\{(1, D_2),$ $(\text{WriteEx1}, D_3)\}$	(x_2)	(x_1)
D_2	$\{P_4, P_5\}$	(x_3)	(x_6, x_5)
P_5	$\varnothing$	(x_4, x_3)	(x_6, x_5)
D_3	$\{P_6, P_7\}$	(x_8, x_7)	(x_{10})

handler when execution of the associated operation generates an exception of the type indicated by its label. The **WriteExHandler** handler receives the **WriteEx** exception on its first root ($\diamond$), closes the offending file, and outputs an error message which is transmitted to the **show** operation.

Table 2 shows part of the structure underlying the example in Figure 2. Note that P_5 has an implicit root, x_6, not part of the visual code, where the thrown exception appears, and D_2 has an implicit terminal x_6.

Complete execution requires 10 steps, however, for the sake of brevity, we will review only those that handle the exception.

Starting with a basic expansion for the operation **Save**, the execution proceeds with successive applications of ϕ_V, the Vivid expansion function, until operation **AppendFile** throws a **WriteEx** exception, at which point, the execution is as follows.

$K = (v, E)$ where
$E = (P_0, \{(1, (), (), \{(P_1, \varnothing), E_1, (P_3, \varnothing)\}\})$
$E_1 = (P_2, \{(1, (x_3), (x_6, x_5), \{(P_4, \varnothing), (P_5, \varnothing)\})\})$
$v(x_1) = v(x_3) =$"Empty Trash"
$v(x_4) =$" /notes.txt"
$v(x_6) = \langle\text{WriteEx}\rangle$ (an instance of WriteEx)

In accordance with Vivid exception handling semantics, execution will now proceed with diagram D_3 for the exception handler. To achieve this, the expansion function reacts to the presence of the exception value on the terminal x_6 of D_2, determining a new expansion for P_2 as follows.

$K' = (v', E')$ where
$E' = (P_0, \{(1, (), (), \{(P_1, \varnothing), E_1, (P_3, \varnothing)\}\})$
$\phi_V(\mathcal{X} - \{x_1, x_2, x_3, x_4, x_6\}, v|_{\{x_1, x_2, x_3, x_5, x_6\}}, E_1) = (u, E_2)$
$E_2 = (P_2, \{(\text{WriteEx1}, (x_7, x_8), (x_{10}), \{(P_6, \varnothing), (P_7, \varnothing)\})\})$
$v'(x_1) = v(x_1) =$"Empty Trash"
$v'(x_8) = u(x_8) = \langle\text{WriteEx}\rangle$
$v'(x_7) = u(x_7) =$"Empty Trash"

After replacing the diagram expansion for D_2 with one for D_3 the exception handler is executed to produce a substitute value for the output of P_2 from x_{10} to x_2 thus completing the handling of the exception.

4. Related Work

An early formalism for flow diagrams, reminiscent of dataflow, was presented as early as 1966 by Bohm and Jacopini who give a normalisation method to show that complex diagrams can be reduced by composition and iteration of two and three simpler diagram forms [2]. Early classical dataflow formalisms [16] represent programs as directed graphs where the nodes represent instructions and the arcs represent dependencies of token flow [8], or unbounded FIFO queues [15].

The semantics of pure dataflow diagrams is presented by Burza and Weide who transform them into Petri nets, to specify execution, and into finite automata, for expressing semantics [3]. In a similar approach Kavi *et al.* [17] have shown isomorphisms between dataflow graphs and Petri net graphs. They cite the ability of dataflow graphs to model parallel systems, their amenability to direct interpretation, and notational compactness (as compared to Petri nets) as benefits. More recently, using coloured Petri nets, Störrle defined the semantics of dataflow control structures and exceptions in UML 2.0 Activity Diagram notation [22]. Procedural abstraction in dataflow graphs has also been defined using networks of Petri nets [18].

Existing CDLs are described in a variety of ways, from informal to precise, from "by-example" to rigorous. For example, the semantics of Prograph are formally defined as function evaluation [21]; the structure and operation of LabVIEW are described in the informal but thorough language of patents [20]; and the semantics of VPP are described algorithmically [25]. Each such description is confined to a particular language, providing no useful characterisation of the common properties of CDLs.

5. Concluding Remarks

Many of the visual languages that have been proposed or implemented are controlled dataflow languages, in which acyclic dataflow diagrams are enclosed in control structures. We have presented a formal definition of this useful class of languages, capturing their common properties and providing a means for incorporating any control structures and any execution ordering. We have presented simple examples that show how the definitions capture iteration, conditional branching, procedure calling and exception handling. Space limitations preclude more complex examples that would illustrate the more subtle details.

In the definitions we have presented, the semantics of all control structures for a particular language are embodied in the omniscient and monolithic expansion function for that language. In future work, we will look at ways to modularise this function, breaking it up into separate functions for different control structures. This should allow us to factor out more of the common features of CDLs, such as iterative constructs.

Another useful consequence of the above definitions is that we now have a basis for proving some common properties of CDLs, such as those mentioned in Section 2. Future work will also be directed to realising language tools such as a CDL implementation upon which to design, test, build, and run new languages of this class.

References

[1] W. Ackerman. Data flow languages. *IEEE Computer*, 15(2):15–25, 1982.

[2] C. Böhm and G. Jacopini. Flow diagrams, Turing machines and languages with only two formation rules. *Communications of the ACM*, 9(5):366–371, 1966.

[3] P. D. Bruza and T. P. van der Weide. The Semantics of Data Flow Diagrams. In N. Prakash, editor, *Proc. of Int. Conf. on Management of Data*, Hyderabad, India, 1989.

[4] P. T. Cox and S. Gauvin. Exceptions in visual data flow programming languages. In *2003 Int. Conf. on Visual Languages and Computing, in Proc. 9th Int. Conf. on Distributed Multimedia Systems*, pages 360–367, 2003.

[5] P. T. Cox, F. Giles, and T. Pietrzykowski. Prograph: A step towards liberating programming from textual conditioning. In *IEEE Workshop on Visual Languages*, pages 150–156, 1989.

[6] A. L. Davis. Data Driven Nets – a class of maximally parallel, output-functional program schemata. Technical report, IRC Report, Burroughs, San Diego, CA, 1974.

[7] A. L. Davis and S. A. Lowder. A sample management application program in a graphical data-driven programming language. *Digest of Papers Compcon Spring*, pages 162–165, 1981.

[8] J. B. Dennis. First version of a data flow procedure language. In *Symposium on Programming*, pages 241–271, Institut de Programmation, University of Paris, Paris, France, 1974.

[9] S. Gauvin and T. Smedley. Vivid: A framework for creating visual programming languages. In *12th Int. Conf. on Intelligent and Adaptive Systems and Software Eng.*, pages 9–11, San Francisco, CA, 2003.

[10] E. Ghittori, M. Mosconi, and N. Porta. Designing and testing new programming constructs in a data flow VL. Technical report, Università di Pavia, Pavia, Italy, 1998.

[11] J. Good. VPLs and novice program comprehension: How do different languages compare? *IEEE: Symposium on Visual Languages*, pages 262–299, 1999.

[12] J. Herath, Y. Yamaguchi, N. Saito, and T. Yuba. Dataflow computing models, languages, and machines for intelligent computations. *IEEE Trans. Softw. Eng.*, 14(12):1805–1828, 1988.

[13] G. Johnson and R. Jennings. *LabVIEW Graphical Programming*. McGraw-Hill, New York, 2001.

[14] W. Johnston, J. Hanna, and R. Millar. Advances in dataflow programming languages. *ACM Comput. Surv.*, 36(1):1–34, 2004.

[15] G. Kahn. The semantics of a simple language for parallel programming. In *IFIP Congress 74*, pages 471–475, Amsterdam, The Netherlands, 1974.

[16] R. Karp and R. Miller. Properties of a model for parallel computations: Determinacy, termination, queueing. *SIAM*, 14:1390–1411, 1966.

[17] K. Kavi, B. Buckles, and U. Bhat. A formal definition of dataflow graph models. *IEEE Trans. on Comp.*, 35(11):940–948, 1986.

[18] A. Kiehn. *A Structuring Mechanism for Petri Nets*. PhD thesis, TU Munchen, 1989.

[19] T. Kimura and P. Mclain. Show and Tell user's manual. Technical Report WUCS-86-4, Washington University, St Louis, MO, 1986.

[20] J. L. Kodosky, J. J. Truchard, and J. E. MacCrisken. *A Graphical System for Modeling a process and associated method.* US Patent No: 4901222, filed 1986.

[21] Prograph International Inc. *Prograph CPX Reference Manual*, 1993.

[22] H. Störrle. Semantics and verification of data-flow in UML 2.0 activities. In *Proc. Intl. Ws. on Visual Languages and Formal Methods*, pages 38–52. IEEE Press, 2004.

[23] W. R. Sutherland. *The On-line Graphical Specification of Computer Procedures*. PhD thesis, MIT, 1966.

[24] S. Wail and D. Abramson. Can dataflow machines be programmed with an imperative language? *Advanced Topics in Dataflow Computing and Multithreading*, pages 229–265, 1995.

[25] R. Wang, L. Wang, C. Geng, and H. Zhou. The design of VPP software development environment. In *IEEE Instr and Meas. Techn. Conf.*, pages 403–408, Anchorage AL, 2002.

[26] Y. Zhang and B. Xu. A survey of semantic description frameworks for programming languages. *ACM SIGPLAN*, 39(3):14–30, 2004.