MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

Computation Structures Group Memo 81-1

Introduction to Data Flow Schemas

by

Jack B. Dennis
John B. Fosseen

September 1973

This research was supported by the National Science Foundation in part under research grant GJ-432, and in part under research grant GJ-34671.

Introduction

A data flow schema is a representation of the logical scheme of a program in a form in which the sequencing of function applications and tests, and the flow of values between applications are specified together. In a data flow schema, an application of a function, or a test of a predicate, is free to proceed as soon as the values required for its application are available. Since the availability of one computed value may simultaneously enable the application of several functions or predicates, concurrency of action is an inherent aspect of data flow schemas.

We present some basic properties of a class of data flow schemas which model the logical schemes of programs that compute with unstructured values. These schemas are a variation and extension of the "program graphs" studied by Rodriguez [12]. A related data flow model for computations on structured data has been described informally by Dennis [4]. Other related models include the work of Adams [1], Bährs [3], and Miller [7]. The material of the present paper is based largely on a thesis by Fosseen [5].

The principal results of this paper are that for free data flow schemas (defined in analogy with Paterson's free program schemas [10]), the equivalence of data links is decidable, and that any free schema may be transformed into an equivalent schema in which no two links are equivalent. This implies that for any free while schema (as defined in Ashcroft and Manna [2]), one can decide whether, for every interpretation, two variables are assigned the same history of values in any execution of the program.

An Example

Consider the following program expressed in an Algol-like notation:

```
begin
   t := x;  y := h(x)
   while p(w, t) do
   begin
      if q(y) then y := f(y)
      t := g(t)
   end
end
```

Variables w and x are input variables of the program and y is the output variable.

A data flow schema for this program is given in Figure 1. The arcs of a data flow schema should be thought of as channels through which tokens flow carrying

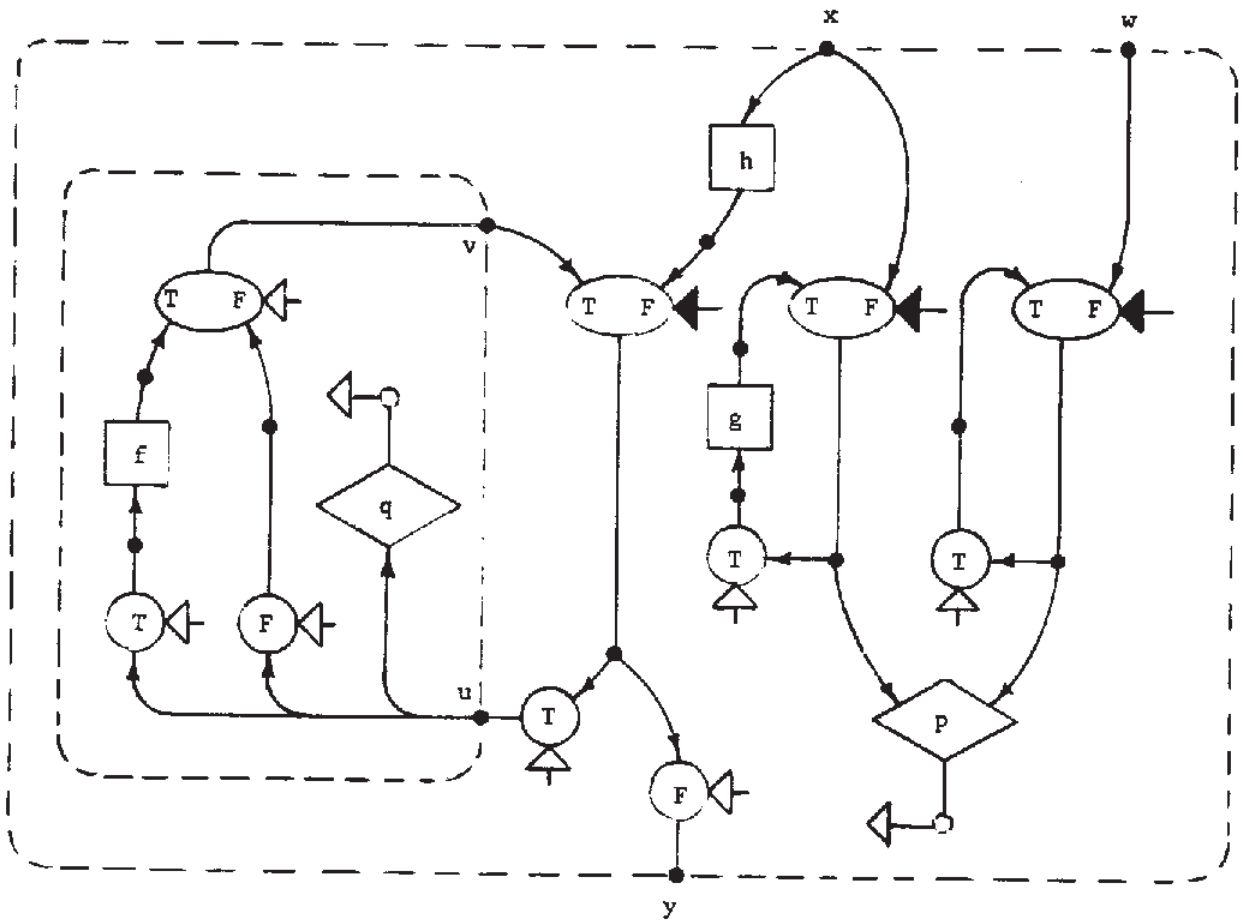


Figure 1. A data flow schema.

values between nodes of the schema. The two kinds of link nodes and the eight kinds of actor nodes are shown in Figure 2.

In general, a node is enabled if tokens with associated values are present on each input arc and no token is present on the output arc of the node. The execution or firing of a node absorbs the tokens on its input arcs and puts tokens on its output arcs. Link nodes are provided so values produced by one actor may be distributed to several actors. Control links pass truth values {true, false}; data links pass arbitrary values from the domain of an interpretation for the schema.

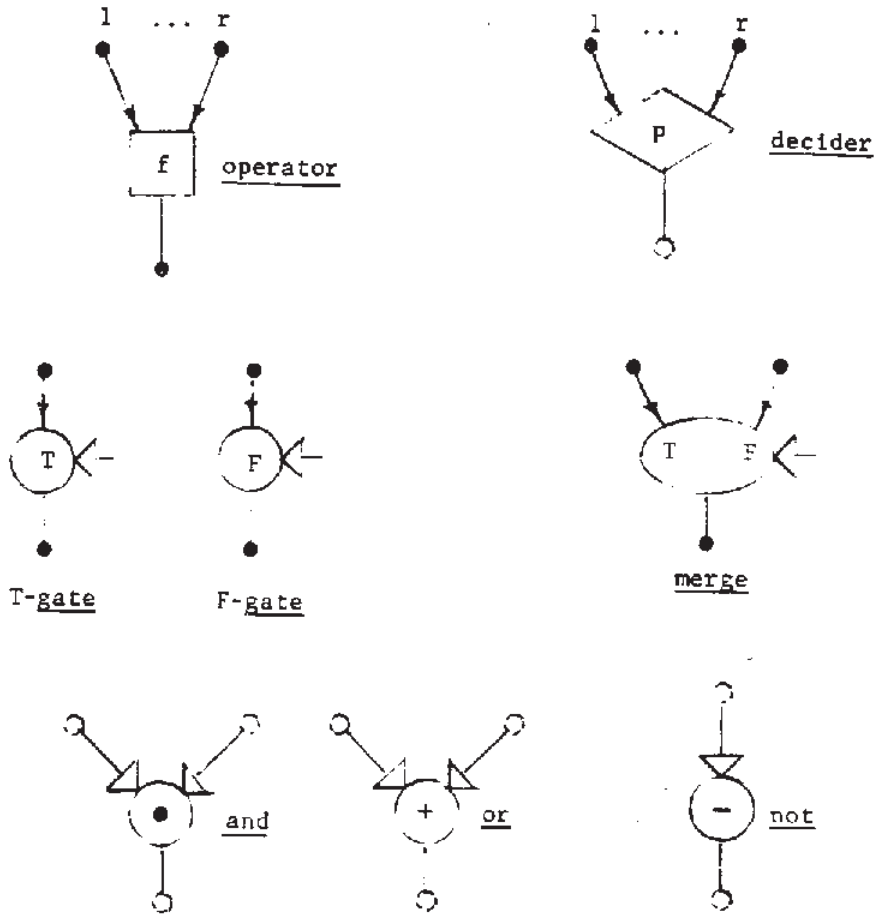
An operator applies to its input values the function associated by an interpretation with the function letter written inside the operator. A decider applies to its input values the predicate associated by an interpretation with the predicate letter written inside the decider. Data values are routed to operators and deciders by T-gate and F-gate actors. A T-gate, for example, passes a value on to its output arc if it receives a true value at its control input arc; the received data value is discarded if a false value is received. The merge node allows a truth value to control which of two sources supplies a data value to its output arc. If the control value false arrives at the control arc the merge passes on the value present or next to arrive at the F-input arc. A value present at the T-input arc is left undisturbed. The complementary action occurs for the control value true.

In Figure 1 the graph of the schema is simplified a bit by the convention that each control link is connected to all gate and merge nodes within the same dashed boundary.

In the representation of a program loop, merge nodes pass initial values into the body of the loop from their F-data input arcs. Values for subsequent repetitions of the body are recycled through the T-data input arcs. The arrowheads of the control arcs of these merge nodes are drawn solid to indicate that these control arcs hold tokens with false values in the initial condition of the schema. In this way the first cycle of the body is begun when values arrive at the data input links.

(a) links

(b) actors



Boolean actors

Figure 2. Node types for data flow schemas.

Definitions

Here we specify the representations for data flow schemas, and develop the means for studying their behavior as a model for computer programs.

An (m,n)-data flow schema S is a bipartite directed graph in which the two classes of nodes are called links and actors. The links (Figure 2a) are of two types: data links and control links. The arcs that go between actors and links are called data arcs or control arcs according to the type of link. An (m,n)-schema S has an ordered set IN(S) of m input links and an ordered set OUT(S) of n output links; IN(S) and OUT(S) need not be disjoint. No arc of S may terminate on any input link; exactly one arc must terminate on each link that is not an input link. At least one arc must originate at each link of S that is not an output link.

The types of actors are shown in Figure 2b.

1. operator: An operator has an ordered set of r input data arcs where $r \geq 0$, and a single output data arc. A function letter f selected from a set F of function letters is written inside the operator symbol. The set F may apply to several schemas; all operators bearing the same function letter must have the same number of input arcs.

2. decider: A decider has an ordered set of r input data arcs where $r \geq 1$, and a single output control arc. A predicate letter p selected from a set P of predicate letters is written inside the decider symbol. The set P may apply to several schemas; all deciders bearing the same predicate letter must have the same number of input arcs.

3. T-gate and F-gate nodes.

4. merge nodes.

5. Boolean actors: and, or, not

The last three actor types have input and output arcs as shown in Figure 2b.

A data flow schema S is an uninterpreted model for computation. Specifying an interpretation for S provides a complete representation of an algorithm.

An interpretation for a data flow schema with function letters in F and predicate letters in P is:

1. A domain $\mathcal{D}$ of values.
2. An assignment of a total function

$$\varphi_f: \mathcal{D}^r \to \mathcal{D}$$

to each $f \in F$, where each operator bearing the function letter f has r input arcs.

3. An assignment of a total predicate

$$\pi_p: \mathcal{D}^r \to \{\underline{\text{true}}, \underline{\text{false}}\}$$

to each $p \in P$, where each decider bearing the predicate letter p has r input arcs.

The activity of a data flow schema is represented by sequences of configurations. A configuration of schema S for an interpretation with domain $\mathcal{D}$ is:

1. An association of a value in $\mathcal{D}$ or the symbol null with each data arc of S.
2. An association of one of the symbols {true, false, null} with each control arc of S.

We depict a configuration of a schema by drawing a solid circle on each arc having a non-null value, and writing a value-denoting symbol beside. These circles are called data tokens, true tokens or false tokens according to the associated value.

A computation by S for a specified initial configuration γ_0 is a sequence of configurations $\gamma_0, \gamma_1, \cdots, \gamma_k, \gamma_{k+1}, \cdots$ where each γ_{i+1} is obtained from γ_i by the firing of some enabled node of S. The rules of firing for the two types of link node and four types of actors are given in Figure 3. Conditions for which a node is enabled are shown on the left (an enabled node is indicated by an asterisk). In addition, a necessary condition for any node to be enabled is that its output arc does not hold a token. Any node which is enabled in a configuration of S may be chosen to fire producing the change in configuration specified in the right part of the figure. The firing rules for the Boolean actors are similar to the rules for operators, and F-gates are identical to T-gates with negated truth values.

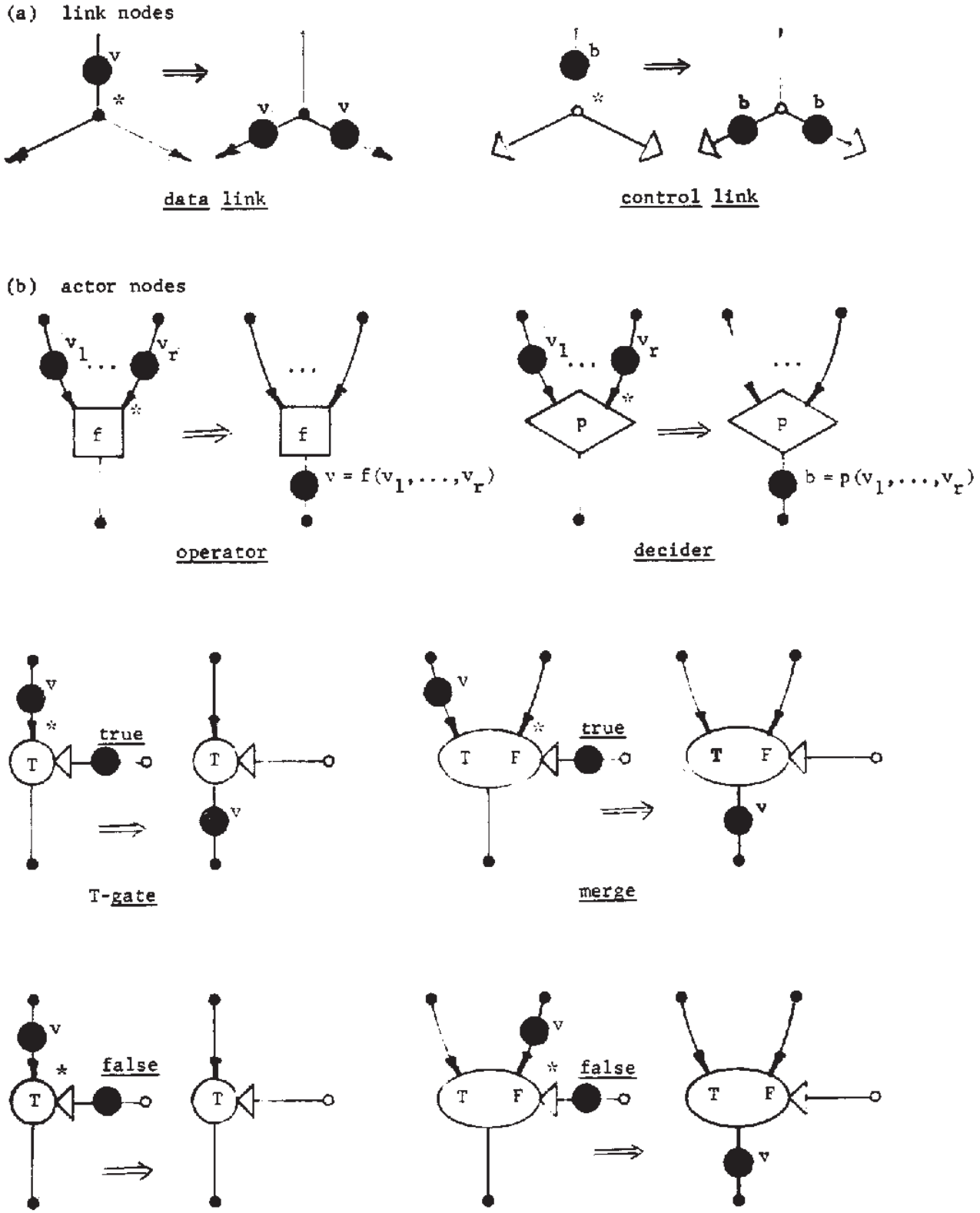


Figure 3. Firing rules for data flow schemas.

For a configuration γ of a schema S, the corresponding marking M is a function that associates an element of {mark, null} with each data arc of S and an element of {true, false, null} with each control arc of S. A marking is identical to the corresponding configuration except that values in the domain $\mathcal{D}$ are replaced by the single element mark.

The firing rules given in Figure 3 also specify the possible sequences of markings of a data flow schema. For any marking in which a decider is enabled, the decider may fire in either of two ways -- placing a true token or a false token on its output arc. An enabled node of any other type can fire in only one way -- determined by the tokens held by its input arcs. The possible sequences of markings of a data flow schema S are determined by the initial marking of S and are independent of the interpretation chosen for the function and predicate letters of S.

Let S be a data flow schema and let:

A be the set of operator nodes of S

D be the set of decider nodes of S

C be the set of gate, merge and Boolean actors of S

L be the set of link nodes of S

The alphabet of actions for S is the set

$$V = \{a \mid a \in A\} \cup \{d^T, d^F \mid d \in D\} \cup \{c \mid c \in C\} \cup \{\ell \mid \ell \in L\}$$

A control sequence of S for a specified initial marking M_0 is any sequence*

$$\tau : \mathcal{N} \to V$$

that defines a sequence of markings

$$M_0 \xrightarrow{\tau(0)} M_1 \xrightarrow{\tau(1)} M_2 \longrightarrow \cdots \xrightarrow{\tau(k-1)} M_k$$

such that:

1. The node specified by $\tau(i)$ is enabled for marking M_i and M_{i+1} is the result of firing this node.
2. If τ is a finite control sequence of k elements, then no node is enabled in the final marking M_k.

The symbols d^T and d^F are used to distinguish firings of a decider with true outcome and false outcome.

*$\mathcal{N} = \{0, 1, \ldots\}$ is the set of natural numbers.

We shall restrict our attention to data flow schemas that are well behaved in the sense that they produce one set of output values for each set of input values presented. Let S be an (m,n)-data flow schema and let M_0 be a marking of S in which no data arc holds a token. Let S' be S with added data arcs as shown in Figure 4. Schema S is <u>well behaved</u> for marking M_0 if and only if each finite control sequence starting from the marking in Figure 4a leaves S' with the marking shown in Figure 4b, in which the marking of S is again M_0.

A data flow schema is a system of interconnected elements that intercommunicate according to a strict discipline. Patil [11] and others [6] have studied such systems and their work shows that the actors of data flow schemas are determinate systems and that since a data flow schema is a well formed interconnection of determinate subsystems, any data flow schema is a determinate system. Therefore, a well behaved (m,n)-data flow schema S defines a function

$$\varphi_S: \mathcal{L}^m \to \mathcal{L}^n$$

Since well behaved schemas are functional, weak and strong equivalence may be defined in the usual manner for uninterpreted program schemas [9]: Two such schemas S and S' are <u>weakly equivalent</u> if and only if for every interpretation φ_S and $\varphi_{S'}$ yield the same value $\underline{y} \in \mathcal{L}^n$ for each $\underline{x} \in \mathcal{L}^m$ at which both are defined; S and S' are <u>strongly equivalent</u> if and only if

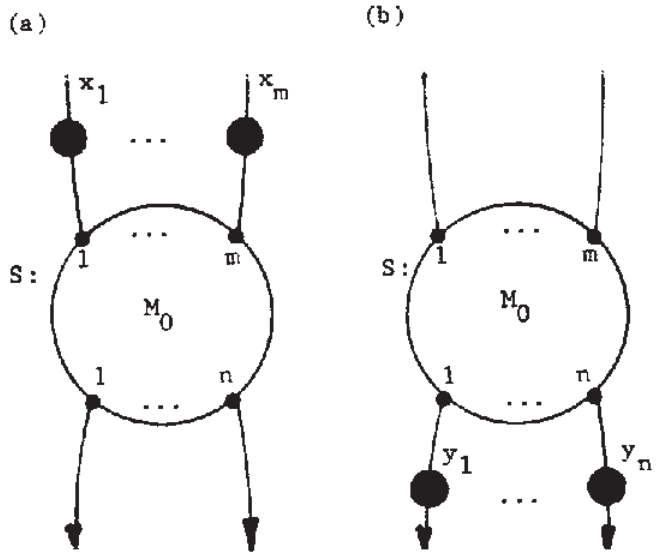


Figure 4. Definition of a well behaved schema.

S and S' are weakly equivalent and $\varphi_{S'}$ is defined for $\underline{x} \in \mathcal{E}^m$ if and only if $\varphi_{S'}$ is defined for $\underline{x}$.

In the remainder of this paper we will be concerned only with data flow schemas that have specified initial markings for which they are well behaved.

Well Formed Data Flow Schemas.

For the remainder of this paper we will study a class of well behaved data flow schemas constructed using formation rules analogous to the *if... then...else...* and *while...do...* constructions of programming languages. This class is characterized by the recursive definition of three kinds of well behaved data flow schemas: *conditional schemas*, *iteration schemas* and *well formed schemas*.

Well formed schemas: An (m, n)-*well formed schema* is any (m, n)-data flow schema formed by an acyclic composition of component data flow schemas, where each component is an operator, a conditional schema, or an iteration schema.

For the definitions of conditional and iteration schemas, we introduce some additional diagramming conventions. In Figure 5a, a broad arc represents a bundle of arcs connecting sets of links and actors. A large dot represents a set of data links; as required by our definitions each link of the set is driven from exactly one arc of the incident bundle and each link must be the origin of at least one arc in some emanating bundle. The *decision structure* shown in Figure 6b represents a set of deciders that provide input control values to an acyclic composition of Boolean actors having a single output control link. Both truth values *true* and *false* must be possible at the output link for suitably chosen outcomes of the deciders.

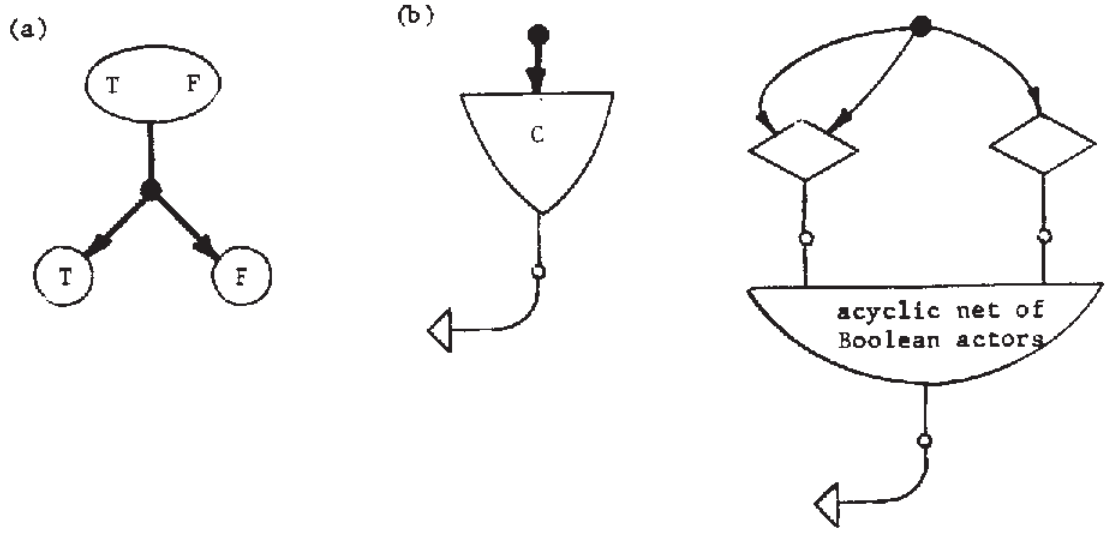


Figure 5. Diagramming conventions.

Conditional schemas: If P is a well formed (p, r)-schema, Q is a well formed (q, s)-schema, and C is a decision structure, then the data flow schema S shown in Figure 6 is a conditional schema. In the initial marking for S, component schemas P and Q are marked as required by their definitions; no tokens are present on arcs of S not contained in P or Q. A data flow schema is a conditional schema only if it is formed in this way.

A conditional schema performs the decision represented by its decision structure and selects either the true alternative P or the false alternative Q for execution to provide output values. The reader may verify that S is well behaved if P and Q are.

Iteration subschemas: if P is a well formed (p, r)-schema and C is a decision structure, then the data flow schema S shown in Figure 7 is an iteration schema. In the initial marking for S, component schema P is marked according to its definition, and the only additional tokens in S are a false token on the control input arc of each merge node not contained in P. A data flow schema is an iteration schema only if it is formed in this way.

An iteration schema uses its decision structure C to test some of its input values and presents some input values to its body P. The output values of P are tested and the cycle is repeated until some test yields a false decision, whereupon certain values are presented as the output of S and the schema is reset for subsequent reactivation. Evidently S is well behaved if P is well behaved.

It is easy to see how any program expressed as a sequence of statements of the types shown below can be represented as a well formed data flow schema. In these statements the x's are variable identifiers, and ⟨Boolean expression⟩ means any predicate of the form $p_k(x_{i_1}, \ldots, x_{i_n})$.

a) $x_j := x_k$

b) $x_j := f_k(x_{i_1}, \ldots, x_{i_n})$

c) if ⟨Boolean expression⟩
then ⟨program$_1$⟩ else ⟨program$_2$⟩

d) while ⟨Boolean expression⟩ do ⟨program⟩

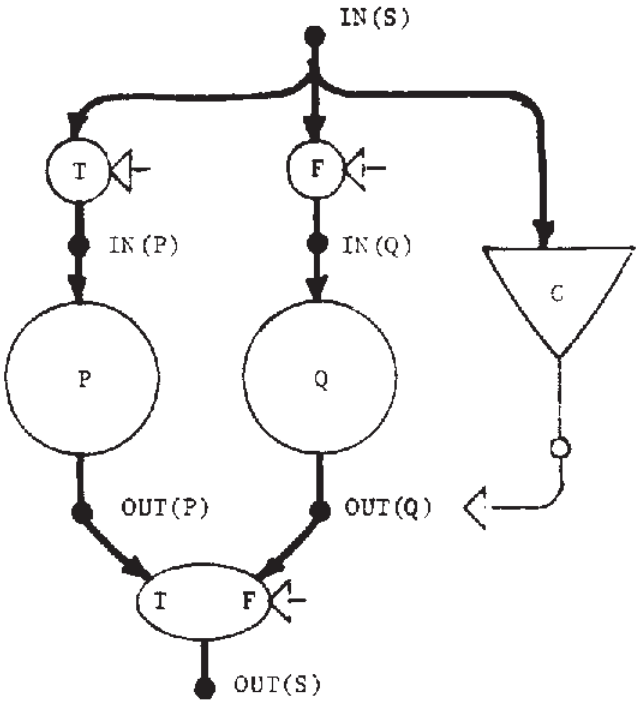


Figure 6. Form of a conditional schema.

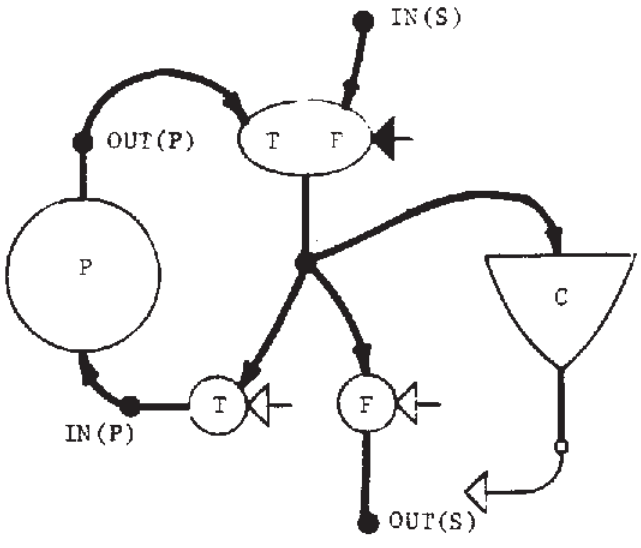


Figure 7. Form of an iteration schema.

Since Ashcroft and Manna [2] have shown that any "goto program" may be translated into a "while program" having only these statement types, well formed schemas are able to represent any "goto program".

Since we are concerned only with well formed schemas, we shall henceforth use the term schema to mean well formed data flow schema.

Data Dependence Graphs.

A data dependence graph (or dadep graph) of a schema is an explicit representation of the generation and testing of values for a particular control sequence of the schema.

Data dependence graphs(dadep graphs): A data dependence graph for an (m, n)-schema S is a finite, directed, acyclic, bipartite graph. The nodes are value nodes and action nodes. Each value node is labelled with a nonempty set of data links of S. There are m value nodes labelled by the distinct input links of S and having only emanating arcs; there are n value nodes labelled by the distinct output links of S and having exactly one incident arc. The action nodes of a dadep graph have the forms shown in Figure 8, where f is a function letter of some operator of S having r input links, and p is a predicate letter of some decider of S having r input links. Each action node in a dadep graph represents an application of some operator of S or a test by some decider of S.

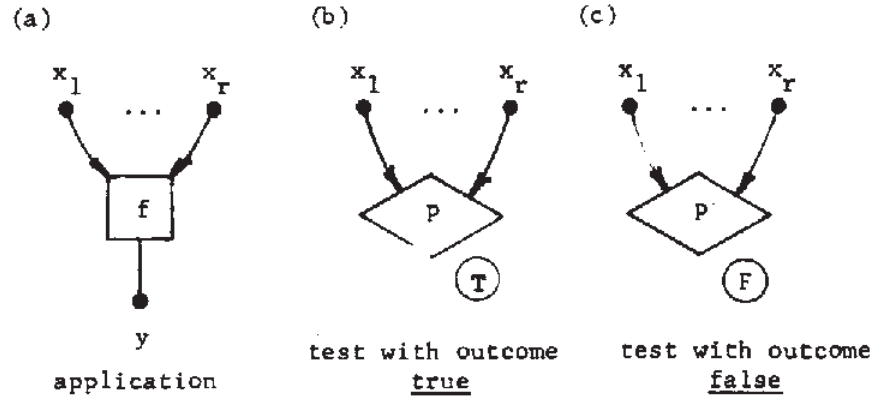


Figure 8. Action nodes for dadep graphs.

The class of dadep graphs for a schema S is denoted DADEPS(S) and is defined recursively according to the recursive construction rules for schemas:

1. If S is an acyclic composition of schemas $S_1, \ldots, S_k$, then $G \in$ DADEPS(S) if and only if G is a similar composition of $G_1, \ldots, G_k$ where $G_i \in$ DADEPS(S_i), i = 1, ..., k.
2. If C is a decision structure, a <u>decision</u> by C consists of a value node for each input link of C and a test for each decider of C, where each test carries a label which is one of the outcomes {<u>true</u>, <u>false</u>}. The <u>outcome</u> of a decision is the truth value obtained by its Boolean actors from the outcomes of its tests.
3. If S is a conditional schema consisting of alternatives P and Q and decision structure C, then $G \in$ DADEPS(S) if and only if G has the form shown in Figure 9 where D is a decision by C and $G' \in$ DADEPS(P) if the outcome of D is <u>true</u>, and $G' \in$ DADEPS(Q) if the outcome of D is <u>false</u>.
4. If S is an iteration schema consisting of body P and decision structure C, then $G \in$ DADEPS(S) if and only if G has the form shown in Figure 10 where $D_0, \ldots, D_{k-1}$ are decisions by C having <u>true</u> outcomes, D_k is a decision by C having a <u>false</u> outcome, and $G_i \in$ DADEPS(P), i = 0, ..., k - 1.

Note that a solitary application qualifies as a dadep graph according to case (1).

In Figure 11 three dadep graphs of the schema in Figure 1 are shown. (Labels corresponding to unnamed links in the schema have been omitted.)

The basic theorems for data flow schemas give conditions for equivalence of schemas that do not refer to the notion of interpretation. These conditions are stated in terms of two properties, similarity and consistency, which we define now.

<u>Similarity</u>: Let S and S' be (m, n)-schemas and let $G \in$ DADEPS(S), $G' \in$ DADEPS(S'). A <u>value node</u> v of G is <u>similar</u> to a <u>value node</u> v' of G' if and only if either

1. v has a label x and v' has a label x' where x is the i^{th} input link of S and x' is the i^{th} input link of S'.

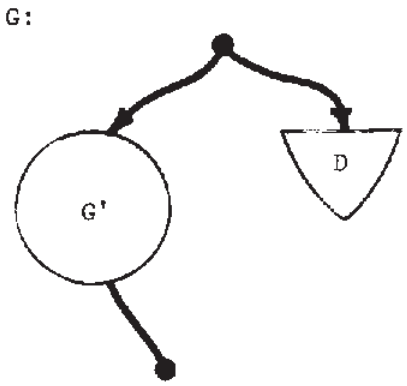


Figure 9. Dadep graph of a conditional schema.

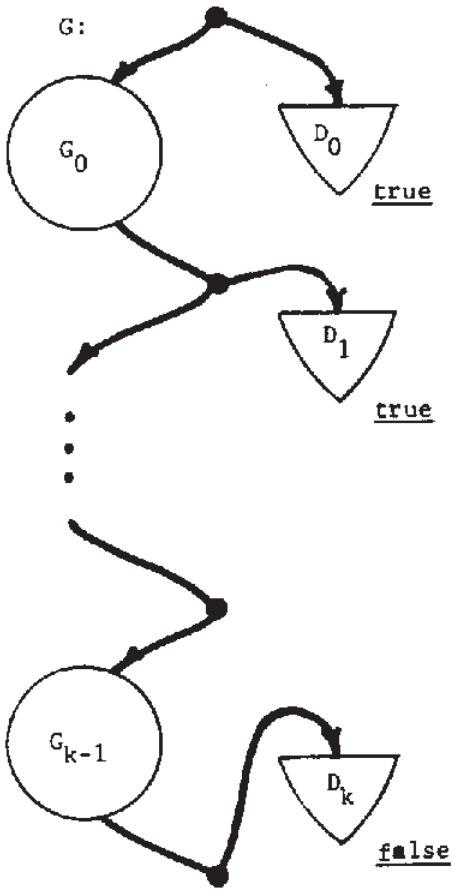


Figure 10. Dadep graph of an iteration schema.

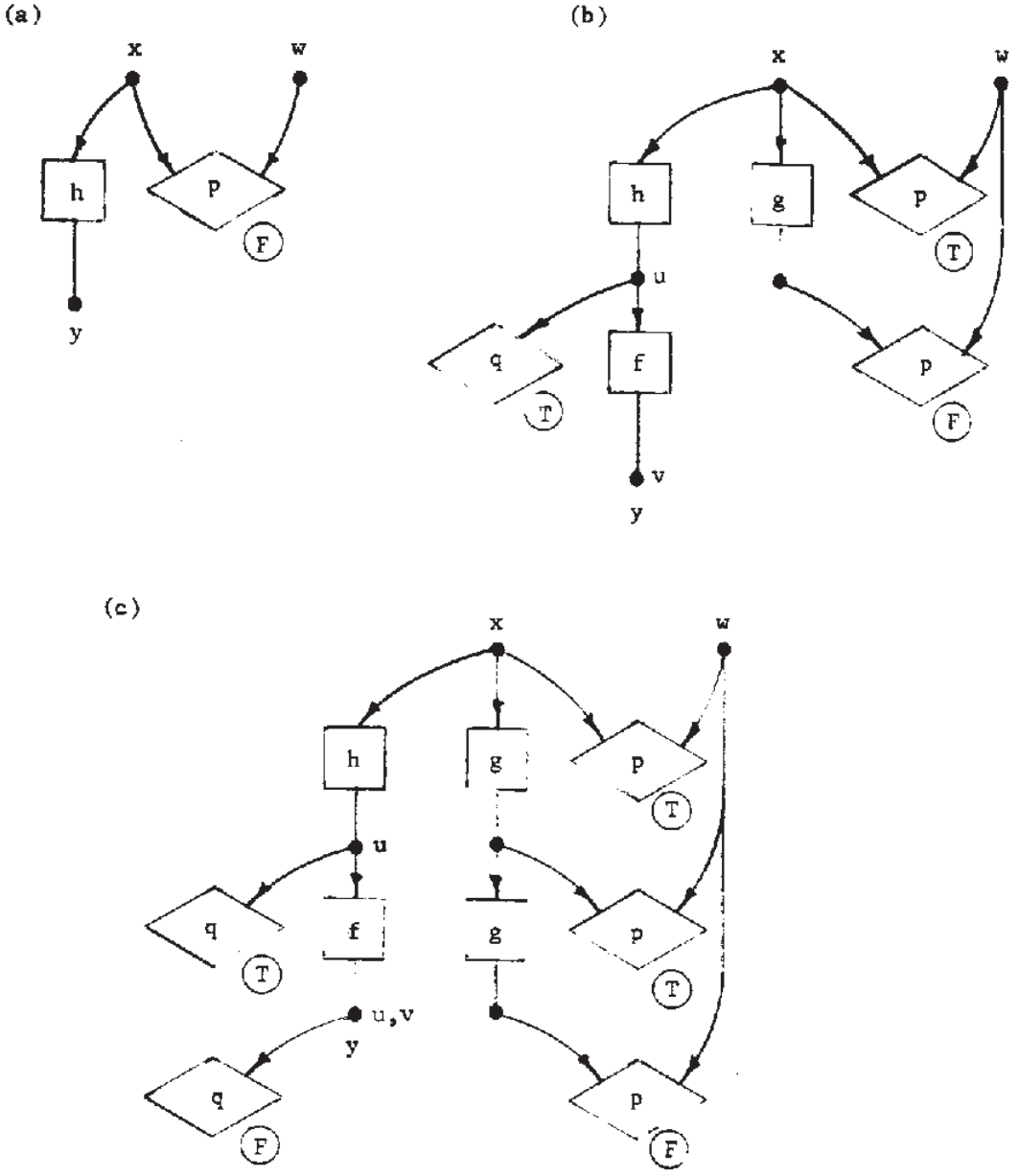


Figure 11. Dadep graphs for a data flow schema.

2. v is the output value node of an application a, v' is the output value node of an application a', applications a and a' bear the same function letter, and corresponding input value nodes of a and a' are similar.

Dadep graphs G and G' are similar if and only if the value nodes v in G and v' in G' labelled with corresponding output links of S and S' are similar. A test t in G and a test t' in G' are similar if and only if t and t' bear the same predicate letter and corresponding input value nodes of t and t' are similar.

Consistency: Let S and S' be (m, n)-schemas and let $G \in$ DADEPS(S), $G' \in$ DADEPS(S'). A test t in G and a test t' in G' are inconsistent if and only if t and t' are similar, but the outcome of one is true and the other is false. Otherwise t and t' are consistent. Dadep graphs G and G' are inconsistent if and only if there is a test t in G and a test t' in G' such that t and t' are inconsistent. Otherwise G and G' are consistent.

Basic Theorems.

The following three theorems are familiar basic results of schematology expressed in the terminology of data flow schemas. The proofs make use of the notion of free interpretations [9] and are only outlined here.

Theorem 1: Let S be an (m, n)-schema and let G_1, $G_2 \in$ DADEPS(S). Then G_1 and G_2 are similar whenever G_1 and G_2 are consistent.

Proof: If G_1 and G_2 were consistent but not similar, we could construct an interpretation for S and choose input values such that computations corresponding to G_1 and G_2 yield differing output values. But this is impossible because any schema is well behaved and therefore functional.

Theorem 2: Let S and S' be (m, n)-schemas. Then S and S' are weakly equivalent if and only if whenever $G \in$ DADEPS(S) and $G' \in$ DADEPS(S'), then G and G' are similar if G and G' are consistent.

Proof (if): Suppose the condition is satisfied but S and S' are not weakly equivalent. There must be computations by S and S' having dadep graphs G and G' such that G and G' are not similar. By the condition G and G' are inconsistent; hence there exist tests in G and G' that are similar but have different outcomes, which is impossible.

(only if): Let $G \in DADEPS(S)$, $G' \in DADEPS(S')$ and suppose G and G' are consistent but not similar. We can construct an interpretation such that test outcomes specified in G and G' are satisfied. For a free interpretation of the function letters, the output values will not be the same and thus S and S' are not weakly equivalent.

Theorem 3: Let S and S' be (m, n)-schemas. Then S and S' are strongly equivalent ($S \equiv S'$) if and only if S and S' are weakly equivalent ($S \simeq S'$), and for each $G \in DADEPS(S)$ there is a $G' \in DADEPS(S')$ such that G and G' are consistent, and for each $G' \in DADEPS(S')$ there is a $G \in DADEPS(S)$ such that G and G' are consistent.

Proof (if): Suppose the condition holds but S and S' are not strongly equivalent. Since $S \simeq S'$ there is some choice of interpretation and input m-tuple for which S has a computation but S' does not (or vice versa). Thus for some $G \in DADEPS(S)$ there is no $G' \in DADEPS(S')$ such that G and G' are consistent -- contradiction.

(only if): Suppose $S \equiv S'$ (and hence $S \simeq S'$) but there exists $G \in DADEPS(S)$ such that no $G' \in DADEPS(S')$ is consistent with G. Since we can choose an interpretation and input m-tuple such that S has a computation described by G, but S' has no (completed) computation, S' and S cannot be strongly equivalent.

Equivalence of Data Links

We shall show how one can transform any free schema into a strongly equivalent free schema such that any pair of data links may be tested for equivalence.

Definition: Let x and x' be any two data links of an (m, n)-schema. Then x and x' are equivalent if and only if, for each interpretation for S and each m-tuple of input values, each execution sequence has exactly k firings of link x and k firings of link x', for some $k \geq 0$, and

$$v_i = v'_i, \qquad i = 1, \ldots, k$$

where $(v_1, \ldots, v_k)$ and $(v'_1, \ldots, v'_k)$ are the sequences of values passed by the firings of x and x'.

We cannot hope to effectively test the equivalence of data links in general since such a test would provide a solution to the general equivalence problem for schemas, which is known to be unsolvable [10]. The procedure we have developed applies to free schemas where freedom is defined for data flow schemas as in Paterson's work: No dadep graph of a free schema may contain similar tests. It follows that every dadep graph of a free schema describes a nonempty set of computations.

It is convenient to introduce some additional terminology: If S is a (well formed) schema then BOUND(S) = IN(S) $\cup$ OUT(S) and contains the boundary links of S. Those data links of S not in BOUND(S) are internal links of S. The set of main links MAIN(S) contains all data links of S that are not internal links of any conditional or iteration schema contained within S. We say that schema R is a subschema of S if and only if S contains R and BOUND(R) $\subseteq$ MAIN(S). Note that the only data links of a schema S that are not main links of some schema contained in S are the output links of iteration schema merge nodes.

The order of a schema S is the number of levels of nested subschemas in S: A schema having no conditional or iteration subschemas is of order 0. The order of a conditional schema is one greater than its highest order alternative. The order of an iteration schema is one greater than the order of its body. The order of a schema in general is equal to the order of its highest order conditional or iteration subschema.

Our algorithm for deciding link equivalence consists of four procedures presented in the proofs of Theorems 7 and 8. The algorithm is recursive in the order of the schema S being tested, and identifies all equivalences that hold among the boundary links of S (and among the boundary links of each schema contained within S). Each of the four procedures corresponds to one of the forms S may have as a well formed schema:

1. S contains only operators -- Theorem 7; Theorem 8, part 1.
2. S is a conditional schema -- Theorem 8, part 2.
3. S is an iteration schema -- Theorem 8, part 3.
4. S is a composition of schemas -- Theorem 8, part 4.

In each case the input to the procedure is the schema S and some partition P of IN(S); the result is a partition P' of BOUND(S) such that: If P divides IN(S) into maximal sets of mutually equivalent links, then P' divides BOUND(S) into maximal sets of equivalent links.

Before presenting the details let us show by several examples some of the problems any decision procedure must deal with. In Figure 12, one cannot determine whether links y and v are equivalent just from the structure of conditional schema C and knowledge of whether links u and v are equivalent: If operators a and b have the same function letter, as shown in the figure, then links y and v are equivalent; but if these operators have distinct function letters, links y and v are not equivalent. Thus our procedure cannot work unless the schemas to be analyzed are first transformed into a restricted form in which such situations as the one in Figure 12 cannot occur. Rules for performing such a transformation on any schema are validated by Lemma 1 and Theorem 5 below. The restricted form of schema that results from applying the transformation rules is called a <u>modified schema</u>. In a modified schema no output link of any <u>gate</u> or <u>merge</u> node can be such that every value passed is produced by operators bearing the same function letter. The schema in Figure 12 is clearly not a modified schema, since link v does not satisfy the condition.

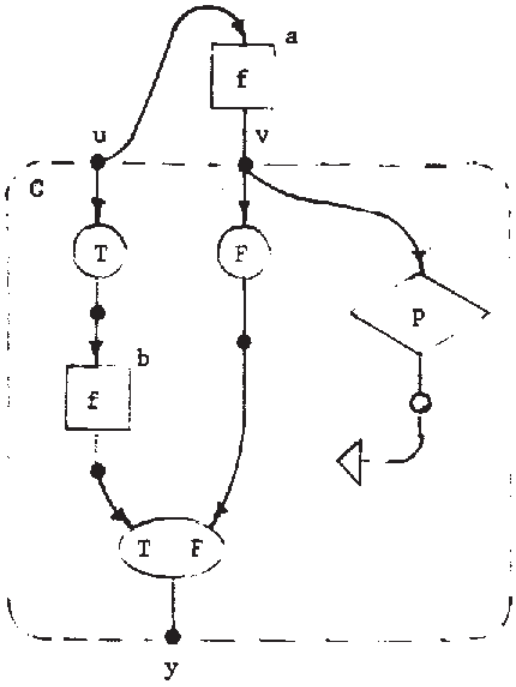


Figure 12. A schema requiring transformation.

One might think it sufficient for our procedures to construct the partition of the output links OUT(S) into maximal sets of equivalent links. The example in Figure 13 shows why it is necessary to partition BOUND(S) so that information about equivalences between input and output links of S is kept. The analysis of schema Q must show that link x is equivalent to links u and v if and only if these input links are equivalent to each other. Likewise, analysis of schema R must show that link y is equivalent to links u and v if and only if u and v are equivalent. Otherwise we would not be able to discover in the analysis of schema S that equivalence of links u and v implies equivalence of links x and y. The consequences of assuming the two possible partitions of IN(S) are as follows:

Partition of IN(S):	{u}{v}	{u,v}
Partition of BOUND(Q):	{u}{v}{x}	{u,v,x}
Partition of BOUND(R):	{u}{v}{y}	{u,v,y}
Partition of BOUND(S):	{u}{v}{x}{y}	{u,v,x,y}

The recognition of such equivalences is essential to the validity of Theorem 8, part 4.

Figure 14 illustrates the problem of analyzing iteration schemas. Output links y and z are equivalent if and only if links u and w are equivalent, and link x is equivalent to links u and w if and only if u and w are equivalent. The required correspondence between the possible partitions of IN(S) and the partitions of BOUND(S) is as follows:

Partition of IN(S)	Partition of BOUND(S)
{u}{v}{w}	{u}{v}{w}{x}{y}{z}
{u,v}{w}	{u,v}{w}{x}{y}{z}
{u,w}{v}	{u,w,x}{v}{y}{z}
{u}{v,w}	{v,w}{u}{x}{y,z}
{u,v,w}	{u,v,w,x}{y,z}

The procedure given in Theorem 8, part 3, is designed to identify all such equivalences in iteration schemas.

First we establish a necessary condition for the equivalence of two data links in a free schema S. This result lets us conclude that two links cannot be equivalent if they are internal links of disjoint schemas within S.

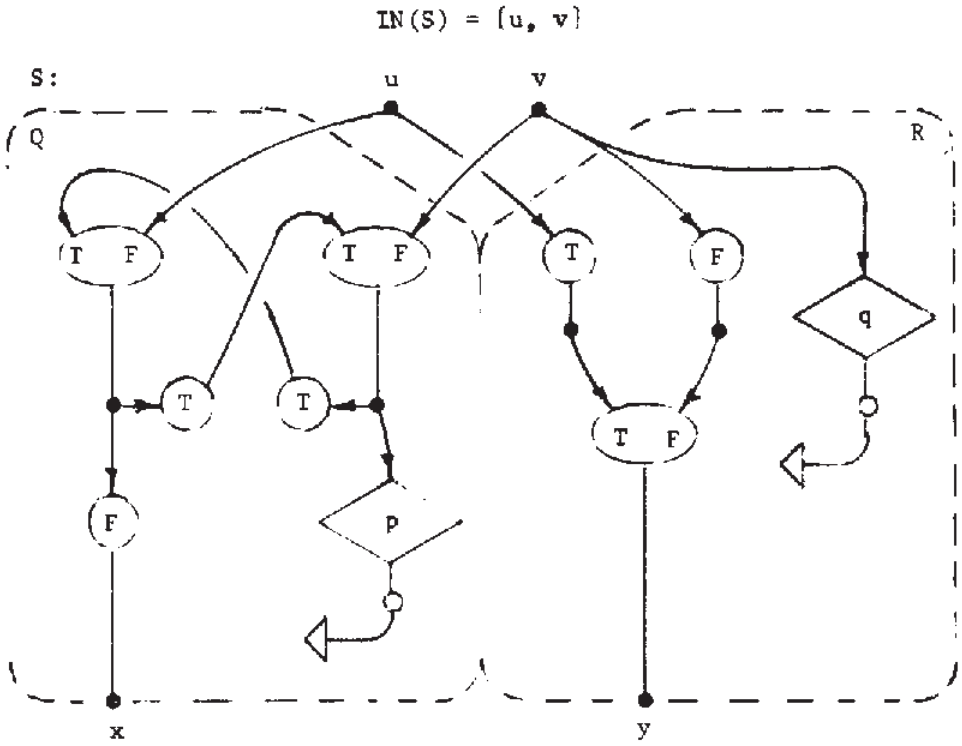


Figure 13. Link equivalence in a composition of two schemas.

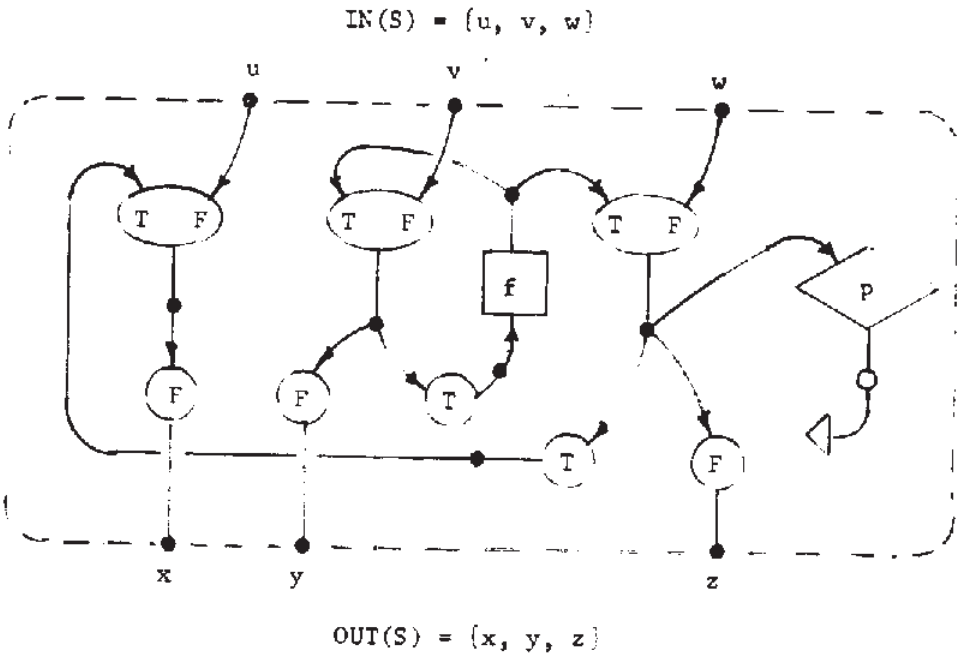


Figure 14. Link equivalence in an iteration schema.

Theorem 4: Let p_1 and p_2 be data links of a free schema S. Then $p_1 \equiv p_2$ only if either:

1. $\{p_1, p_2\} \subseteq$ MAIN(R) where schema R is contained in (and possibly equal to) S.
2. Links p_1 and p_2 are output links of merge nodes of the same iteration schema in S.

Proof: Let R be the smallest schema in S that contains both link p_1 and link p_2. Denote by M(R) the set of output links of the merge nodes of R if R is an iteration schema, and the empty set if R is not an iteration schema.

Suppose $p_1 \notin$ MAIN(R) $\cup$ M(R). Then either $p_1 \in$ MAIN(P) or $p_1 \in$ M(P) where P is some schema contained in but distinct from R. It cannot be that p_2 is a link of schema P, for then P would be a smaller schema than R that contains both p_1 and p_2. We consider three cases:

1. $p_1 \in$ MAIN(P) where P is one alternative of some conditional schema Q. Since S is free, either 0 or 1 tokens may be passed over link p_1 in an execution of Q. This choice is independent of the number of tokens passed over link p_2 unless p_2 is a link of the opposite alternative of conditional schema Q. But in that case the number of tokens passed over p_2 is nonzero only if the number passed over p_1 is zero. Thus $p_1 \not\equiv p_2$.

2. $p_1 \in$ MAIN(P) where P is the body of an iteration schema Q. Since S is free, the number of tokens passed over link p_1 may be 0, 1, 2... independent of the number of tokens passed by p_2, hence $p_1 \not\equiv p_2$.

3. $p_1 \in$ M(P) where P is an iteration schema. Since S is free the number of tokens passed by link p_1 may be 1, 2, 3, ... independent of the number of tokens passed by p_2, hence $p_1 \not\equiv p_2$.

Similarly, assuming $p_2 \notin$ MAIN(R) $\cup$ M(R) leads to the conclusion $p_1 \not\equiv p_2$. Therefore,

$$\{p_1, p_2\} \subseteq \text{MAIN(R)} \cup \text{M(R)}$$

It cannot be that $p_1 \in$ MAIN(R) and $p_2 \in$ M(R), because the number of tokens passed by link p_2 would depend on decisions made by R whereas the number of tokens passed by link p_1 would not.

Transformation Into Modified Form

To apply our method for testing equivalence of links, we must transform the given schema into a modified form such that information about equivalence of links in IN(R) is sufficient to determine equivalence of links in OUT(R).

Modified schemas: By an empty data path in a schema we mean a path containing only data links, gate nodes and merge nodes. A data link y in a schema S is admissible if y meets these conditions:

1. Link y is the output link of a gate node of S, or the output link of a conditional schema within S.
2. Each operator with output link x, such that S contains an empty data path from x to y, has the same function letter.
3. Schema S contains no empty data path from x to y for any $x \in IN(S)$.

The depth of an admissible link in S is the minimum number of operators on any path from an input link of S to link y. A schema is a modified schema if and only if it has no admissible links.

A schema is put into modified form by moving operators past gate and merge nodes. This is done by repeated use of Transformation T given below:

Transformation T: Let S be a schema with admissible link y. Construct schema S' as follows:

Step 1. Let $A = \{a_1, \ldots, a_m\}$ contain the operators of S such that S has an empty data path from each x_i to y where x_i is the output link of a_i. Let the input links of a_i be $u_{i1}, \ldots, u_{ir}$.

Step 2. Let Z be the subgraph of S that contains each empty data path from each x_i to y. If Z contains the output link w of an iteration schema merge node, then include in Z each gate node having w as its input data link. The input links of Z are $IN(Z) = \{x_1, \ldots, x_m\}$. The output links OUT(Z) include each data link of Z which is an input link of some actor in S but not in Z.

Step 3. For each $y \in OUT(Z)$ that is an input link of some decider d in the decision structure of an iteration schema, perform the transformation shown in Figure 15. The new merge node m' is not part of Z, hence w_t and w_f become output links of Z if they are not already. Link y is no longer an output link of Z unless it is an input link of actors other than d that are not in Z.

Step 4: Schema S' is the result of rearranging S using r copies of Z as shown in Figure 16.

Figure 17 illustrates application of transformation T to admissible link w of an iteration schema. In Figure 17a empty paths lead to link w from links x and v, and links x and v are output links of operators a and b which have the same function letter. The subgraph Z for admissible link w is indicated in Figure 17b and consists of the merge and gate nodes and links x, v, y, w and z. Since link y is an input link of decider d, a copy of the merge node is brought outside Z as in Figure 17c. Since operators a and b have one input link, just one copy of Z is required, and the transformed schema is shown in Figure 17d.

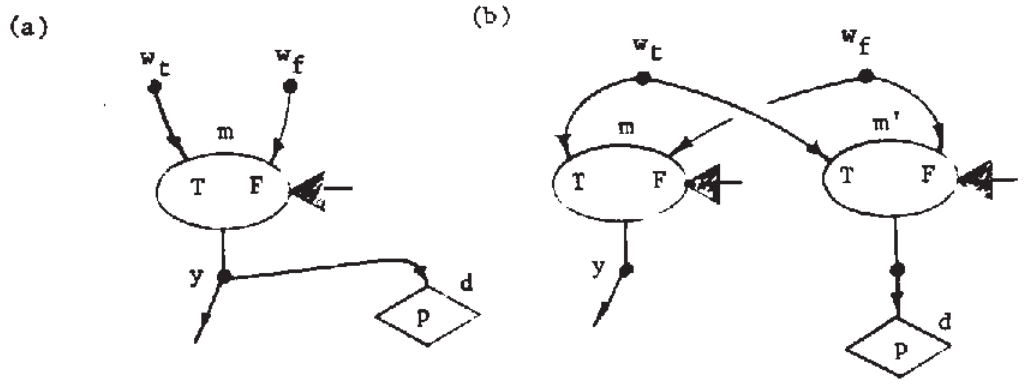
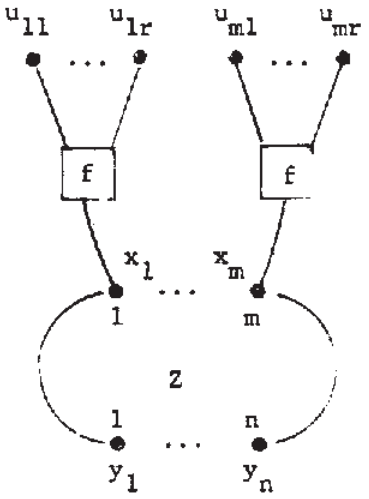


Figure 15. Step 3 of transformation T.

(a)



(b)

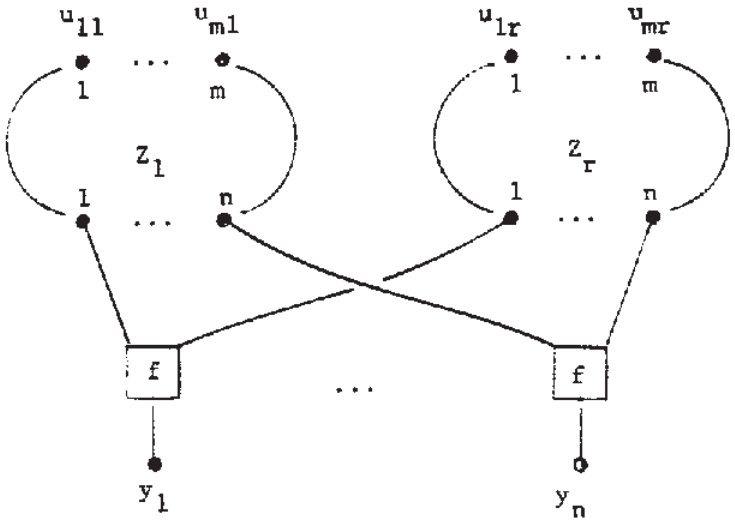


Figure 16. Step 4 of transformation T.

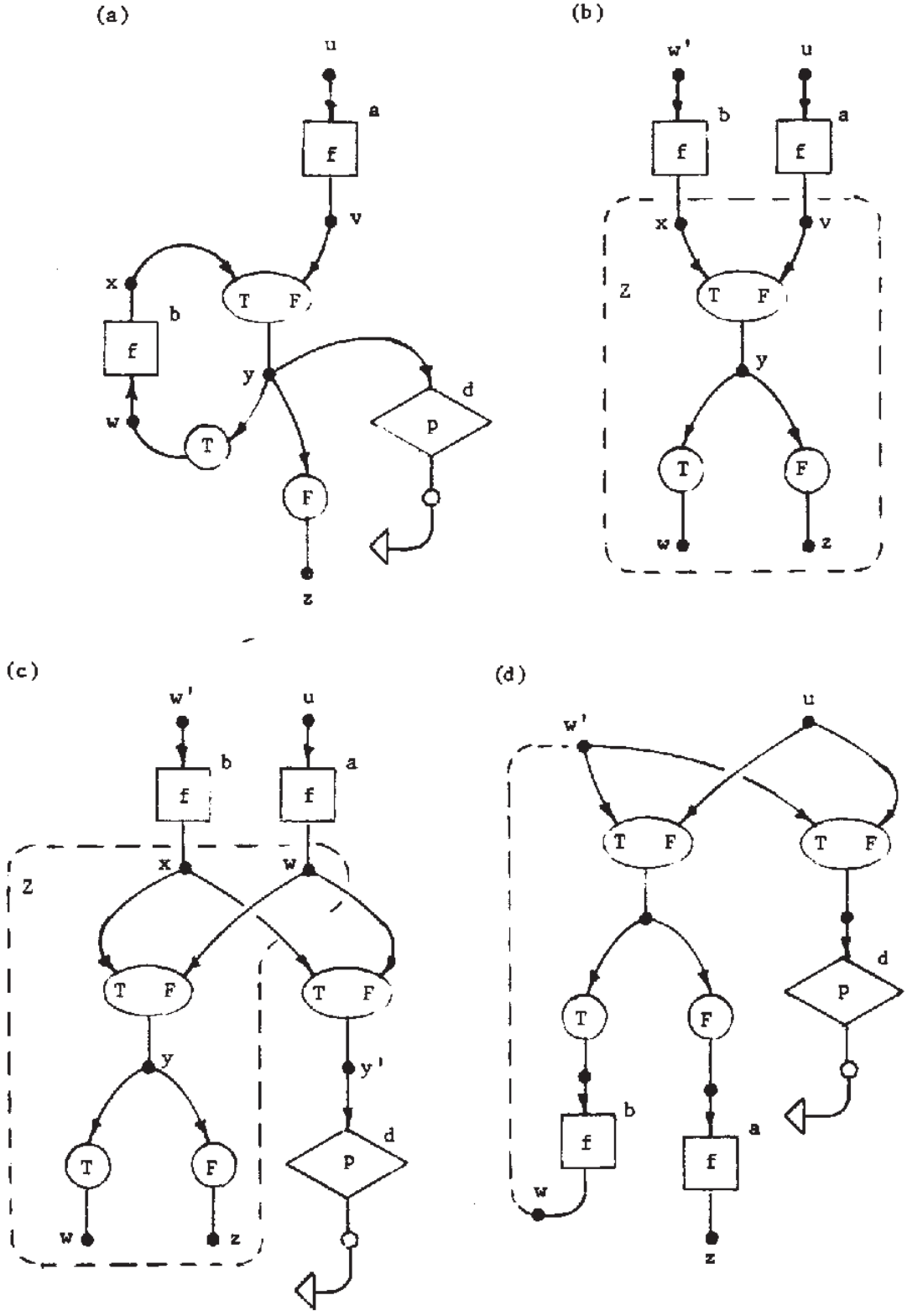


Figure 17. Application of transformation T for admissible link w.

Lemma 1: If S' is the result of applying transformation T to schema S, then S' is a (well formed) schema, and S' is strongly equivalent to S. Furthermore, S' is free if S is free.

Proof: 1. (S' is well formed.) The association of decision structures and gate and merge nodes with conditional and iteration schemas in S' is exactly as in S. In fact, the only change is that operators occur in S' where they are not present in S and vice versa. There are two places where introduction of an operator can make a schema not well formed:

a. Between a merge node and a T-gate or F-gate of an iteration schema.

b. Between a merge node and the decision structure of an iteration schema.

But the construction of Z in Step 2 and the modification of S in Step 3 ensure that no output link of an iteration schema merge node is an output link of Z. Hence S' is well formed.

2. (S' is strongly equivalent to S.) It is clear from Figures 12 and 13 that transformation T preserves functionality and therefore S' is strongly equivalent to S.

3. (S' is free if S is free.) The decision structures of S' correspond one-to-one with decision structures of S, and perform identical decisions.

Theorem 5: If S is a schema, one can construct a modified schema S' such that $S' \equiv S$. Furthermore, S' is free if S is free.

Proof: We show how the required schema S' may be obtained by repeated application of Transformation T until no further application is possible.

Step 1. Let $S_0 = S$; set $k = 0$.

Step 2. Let N_k contain the admissible links of S_i. If N_k is empty, then $S' = S_k$ is the transformed schema.

Step 3. Let $y \in N_k$ be some admissible link having greatest depth in S_k. Use Transformation T to convert S_k into S_{k+1}.

Step 4. Set $k = k + 1$ and return to Step 2.

If this procedure terminates, the resulting schema will have no admissible links. If S is a free schema, then Lemma 1 shows that if the procedure terminates, S' is well formed, free, and strongly equivalent to S. The procedure terminates because each application of Transformation T reduces by one the number of admissible links having greatest depth in S_k, and creates a bounded number of admissible links of lesser depth. Hence S' is a modified schema.

A Necessary Condition for Equivalence of Main Links

Next we show that in testing the equivalence of main links of a schema R containing a conditional or iteration subschema R', we need only be concerned with identifying those links in OUT(R') that are equivalent to one another, or are equivalent to links in IN(R').

The union of a conditional schema: Let R be a conditional schema within schema S and let P and Q be the true and false alternative schemas of R. The union of R is the schema R' obtained by deleting the gate and merge nodes of R, merging the output node of each gate with its data input node, and taking OUT(R') = OUT(P) $\cup$ OUT(Q).

Theorem 6: Let R be a schema within a free, modified schema S (possibly R = S), and let R' be a conditional or iteration subschema of R. If $y \in OUT(R')$ then either

1. $y \equiv x$ for some $x \in IN(R')$

or

2. If $z \in MAIN(R)$ then $z \equiv y$ only if R contains an empty data path from y to z.

Proof: We consider two cases according as R' is an iteration schema or conditional schema.

Part 1. (R' is an iteration subschema of R.) Let $y \in OUT(R')$ and suppose $y \not\equiv x$, every $x \in IN(R')$. Let $H_1' \in DADEPS(R')$ where the first decision in H_1' has outcome false and H_1' therefore contains zero repetitions of the body of R'. Since S is free, there exists $G_1 \in DADEPS(S)$

containing some $H_1 \in$ DADEPS(R) such that H_1 contains H_1'. Let v_1 be the value node labelled y in H_1. Node v_1 is also labelled x for some $x \in$ IN(R') because H_1' contains no operator applications. Since by hypothesis $y \not\equiv x$, let H_2 be the result of replacing H_1' by H_2' in H_1, and let G_2 be the result of replacing H_1 by H_2 in G_1, where $H_2' \in$ DADEPS(R') is chosen so value node u_2 labelled x in H_2 and value node v_2 labelled y in H_2 are not similar in G_2. If $z \in$ MAIN(R), then z cannot label v_1 in G_1 and v_2 in G_2 unless R contains an empty data path from link y to link z. Hence $z \not\equiv y$ unless there is an empty data path in R from y to z.

Part 2. (R' is a conditional subschema of R.) We use an induction on r, the order of schema R.

Basis: The Theorem holds trivially if R contains no conditional or iteration subschemas.

Induction: Assume the Theorem is valid whenever schema R is of order less than r. Let R be of order r, let R' be a conditional subschema of R, and let $y \in$ OUT(R'). Suppose $y \not\equiv x$, every $x \in$ IN(R'). Let P and Q be the true and false alternative schemas of R', and let $u \in$ OUT(P) and $v \in$ OUT(Q) be the T- and F-input links of the merge with output link y. Let $z \in$ MAIN(R). Since S is a free schema, $z \equiv y$ implies that either R contains an empty data path from y to z, or that $u \equiv v$ in the union of R'. We shall demonstrate that the latter assumption leads to a contradiction.

Let n be the sum of the largest number of conditional and iteration subschemas on any path from IN(P) to link $u \in$ OUT(P) and the largest number of conditional and iteration subschemas on any path from IN(Q) to link $v \in$ OUT(Q). We show that $u \not\equiv v$ in the union of R' by induction on n.

Basis: Schemas P and Q contain only operators. We distinguish three cases.

1. u and v are input links of P and Q, respectively. If $u \equiv v$ in the union of R', then in the union of R' links u and v must coincide with data input links x and x' where $x \equiv x'$. But this implies that $y \equiv x$, which contradicts our assumption that $y \not\equiv x$ for each $x \in$ IN(R').

2. Links u and v are output links of operators a and b, respectively. Then operators a and b must have the same function letter, and S cannot be a modified schema with respect to link y.

3. Just one of links u and v is the output link of an operator. Suppose link u is the output link of an operator with function letter f. Then v must be an output link of a gate node or an output link of a conditional schema within Q. If $u \equiv v$ in the union of R', then every empty data path in S leading to link v must originate at an operator with function letter f. Hence S cannot be a modified schema with respect to link v.

Induction: We distinguish two cases:

1. Links u and v are output links of operators a and b, respectively. Then operators a and b must have the same function letter and S cannot be a modified schema with respect to link y.

2. At least one of u and v is an output link of a conditional or iteration schema. First, suppose $u \in OUT(L)$ where L is an iteration subschema of P. By Part 1 of the Theorem, either $u \equiv u'$ for some $u' \in IN(L)$, or $u \not\equiv v$ in the union of R' since the union cannot contain a data path from link u to link v. But the largest number of conditional and iteration subschemas on any path from IN(P) to link u' is one less than the corresponding number for link u. Hence, by the hypothesis of induction, $u' \not\equiv v$ in the union of R'. Therefore, $u \not\equiv v$ in the union of R'. Second, suppose $u \in OUT(C)$ where C is a conditional subschema of P. Since C is of order less than r, either $u \equiv u'$ for some $u' \in IN(C)$ or $u \not\equiv v$ in the union of R'. Again, the largest number of conditional and iteration subschemas on any path from IN(P) to link u' is one less than the corresponding number for link u. By induction $u' \not\equiv v$ in the union of R'. Therefore $u \not\equiv v$ in the union of R'. The corresponding argument applies if link v is an output link of a conditional or iteration schema.

The Decision Procedure

Procedures for deciding equivalence of any two main links of a schema are given in the proofs of Theorem 7 and Theorem 8. These procedures yield for each subschema R a partition of the input and output links of R into sets of equivalent links.

Equivalence Partitions: If X is a set of data links in a schema, then P is the equivalence partition of X if and only if P is the partition of X into maximal sets of equivalent links. We write P = EQPART(X). For simplicity, we shall regard a partition P of a set X as a relation

$$P \subseteq X \times X$$

and write

$$(x, x') \in P$$

to mean: x and x' are in the same part of P.

Similar paths: Two paths α and β in a schema containing only operators and data links are similar ($\alpha \sim \beta$) if and only if both paths contain equally many operators, and, if a is the i^{th} operator on path α and b is the i^{th} operator on path β, then a and b have the same function letter and the input arcs of a and b that lie on paths α and β have the same index.

Thoerem 7: Let R be a schema in a free modified schema S (possibly R = S). Let R' be a subschema of R where R' contains only operators and data links, and each $x \notin IN(R')$ is in IN(R) or is an output link of some conditional or iteration subschema of R. Then if P = EQPART(IN(R')) is given, one can construct P' = EQPART(BOUND(R')).

Proof: In this proof we use the convention that $x, x' \in IN(R')$ and $y, y' \in OUT(R')$. Define P' as follows:

$(x, x') \in P'$ if and only if $(x, x') \in P$.

$(y, y') \in P'$ if and only if, for each path in R' from some x to y (or y') there is a similar path in R' from some x' to y' (or y) where $(x, x') \in P$.

By construction $x \equiv x'$ if and only if $(x, x') \in P'$. We prove that $P' = EQPART(BOUND(R'))$ by showing that: (1) $y \equiv y'$ if and only if $(y, y') \in P'$; and (2) $y \equiv x$ if and only if $y = x'$ where $(x, x') \in P$.

Part 1. ($y \equiv y'$ if and only if $(y, y') \in P'$.)

Sufficiency: By construction of P' the value nodes labelled y and y' in any $G \in DADEPS(R)$ are similar if $(y, y') \in P'$.

Necessity: Suppose $y \equiv y'$ but $(y, y') \notin P'$. There must be paths in R'

$$x \xrightarrow{\alpha} y \quad \text{and} \quad x' \xrightarrow{\alpha'} y'$$

such that one of the following is true:

1. $\alpha \sim \alpha'$ but $(x, x') \notin P$.
2. $(x, x') \in P$ but $\alpha \not\sim \alpha'$.
3. $\alpha \not\sim \alpha'$ and $(x, x') \notin P$.

We show that each of (1), (2) and (3) leads to a contradiction.

1. If $\alpha \sim \alpha'$ but $(x, x') \notin P$, then $x \neq x'$ and we can choose $G \in DADEPS(S)$ containing $H \in DADEPS(R)$ such that the value nodes labelled x and x' in H are not similar. Then the nodes labelled y and y' in H are not similar, contradicting $y \equiv y'$.

2. If $(x, x') \in P$ but $\alpha \not\sim \alpha'$, then $x \equiv x'$ and in any $G \in DADEPS(S)$ containing some $H \in DADEPS(R)$, the nodes labelled x and x' in H will be similar. Hence the nodes labelled y and y' in H cannot be similar and $y \neq y'$.

3. Suppose $x \neq x'$ and $\alpha \not\sim \alpha'$. If $y \equiv y'$, α must be similar to a proper subpath of α' (or vice versa), as illustrated in Figure 18. Suppose the last operator in α' having no similar operator in α has function letter f. Let $G \in DADEPS(S)$ contain $H \in DADEPS(R)$. The value node labelled x in H must be the output value node of an application having function letter f. This must be true for every $H \in DADEPS(R)$ contained in any $G \in DADEPS(S)$. Since S is free, S can have no empty data path leading to link x from any link in $IN(S)$ or from the output link of any operator having function letter $g \neq f$. But this is impossible because S is a modified schema.

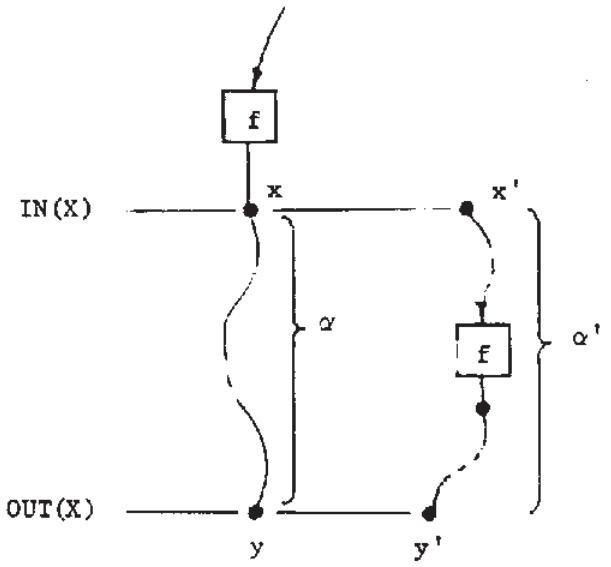


Figure 18. Necessity argument for Theorem 7.

Part 2. ($y \equiv x$ if and only if $y = x'$ for some x' where $(x, x') \in P$.): If $y = x'$ but $x' \not\equiv x$ then $y \not\equiv x$. Otherwise R contains a nonempty path to link y from some x'. If $x' \equiv x$, the argument of Part 1, case 2 applies to show $y \not\equiv x$; if $x' \not\equiv x$ the argument of Part 1, case 3 shows that $y \not\equiv x$.

Theorem 8: Let R be a schema within a free modified schema S (possibly R = S). Let Q be any subschema in R where Q is some composition of operators, conditional schemas and iteration schemas of R, and $IN(Q) \subseteq IN(R)$. Then one can construct P = EQPART(BOUND(Q)).

Proof: We validate the following three predicates by induction on the order of any schema R contained in S.

WF(r): If R is a (well formed) schema of order r and P = EQPART(IN(R)), one can construct P' = EQPART(BOUND(R)).

COND(r): If R is a conditional schema of order r and P = EQPART(IN(R)), one can construct P' = EQPART(BOUND(R)).

ITER(r): If R is an iteration schema of order r and P = EQPART(IN(R)), one can construct P' = EQPART(BOUND(R)).

The proof is in four parts:

Part 1: WF(0) is true.

Part 2: WF(r) implies COND(r + 1)

Part 3: WF(r) implies ITER(r + 1)

Part 4: COND(r) and ITER(r) imply WF(r)

Part 1. (WF(0) is true.) If R is of order 0, R contains only operators. By Theorem 7 one can construct P' = EQPART(BOUND(R)).

Part 2. (WF(r) implies COND(r+1).). Let C be a conditional schema of order r + 1 within S and suppose P = EQPART(IN(C)). Let A and B be the schemas that are the true and false alternatives of C (see Figure 19). We use the convention that

x, x' ∈ IN(C)	t, t' ∈ IN(A)	v, v' ∈ IN(B)
y, y' ∈ OUT(C)	u, u' ∈ OUT(A)	w, w' ∈ OUT(B)

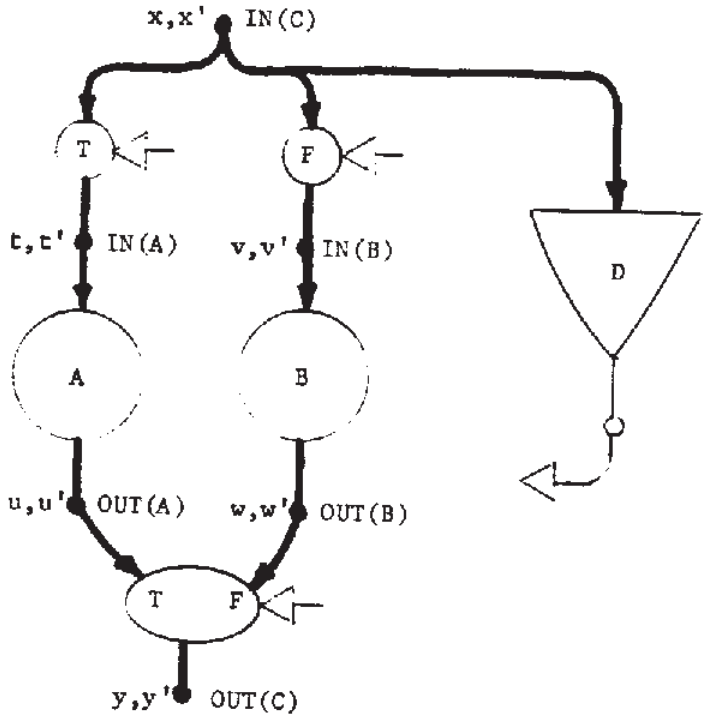


Figure 19. Part 2 of Theorem 8.

Construct partition P' of BOUND(C) as follows:

Step 1. Construct partition Q_A of IN(A):

$(t, t') \in Q_A$ if and only if t and t' are output links of T-gates with input links x and x', and $(x, x') \in P$.

Construct partition Q_B of IN(B):

$(v, v') \in Q_B$ if and only if v and v' are output links of F-gates with input links x and x', and $(x, x') \in P$.

Assertion 1: $P = EQPART(IN(C))$ implies $Q_A = EQPART(IN(A))$ and $Q_B = EQPART(IN(B))$.

Step 2. Use WF(r) to construct partition Q'_A of BOUND(A) from Q_A, and partition Q'_B of BOUND(B) from Q_B.

Assertion 2: $Q_A = EQPART(IN(A))$ implies $Q'_A = EQPART(BOUND(A))$; $Q_B = EQPART(IN(B))$ implies $Q'_B = EQPART(BOUND(B))$.

Step 3. Construct partition P' of BOUND(C):

$(x, x') \in P'$ if and only if $(x, x') \in P$.

$(x, y) \in P'$ if and only if: the merge with output link y has data input links u and w; some T-gate has input link x' and output link t where $(x, x') \in P$ and $(t, u) \in Q'_A$; some F-gate has input link x'' and output link v where $(x, x'') \in P$ and $(v, w) \in Q'_B$.

$(y, y') \in P'$ if and only if: the merge with output link y has input links u and w; the merge with output link y' has input links u' and w'; $(u, u') \in Q'_A$ and $(w, w') \in Q'_B$.

Assertion 3: $Q'_A = EQPART(BOUND(A))$, $Q'_B = EQPART(BOUND(B))$, and $P = EQPART(IN(C))$ imply $P' = EQPART(BOUND(C))$,

The validity of Assertions 1 and 3 follows directly from the construction of partitions Q_A, Q_B and P'. The validity of Assertion 2 is provided by WF(r). The validity of Part 2 follows directly from the three assertions.

Part 3. WF(r) implies ITER(r + 1).): Let L be an iteration schema of order r + 1 within S and suppose P = EQPART(IN(L)). Let schema B be the body of L and let M(L) contain the output links of the merge nodes of L (see Figure 20). We use the convention that

$x, x' \in IN(L)$ $\quad w, w' \in M(L)$ $\quad u, u' \in IN(B)$

$y, y' \in OUT(L)$ $\quad v, v' \in OUT(B)$

Construct partition P' of BOUND(L) as follows:

Step 1. Let $Q_0' = \{IN(L) \cup OUT(B)\}$ be an initial (trivial) partition of $IN(L) \cup OUT(B)$. Set i = 0.

Step 2. Construct partition P_i of $IN(L) \cup M(L)$:

$(x, x') \in P_i$ if and only if $(x, x') \in P$.

$(w, w') \in P_i$ if and only if: the merge nodes with output links w and w' have F-input links x and x' and T-input links v and v'; $(v, v') \in Q_i'$; and $(x, x') \in P$.

$(x, w) \in P_i$ if and only if: the merge node with output link w has F-input link x' and T-input link v; $(x, x') \in P$; and $(x, v) \in Q_i'$.

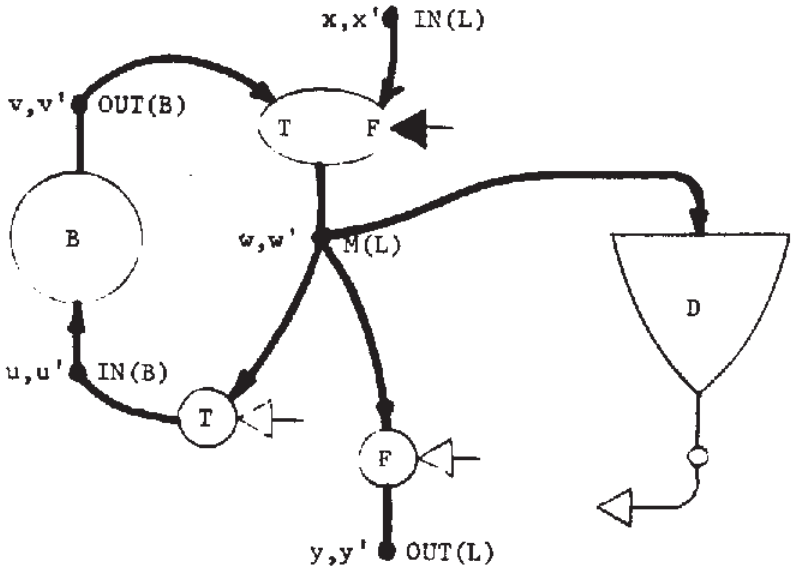


Figure 20. Part 3 of Theorem 8.

Assertion 1: $P = EQPART(IN(L))$ and $Q_i' = EQPART(IN(L) \cup OUT(B))$ imply $P_i = EQPART(IN(L) \cup M(L))$.

Step 3. Construct partition Q_{i+1} of $IN(L) \cup IN(B)$:

$(x, x') \in Q_{i+1}$ if and only if $(x, x') \in P_i$.

$(u, u') \in Q_{i+1}$ if and only if the T-gates having output links u and u' have data input links w and w', and $(w, w') \in P_i$.

$(x, u) \in Q_{i+1}$ if and only if the T-gate having output link u has data input link w, and $(x, w) \in P_i$.

Assertion 2. $P_i = EQPART(IN(L) \cup M(L))$ implies $Q_{i+1} = EQPART(IN(L) \cup IN(B))$.

Step 4. Construct partition Q_{i+1}' of $IN(L) \cup OUT(B)$:
Define partition R of IN(B) by:

$(u, u') \in R$ if and only if $(u, u') \in Q_{i+1}$.

Assume $R = EQPART(IN(B))$ and use the validity of WF(r) to construct $R' = EQPART(BOUND(B))$. Define Q_{i+1}' by:

$(x, x') \in Q_{i+1}'$ if and only if $(x, x') \in Q_{i+1}$.

$(v, v') \in Q_{i+1}'$ if and only if $(v, v') \in R'$.

$(x, v) \in Q_{i+1}'$ if and only if $(x, u) \in Q_{i+1}$ and $(u, v) \in R'$ for some u.

Assertion 3. $Q_{i+1} = EQPART(IN(L) \cup IN(B))$ implies $Q_{i+1}' = EQPART(IN(L) \cup OUT(B))$.

Step 5. If $Q_{i+1}' \neq Q_i'$, set $i = i + 1$ and return to Step 2. Otherwise let $P'' = P_i$.

Step 6. Define partition P' by:

$(x, x') \in P'$ if and only if $(x, x') \in P''$

$(y, y') \in P'$ if and only if the F-gates having output links y and y' have data input links w and w', and $(w, w') \in P''$

$(x, y) \in P'$ if and only if the F-gate having output link y has data input link w, and $(x, w) \in P''$.

Assertion 4. P" = EQPART(IN(L) ∪ M(L)) implies P' = EQPART(BOUND(L)).

The validity of Assertions (1), (2) and (4) follows by construction; that S is free is used in Assertion (1). The validity of WF(r) establishes Assertion (3). Termination of the procedure is guaranteed because each Q'_{i+1} is a refinement of Q'_i, and L has finitely many links. Then P" = EQPART(IN(L) ∪ M(L)) holds by an induction on the length of the procedure. With Assertion (4), the validity of ITER(r + 1) follows.

Part 4. (COND(r) and ITER(r) imply WF(r + 1).): Let R be a schema within S and suppose Q = EQPART(IN(R)). As shown in Figure 21, let $R_1, \ldots, R_n$ be the conditional and iteration subschemas of R, with the numbering chosen so R contains no data path from R_j to R_i if

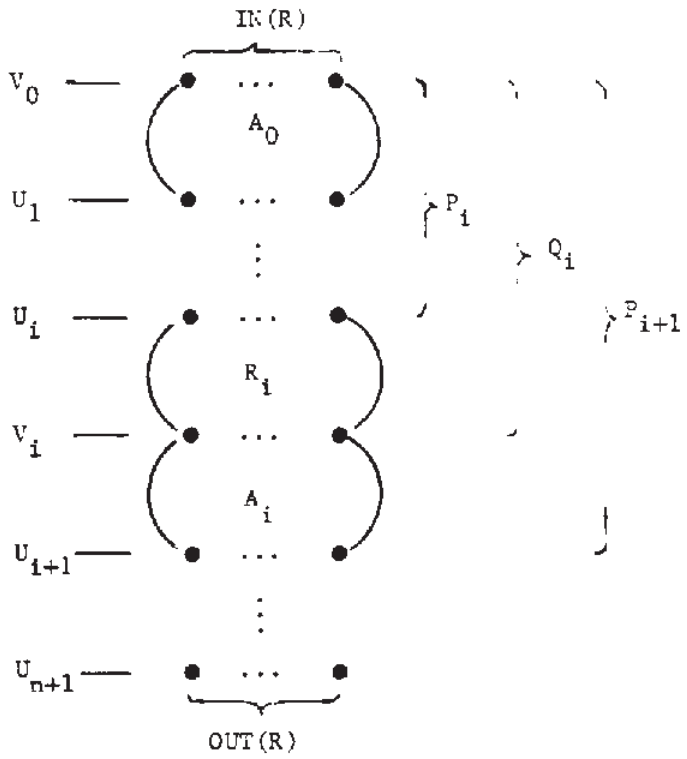


Figure 21. Structure of Schema R for Part 4 of Theorem 8.

$j \geq i$. Let

$$V_0,\ U_1,\ V_1,\ \ldots,\ U_n,\ V_n,\ U_{n+1}$$

be sets of data links of R such that:

1. $V_0 = IN(R)$; $U_{n+1} = OUT(R)$.
2. Each V_i and U_i is a subset of MAIN(R) that is a cut set of R.
3. The sequence of cut sets strictly "progresses" through R; that is, R contains no path from $v \in V_i$ to $u \in U_j$ where $j \leq i$, and no path from u to v where $j < i$.
4. The part of R between U_i and V_i is precisely R_i.

The actors between V_i and U_{i+1} are operators; let these operators constitute schema A_i. Thus:

$$IN(R_i) \subseteq U_i \qquad IN(A_i) = V_i$$

$$OUT(R_i) \subseteq V_i \qquad OUT(A_i) = U_{i+1}$$

We use the convention that

$$x,\ x' \in IN(R) \qquad u,\ u' \in U_i$$

$$y,\ y' \in OUT(R) \qquad v,\ v' \in V_i$$

Construct Q' = EQPART(BOUND(R)) as follows:

Step 1. Let $Q_0 = Q$; set $i = 0$.

Step 2. Given partition Q_i of $IN(R) \cup V_i$, construct partition P_{i+1} of $IN(R) \cup U_{i+1}$:

$(x, x') \in P_{i+1}$ if and only if $(x, x') \in Q_i$.

$(u, u') \in P_{i+1}$ if and only if either $\{u, u'\} \subseteq V_i$ and $(u, u') \in Q_i$, or $\{u, u'\} \subseteq OUT(A_i)$ and A_i contains similar paths to links u and u' from links v and v' where $(v, v') \in Q_i$.

$(x, u) \in P_{i+1}$ if and only if there exists some $v \in V_i$ such that $v = u$ and $(x, v) \in Q_i$.

Assertion 1. $Q_i = EQPART(IN(R) \cup V_i)$ implies $P_{i+1} = EQPART(IN(R) \cup U_{i+1})$.

Step 3. If $i = n$, let $Q' = P_{i+1}$ and stop. Otherwise set $i = i + 1$.

Step 4. Given partition P_i of $IN(R) \cup U_i$, define partition P_i' of $IN(R_i)$ by:

$$(u, u') \in P_i' \text{ if and only if } (u, u') \in P_i$$

Assertion 2. $P_i = EQPART(IN(R) \cup U_i)$ implies $P_i' = EQPART(IN(R_i))$.

Step 5. Use the validity of COND(r) and ITER(r) to construct partition Q_i' of $BOUND(R_i)$ from P_i'.

Assertion 3. $P_i' = EQPART(IN(R_i))$ implies $Q_i' = EQPART(BOUND(R_i))$.

Step 6. Construct partition Q_i of $IN(R) \cup V_i$:

$(x, x') \in Q_i$ if and only if $(x, x') \in P_i$.

$(v, v') \in Q_i$ if and only if, for some u and u':
either $v = u$ or $(u, v) \in Q_i'$;
either $v' = u'$ or $(u', v') \in Q_i'$; and $(u, u') \in P_i$.

$(x, v) \in Q_i$ if and only if, for some u:
either $v = u$ or $(u, v) \in Q_i'$; and $(x, u) \in P_i$.

Return to Step 2.

Assertion 4. $P_i = EQPART(IN(R) \cup U_i)$ and $Q_i' = EQPART(BOUND(R_i))$ imply $Q_i = EQPART(IN(R) \cup V_i)$.

Assertion (1) holds by construction and Theorem 7. Assertion (2) holds by construction. Assertion (3) follows from the validity of COND(r) and ITER(r). Assertion (4) holds because by Theorem 6, $v \equiv v'$ is possible only if $v, v' \in U_i$ or $v, v' \in OUT(R_i)$. Step 1 validates $Q_i = EQPART(IN(R) \cup V_i)$ for $i = 0$. Hence by induction using Assertions (1), (2), (3) and (4), $Q' = P_{n+1} = EQPART(IN(R) \cup U_{n+1})$ $= EQPART(BOUND(R))$.

Theorem 9: Let S be a free, modified schema, and let p_1 and p_2 be any data links of S. It is decidable whether p_1 and p_2 are equivalent.

Proof: We consider three cases:

1. Suppose S contains a schema R such that $\{p_1, p_2\} \subseteq$ MAIN(R). By Theorem 8 one can construct P = EQPART(BOUND(R)). Then $p_1 \equiv p_2$ if and only if $(p_1, p_2) \in P$.
2. Suppose S contains an iteration schema R such that p_1 and p_2 are output links of merge nodes of R. Let x_1 and x_2 be the F-input links and let y_1 and y_2 be the T-input links of these merge nodes. Then $p_1 \equiv p_2$ if and only if $x_1 \equiv x_2$ and $y_1 \equiv y_2$. Since $\{x_1, x_2\} \subseteq$ MAIN(R) and $\{y_1, y_2\} \subseteq$ MAIN(B) where B is the body of R, $p_1 \equiv p_2$ is decidable.
3. If neither case 1 nor case 2 applies, Theorem 4 shows that $p_1 \not\equiv p_2$.

Corollary: If S is a free schema, one can construct a strongly equivalent schema S' such that no two data links of S' are equivalent. (We assume that sets of equivalent output links in S are combined for comparison with S'.)

Proof: Transform S into a free, modified schema, then merge equivalent links and delete redundant parts to obtain S'.

This work, unfortunately, does not apply as it stands to goto programs or program schemas because there are free program schemas for which no equivalent free data flow schema exists [8].

Having a procedure for testing equivalence of data links in any free schema does not yield a test for equivalence of free schemas because combining two schemas so our test may be applied to corresponding pairs of output links necessarily destroys freedom if the schemas are equivalent. Nevertheless, we hope to extend the concepts and methods developed here to obtain decision procedures for equivalence of schemas.

Acknowledgement

Many useful comments, ideas and counterexamples were contributed in the course of this work by Paul Fox, John Linderman, Joe Qualitz and James Rumbaugh.

References

1. Adams, D. A. A Computation Model With Data Flow Sequencing. Technical Report CS 117, Computer Science Department, School of Humanities and Sciences, Stanford University, Stanford Calif., December 1968.

2. Ashcroft, E., and Manna, A. The translation of 'go to' programs to 'while' programs. Information Processing 71, North-Holland Publishing Co., Amsterdam 1972, pp 250-255.

3. Bahrs, A. Operation patterns (An extensible model of an extensible language). Symposium on Theoretical Programming, Novosibirsk, USSR, August 1972 (preprint).

4. Dennis, J. B. Programming generality, parallelism and computer architecture. Information Processing 68, North-Holland Publishing Co., Amsterdam 1969, pp 484-492.

5. Fosseen, J. B. Representation of Algorithms by Maximally Parallel Schemata. S. M. Thesis, Department of Electrical Engineering, Massachusetts Institute of Technology, Cambridge, Mass., June 1972.

6. Kahn, G. A Preliminary Theory for Parallel Programs. Internal Memo, Institut de Recherche d'Informatique et d'Automatique, Rocquencourt, Yvelines, France, 1973.

7. Karp, R. M., and Miller, R. E. Properties of a model for parallel computations: determinacy, termination, queueing. SIAM J. Appl. Math., Vol. 14, November 1966, pp 1390-1411.

8. Linderma

9. Luckham, , M. S. On formalised computer programs s, Vol. 4, No. 3 (June 1970), pp 220-2

10. Paterson el of Computation. Ph.D. Thesis, Trinity 57.

11. Patil, S lections of determinate systems. Record o: rrent Systems and Parallel Computation

12. Rodrigue: Computation. Report MAC-TR-64, Project MAC, Massachusetts Institute of Technology, Cambridge, Mass., September 1969.