Report Number: WUCS-86-14

1986-07-01

The Syntax and Parsing of the Two-Dimensional Languages

Will D. Gillett and T. D. Kimura

This report introduces the idea of expressing programming concepts in a two-dimensional (pictorial) language. A specific two-dimensional language, Show and Tell, is briefly presented and formalisms that might be used to define the syntax of such a language are discussed. An abstraction of Show and Tell is defined, and a specific grammar formalism is presented for defining the syntax of this abstraction. The mechanisms found in expert systems are shown to be sufficient to parse languages defined by this formalism.

THE SYNTAX AND PARSING OF TWO-DIMENSIONAL LANGUAGES

Will D. Gillett and T. D. Kimura

WUCS-86-14

July 1986

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899

Abstract

This report introduces the idea of expressing programming concepts in a two-dimensional (pictorial) language. A specific two-dimensional language, Show and Tell, is briefly presented and formalisms that might be used to define the syntax of such a language are discussed. An abstraction of Show and Tell is defined, and a specific grammar formalism is presented for defining the syntax of this abstraction. The mechanisms found in expert systems are shown to be sufficient to parse languages defined by this formalism.

1. Introduction: The Problem

The recent advancement of VLSI technology has made it possible to provide end users with high resolution graphics capabilities as a primary computer interface mechanism. Such capabilities can offer a high bandwidth and opportunities for more efficient man-machine communication than the traditional text based interface mechanism.

Visual programming[1] is a new concept in software engineering that is suited to such graphics hardware development. There are two main areas of research in visual programming. One activity is to develop graphics environments in which two-dimensional pictures can be used for monitoring computational processes (and displaying their activities) specified in the traditional one-dimensional programming languages such as LISP, Pascal, and Ada. The other area of research is to develop two-dimensional icon-driven programming languages in which programs are easy to understand and maintain.

A key problem in visual programming language design is to define a pictorial representation of various programming concepts, such as iteration, assignment, block structure, concurrency, exception handling, recursion, and so forth, with as little textual input as possible. This leads to the problem of finding a formalism for specifying the syntax of such languages and a parsing algorithm based on that formalism.

There are several different approaches to specify a formal syntax for two-dimensional languages. One is to extend the concept of a formal grammar designed for one-dimensional languages, such as a phrase structure grammar, into the concept of a two-dimensional grammar; that is, to define a two-dimensional meta-language for specifying the syntax of a two-dimensional object language. Lexical elements are also two-dimensional objects. One difficulty encountered in this approach is the lack of a well-established definition for concatenation of two-dimensional objects. Another difficulty is the lack of a parsing algorithm for two-dimensional grammars.

Another approach is to use a known one-dimensional meta-language to specify the set of one-dimensional representations of two-dimensional program constructs. For example, since a box and a line can be represented by a pair of XY-coordinates, a flowchart program consisting of boxes and lines can be represented by a set of such tagged pairs. In this case, lexical elements are already one-dimensional objects. The two major difficulties in this approach are: to find a proper translation for two-dimensional language constructs into one-dimensional lexical elements, and to find a grammar that specifies exactly the set of combinations of lexical elements representing the two-dimensional language. In this paper, we will introduce a formalism for defining the syntax of a visual programming language, based on the second approach. Parsing, using this formalism, will also be addressed.

2. An Application: Show and Tell Language (STL)

The Show and Tell Language[2] is an icon-driven visual programming language. Since both programs and the structure of data in STL are pictorial, a program can be created solely by using a pointing device, such as a mouse. A keyboard is needed only for entering textual or numerical data. STL is designed for school children and novice computer users. "Keyboardless programming" is the main goal of the language.

The semantic model of STL is based on the concepts of dataflow and completion[3]. Completion refers to a problem in which the missing portions of an incomplete pattern must be filled in. An STL program is called a "puzzle" because it defines a completion problem in the same way that a jig-saw puzzle defines a completion problem.

The language is implemented on the Apple Macintosh™ personal computer with the AppleTalk™ local area network. The current version runs on a 512K Mac with external disk drive.

An STL program (puzzle) consists of nested boxes connected by arrows. Loops (cycles) may not exist in an STL puzzle. A box may be empty or may contain a data value, an icon which represents a system or user-defined operation, or another STL puzzle. Arrows allow data to flow from one box to another. An operation in a box will be executed when and only when all incoming values have arrived at a box. Except for this data dependency, there is no inherent sequencing mechanism in STL. Since there is no loop in an STL program, once an operation is executed and the result is registered in an empty box, the value will never be changed. There are no side effects. Thus, STL is a functional parallel programming language[4].

An empty box can be filled with a data object. Some empty boxes are used for communicating with the environment of the puzzle. They are called the base boxes of the puzzle and are depicted by a thicker box frame. They correspond to formal parameters of a subroutine in traditional programming languages.

An STL puzzle is called inconsistent if two neighboring boxes (boxes connected by arrows) contain conflicting objects; otherwise it is consistent. The primary function of the STL interpreter is to decide whether a given puzzle is consistent or not. There are two kinds of control mechanisms for propagation of the effects of inconsistency: the closed box and the open box. When a closed box contains an inconsistent puzzle, the inconsistency is confined within the box. No communication is possible with a box containing an inconsistent puzzle, i.e., the box becomes non-existent from the viewpoint of the rest of the puzzle. If an box contains a consistent puzzle, then the box has no effect on the rest of the puzzle, i.e., it becomes transparent. When an open box contains an inconsistent puzzle, the inconsistency propagates out of the open box and into the surrounding environment. These two language constructs correspond to the block structure of traditional programming languages.

Inconsistency propagates in the broadcasting mode within the boundary of the smallest closed box containing the inconsistent puzzle, while a data object propagates in a point-to-point mode to neighboring boxes. When an arrow goes through an inconsistent box, data on the arrow does not reach the destination, i.e., the inconsistency of a puzzle switches off the communication path provided by the arrow. There is no explicit Boolean data type in STL.

An STL puzzle can be named by a user-defined icon. Any Macpaint™ picture can be used as a puzzle name. When a box containing a user defined icon is evaluated by the STL interpreter, the puzzle (program) named by the icon will be evaluated, after the incoming data values are moved into the base boxes of the called puzzle. An icon is a subroutine name in STL. Recursive definition of a puzzle is also allowed.

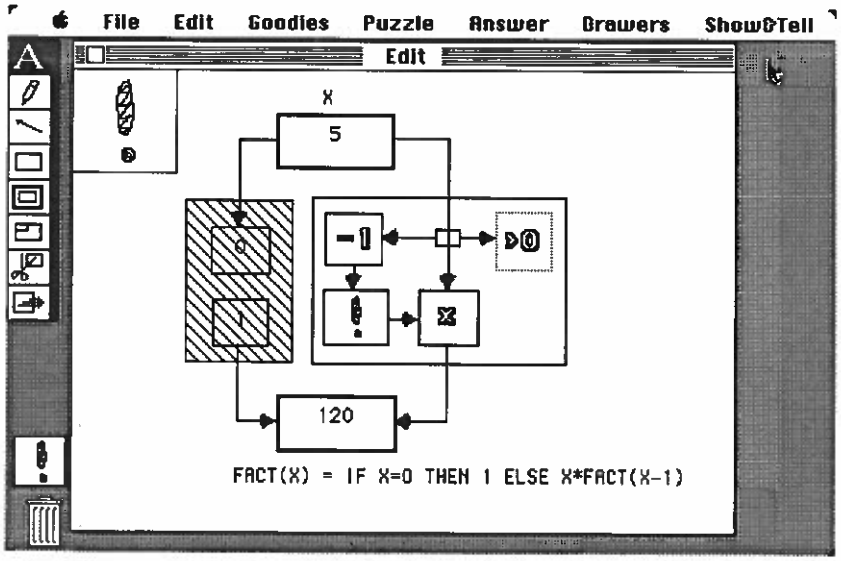


Figure 2.1: A recursive STL Puzzle (Factorial Function)

An example of an STL puzzle is given in Figure 2.1. The puzzle defines the factorial function recursively, and shows the result after computing factorial(5). The name of the puzzle is given in the upper left corner of the edit window. Note that the puzzle has one input and one output parameter. It consists of four boxes (besides the puzzle name box), one of which contains five boxes. The inconsistent box containing two boxes is shaded by the interpreter. The box is inconsistent because there is a conflict in value 5 "flowing" into value 0. The data flows from the top to the bottom of the puzzle. The leftmost column provides the editing tools for puzzle construction. Any box that overlaps with other boxes or any arrow that forms a cycle will be rejected by the editing program of the STL system.

3. An Abstraction: Data Flow Networks

In this section, the basic concept of data flow (from box to box through arrows) present in the STL language of Section 2 is abstracted into the concept of a data flow network. The abstraction presented here is just that of a rooted Directed Acyclic Graph (DAG)[5] (i.e., no cycles are allowed). The abstraction of data flow networks was chosen because most two-dimensional languages utilize the concept of data flowing from one "place" to another. The semantics of what is inside a box (eg., a value or another data flow network) are abstracted out. Single-rooted data flow networks are discussed first, and multi-rooted data flow networks are presented as an extension of the single-rooted case.

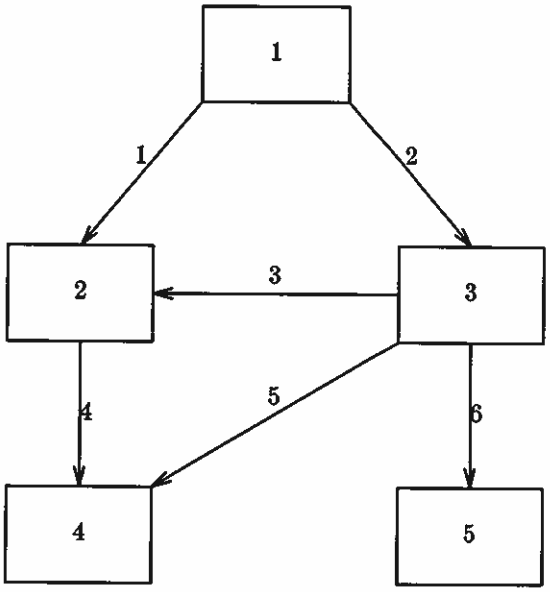
Definition:

A Data Flow Network (DFN) is a rooted directed acyclic graph in which a node is represented by a box, and an arc is represented by an arrow. More specifically, there is a distinguished head node (*head*), which has no arcs (arrows) entering it. There is a set of internal nodes (*internals*), which have arcs entering and leaving them. There is a set of tail nodes (*tails*), which have no arcs leaving them. There is a set of arcs (*arcs*) which connect nodes of the DFN. (Notation: DFN[*head,internals,tails,arcs*]) A trivial DFN is a DFN composed of only one node. A head-maximal DFN is a DFN contained in no other DFN with the same head. A maximal DFN is a DFN contained in no other DFN.

Example 3.1

DFN[1,{},{1},{}]

Example 3.2

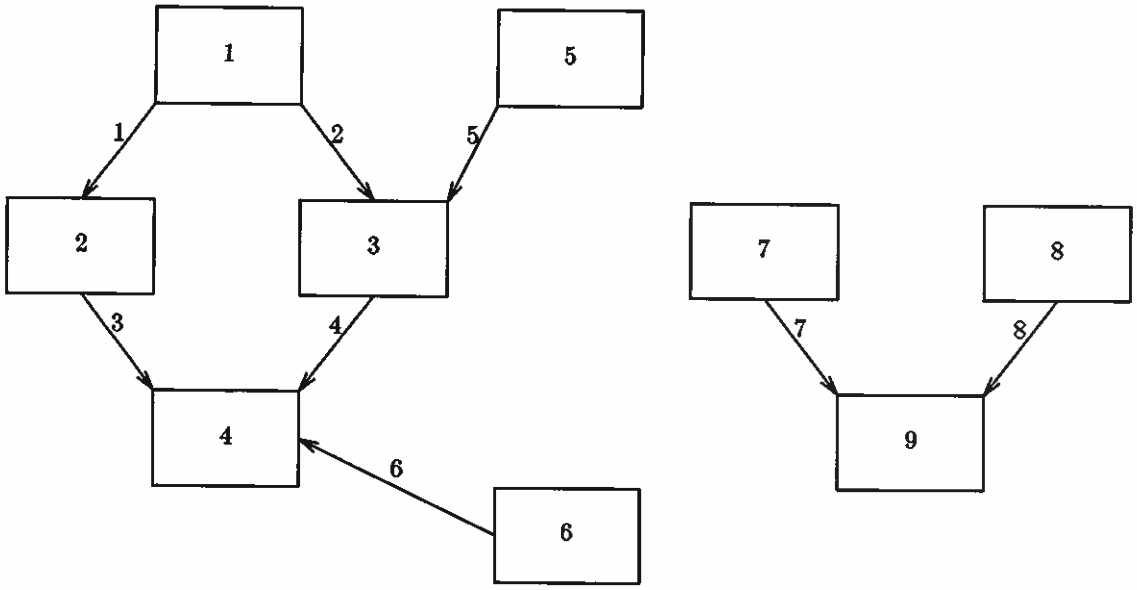


DFN[1,{2,3},{4,5},{1,2,3,4,5,6}]

The five head-maximal DFNs in Example 3.2 are: DFN[1,{2,3},{4,5},{1,2,3,4,5,6}], DFN[2,{},{4},{}], DFN[3,{2},{4,5},{3,4,5,6}], DFN[4,{},{4},{}], and DFN[5,{},{5},{}]. The first is maximal and contains the other four. DFN[5,{},{5},{}] is a trivial DFN. DFN[1,{2,3},{4,5},{2,3,4,5,6}] is another DFN, but it is neither head-maximal nor maximal; it is contained in DFN[1,{2,3},{4,5},{1,2,3,4,5,6}]. Given these five head-maximal DFNs, it is possible to construct the structure of the maximal DFN (although this can be done during the processing of the parse itself).

The concept of a Multi-headed Data Flow Network (MDFN) is very similar; the generalization is that instead of the directed acyclic graph having one distinguished head (root), there is a set of head nodes which have no arcs entering them. The definitions and notation for MDFNs is the same as that for DFNs except that the *head* component is a set of nodes instead of a single node.

Example 3.3



MDFN[{1,5,6},{2,3},{4},{1,2,3,4,5,6}]
MDFN[{7,8},{},{9},{7,8}]

For instance, in Example 3.3, there are two maximal MDFNs: MDFN[{1,5,6},{2,3},{4},{1,2,3,4,5,6}] and MDFN[{7,8},{},{9},{7,8}]. Note that there are five maximal DFNs here (DFN[1,{2,3},{4},{1,2,3,4}], DFN[5,{3},{4},{4,5}], DFN[6,{},{4},{6}], DFN[7,{},{9},{7}] and DFN[8,{},{8},{9}]) but only two MDFNs.

4. A Specification Formalism: Index Set Grammar

In this section, the concept of an Index Set Grammar (ISG) is introduced as a formalism capable of defining the syntax of DFNs (and all the constructs in STL). An ISG specifying the syntax of DFNs is given.

4.1 Definition of an Index Set Grammar

An Index Set Grammar is G = <T,N,S,R>, where:

T is a finite set of terminal *classes*
N is a finite set of nonterminal *classes*
S is the starting symbol, S ∈ N
R is a finite set of rewriting rule *classes*
with associated semantics
V = T+N, the vocabulary

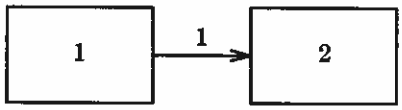
Elements in R have the form <A → α,*S*>, where A ∈ N, $\alpha \in V^*$, and *S* is a set of first-order predicate calculus expressions. Each $v \in V$ has a fixed number (possibly zero) of indices[6] which can be thought of as subscripts on the object. These indices are <u>like</u> attributes[7]; however unlike attributes, they are an integral part of the <u>parsing</u> of the puzzle. The notation used to indicate indices will be square brackets, eg., A[1,a,{b,c}]. (Here, A has 3 indices: "1", "a", and "{b,c}".) The interpretation of α in A → α has no connotation of concatenation or order; the

elements of α represent a set. A more precise notation for A → a b c might be A → {a,b,c}. A rewriting rule can be applied only if all the semantics in S are true. The scope of S is restricted to the rewriting rule to which it is attached.

A lexical analyzer is assumed. Tokens are produced for every elementary object in the puzzle (eg., box and arrow) and every relationship (of interest) between the elementary objects (eg., an arrow touching a box). The lexical analyzer gives each token its appropriate indices.

Example 4.1

The DFN



might produce tokens:

```
B[1]          /* box 1 */
B[2]          /* box 2 */
A[1]          /* arrow 1 */
BATB[1,1]     /* Beginning of Arrow 1 Touches Box 1 */
EATB[1,2]     /* End of Arrow 1 Touches Box 2 */
```

4.2 An ISG for DFNs

An ISG for DFNs is given by:

$$G_{DFN} = <\{B,A,BATB,EATB\},\{DFN\},DFN,R>$$

where R is given by:

(R1) DFN[h1,i1,t1,a1] → B[h2]

indices.1: h1 = h2 ∧
i1 = {} ∧
t1 = {h2} ∧
a1 = {}

(R2) DFN[h1,i1,t1,a1] →
DFN[h2,i2,t2,a2]
BATB[ar3,n3]
A[ar4]
EATB[ar5,n5]
DFN[h6,i6,t6,a6]

indices.2: $h1 = h2 \wedge$
$i1 = i2 \cup i6 \cup (\{n6\} - t6) \wedge$
$t1 = ((t2 - \{h2\}) \cup t6) - i1 \wedge$
$a1 = a2 \cup a6 \cup \{ar4\} \wedge$
$h2 = n3 \wedge$
$ar3 = ar4 = ar5 \wedge$
$h6 = n5$
nocycle.2: $h2 \notin \{h6\} \cup (t2-\{h2\}) \cup i2 \cup i6 \cup t6$
bigger.2: $a2 \neq a1 \vee$
$i2 \neq i1 \vee$
$t2 \neq t1$

The terminals and nonterminals have the indices implicitly defined in Example 4.1. The relationship between the indices in the body of the rules defines how DFNs are built from other objects. The semantics (below the rules) indicate when the rule can be applied. Semantic **indices.1** indicates that R1 can only be applied if the stated relationships between the indices hold; similarly for **indices.2**. Semantic **nocycle.2** states that no cycles are present in the DFN created. Semantic **bigger.2** states that a "larger" DFN has been created. A parse for Example 3.2 might look something like the following. (Here "=>" means "produces a new instance of vocabulary symbol".)

(R1) B[5] => DFN[5,{},{5},{}]
(R1) B[4] => DFN[4,{},{4},{}]
(R1) B[3] => DFN[3,{},{3},{}]
(R1) B[2] => DFN[2,{},{2},{}]
(R1) B[1] => DFN[1,{},{1},{}]
(R2) DFN[3,{},{3},{}] BATB[6,3] A[6] EATB[6,5] DFN[5,{},{5}{}] =>
DFN[3,{},{5},{6}]
(R2) DFN[3,{},{5},{6}] BATB[5,3] A[5] EATB[5,4] DFN[4,{},{4},{}] =>
DFN[3,{},{4,5},{5,6}]
(R2) DFN[2,{},{2},{}] BATB[4,2] A[4] EATB[4,4] DFN[4,{},{4},{}] =>
DFN[2,{},{4},{4}]
(R2) DFN[3,{},{4,5},{5,6}] BATB[3,3] A[3] EATB[3,2] DFN[2,{},{4},{4}] =>
DFN[3,{2},{4,5},{3,4,5,6}]
(R2) DFN[1,{},{1},{}] BATB[2,1] A[2] EATB[2,3] DFN[3,{2},{4,5},{3,4,5,6}] =>
DFN[1,{2,3},{4,5},{2,3,4,5,6}]
(R2) DFN[1,{2,3},{4,5},{2,3,4,5,6}] BATB[1,1] A[1] EATB[1,2] DFN[2,{},{4},{4}] =>
DFN[1,{2,3},{4,5},{1,2,3,4,5,6}]

Note that there are many other parses possible for this particular DFN. The grammar is ambiguous in the sense that there are such multiple parses. For all practical purposes, it is probably impossible to write an unambiguous grammar for such a language because of its unstructured nature. G_{DFN} is able to exclude cycles because it "builds" DFNs from the bottom to the top of the DFN.

4.3 Extension to MDFNs

Once the specification for DFNs is available, it is a trivial matter to specify MDFNs by simply "gluing together" DFNs with common internal and/or tail nodes. An ISG for MDFNs is given by:

$$G_{MDFN} = <\{B,A,BATB,EATB\},\{MDFN,DFN\},MDFN,R>$$

where R is given by:

(R1) Same as in G_{DFN}.

(R2) Same as in G_{DFN}.

(R3) MDFN[h1,i1,t1,a1] $\rightarrow$ DFN[h2,i2,t2,a2]

indices.3: $h1 = \{h2\} \wedge$
$i1 = i2 \wedge$
$t1 = t2 \wedge$
$a1 = a2$

(R4) MDFN[h1,i1,t1,a1] $\rightarrow$
MDFN[h2,i2,t2,a2]
MDFN[h3,i3,t3,a3]

indices.4: $h1 = h2 \bigcup h3 \wedge$
$i1 = i2 \bigcup i3 \wedge$
$t1 = (t2 \bigcup t3) - i1 \wedge$
$a1 = a2 \bigcup a3 \wedge$
common.4: $(i2 \bigcup t2) \bigcap (i3 \bigcup t3) \neq \phi$
nocycle.4: $h1 \bigcap (i1 \bigcup t1) = \phi$
bigger.4: $(h1 \neq h2 \wedge h1 \neq h3) \vee$
$(i1 \neq i2 \wedge i1 \neq i3) \vee$
$(t1 \neq t2 \wedge t1 \neq t3) \vee$
$(a1 \neq a2 \wedge a1 \neq a3)$

For example, given the network of Example 3.3 and the fact that the nine head-maximal DFNs (DFN[1,{2,3},{4},{1,2,3,4}], DFN[2,{},{4},{3}], DFN[3,{},{4},{4}], DFN[4,{},{4},{}], DFN[5,{3},{4},{4,5}], DFN[6,{},{4},{6}], DFN[7,{},{9},{7}], DFN[8,{},{8},{9}] and DFN[9,{},{9},{}]) have already been parsed, the parse to find MDFNs might look something like the following (only the 5 maximal DFNs are used).

(R3) DFN[1,{2,3},{4},{1,2,3,4}] => MDFN[{1},{2,3},{4},{1,2,3,4}]
(R3) DFN[5,{3},{4},{4,5}] => MDFN[{5},{3},{4},{4,5}]
(R3) DFN[6,{},{4},{6}] => MDFN[{6},{},{4},{6}]
(R3) DFN[7,{},{9},{7}] => MDFN[{7},{},{9},{7}]
(R3) DFN[8,{},{8},{9}] => MDFN[{8},{},{8},{9}]
(R4) MDFN[{1},{2,3},{4},{1,2,3,4}] MDFN[{5},{3},{4},{4,5}] =>
MDFN[{1,5},{2,3},{4},{1,2,3,4,5}]
(R4) MDFN[{1,5},{2,3},{4},{1,2,3,4,5}] MDFN[{6},{},{4},{6}] =>
MDFN[{1,5,6},{2,3},{4},{1,2,3,4,5,6}]
(R4) MDFN[{7},{},{9},{7}] MDFN[{8},{},{8},{9},{8}] =>
MDFN[{7,8},{},{9},{7,8}]

5. A Parsing Mechanism: Expert Systems

After studying mechanisms for parsing ISGs, it becomes obvious that expert systems have sufficient power. This includes not only the ability to relate the indices of different objects but also the ability to specify the semantic restrictions (eg., **nocycle** and **bigger**). The isomorphism between ISGs and their instantiation in expert systems is striking. Here, we show the instantiation in a specific expert system language, OPS83[8,9]. We will not explain the syntax or semantics of OPS83; this is assumed from a general knowledge of expert systems. For brevity, we only present segments of the solution in the body of this report. Supporting routines will not be fully explained; their semantics are assumed by the mnemonics of their names. The complete OPS83 code along with supporting explanation can be found in Appendix A.

5.1 Definitions

The pertinent definitions are:

```
type set = array(15:logical);
type b=element(name:integer;);
type a=element(name:integer;);
type batb= element(arrow:integer; box:integer;);
type eatb=element(arrow:integer; box:integer;);
type dfn = element
(
    name:integer;
    head:integer;
    internal:set;
    tail:set;
    arcs:set;
);
type mdfn = element
(
    name:integer;
    head:set;
    internal:set;
    tail:set;
    arcs:set;
);
type goal = element(activity:symbol;);
type arrowdelete = element(arrow:integer;);
```

The first type (`set`) simulates a set by an array of Boolean (logical). The next four types (`b`, `a`, `batb`, and `eatb`) correspond to the four classes of terminal tokens. The next type (`dfn`) corresponds to the DFN class of nonterminal. The next type (`mdfn`) corresponds to the MDFN class of nonterminal. The next type (`goal`) is used to specify the activity to be performed. The next type (`arrowdelete`) is used to report arrow that should be deleted becaused their presence produces a cycle.

5.2 Creating DFNs

The OPS83 instantiation of the two rules in G_{DFN} are as follows.

```
rule RULE_upgrade_box_to_dfn
{
    &1 (b);
    &g (goal activity=create);
    [1.75]
        -->
    make (dfn call sh_attrib1(@,&1.name));
};

rule RULE_combine_dfns
{
    &1 (dfn);
    &2 (batb box = &1.head);
    &3 (a name = &2.arrow);
    &4 (eatb arrow = &3.name);
    &5 (dfn head =  &4.box; sh_nocycle(&1,&5); sh_bigger(&1,&5,&3););
    &g (goal activity=create);
    [1.5]
        -->
    make (dfn call sh_attrib2(@,&1,&5,&3));
};
```

Here, the value in square brackets is used to specify a priority scheme for which rules are fired; rules with higher values are fired first. Routines `sh_attrib1` and `sh_attrib2` set the indices of the newly created object; these two routines correspond to semantics **indices.1** and **indices.2**, respectively, of the grammar. Routines `sh_nocycle` and `sh_bigger` correspond to semantics **nocycle.2** and **bigger.2**, respectively. These two rules alone create many sub-DFNs of head-maximal DFNs. Since we are only interested in head-maximal DFNs, it is appropriate to remove these sub-DFNs (if for no other reason than efficiency). This is achieved by the following rule.

```
rule RULE_delete_subset_dfns
{
    &5 (dfn);
    &6 (dfn head=&5.head; name <> &5.name; dfn_subset(&5,&6););
    &g (goal activity=create);
    [2.0]
        -->
    remove &5;
};
```

Applying these rules to Example 3.2, which has lexical tokens `b[1]`, `b[2]`, `b[3]`, `b[4]`, `b[5]`, `batb[1,1]`, `a[1]`, `eatb[1,2]`, `batb[2,1]`, `a[2]`, `eatb[2,3]`, `batb[3,3]`, `a[3]`, `eatb[3,2]`, `batb[4,2]`, `a[4]`, `eatb[4,4]`, `batb[5,3]`, `a[5]`, `eatb[5,4]`, `batb[6,3]`, `a[6]`, and `eatb[6,5]`, produces the following results.

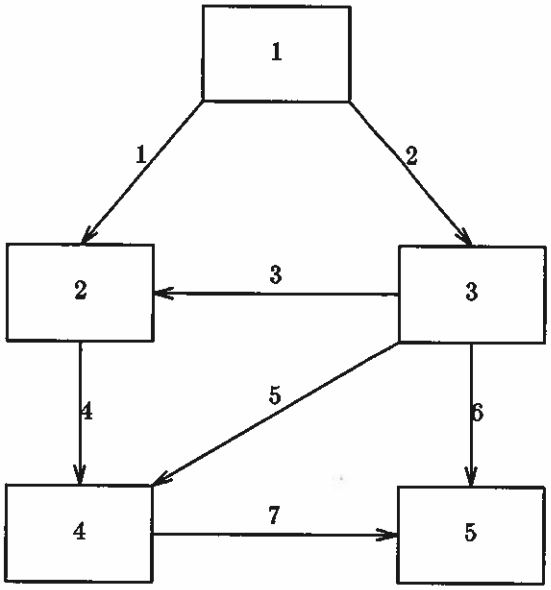
```
all head-maximal DFNs without loops are:
dfn(13)[1,{ 2 3 },{ 4 5 },{ 1 2 3 4 5 6 }]
dfn(12)[3,{ 2 },{ 4 5 },{ 3 4 5 6 }]
dfn(8)[2,{ },{ 4 },{ 4 }]
dfn(2)[4,{ },{ 4 },{ }]
dfn(1)[5,{ },{ 5 },{ }]
```

These are precisely the head-maximal DFNs desired. (The numbers in parentheses are internal names used to uniquely identify each DFN.)

5.3 Adding Tokens

If tokens are added to the network, DFNs produced prior are used in incorporating the new information. Since the processing works from the bottom of the DFN to the top of the DFN, little processing is required for additions near the top of a maximal DFN; more processing is required for additions near the bottom of a maximal DFN. However, no new code is required.

For instance, if we add a new arrow (arrow 7) from box 4 to box 5 to the network of Example 3.2, the following results are obtained.

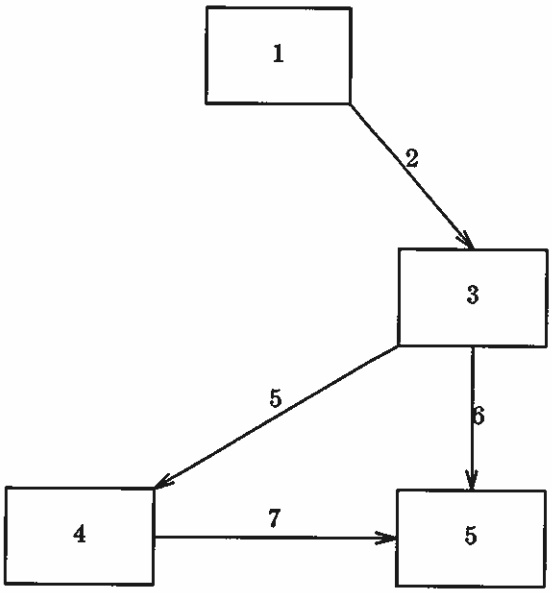



```
all head-maximal DFNs without loops are:
dfn(27) [1,{ 2 3 4 },{ 5 },{ 1 2 3 4 5 6 7 }]
dfn(26) [3,{ 2 4 },{ 5 },{ 3 4 5 6 7 }]
dfn(25) [2,{ 4 },{ 5 },{ 4 7 }]
dfn(24) [4,{ },{ 5 },{ 7 }]
dfn(1) [5,{ },{ 5 },{ }]
```

5.4 Deleting Tokens

As tokens are deleted from the network, DFNs must be modified. The OPS83 code to do this is somewhat extensive and will not be shown here (see Appendix A). There are many approaches to modifying the DFNs. The simplest one was used here; the deletion of any box, arrow or connection causes the deletion of any DFN in which the deleted token took part. Then the creation process is reapplied.

For instance, if we delete box 2 and arrows 1, 3, and 4 from the network of Example 3.2 as modified above in Section 5.3, the following results are obtained.




```
all head-maximal DFNs without loops are:
dfn(40)[1,{ 3 4 },{ 5 },{ 2 5 6 7 }]
dfn(39)[3,{ 4 },{ 5 },{ 5 6 7 }]
dfn(1)[5,{ },{ 5 },{ }]
dfn(24)[4,{ },{ 5 },{ 7 }]
```

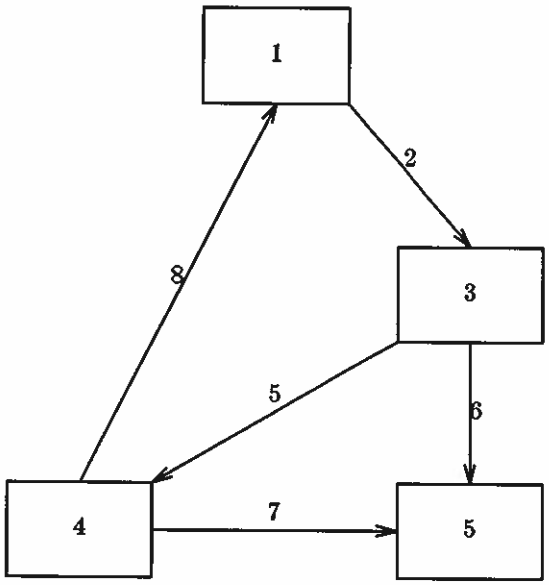
Of course, as boxes and arrows are deleted, the corresponding connections must also be deleted. This is done "automatically" by the OPS83 code (see Appendix A).

5.5 Searching for Cycles

Searching for cycles is relatively simple once the head-maximal DFNs have been created. Searching for cycles is appropriate because the processing shown so far only produced DFNs without loops (as specified by G_{DFN}) and ignores any tokens whose incorporation into a DFN would cause a loop. If such loops are not explicitly searched for and reported, the user may incorrectly infer that all the tokens of the input network have taken part in the processing.

```
rule RULE_search_for_loop
{
    &1 (dfn);
    &2 (eatb box=&1.head);
    &3 (a name=&2.arrow);
    &4 (batb arrow=&3.name; dfn_loop(&4.box,&1););
    &g (goal activity=loopsearch);
    [1.0]
        -->
    make (arrowdelete arrow=&3.name);
};
```

For instance, if we now add an arrow (arrow 8) from box 4 to box 1 of the network of example 3.2 as modified above in Section 5.4, the following results are obtained.




```
deletions:
delete: arrow(8)
unused tokens:
not used: arrow[8]
all head-maximal DFNs without loops are:
dfn(24)[4,{ },{ 5 },{ 7 }]
dfn(1)[5,{ },{ 5 },{ }]
dfn(39)[3,{ 4 },{ 5 },{ 5 6 7 }]
dfn(40)[1,{ 3 4 },{ 5 },{ 2 5 6 7 }]
```

Note the increased content of this output indicating deletion of arrows and unused tokens. This content was suppressed in previous output because no deletions and/or unused tokens were present.

5.6 Extension to MDFNs

The extension to MDFNs is relatively simple. The OPS83 code for R3 and R4 of G_{MDFN} are as follows.

```
rule RULE_upgrade_dfn_to_mdfn
{
    &dfn (dfn);
    &g (goal activity=create);
    [0.5]
        -->
    make (mdfn call mh_attrib1(@,&dfn));
};

rule RULE_combine_mdfns
{
    &mdfn1 (mdfn);
    &mdfn2 (mdfn mh_common(&mdfn1,&mdfn2);
             mh_nocycle(&mdfn1,&mdfn2);
             mh_bigger(&mdfn1,&mdfn2);
        );
    [0.25]
        -->
    make (mdfn call mh_attrib2(@,&mdfn1,&mdfn2));
};
```

In this code, routines mh_attrib1, mh_attrib2, mh_common, mh_nocycle, and mh_bigger correspond to semantics **indices.3**, **indices.4**, **common.4**, **nocycle.4**, and **bigger.4**, respectively.

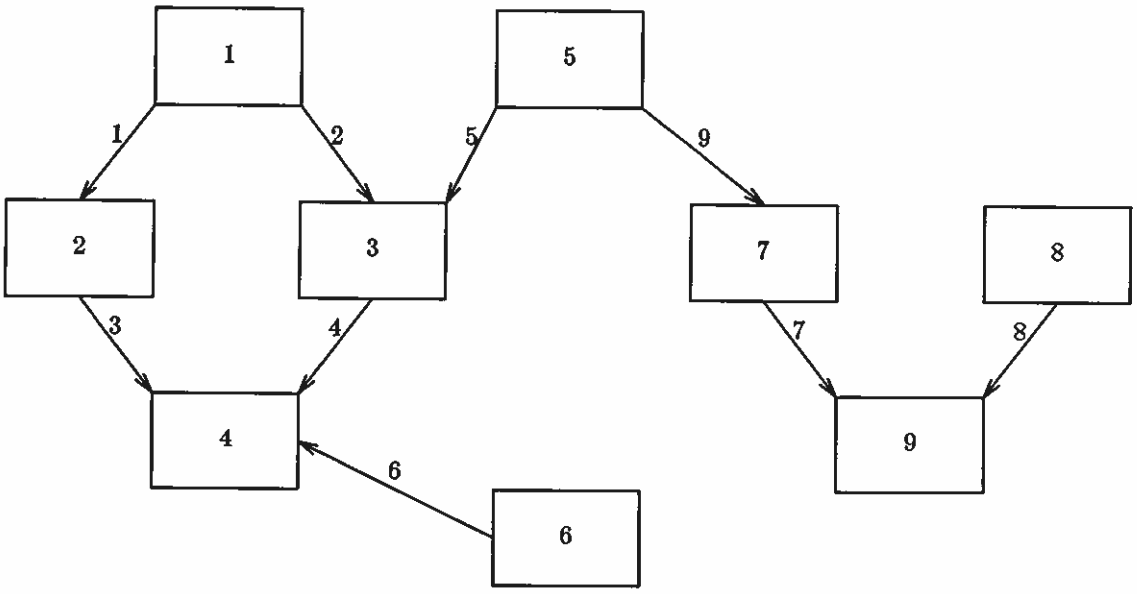
Again, for efficiency, we delete MDFNs that are subsets of others. The OPS83 code is as follows.

```
rule RULE_delete_subset_mdfns
{
    &5 (mdfn);
    &6 (mdfn name <> &5.name; mdfn_subset(&5,&6););
    &g (goal activity=create);
    [0.75]
        -->
    remove &5;
};
```

If we now parse the network of Example 3.3 for MDFNs, the following results are obtained.

```
all head-maximal DFNs without loops are:
dfn(19) [8,{ },{ 9 },{ 8 }]
dfn(18) [7,{ },{ 9 },{ 7 }]
dfn(17) [6,{ },{ 4 },{ 6 }]
dfn(16) [5,{ 3 },{ 4 },{ 4 5 }]
dfn(15) [1,{ 2 3 },{ 4 },{ 1 2 3 4 }]
dfn(14) [3,{ },{ 4 },{ 4 }]
dfn(12) [2,{ },{ 4 },{ 3 }]
dfn(6) [4,{ },{ 4 },{ }]
dfn(1) [9,{ },{ 9 },{ }]
all maximal mdfns are:
mdfn(31) [{ 7 8 },{ },{ 9 },{ 7 8 }]
mdfn(30) [{ 1 5 6 },{ 2 3 },{ 4 },{ 1 2 3 4 5 6 }]
```

If we now add a new arrow (arrow 9) from box 5 to box 7, the following results are obtained.




```
all head-maximal DFNs without loops are:
dfn(41)[5,{ 3 7 },{ 4 9 },{ 4 5 7 9 }]
dfn(1)[9,{ },{ 9 },{ }]
dfn(6)[4,{ },{ 4 },{ }]
dfn(12)[2,{ },{ 4 },{ 3 }]
dfn(14)[3,{ },{ 4 },{ 4 }]
dfn(15)[1,{ 2 3 },{ 4 },{ 1 2 3 4 }]
dfn(17)[6,{ },{ 4 },{ 6 }]
dfn(18)[7,{ },{ 9 },{ 7 }]
dfn(19)[8,{ },{ 9 },{ 8 }]
all maximal mdfns are:
mdfn(53)[{ 1 5 6 8 },{ 2 3 7 },{ 4 9 },{ 1 2 3 4 5 6 7 8 9 }]
```

6. Conclusions

We have presented the concept of a two-dimensional language, a specific instance of a two-dimensional language (STL), a formalism for defining the syntax of such languages (ISGs), an abstraction of a common concept found in most two-dimensional languages (DFNs), and a mechanism (expert systems) for parsing them. OPS83 was the specific expert system language used to show the applicability of using expert systems for parsing ISGs.

The semantics abstracted out of STL in producing the DFN abstraction of Section 3 can be reintroduced in a fairly simple manner. For instance, the concept that MDFNs can be present in a box can be handled by parsing the MDFNs inside the box and creating a new kind of box, complex box, which then takes part in parsing (as above) as if it were a simple box. Of course, this requires the introduction of new relations, such as an object being inside a box and an arrow crossing over the boundary of a box. This extends the responsibilities of the lexical analyser, but the lexical analyser requires no new global contextual information.

The specific use of OPS83 (or any specific expert system language) may not necessarily be the most appropriate mechanism to use for parsing unstructured data. Expert systems supply a very powerful mechanism (because of their generality), but may be inappropriate for efficiency

reasons. Instead, once the exact method of parsing is determined, a more specific (less general) system may be built which is more efficient because it can be tuned to the specific problem of parsing the particular language. However, we have found OPS83 to be very useful in organizing our thoughts and refining our understanding of the specific problem of parsing DFNs. For instance, the concept of head-maximal DFN was arrived at after seeing what was produced by OPS83 when operating in an unconstrained manner; other fundamental insights into this problem have been uncovered by analysing the output (activity) of OPS83. Other types of languages, such as Prolog, may also be applicable, but may not have the same kind of isomorphism that is evident between ISGs and expert systems.

The concepts and mechanisms presented in Sections 3 through 5 of this paper were not used during the development and implementation of STL; instead, ad hoc methods were used to understand the syntax and perform the parsing. This work is an attempt to formalize the syntax and parsing of languages in which the relationships between tokens is much more robust than that found in standard one-dimensional languages.

It is an overstatement to claim that the concepts and mechanisms presented here are sufficient to define and parse *all* two-dimensional languages. The major reason for this is that the field is too young and ill-defined to say what the final form of such two-dimensional languages eventually will be. However, it should be clear that these concepts and mechanisms are capable of handling much more robust relationships between tokens than the one-dimensional concatenation relationship, the only one found in one-dimensional languages.

7. References

[1] G. Raeder, "A Survey of Current Graphical Programming Techniques," *IEEE Computer*, August 1985, pp 11-25.

[2] P. McLain and T. D. Kimura, *Show and Tell User's Manual*, Technical Report WUCS-86-4, Department of Computer Science, Washington University, St. Louis, February 1986.

[3] T. D. Kimura, *Hierarchical Dataflow Model*, Technical Report WUCS-85-5, Department of Computer Science, Washington University, St. Louis, May 1985.

[4] Hugh Glaser, Chris Hankin and David Till, *Principles of Functional Programming*, Prentice-Hall, 1984.

[5] Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullman, *Compilers: Principles, Techniques and Tools*, Addison-Wesley, 1985, pp 290-293.

[6] A. V. Aho, "Indexed Grammars: An Extension of Context-free Grammars," *JACM*, 15 (1968), pp 647-671.

[7] Philip M. Lewis II, Daniel J. Rosenkrantz and Richard E. Stearns, *Compiler Design Theory*, Addison-Wesley, 1976, pp 190-197.

[8] Charles L. Forgy, *The OPS83 User's Manual*, Production Systems Technologies, Inc., May 1985.

[9] Charles L. Forgy, *The OPS83 Report: System Version 2.1*, Production Systems Technologies, Inc., October 1984.

Appendix A: OPS83 Implementation

In this appendix, the entire OPS83 program for parsing DFNs and MDFNs will be presented. Section A.1 will explain the intent and structure of the code, and Section A.2 will present the input and output from which the examples in Section 5 were extracted.

A.1 OPS83 Code

The OPS83 code is divided into three files: `dfndefs.ops`, `dfnrules.ops`, and `dfnmain.ops`. This code will be presented in Figures A.1, A.2, and A.3, respectively. For purposes of presentation, these figures will be broken down into subfigures containing logically similar concepts. Figure A.1(a) shows global variables used across procedure boundaries.

```
-------------------------------------------------------------------
-------------------------------------------------------------------
-- An OPS83 program for a version of the Data Flow Network problem --
--       Definitions and Supporting Routines                       --
-------------------------------------------------------------------
-------------------------------------------------------------------
module dfndefs ()
{
--
-- GLOBAL VARIABLES
--
global &HALTFLAG:logical,
    &RT:logical,
    &WT:logical,
    &dfnname:integer,
    &DEBUG:logical,
    &NRF:integer,
    &INNAME:symbol,
    &INFILE:integer,
    &maxrules:integer;
```

Figure A.1(a): Global Variables

Figure A.1(b) shows the working memory database types. Note that sets are simulated by an array of Boolean (logical). Many of these types have been presented and described in the main body of this report and will not be described here. Type `arrowdelete` is used to report to the user that an arrow of the network should be deleted because its presence causes a loop. Types `deletea`, `deleteb`, `deletebatb`, and `deleteeatb` are input to the system by the user to indicate that arrows, boxes and connections should be removed from the existing network.

```
--
-- TYPES
--
-- box
type b = element(name:integer;);
-- arrow
type a = element(name:integer;);
-- set simulation by an array of Boolean
type set = array(15:logical);
-- base of arrow touches box
type batb = element(arrow:integer; box:integer;);
-- end of arrow touches box
type eatb = element(arrow:integer; box:integer;);
-- single-headed data flow network
type dfn = element
(
    name:integer;
    head:integer;
    internal:set;
    tail:set;
    arcs:set;
);
-- multi-headed data flow network
type mdfn = element
(
    name:integer;
    head:set;
    internal:set;
    tail:set;
    arcs:set;
);
-- current activity to be executed
type goal = element(activity:symbol;);
-- output imperative that arrow should be deleted because of loop
type arrowdelete = element(arrow:integer;);
-- input imperative that arrow should be deleted from network
type deletea = element(name:integer;);
-- input imperative that box should be deleted from network
type deleteb = element(name:integer;);
-- input imperative that connection should be deleted
type deletebatb = element(arrow:integer; box:integer;);
-- input imperative that connection should be deleted
type deleteeatb = element(arrow:integer; box:integer;);
```

Figure A.1(b): Working Memory Types

Figure A.1(c) shows utility routines used for performing the set operations within the simulating Boolean arrays.

```
--
-- Utilities for set operations
--
-----------------------------------------
-- B <- A
-----------------------------------------
simple procedure setcopy(&A:set, &B: out set)
{
local &i:integer;
    for &i = (1 to 15) &B[&i] = &A[&i];
};
-----------------------------------------
-- A <- PHI
-----------------------------------------
simple procedure setempty(&A:out set)
{
local &i:integer;
    for &i = (1 to 15) &A[&i] = 0b;
};
-----------------------------------------
-- A <- A union {a}
-----------------------------------------
simple procedure makemember(&a:integer, &A:out set)
{
    &A[&a] = 1b;
};
-----------------------------------------
-- C <- A union B
-----------------------------------------
simple procedure setunion(&A:set, &B:set, &C:out set)
{
local &i:integer;
    call setcopy(&A,&C);
    for &i = (1 to 15) if (&B[&i]) call makemember(&i,&C);
};
-----------------------------------------
-- C <- A intersect B
-----------------------------------------
simple procedure setintersect(&A:set, &B:set, &C:out set)
{
local &i:integer;
    for &i = (1 to 15) &C[&i] = &A[&i] /\ &B[&i];
};
-----------------------------------------
-- A <- {a}
-----------------------------------------
simple procedure setsingleton(&a:integer, &A:out set)
{
    call setempty(&A);
    call makemember(&a,&A);
};
-----------------------------------------
-- is a member.of A
-----------------------------------------
```

```
simple function memberof(&a:integer, &A:set):logical
{
local &B:set, &C:set;
    return(&A[&a]);
};
-----------------------------------------
-- is a not member.of A
-----------------------------------------
simple function notmemberof(&a:integer, &A:set):logical
{
    return(~memberof(&a,&A));
};
-----------------------------------------
-- C <- A - B
-----------------------------------------
simple procedure setsubtract(&A:set, &B:set, &C:out set)
{
local &i:integer;
    call setcopy(&A,&C);
    for &i = (1 to 15) &C[&i] = &C[&i] /\ (~&B[&i]);
};
-----------------------------------------
-- is A a subset of B?
-----------------------------------------
simple function setsubset(&A:set,&B:set):logical
{
local &i:integer, &C:set;
    for &i = (1 to 15) if (&A[&i] /\ (~&B[&i])) return(0b);
    return(1b);
};
-----------------------------------------
-- is A equal to B?
-----------------------------------------
simple function setequal(&A:set, &B:set):logical
{
local &i:integer;
    for &i = (1 to 15) if (&A[&i] <> &B[&i]) return(0b);
    return(1b);
};
```

Figure A.1(c): Set Operations

Figure A.1(d) presents routines used to output information to the user.

```
--
-- Utilities for printing output
--
-----------------------------------------
-- return end-of-line
-----------------------------------------
function crlf():char
  {
    return('\n');
  };
-----------------------------------------
-- print out a box
-----------------------------------------
procedure output_box(&b:b)
{
    write() crlf(),|box[|,&b.name,|]|;
};
-----------------------------------------
-- print out a ->
-----------------------------------------
procedure output_goesto()
{
    write() crlf(),|->|;
};
-----------------------------------------
-- print out an arrow
-----------------------------------------
procedure outputarrow(&a:a)
{
    write() crlf(),|arrow[|,&a.name,|]|;
};
-----------------------------------------
-- print out a set
-----------------------------------------
procedure output_set(&S:set)
{
local &i:integer;
    write() |{ |;
    for &i = (1 to 15)
        if (memberof(&i,&S)) write() &i,| |;
    write() |}|;
};
-----------------------------------------
-- print out a single-headed data flow network
-----------------------------------------
procedure output_dfn(&dfn:dfn)
{
    write() crlf(),|dfn(|,&dfn.name,|)[|,&dfn.head,|,|;
    call output_set(&dfn.internal);
    write() |,|;
    call output_set(&dfn.tail);
    write() |,|;
    call output_set(&dfn.arcs);
    write() |]|;
```

```
};
-----------------------------------------
-- print out a multi-headed data flow network
-----------------------------------------
procedure output_mdfn(&mdfn:mdfn)
{
    write() crlf(),|mdfn(|,&mdfn.name,|)[|;
    call output_set(&mdfn.head);
    write() |,|;
    call output_set(&mdfn.internal);
    write() |,|;
    call output_set(&mdfn.tail);
    write() |,|;
    call output_set(&mdfn.arcs);
    write() |]|;
};
-----------------------------------------
-- print out a connection
-----------------------------------------
procedure output_batb(&batb:batb)
{
    write() crlf(),|batb[a:|,&batb.arrow,|,b:|,&batb.box,|]|;
};
-----------------------------------------
-- print out a connection
-----------------------------------------
procedure output_eatb(&eatb:eatb)
{
    write() crlf(),|eatb[a:|,&eatb.arrow,|,b:|,&eatb.box,|]|;
};
```

Figure A.1(d): Printing Routines

Figure A.1(e) presents routines used to define the semantics of DFNs.

```
--
-- Utilities for DFN semantics
--
-----------------------------------------
-- determine new internal nodes based on two dfns
-----------------------------------------
simple procedure newinternal(&dfn1:dfn,&dfn2:dfn,&S:out set)
{
local &A:set, &B:set, &C:set;
     call setunion(&dfn1.internal,&dfn2.internal,&A);
     call setsingleton(&dfn2.head,&B);
     call setsubtract(&B,&dfn2.tail,&C);
     call setunion(&A,&C,&S);
};
-----------------------------------------
-- determine new tail nodes based on two dfns
-----------------------------------------
simple procedure newtail(&dfn1:dfn, &dfn2:dfn, &S:out set)
{
local &A:set, &B:set, &C:set, &D:set;
     call setsingleton(&dfn1.head,&A);
     call setsubtract(&dfn1.tail,&A,&B);
     call setunion(&B,&dfn2.tail,&C);
     call newinternal(&dfn1,&dfn2,&D);
     call setsubtract(&C,&D,&S);
};
-----------------------------------------
-- determine new arcs nodes based on two dfns and an arc
-----------------------------------------
simple procedure newarcs(&1:dfn, &2:dfn, &a:a, &A:out set)
{
     call setunion(&1.arcs,&2.arcs,&A);
     call makemember(&a.name,&A);
};
-----------------------------------------
-- single-headed no cycle semantics
-----------------------------------------
simple function sh_nocycle(&dfn1:dfn, &dfn2:dfn):logical
{
local &A:set, &B:set, &C:set, &D:set, &E:set;
     if (&dfn1.head = &dfn2.head) return(Ob);
     call setsingleton(&dfn1.head,&A);
     call setsubtract(&dfn1.tail,&A,&B);
     call setunion(&B,&dfn1.internal,&C);
     call setunion(&C,&dfn2.internal,&D);
     call setunion(&D,&dfn2.tail,&E);
     return(~memberof(&dfn1.head,&E));
};
-----------------------------------------
-- single-headed bigger semantics
-----------------------------------------
simple function sh_bigger(&dfn1:dfn, &dfn2:dfn,&a:a):logical
{
local &A:set;
```

```
    call newarcs(&dfn1,&dfn2,&a,&A);
    if (~setequal(&A,&dfn1.arcs)) return(1b);
    call newinternal(&dfn1,&dfn2,&A);
    if (~setequal(&A,&dfn1.internal)) return(1b);
    call newtail(&dfn1,&dfn2,&A);
    if (~setequal(&A,&dfn1.tail)) return(1b);
    return(0b);
};
-----------------------------------------
-- single-headed attribute assignment for trivial dfn
-----------------------------------------
procedure sh_attrib1(&new:out dfn, &a:integer)
{
    &dfnname = &dfnname+1;
    &new.name = &dfnname;
    &new.head = &a;
    call setempty(&new.internal);
    call setsingleton(&a,&new.tail);
    call setempty(&new.arcs);
    if (&DEBUG) {
        call output_goesto();
        call output_dfn(&new);
    };
};
-----------------------------------------
-- single-headed attribute assignment for combined dfns
-----------------------------------------
procedure sh_attrib2(&new:out dfn, &1:dfn, &2:dfn, &3:a)
{
local &A:set ,&B:set ,&C:set;
    &dfnname = &dfnname + 1;
    &new.name = &dfnname;
    &new.head = &1.head;
    call newinternal(&1,&2,&new.internal);
    call newtail(&1,&2,&new.tail);
    call newarcs(&1,&2,&3,&new.arcs);
    if (&DEBUG) {
        call output_goesto();
        call output_dfn(&new);
    };
};
-----------------------------------------
-- is dfn1 a subset of dfn2?
-----------------------------------------
simple function dfn_subset(&dfn1:dfn, &dfn2:dfn):logical
{
    if (&dfn1.head <> &dfn2.head) return(0b);
    if (setsubset(&dfn1.arcs,&dfn2.arcs)) return(1b);
    return(0b);
};
-----------------------------------------
-- is there a loop in a dfn?
-----------------------------------------
simple function dfn_loop(&b:integer, &dfn:dfn):logical
```

```
{
    if (memberof(&b,&dfn.internal)) return(1b);
    if (memberof(&b,&dfn.tail)) return(1b);
    return(0b);
};
-----------------------------------------
-- make all dfns active again by touching them
-----------------------------------------
procedure touch_dfn()
{
local &i:integer, &name:integer, &L:logical;
    for &i = (wsize() downto 1)
        if (wtype(&i) = dfn /\ wextract(&name,&i,name))
            &L = wstore(&name,&i,name);
};
-----------------------------------------
-- is box b contained in dfn?
-----------------------------------------
simple function box_in_dfn(&b:integer, &dfn:dfn):logical
{
    if (&b = &dfn.head) {
        if (memberof(&b,&dfn.tail)) return(0b)
        else return(1b);
    };
    if (memberof(&b,&dfn.internal)) return(1b);
    if (memberof(&b,&dfn.tail)) return(1b);
    return(0b);
};
-----------------------------------------
-- is arrow a contained in dfn?
-----------------------------------------
simple function arrow_in_dfn(&a:integer, &dfn:dfn):logical
{
    if (memberof(&a,&dfn.arcs)) return(1b);
    return(0b);
};
```

Figure A.1(e): DFN Semantics

Figure A.1(f) presents routines used to define the semantics of MDFNs.

```
--
-- Utilities for MDFN semantics
--
-----------------------------------------
-- multi-headed common semantics
-----------------------------------------
simple function mh_common(&1:mdfn, &2:mdfn):logical
{
local &A:set, &B:set, &C:set, &D:set;
    call setunion(&1.internal,&1.tail,&A);
    call setunion(&2.internal,&2.tail,&B);
    call setintersect(&A,&B,&C);
    call setempty(&D);
    return(~setequal(&C,&D));
};
-----------------------------------------
-- multi-headed no cycle semantics
-----------------------------------------
simple function mh_nocycle(&1:mdfn, &2:mdfn):logical
{
local &h3:set, &i3:set, &t3:set, &A:set, &B:set, &C:set;
    call setunion(&1.head,&2.head,&h3);
    call setunion(&1.internal,&2.internal,&i3);
    call setunion(&1.tail,&2.tail,&t3);
    call setunion(&i3,&t3,&A);
    call setintersect(&h3,&A,&B);
    call setempty(&C);
    return(setequal(&B,&C));
};
-----------------------------------------
-- multi-headed bigger semantics
-----------------------------------------
simple function mh_bigger(&1:mdfn, &2:mdfn):logical
{
local &3:set, &t3:set, &i3:set;

    call setunion(&1.head,&2.head,&3);
    if ((~setequal(&3,&1.head)) /\ (~setequal(&3,&2.head)))
        return(1b);
    call setunion(&1.arcs,&2.arcs,&3);
    if ((~setequal(&3,&1.arcs)) /\ (~setequal(&3,&2.arcs)))
        return(1b);
    call setunion(&1.internal,&2.internal,&i3);
    if ((~setequal(&i3,&1.internal)) /\ (~setequal(&i3,&2.internal)))
        return(1b);
    call setunion(&1.tail,&2.tail,&t3);
    call setsubtract(&t3,&i3,&t3);
    if ((~setequal(&t3,&1.tail)) /\ (~setequal(&t3,&2.tail)))
        return(1b);
    return(0b);
};
-----------------------------------------
-- multi-headed attribute assignment for trivial dfn
-----------------------------------------
```

```
procedure mh_attrib1(&new:out mdfn, &1:dfn)
{
    &dfnname = &dfnname + 1;
    &new.name = &dfnname;
    call setsingleton(&1.head,&new.head);
    call setcopy(&1.internal,&new.internal);
    call setcopy(&1.tail,&new.tail);
    call setcopy(&1.arcs,&new.arcs);
    if (&DEBUG) {
        call output_goesto();
        call output_mdfn(&new);
    };
};
-----------------------------------------
-- multi-headed attribute assignment for combining mdfns
-----------------------------------------
procedure mh_attrib2(&new:out mdfn, &1:mdfn, &2:mdfn)
{
local &A:set;
    &dfnname = &dfnname + 1;
    &new.name = &dfnname;
    call setunion(&1.head,&2.head,&new.head);
    call setunion(&1.arcs,&2.arcs,&new.arcs);
    call setunion(&1.internal,&2.internal,&new.internal);
    call setunion(&1.tail,&2.tail,&new.tail);
    call setsubtract(&new.tail,&new.internal,&new.tail);
    if (&DEBUG) {
        call output_goesto();
        call output_mdfn(&new);
    };
};
-----------------------------------------
-- is mdfn1 a subset of mdfn2?
-----------------------------------------
simple function mdfn_subset(&mdfn1:mdfn, &mdfn2:mdfn):logical
{
    if (setsubset(&mdfn1.arcs,&mdfn2.arcs)) return(1b);
    return(0b);
};
```

Figure A.1(f): MDFN Semantics

Figure A.1(g) shows the utility routine used to get information from the user about the configuration of the network to be analyzed. Note that input can come from either the terminal or from an auxiliary file. Also, different forms of debugging can be turned on and off. The `c` command indicates to continue, i.e., return and process the database given the new information. The `s` command indicates to stop (quit).

```
--
-- Utility for defining network and debugging
--
-----------------------------------------
-- get modifications to the system
-----------------------------------------
function getmod():logical
{
local    &do:symbol,
    &an:integer,
    &bn:integer;
    while (1b) {
        write() crlf();
        read(&INFILE) &do;
        if (&do = aa) {
            read(&INFILE) &an;
            write() crlf(),|add  arrow |,&an;
            make (a name=&an);
        }
        else if (&do = ab) {
            read(&INFILE) &bn;
            write() crlf(),|add  box |,&bn;
            make (b name=&bn);
        }
        else if (&do = abatb) {
            read(&INFILE) &an,&bn;
            write() crlf(),|add  batb |,&an,|  |,&bn;
            make (batb arrow=&an; box=&bn);
        }
        else if (&do = aeatb) {
            read(&INFILE) &an,&bn;
            write() crlf(),|add  eatb |,&an,|  |,&bn;
            make (eatb arrow=&an; box=&bn);
        }
        else if (&do = db) {
            read(&INFILE) &bn;
            write() crlf(),|delete box |,&bn;
            make (deleteb name=&bn);
        }
        else if (&do = da) {
            read(&INFILE) &an;
            write() crlf(),|delete arrow |,&an;
            make (deletea name=&an);
        }
        else if (&do = dbatb) {
            read(&INFILE) &an,&bn;
            write() crlf(),|delete datb |,&an,|  |,&bn;
            make (deletebatb arrow=&an; box=&bn);
        }
        else if (&do = deatb) {
            read(&INFILE) &an,&bn;
            write() crlf(),|delete eatb |,&an,|  |,&bn;
            make (deleteeatb arrow=&an; box=&bn);
        }
```

```
        else if (&do = rton) {
            &RT = 1b;
            write() crlf(),|R trace on|;
        }
        else if (&do = rtoff) {
            &RT = 0b;
            write() crlf(),|R trace off|;
        }
        else if (&do = wton) {
            &WT = 1b;
            write() crlf(),|WM trace on|;
        }
        else if (&do = wtoff) {
            &WT = 0b;
            write() crlf(),|WM trace off|;
        }
        else if (&do = dbon) {
            &DEBUG = 1b;
            write() crlf(),|debug on|;
        }
        else if (&do = dfoff) {
            &DEBUG = 0b;
            write() crlf(),|debug off|;
        }
        else if (&do = input) {
            read(&INFILE) &INNAME;
            write() crlf(),|input from: |,&INNAME,crlf();
            if (&INNAME = terminal) &INFILE = 0
            else {
                &INFILE = open(&INNAME);
                if (&INFILE = -1) {
                    &INFILE = 0;
                    write() crlf(),|could not open file '|,
                        &INNAME,|'. input set to terminal.|;
                };
            };
        }
        else if (&do = c) return(1b)
        else if (&do = s) return(0b)
        else {
            write() |'|,&do,|' is illegal; try again|;
        };
    };
};
```

Figure A.1(g): Network Definition

Figure A.1(h) presents system utility routines. The routine `wmtrace` echos information as changes to the working memory are performed. The routine `select` determines the specific instantiation of a rule that will be fired next. The routine `run` controls selection and firing of rules.

```
--
-- System Utilities
--
-----------------------------------------
-- print trace info about wm changes
-----------------------------------------
procedure wmtrace()
  {
      local &C:char;
      if (wmcadd()) &C='+' else &C='-';
      write() | (|, wmcact(), ' ', wmctype(), ')';
  };
-----------------------------------------
-- print no trace information
-----------------------------------------
procedure notrace()
  {
  };
-----------------------------------------
-- select the dominant instantiation and return its number;
-- return 0 if there are no unfired instantiations
-----------------------------------------
function select(&I:integer):integer
  {
    local
    &C:integer,              -- current best choice
    &CTAG:integer,           -- time tag of goal of &C
    &CLEN:integer,           -- length of inst &C
    &CUSE:integer,           -- number of times &C has fired
    &CRANK:real,             -- rank order of &C
    &CCNT:integer,           -- number of CEs in &C
    &N:integer,              -- next inst to look at
    &NTAG:integer,           -- time tag of goal of &T
    &NLEN:integer,           -- length of inst &N
    &NUSE:integer,           -- number of times &C has fired
    &NRANK:real,             -- rank number for &N
    &NCNT:integer,           -- number of CEs in &N
    &CSS:integer;            -- CS size
    -- get the first unfired inst
    &CSS = cssize();
    if (&DEBUG) write() crlf(),|select cycle |,&I,|,
        number of rules |,&CSS;
    if (&CSS > &maxrules) &maxrules = &CSS;
    &C=0;
    &N=&CSS;
    while (&C=0)
      {
        if (instance(&N,&CTAG,&CUSE,&CRANK,&CLEN,&CCNT) = 0b)
        return(0);
        if (&CUSE=0)
        &C=&N
    else
        &N=&N-1;
      };
```

```
    -- check the rest to find the dominant one
    &N = &C-1;
    while(1b)
      {
        if (instance(&N,&NTAG,&NUSE,&NRANK,&NLEN,&NCNT) = 0b)
        return(&C);
        if (&NUSE = 0)
      {
        if (&NRANK>&CRANK \/
           (&NRANK=&CRANK /\
            (&NLEN>&CLEN \/
                (&NLEN=&CLEN /\ &NTAG> &CTAG)
            )
            )
        )
          {&C=&N; &CLEN=&NLEN; &CTAG=&NTAG; &CRANK=&NRANK;};
      };
    &N = &N-1;
      };
  };
-------------------------------------------
-- fire rules for up to &L cycles
-------------------------------------------
procedure run(&L:integer)
  {
    local &I:integer,&C:integer,&NL:integer,&N:name,&TMP:integer;
    &HALTFLAG = 0B;
    if (&WT=1B)
        on wmchange call wmtrace
    else
        on wmchange call notrace;
    for &I = (1 to &L)
      {
        &C = select(&I);
    if (&C < 1)
        {write() crlf(), |No satisfied rules|, crlf(); return;};
    if (&RT)
      {
        write() crlf();
        &NL = irule(&N,&C);
            for &TMP = (1 to &NL) write() &N[&TMP];
        write() |. |;
      };
    &NRF = &NRF + 1;
    fire &C;
        if (&HALTFLAG = 1B)
        {write() crlf(), |Explicit halt|, crlf(); return;};
      };
    write() crlf(), |Cycle limit reached|, crlf();
  };
};
```

Figure A.1(h): System Utilities

The rules for manipulating the working memory database will also be divided into subfigures. Since the system is invoked iteratively as tokens are added to or deleted from the network, working memory objects may already be in place when the system is invoked. Thus, rules for deleting objects are addressed first. Figure A.2(a) shows the rules for deleting boxes and associated connections. First boxes to be deleted are removed; next, any connections associated with these boxes are marked for deletion; last, any head-maximal DFNs with these boxes as head are removed.

```
------------------------------------------------------------------------
------------------------------------------------------------------------
-- An OPS83 program for a version of the Data Flow Network problem --
--                    Rules of Inference                               --
------------------------------------------------------------------------
------------------------------------------------------------------------
module dfnrules ()
{
use dfndefs;
--
-- Rules for deletion of objects
--

--
-- Rules for deleting boxes and their associations
--
rule RULE_D_box
-- if a box is to be deleted,
-- then delete it
{
    &d (deleteb);
    &b (b name=&d.name);
    &g (goal activity=delete);
    [1.0]
        -->
    remove &b;
};
rule RULE_D_box_batb
-- if a box is to be deleted,
-- then mark to delete any attachments to arrows
{
    &d (deleteb);
    &b (batb box=&d.name);
    &g (goal activity=delete);
    [1.0]
        -->
    make (deletebatb arrow=&b.arrow; box=&b.box);
};
rule RULE_D_box_eatb
-- if a box is to be deleted,
-- then mark to delete any attachments to arrows
{
    &d (deleteb);
    &b (eatb box=&d.name);
    &g (goal activity=delete);
    [1.0]
        -->
    make (deleteeatb arrow=&b.arrow; box=&b.box);
};
rule RULE_D_box_dfn
-- if a box is to be deleted,
-- then delete any DFN with it as head
{
    &d (deleteb);
```

```
    &dfn (dfn head=&d.name);
    &g (goal activity=delete);
    [1.0]
        -->
    remove &dfn;
};
```

Figure A.2(a): Rules for Deleting Boxes

Figure A.2(b) shows the rules for deleting arrows and associated connections. First, any arrows to be deleted are removed; then, any connections associated with these arrows are marked for deletion.

```
--
-- Rules for deleting arrows and their associations
--
rule RULE_D_arrow
-- if arrow is to be deleted,
-- then delete it
{
    &d (deletea);
    &a (a name=&d.name);
    &g (goal activity=delete);
    [0.75]
        -->
    remove &a;
};
rule RULE_D_arrow_batb
-- if an arrow is to be deleted,
-- then mark for deletion any attachments
{
    &d (deletea);
    &b (batb arrow=&d.name);
    &g (goal activity=delete);
    [0.75]
        -->
    make (deletebatb arrow=&b.arrow; box=&b.box);
};
rule RULE_D_arrow_eatb
-- if an arrow is to be deleted,
-- then mark for deletion any attachments
{
    &d (deletea);
    &b (eatb arrow=&d.name);
    &g (goal activity=delete);
    [0.75]
        -->
    make (deleteeatb arrow=&b.arrow; box=&b.box);
};
```

Figure A.2(b): Rules for Deleting Arrows

Figure A.2(c) shows the rules for deleting connections. First, any connections to be deleted are removed; these deletions may have been explicitly entered by the user or may have been introduced implicitly by the deletion of boxes and/or arrows. Next, any DFNs that may have used these connections are removed.

```
--
-- Rules for deleting connections and their associations
--
rule RULE_D_batb
-- if attachment to be deleted,
-- then delete it
{
     &d (deletebatb);
     &b (batb arrow=&d.arrow; box=&d.box);
     &g (goal activity=delete);
     [0.5]
          -->
     remove &b;
};
rule RULE_D_eatb
-- if attachment to be deleted,
-- then delete it
{
     &d (deleteeatb);
     &b (eatb arrow=&d.arrow; box=&d.box);
     &g (goal activity=delete);
     [0.5]
          -->
     remove &b;
};
rule RULE_D_batb_dfn
-- if attachment to be deleted,
-- then remove any DFN containing it
{
     &d (deletebatb);
     &dfn (dfn memberof(&d.arrow,&dfn.arcs));
     &g (goal activity=delete);
     [0.5]
          -->
     remove &dfn;
};
rule RULE_D_eatb_dfn
-- if attachment to be deleted,
-- then remove any DFN containing it
{
     &d (deleteeatb);
     &dfn (dfn memberof(&d.arrow,&dfn.arcs));
     &g (goal activity=delete);
     [0.5]
          -->
     remove &dfn;
};
```

Figure A.2(c): Rules for Deleting Connections

Figure A.2(d) shows the rules for removing all MDFNs that are present in the working memory database.

```
--
-- Rules for deleting MDFNs
--
rule RULE_D_all_mdfns
-- delete all mdfns
{
     &mdfn (mdfn);
     &g (goal activity=delete);
     [0.25]
          -->
     remove &mdfn;
};
rule RULE_switch_to_cleanup_activity
-- go to the cleanup (remove deletions) activity
{
     &g (goal activity=delete);
     [0.0]
          -->
     modify &g (activity=cleanup);
     call touch_dfn();
};
```

Figure A.2(d): Rules for Deleting MDFNs

After the appropriate primary objects have been removed from the working memory, a cleanup activity is performed to remove the imperatives that caused the deletion of the primary objects. The rules for this are shown in Figure A.2(e).

```
--
-- Rules for cleanup after deletion activity
--
rule RULE_R_db
-- remove box deletion
{
    &1 (deleteb);
    &g (goal activity=cleanup);
    [1.0]
        -->
    remove &1;
};
rule RULE_R_da
-- remove arrow deletion
{
    &1 (deletea);
    &g (goal activity=cleanup);
    [1.0]
        -->
    remove &1;
};
rule RULE_R_dbatb
-- remove connection deletion
{
    &1 (deletebatb);
    &g (goal activity=cleanup);
    [1.0]
        -->
    remove &1;
};
rule RULE_R_deatb
-- remove connection deletion
{
    &1 (deleteeatb);
    &g (goal activity=cleanup);
    [1.0]
        -->
    remove &1;
};
rule RULE_switch_to_create_activity
-- go to creation of dfns activity
{
    &g (goal activity=cleanup);
    [0.0]
        -->
    modify &g (activity=create);
};
```

Figure A.2(e): Rules for Cleanup Activity

The rules for creating DFNs and removing subset DFNs are shown in Figure A.2(f). These are extensions of the rules shown in the main body.

```
--
-- Rules for creating DFNs
--
rule RULE_upgrade_box_to_dfn_1
-- make a dfn from a box
{
    &1 (b);
    &g (goal activity=create);
    [1.75]
        -->
    if (&DEBUG) call output_box(&1);
    make (dfn call sh_attrib1(@,&1.name));
};
rule RULE_combine_dfns
-- hook two dfns together
{
    &1 (dfn);
    &2 (batb box = &1.head);
    &3 (a name = &2.arrow);
    &4 (eatb arrow = &3.name);
    &5 (dfn head =  &4.box; sh_nocycle(&1,&5); sh_bigger(&1,&5,&3););
    &g (goal activity=create);
    [1.5]
        -->
    if (&DEBUG) {
        call output_dfn(&1);
        call output_batb(&2);
        call outputarrow(&3);
        call output_eatb(&4);
        call output_dfn(&5);
    };
    make (dfn call sh_attrib2(@,&1,&5,&3));
};
rule RULE_delete_subset_dfns
-- get rid of subset dfns
{
    &5 (dfn);
    &6 (dfn head=&5.head; name <> &5.name; dfn_subset(&5,&6););
    &g (goal activity=create);
    [2.0]
        -->
    if (&DEBUG) {
        write() crlf(),|removing:|;
        call output_dfn(&5);
    };
    remove &5;
};
```

Figure A.2(f): Rules for Creating DFNs

The rules for creating MDFNs and removing subset MDFNs are shown in Figure A.2(g). These are extensions of the rules shown in the main body.

```
--
-- Rules for creating MDFNs
--
rule RULE_upgrade_dfn_to_mdfn
-- make a multi-headed dfn from a single-headed dfn
{
    &dfn (dfn);
    &g (goal activity=create);
    [0.5]
        -->
    if (&DEBUG) call output_dfn(&dfn);
    make (mdfn call mh_attrib1(@,&dfn));
};
rule RULE_combine_mdfns
-- hook together multi-headed dfns
{
    &mdfn1 (mdfn);
    &mdfn2 (mdfn mh_common(&mdfn1,&mdfn2);
             mh_nocycle(&mdfn1,&mdfn2);
             mh_bigger(&mdfn1,&mdfn2);
        );
    [0.25]
        -->
    if (&DEBUG) {
        call output_mdfn(&mdfn1);
        call output_mdfn(&mdfn2);
    };
    make (mdfn call mh_attrib2(@,&mdfn1,&mdfn2));
};
rule RULE_delete_subset_mdfns
-- get rid of subset mdfns
{
    &5 (mdfn);
    &6 (mdfn name <> &5.name; mdfn_subset(&5,&6););
    &g (goal activity=create);
    [0.75]
        -->
    if (&DEBUG) {
        write() crlf(),|removing:|;
        call output_mdfn(&5);
    };
    remove &5;
};
rule RULE_switch_to_loopsearch_activity
-- go to loop searching activity
{
    &g (goal activity=create);
    [0.0]
        -->
    modify &g (activity=loopsearch);
};
```

Figure A.2(g): Rules for Creating MDFNs

The rules for searching for loops are shown in Figure A.2(h).

```
--
-- Rules for searching for loops
--
rule RULE_search_for_loop
-- find arrows that would create loops
{
    &1 (dfn);
    &2 (eatb box=&1.head);
    &3 (a name=&2.arrow);
    &4 (batb arrow=&3.name; dfn_loop(&4.box,&1););
    &g (goal activity=loopsearch);
    [1.0]
        -->
    if (&DEBUG) {
        write() crlf(),|****************************|;
        write() crlf(),|loop   box(|,&4.box,|):arrow(|,
            &4.arrow,|):box(|,&2.box,|)  in|;
        call output_dfn(&1);
        write() crlf(),|****************************|;
    };
    make (arrowdelete arrow=&3.name);
};
rule RULE_switch_to_print_activity
-- go to printing activity
{
    &g (goal activity=loopsearch);
    [0.0]
        -->
    modify &g (activity=print);
};
```

Figure A.2(h): Rules for Finding Loops

The rules for printing results are shown in Figure A.2(i). Note that there are different phases of the output activity. First, arrows that should be removed because there presence causes a loop are presented; next, tokens that are present in the network but do not take part in creating DFNs are presented; then, head-maximal DFNs are presented; last, maximal MDFNs are presented.

```
--
-- Rules for printing results
--
rule RULE_print_deletion_header
-- start into deletion output
{
    &g (goal activity=print);
    [4.0]
        -->
    write() crlf(),|deletions:|;
};
rule RULE_output_deleted_arrow
-- output an arrow that should be deleted because it causes a loop
{
    &1 (arrowdelete);
    &g (goal activity=print);
    [3.5]
        -->
    write() crlf(),|delete: arrow(|,&1.arrow,|)|;
};
rule RULE_print_unused_header
-- indicate tokens not used in creating dfns
{
    &g (goal activity=print);
    [3.0]
        -->
    write() crlf(),|unused tokens:|;
};
rule RULE_output_unused_box
-- output unused boxes
{
    &b (b);
    ~ (dfn box_in_dfn(&b.name,@));
    &g (goal activity=print);
    [2.75]
        -->
    write() crlf(),|not used: box[|,&b.name,|]|;
};
rule RULE_output_unused_arrow
-- output unused arrows
{
    &a (a);
    ~ (dfn arrow_in_dfn(&a.name,@));
    &g (goal activity=print);
    [2.50]
        -->
    write() crlf(),|not used: arrow[|,&a.name,|]|;
};
rule RULE_print_dfn_header
-- indicate dfn structure
{
    &g (goal activity=print);
    [2.0]
        -->
```

```
    write() crlf(),|all head-maximal DFNs without loops are:|;
};
rule RULE_output_dfn
-- output all head-maximal dfns
{
    &1 (dfn);
    &g (goal activity=print);
    [1.5]
        -->
    call output_dfn(&1);
};
rule RULE_print_mdfn_header
-- indicate maximal mdfns
{
    &g (goal activity=print);
    [1.0]
        -->
    write() crlf(),|all maximal mdfns are:|;
};
rule RULE_output_mdfn
-- output all maximal mdfns
{
    &1 (mdfn);
    &g (goal activity=print);
    [0.5]
        -->
    call output_mdfn(&1);
};
rule RULE_remove_goal
-- set up for next iteration
{
    &g (goal activity=print);
    [-0.5]
        -->
    remove &g;
};
};
```

Figure A.2(i): Rules for Printing Results

The main driver is shown in Figure A.3. It first defines some global information. Next, it loops to get modifications to the network and parse new DFNs and MDFNs. At the end, it prints some summary statistics.

```
module  dfnmain(start)
{
-------------------------------------------------------------------------
-------------------------------------------------------------------------
-- An OPS83 program for a version of the Data Flow Network problem --
--                          Main Driver                                --
-------------------------------------------------------------------------
-------------------------------------------------------------------------
use dfndefs;
use dfnrules;
procedure start ()
  {
    &dfnname = 0;
    &DEBUG = 0b;
    &RT = 0b;
    &WT = 0b;
    &NRF = 0;
    &maxrules = 0;
    &INFILE = 0;
    while (getmod()) {
        make(goal activity=delete);
        call run(500);
        write() crlf(),|number of rules fired |,&NRF,crlf();
    };
    write() crlf(),|maxrules = |,&maxrules,crlf();
    write() crlf(),|number of rules fired |,&NRF,crlf();
  };

};
```

Figure A.3: Main Driver

A.2 Defining Networks and the Resulting Parse

The mechanism used to define the content of a network is to read in tokens from an external file (using getmod). The input used to create the DFN examples presented in Sections 5.2 through 5.5 is shown in Figure A.4.

```
ab 1
ab 2
ab 3
ab 4
ab 5
aa 1
aa 2
aa 3
aa 4
aa 5
aa 6
abatb 1 1
abatb 2 1
abatb 3 3
abatb 4 2
abatb 5 3
abatb 6 3
aeatb 1 2
aeatb 2 3
aeatb 3 2
aeatb 4 4
aeatb 5 4
aeatb 6 5
c
aa 7
abatb 7 4
aeatb 7 5
c
db 2
da 1
da 3
da 4
c
aa 8
abatb 8 4
aeatb 8 1
c
s
```

Figure A.4: Input for DFN Examples

The corresponding output is shown in Figure A.5, which is again broken down for presentation purposes. This output has been edited to remove white space. Figure A.5(a) shows the output produced for Section 5.2.

```
add  box 1
add  box 2
add  box 3
add  box 4
add  box 5
add  arrow 1
add  arrow 2
add  arrow 3
add  arrow 4
add  arrow 5
add  arrow 6
add  batb 1  1
add  batb 2  1
add  batb 3  3
add  batb 4  2
add  batb 5  3
add  batb 6  3
add  eatb 1  2
add  eatb 2  3
add  eatb 3  2
add  eatb 4  4
add  eatb 5  4
add  eatb 6  5
deletions:
unused tokens:
all head-maximal DFNs without loops are:
dfn(13)[1,{ 2 3 },{ 4 5 },{ 1 2 3 4 5 6 }]
dfn(12)[3,{ 2 },{ 4 5 },{ 3 4 5 6 }]
dfn(8)[2,{ },{ 4 },{ 4 }]
dfn(2)[4,{ },{ 4 },{ }]
dfn(1)[5,{ },{ 5 },{ }]
all maximal mdfns are:
mdfn(14)[{ 1 },{ 2 3 },{ 4 5 },{ 1 2 3 4 5 6 }]
No satisfied rules
number of rules fired 45
```

Figure A.5(a): Output for Section 5.2

Figure A.5(b) shows the output produced for Section 5.3.

```
add  arrow 7
add  batb 7  4
add  eatb 7  5
deletions:
unused tokens:
all head-maximal DFNs without loops are:
dfn(27)[1,{ 2 3 4 },{ 5 },{ 1 2 3 4 5 6 7 }]
dfn(26)[3,{ 2 4 },{ 5 },{ 3 4 5 6 7 }]
dfn(25)[2,{ 4 },{ 5 },{ 4 7 }]
dfn(24)[4,{ },{ 5 },{ 7 }]
dfn(1)[5,{ },{ 5 },{ }]
all maximal mdfns are:
mdfn(28)[{ 1 },{ 2 3 4 },{ 5 },{ 1 2 3 4 5 6 7 }]
No satisfied rules
number of rules fired 88
```

Figure A.5(b): Output for Section 5.3

Figure A.5(c) shows the output produced for Section 5.4.

```
delete box 2
delete arrow 1
delete arrow 3
delete arrow 4
deletions:
unused tokens:
all head-maximal DFNs without loops are:
dfn(40)[1,{ 3 4 },{ 5 },{ 2 5 6 7 }]
dfn(39)[3,{ 4 },{ 5 },{ 5 6 7 }]
dfn(1)[5,{ },{ 5 },{ }]
dfn(24)[4,{ },{ 5 },{ 7 }]
all maximal mdfns are:
mdfn(41)[{ 1 },{ 3 4 },{ 5 },{ 2 5 6 7 }]
No satisfied rules
number of rules fired 159
```

Figure A.5(c): Output for Section 5.4

Figure A.5(d) shows the output produced for Section 5.5.

```
add  arrow 8
add  batb 8  4
add  eatb 8  1
deletions:
delete: arrow(8)
unused tokens:
not used: arrow[8]
all head-maximal DFNs without loops are:
dfn(24)[4,{ },{ 5 },{ 7 }]
dfn(1)[5,{ },{ 5 },{ }]
dfn(39)[3,{ 4 },{ 5 },{ 5 6 7 }]
dfn(40)[1,{ 3 4 },{ 5 },{ 2 5 6 7 }]
all maximal mdfns are:
mdfn(52)[{ 1 },{ 3 4 },{ 5 },{ 2 5 6 7 }]
No satisfied rules
number of rules fired 192
maxrules = 35
number of rules fired 192
```

Figure A.5(d): Output for Section 5.5

The input used to produce the MDFN examples in Section 5.6 is shown in Figure A.6.

```
ab 1
ab 2
ab 3
ab 4
ab 5
ab 6
ab 7
ab 8
ab 9
aa 1
aa 2
aa 3
aa 4
aa 5
aa 6
aa 7
aa 8
abatb 1 1
abatb 2 1
abatb 3 2
abatb 4 3
abatb 5 5
abatb 6 6
abatb 7 7
abatb 8 8
aeatb 1 2
aeatb 2 3
aeatb 3 4
aeatb 4 4
aeatb 5 3
aeatb 6 4
aeatb 7 9
aeatb 8 9
c
aa 9
abatb 9 5
aeatb 9 7
c
s
```

Figure A.6: Input for MDFN Examples

The output produced for Example 3.3 is shown in Figure A.7(a).

```
add  box 1
add  box 2
add  box 3
add  box 4
add  box 5
add  box 6
add  box 7
add  box 8
add  box 9
add  arrow 1
add  arrow 2
add  arrow 3
add  arrow 4
add  arrow 5
add  arrow 6
add  arrow 7
add  arrow 8
add  batb 1  1
add  batb 2  1
add  batb 3  2
add  batb 4  3
add  batb 5  5
add  batb 6  6
add  batb 7  7
add  batb 8  8
add  eatb 1  2
add  eatb 2  3
add  eatb 3  4
add  eatb 4  4
add  eatb 5  3
add  eatb 6  4
add  eatb 7  9
add  eatb 8  9
deletions:
unused tokens:
all head-maximal DFNs without loops are:
dfn(19)[8,{ },{ 9 },{ 8 }]
dfn(18)[7,{ },{ 9 },{ 7 }]
dfn(17)[6,{ },{ 4 },{ 6 }]
dfn(16)[5,{ 3 },{ 4 },{ 4 5 }]
dfn(15)[1,{ 2 3 },{ 4 },{ 1 2 3 4 }]
dfn(14)[3,{ },{ 4 },{ 4 }]
dfn(12)[2,{ },{ 4 },{ 3 }]
dfn(6)[4,{ },{ 4 },{ }]
dfn(1)[9,{ },{ 9 },{ }]
all maximal mdfns are:
mdfn(31)[{ 7 8 },{ },{ 9 },{ 7 8 }]
mdfn(30)[{ 1 5 6 },{ 2 3 },{ 4 },{ 1 2 3 4 5 6 }]
No satisfied rules
number of rules fired 71
```

Figure A.7(a): Output for Example 3.3

The output produced for the modified network (after adding arrow 9) is shown in Figure A.7(b).

```
add  arrow 9
add  batb 9  5
add  eatb 9  7
deletions:
unused tokens:
all head-maximal DFNs without loops are:
dfn(41) [5,{ 3 7 },{ 4 9 },{ 4 5 7 9 }]
dfn(1) [9,{ },{ 9 },{ }]
dfn(6) [4,{ },{ 4 },{ }]
dfn(12) [2,{ },{ 4 },{ 3 }]
dfn(14) [3,{ },{ 4 },{ 4 }]
dfn(15) [1,{ 2 3 },{ 4 },{ 1 2 3 4 }]
dfn(17) [6,{ },{ 4 },{ 6 }]
dfn(18) [7,{ },{ 9 },{ 7 }]
dfn(19) [8,{ },{ 9 },{ 8 }]
all maximal mdfns are:
mdfn(53) [{ 1 5 6 8 },{ 2 3 7 },{ 4 9 },{ 1 2 3 4 5 6 7 8 9 }]
No satisfied rules
number of rules fired 135
maxrules = 33
number of rules fired 135
```

Figure A.7(b): Output for Modified Network