stichting
mathematisch
centrum

MC

AFDELING INFORMATICA
(DEPARTMENT OF COMPUTER SCIENCE)

IW 179/81 OKTOBER

A.H. VEEN

A FORMAL MODEL FOR DATA FLOW PROGRAMS WITH TOKEN COLORING

Preprint

kruislaan 413 1098 SJ amsterdam

Printed at the Mathematical Centre, 413 Kruislaan, Amsterdam.

The Mathematical Centre, founded the 11-th of February 1946, is a non-profit institution aiming at the promotion of pure mathematics and its applications. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

1980 Mathematics subject classification: 68B10, 68C99

ACM-Computing Reviews-category: 4.12, 4.6, 5.24

A Formal Model for Data Flow Programs with Token Coloring†

by

Arthur H. Veen

ABSTRACT

A formal model is presented that is intended to be general enough to allow description of the execution of programs on different types of data flow computers, including those that use token coloring to separate activations of reentrant graphs. The model regards nodes and ports as basic domains and associates with each node a firing rule which fully defines the behavior of the node including, for each firing, the destinations of its output arcs. The behavior of non-functional and non-deterministic data flow instructions and the dynamic creation of data paths can be expressed in this model. It can be a useful vehicle for exploring semantic properties of data flow programs, including conditions that guarantee deadlock-free, conflict-free and deterministic execution. Some of these present thorny issues in the context of token coloring and dynamic data paths. The model could also be used as a descriptive tool for a formal comparison of different data flow machines.

KEY WORDS & PHRASES: data flow machines, data driven machines, machine models, data flow graphs, safety

†This paper is not for review; it is intended for publication elsewhere.

1. INTRODUCTION

Most computers can reach pathological states, which all meaningful programs should avoid. Programs for conventional computers should, for instance, not lead to infinite loops or to violations of array boundaries. Programs for data flow machines should avoid deadlocks and conflicts. For conventional and data flow programs alike, one would like to formulate properties that can be easily checked statically and that, if satisfied, guarantee that the computer never reaches such a state.

A formal approach is even more compelling for data flow programs than for conventional ones, as intuition, always a doubtful tool, is still less to be trusted when used to validate parallel processes. This became painfully clear when one of the translation algorithms described in an earlier publication[7] was discovered to be incorrect. The compiler described in that paper would translate a program with nested loops to a data flow program that could bring the machine into a state where two tokens of the same color reside at the same port. The machine that the compiler was intended for regards this situation as a fatal error called a token clash. More interesting than the incorrect graph itself is the fact that the error had been overlooked so easily. The graph contains not even twenty nodes, so the oversight can hardly be blamed on its size. It confirms the impression that graphs that manipulate colors or that dynamically create arcs are often more complicated than their appearance would suggest.

Before one can even begin to formulate properties, it is necessary to have a model of data flow programs and their execution. A few of these models have been reported in the literature. Rumbaugh, who was the first to publish a detailed description of a data flow machine, included in his thesis [6] an abstract model for a data flow language, which is closely tailored to the machine he designed. He was able to prove implications of the form: "A well nested-graph guarantees conflict-free execution." Brock [1] defined a graph assembly language to be able to describe data flow programs formally. This language can be seen as a limited but more readable variation of our model. He presents a translation function from a simple high level single assignment language to graphs and defines the semantics of his graphs. He is not concerned with safety, since he allows unbounded queues of tokens to collect on the data paths. Jaffe [4] discusses the effects of bounded queues on the parallelism of data flow programs. He presents a formal model, which resembles ours. The execution of his programs is more deterministic than ours, however, since in his model the order of tokens on a queue is preserved. All of these models describe the input programs for a data flow machine as directed graphs, with a node representing a machine instruction and an arc representing the possible flow of data from the ancestor to the descendant. Such models, however, do not allow the expression of input programs for computers like the Manchester Data Flow Machine [3] since they cannot handle the presence of colored tokens, the dynamic creation of data paths, non-deterministic merging of data paths and non-functional behavior of nodes.

We therefore decided to develop a more sophisticated model, which is intended to cover a large class of machines. The formulation of this model is the main topic of this paper. In addition a simple application of the model is presented. This application concerns the safety of a graph, which is the property that during execution no token clash can occur. For data flow machines without colors, safety implies that a data path holds at most one token. We prove that graphs that satisfy a few easily checked conditions are safe. These conditions are rather severe since they exclude all graphs that manipulate colors or contain cycles. We present a formal description of a translation algorithm from a simple conventional language to data flow graphs and prove that all graphs produced by the algorithm are safe.

After a few remarks on notation, this paper proceeds with the description of the model for a data flow program. Usually each formal definition is preceded by a, hopefully more readable, informal description. In section 3 the model for the execution of such a program is described and the safety concept is defined. Section 4 contains the theorems on safe graphs. Section 5 describes the translation algorithm and the proof that it produces safe graphs. The paper concludes with some suggestions for further research along the same lines. Full proofs of the lemmas can be found in the appendix.

2. DATA FLOW NETS

2.1. Notation

In this paper we will use the notion of bags. A *bag* or multiset is like a set but it allows multiple occurrences of identical elements. The ordering of the elements of a bag is irrelevant. The {...} notation will be used to denote a specific bag or a set. The notation $\varnothing$ will be used for both the empty set and the empty bag. Subbag, union ($\cup$) intersection ($\cap$) and difference ($-$) are defined on bags in the obvious way. [5] BAGS(T), where T is a set, denotes the set of all countable bags whose elements are in T. $|S|$, where S is a bag or a set, denotes the number of (different or identical) elements in S.

If $C \subseteq A$ and f a function defined on A then $f[C]$ denotes the image of C.
N denotes the set of natural numbers.

2.2. Informal Description

We will need a definition for a data flow program or graph. The obvious model would be a directed graph consisting of nodes and arcs. Tokens travel over the arcs and a function is associated with each node, that describes how tokens are transformed when they pass the node. Since we want to model dynamically created arcs this would not be a fruitful approach. Instead we define an entity called a *data flow net*, where a node is a primitive concept, but where the connection between the nodes will be dynamically determined by the functions that are associated with the nodes. To this end arcs will be replaced by more or less free-floating *ports*. The function associated with each node (henceforth called *firing rule*) describes which input tokens are taken from which ports, which output tokens are produced and to which ports they are added. The input and output ports of a node are thus defined by its firing rule. Nodes which can dynamically create arcs have as output ports all those ports to which they can possibly direct their output tokens. In many cases this will be the complete set of ports in the program. The following property is an important characteristic of our model: a port can serve as a shared output port for many different nodes but as input port to only one node. Because we want to include non-functional nodes, the firing rule can manipulate the *state* of a global memory. Because we also want to describe non-deterministic behavior the firing rule is not a function but a relation. Collections of tokens reside at the ports. Such a collection will be described by a bag. A tuple would not be appropriate since we want the model to be insensitive to the order in which the tokens on an arc are produced (this insensitivity corresponds to the asynchronous communication networks of data flow machines). A set would not be appropriate either because the model should be sensitive to multiple occurrences of the same token at a port.

We thus have the basic domains *tokens*, *states*, *nodes* and *ports*. *entry* and *exit*, two subsets of *ports*, will be used later to describe interaction with other data flow nets. The function *color* is used to differentiate between tokens. The function *prog* associates a firing rule with each node.

Although the definition is very general, some properties are built in. We already mentioned that the order in which tokens arrive at a port is not preserved. Exit ports do not serve as input ports to any node, so the tokens that reside at an exit port can be considered to have left the net. Input ports are not shared between nodes. Firings are modest but not spontaneous, i.e. of each input port at most one token is absorbed and there is at least one input port from which a token is absorbed. To include the notion of colored (or tagged or labeled) tokens we include in the definition a function *color*, which associates a color with each token, and put two additional restrictions on the firing rule. The first restriction is that for each element of a firing rule all input tokens should have the same color. The second restriction is that if the firing rule accepts tokens of one color it should accept tokens of all colors.

2.3. Formal Definition

A **data flow net** is a tuple

$< tokens, states, nodes, ports, entry, exit, color, prog >$

- $tokens$, $states$, $nodes$ and $ports$ are all disjoint and countable sets.
- $entry$ and $exit \subseteq ports$.
- $color: tokens \rightarrow \mathbf{N}$
- $prog : nodes \rightarrow$ sets of firing elements

  a **firing element** is a tuple

  $(<I, s>, <O, s'>)$

  with

  - $s, s' \in states$
  - $I : ports \rightarrow (tokens \cup \{\text{NIL}\})$
  - $O: ports \rightarrow \text{BAGS}(tokens)$

The set $prog(N)$ of firing elements for a node N is called the **firing rule** of N. Each firing element of a node represents a possible firing. The s and s' indicate the state of the global memory before and after the firing. I indicates from what ports what input tokens are absorbed; O indicates on what ports what output tokens are produced. For a particular firing element, $I(p)=t$ means that this firing removes token t from port p. If $I(p)=$ NIL no token is removed: p is not an input port for this element. Note that each firing can remove at most one token from a port. The range of O however is bags of tokens, so a firing may add more than one token to a port. If $O(p)$ is an empty bag than the firing does not add any tokens to p: p is not an output port for this element.

As an aid in the further description we first define four functions. For a firing element, $inel$ and $outel$ are the sets of ports from which a token is removed or to which tokens are added, respectively. For a port p, $I(p)$ is called an **input token** and elements of $O(p)$ are called **output tokens.** For a node, in and out are the unions of $inel$ and $outel$ over all firing elements. Elements of in and out are called **input** and **output ports.**

We define

- The functions $inel$ and $outel$ from the set of firing elements to sets of ports:

  $inel((<I, s>, <O, s'>)) = \{p \in ports \mid I(p) \neq \text{NIL}\}$

  $outel((<I, s>, <O, s'>)) = \{p \in ports \mid O(p) \neq \emptyset\}$
- The functions in and out from $nodes$ to sets of ports:

$$in(N) = \bigcup_{\{f \in prog(N)\}} inel(f)$$

$$out(N) = \bigcup_{\{f \in prog(N)\}} outel(f)$$

Certain conditions have to be satisfied before a tuple as described above deserves to be called a data flow net. So the restrictions given below are part of the definition.

1 Exit ports do not serve as input port to any node:

$\forall N \in nodes: \quad in(N) \cap exit = \emptyset$

2 Input ports are not shared between nodes:

$\forall N, K \in nodes: \quad in(N) \cap in(K) = \emptyset$, if $N \neq K$

We allow output ports to be shared between nodes. Such *knots* will be defined below.

3 Nodes cannot fire spontaneously:

$\forall N \in nodes$ and $f \in prog(N): \quad inel(f) \neq \emptyset$

4 All input tokens have the same color:

$\forall N \in nodes$ and $f \in prog(N)$

$\exists\ c \in \mathbf{N}$ such that $\forall p \in inel(f)$

$color(I(p)) = c$

f is said to have color c, so the color of a firing element is the color of its *input* tokens.

5 Whether a node will fire or not does not depend on the color of the input tokens as long as they are equal:

$\forall N \in nodes$ and $f = (<I_1, s_1>, <O_1, s'_1>) \in prog(N)$

$\forall c \in color[tokens]$

$\exists f' = (<I_2, s_1>, <O_2, s'_2>) \in prog(N)$ of color c with

$inel(f') = inel(f)$

EXAMPLE

Throughout this paper we will use an almost trivial data flow net as the standard example. The net has three nodes, four entry ports and one exit port. In the traditional way the net is modeled by the graph

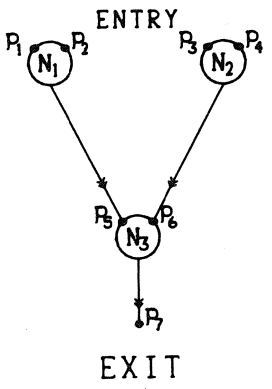

Figure 1
SIMPLENET

in figure 1. The structure of the graph is identical to that of a program that would compute $(a+b)\times(c+d)$, but for the sake of simplicity we will assume that each node sends a copy of one of its input tokens to its output ports. The definition of the net is as follows:

SIMPLENET $= <tokens, states, nodes, ports, entry, exit, color, prog>$

where

- *tokens* is a countable set
- *states* $= \{s_0\}$
- *nodes* $= \{N_1, N_2, N_3\}$
- *ports* $= \{p_1, \ldots, p_7\}$
- *entry* $= \{p_1, p_2, p_3, p_4\}$
- *exit* $= \{p_7\}$
- *color* $: tokens \rightarrow \{0\}$
- *prog* $= \left| \begin{array}{l} N_1 \rightarrow [\text{COPY FROM } \{p_1, p_2\} \text{ TO } \{p_5\}], \\ N_2 \rightarrow [\text{COPY FROM } \{p_3, p_4\} \text{ TO } \{p_6\}], \\ N_3 \rightarrow [\text{COPY FROM } \{p_5, p_6\} \text{ TO } \{p_7\}]) \end{array} \right.$

The shorthand notation

[COPY FROM *inports* TO *outports*]

is used to indicate a firing rule in which for each element

$f = (<I, s_0>, <O, s_0>)$

1 - $inel(f) = inports$

2 - $outel(f) = outports$

3 - Each output port receives exactly one copy of one of the input tokens.

2.4. Properties

In the sequel we will simply talk about a net when we mean a data flow net. In this section we will define certain special properties of a particular net *NET*. Most of these properties we will need later when we prove theorems.

The notions of *static* and *dynamic* arcs are described. We further define that a node is *strict* if it always absorbs one token from each of its input ports. Since firings are not spontaneous, nodes with only one input port are always strict. A *knot* is introduced, which is a port that is a destination for more than one node. For loops and the merging of the branches of a condition the net either needs to have knots or to include non-strict nodes. In order to simplify the separation between matching store and instruction store the designers of the Manchester Data Flow Machine have chosen for strict nodes and knots. Most other designers have made the other choice. Data flow nets can model both designs.

We further give a definition of *functionality* and *determinism*. These two notions have frequently been confused in the data flow literature. This is not at all surprising, since the input-output behavior of a node does not give enough information to deduct whether the node is either non-functional or non-deterministic, unless one is able to initialize or inspect the state of the global memory.

A node N is called

- **unitary** if each firing adds at most one token to a port:

  $\forall p \in ports: \quad |O(p)| \leq 1$
- **static** if each firing adds tokens to each output port:

  $\forall f \in prog(N): \quad outel(f) = out(N)$
- **dynamic** if it is not static
- **strict** if each firing removes one token from each input port:

  $\forall f \in prog(N): \quad inel(f) = in(N)$
- **color preserving** if all output tokens have the same color as the input tokens:

  $\forall f = (<I, s>, <O, s'>) \in prog(N), p \in outel(f)$ and $t \in O(p)$

  t and f have the same color
- **deterministic**, if the firing rule is a function.

  This means that there is only one firing for each combination of state and input tokens. The behavior of non-deterministic nodes is influenced by factors that lie outside of our model, such as the creativity of an interactive user or the order of updates in a shared data base.
- **functional**, if $\forall (<I, s>, <O, s'>) \in prog(N): \quad s = s'$

  This means that the node has no effect on the state of the global memory. For functional nodes we will often omit all occurrences of states and we will denote a firing element of such a node simply with $<I, O>$.

A port p is called

- a **knot** if it is an output port for more than one node:

  there are different nodes N and K such that $p \in (out(N) \cap out(K))$

2.5. Example

We can make the following statements about *SIMPLENET*:
- All nodes are strict, unitary, static, deterministic, functional and color preserving.
- The net is free of knots.

3. MARKINGS

We will describe the execution of a net as a collection of state transitions. To prevent confusion with the state of the global memory and to emphasize the analogy with Petri nets we will call the state of a net a *marking*. A marking assigns bags of tokens to ports and a state to the global memory. A *firing* of a node is a pair of markings, such that the transition from the first marking to the second is consistent with the firing rule of the node. This is equivalent with saying that a firing of a node transforms one marking into another by removing tokens from input ports of the node, adjusting the state of the global memory and adding the result tokens to output ports. An *execution path* is a series of markings in which each subsequent pair is a firing.

3.1. Definitions

A **marking** M of NET is a pair $<M_{state}, M_{tokens}>$
- $M_{state} \in states$
- $M_{tokens}: ports \rightarrow$ BAGS$(tokens)$

If $p \in ports$ we will use the notation $M(p)$ as shorthand for $M_{tokens}(p)$.

A **firing** of node N of NET is a pair $<M, M'>$
where M and M' are markings of NET such that
$\exists$ firing element $f = <(I, s), (O, s')> \in prog(N)$ with
1 - $M_{state} = s$
2 - $M'_{state} = s'$
3 - $\forall p \in inel(f)$
- $I(p) \in M(p)$ (the enabling condition)
- $M'(p) = M(p) \cup O(p) - \{I(p)\}$

4 - $\forall p \in (ports - inel(f))$
$M'(p) = M(p) \cup O(p)$

s and s' are the states of the global memory before and after the firing. The input tokens are removed from input ports and the output tokens added to output ports. Note that for most ports p, $O(p) = \emptyset$. We will say that $<M, M'>$ has the same color as f.

An **execution path** is a series of markings $<M_1, \ldots, M_n>$ where
- $n > 1$
- $\forall_{i<n}$, $<M_i, M_{i+1}>$ is a firing of a node $\in NET$

3.2. Properties of Markings

Now that we can describe the execution of a net we can formulate dynamic restrictions for a certain machine by declaring certain markings illegal. Among the interesting restrictions are:

Freedom from Deadlock

The net cannot get into a state from which no more output can be produced.

Cleanliness

Each series of firings will eventually leave all non-exit ports of the net empty.

Deterministic Execution

For each set of inputs there is only one set of outputs.

Safety

The net cannot get into a state where more than one token of the same color resides at a port. If all tokens have the same color this degenerates to the restriction that a port contains at most one token.

In order to prove that certain nets obey these restrictions we will first have to define these notions precisely. In this paper we will concentrate on safety. We define a *reachability* relation between markings and we call a net *safe* if no unsafe marking is reachable from any safe input marking.

A marking M of a net *NET* is called

- **safe with respect to** p, if p is a port such that
  there are no two tokens $\in M(p)$ of the same color
- **safe**, if $\forall p \in ports$
  M is safe with respect to p
- **input**, if there are only tokens on entry ports:
  $\forall p \in (ports - entry)$
  $M(p) = \varnothing$
  So we do not require that there are tokens on all entry ports.
- **output**, if there are only tokens on exit ports:
  $\forall p \in (ports - exit)$
  $M(p) = \varnothing$
- **reachable** from a marking M', if
  $\exists$ execution path $<M',..., M>$
  Note that the reachability relation between markings is not necessarily reflexive.

A net is called

- **safe**, if for all safe input markings M_I, all markings M that are reachable from M_I are safe.

It may come as a surprise that in a model for such a parallel machine as a data flow machine, no mention is made of parallel execution. It turns out however that due to the enabling condition and the asynchronous nature of data flow nets, sequentializing the parallel firing of nodes does not influence the set of reachable markings. This is proven in appendix I.

3.3. Example

In the following descriptions of markings of *SIMPLENET* the component M_{state} and the mappings from ports to empty bags are omitted.

- $M_{in} = (p_1 \rightarrow \{t_1\}, p_2 \rightarrow \{t_2\}, p_3 \rightarrow \{t_3\}, p_4 \rightarrow \{t_4\})$ is a safe input marking
- $M_1 = (p_3 \rightarrow \{t_3\}, p_4 \rightarrow \{t_4\}, p_5 \rightarrow \{t_1\})$ is reachable from M_{in} since $<M_{in}, M_1>$ is a firing of N_1.
- $M_2 = (p_7 \rightarrow \{t_1\})$ is an output marking reachable from M_{in}.

4. THEOREMS

In this section we will prove that nets that satisfy certain conditions are safe. This will be done in two steps. In the first step nets will be considered that are free of knots, only contain unitary, strict, color preserving nodes and in which no entry port serves as output port to any node (i.e. tokens do not enter the net on a path between nodes). The most severe of these restrictions is the absence of knots and non-strict nodes. This combination outlaws dynamic nodes. Loops, conditional flow and procedure calls can therefore not be implemented by these nets. Simple programs without these constructs can however be modeled. In the second step nets are considered for which the restriction on knots is somewhat relaxed and which allow conditional flow.

For loops and procedure calls, the restriction on color preservation has to be relaxed, but this will be a topic for future research. In the next section we will describe a translation scheme from a simple multiple assignment language into nets that (not by coincidence) satisfy exactly these conditions.

For lemmas only a proof outline is given. Complete proofs can be found in Appendix II.

4.1. Nets without Knots

We first define a partial distance function from a set of ports to descendant nodes:

If S is a set of ports and N a node than

$$dist(S,N) = \left| \begin{array}{l} 0, \text{ iff } in(N) \subseteq S \\ n, \text{ iff } n \text{ is the smallest integer such that} \\ \quad \forall p \in (in(N) - S), \exists L \in nodes \text{ such that} \\ \quad 1\ p \in out(L) \\ \quad 2\ dist(S,L) < n \end{array} \right.$$

If for some p there is no such L than $dist(S,N)$ is undefined.

A node N is *covered* by a set of ports S if $dist(S,N)$ is defined.

Intuitively, this distance $dist(S,N)$ is equal to the minimum number of firings that is needed before tokens on a port in S have percolated to all input ports of N. If tokens are present at all ports of S but at no other ports and all nodes are strict than only those nodes that are covered by S will eventually fire. This is formalized in the following lemma:

Lemma A1:

If M_1 is a marking of a net with only strict nodes than for each node K holds
if there is an execution path $<M_1, \ldots, M_m>$ with $<M_{m-1}, M_m>$ a firing of K
then K is covered by $\{ p \mid M_1(p) \neq \varnothing \}$

Proof outline:

Let $<M_1, \ldots, M_m>$ be an execution path with $<M_i, M_{i+1}>$ a firing of node N_i and $S = \{ p \mid M_1(p) \neq \varnothing \}$. By induction on i the following properties are proven

A - $dist(S,N_i) < i$

B - $\forall K \in nodes: \ dist(S,K) > i \implies \exists p \in in(K)$ with $M_{i+1}(p) = \varnothing$.

The basis of the induction (firing of N_1) is trivial.

The induction step considers the firing of N_{n+1} assuming that properties A and B hold for $i \leqslant n$.

Induction assumption B combined with the strictness of N_{n+1} implies property A for $i = n+1$.

A node is then considered which has all its input ports loaded in M_{n+2}. It follows from property A for $i \leqslant n+1$ that the distance of this node to S is less than $n+2$. This implies property B for $i = n+1$.

Property A implies that all N_i are covered by S. □

In a strict, knot-free net there are interesting correlations between *dist* and the execution of the net on a parallel data flow machine. The maximum of $dist(entry, N)$ is a measure of the minimum execution time of the program. The maximum number of nodes with the same $dist(entry, N)$ is an indication of the maximum number of processors that can be used concurrently.

In the proof of the above mentioned theorem we will make extensive use of the property of a net that for each safe input marking each node will fire each color at most once. We define this formally:

For a node K in NET, $ONCE(K)$ holds iff
there is no safe input marking M_I, and execution path $<M_I, \ldots, M>$
with two separate firings of K with the same color.

The following lemma states that all nodes of nets of the type we are considering will fire each color at most once.

Lemma A2:
If NET is such that
1 NET is free of knots
2 all nodes are strict, unitary and color preserving
3 $\forall K \in nodes: out(K) \cap entry = \varnothing$
than $ONCE(K)$ holds $\forall K \in nodes.$
Proof outline:
The proof is split into the following steps:
1 - According to lemma A1 nodes not covered by *entry* will not fire for any input marking.
2 - $ONCE(K)$ holds for all remaining nodes by induction on $dist(entry, K)$.

- a For all nodes K with only entry ports $ONCE(K)$ holds. This follows from restriction 3 and the strictness of K.
- b Suppose $ONCE(K)$ and $dist(entry, K) \leqslant n$. Now consider a node L with $dist(entry, L) = n+1$. Suppose $ONCE(L)$ does not hold. Because L is strict, at least two different tokens of the same color must have passed through one of its input ports. Since the net is free of knots, both tokens must have been produced by one node N. It follows that $dist(entry, N) \leqslant n$. But since N is unitary and color preserving, it must have fired at least one color twice, contrary to the induction hypothesis.□

We now get to the safety theorem for simple, tree-like nets:

Theorem A:
If NET is such that
1 - NET is free of knots
2 - all nodes are unitary, strict and color preserving
3 - $\forall K \in nodes: out(K) \cap entry = \varnothing$
than NET is safe.
Proof:
Let M be a marking of NET reachable from some safe input marking M_I . Let q be a port.
If $q \in entry$ than
q is not an output port to any node, so $M(q) \subseteq M_I(q)$. This implies that M is safe with respect to q.
If $q \not\in entry$ than
Since NET is free of knots, there is exactly one node K such that $q \in out(K)$. Since
- $ONCE(K)$ holds according to lemma A2
- K is unitary and color preserving

M is safe with respect to q.
So M is safe.□

It is interesting to note that this theorem is equally valid for nets with non-functional, non-deterministic and dynamic nodes as long of course the dynamic arcs do not end in knots.

4.2. Nets with Conditional Constructs

We will now prove a second theorem, which relaxes the restriction on knots somewhat. These nets allow the kind of data flow programs that are usually suggested to implement conditional statements.[2,6,7,8] We will need an additional tool to describe the structure of a net, with the effect that if we can call a node *insulated* from a set of ports by a second set of ports than no token residing in a port in the first set can reach the node without passing through a port in the second set. We define this structural concept as follows:

A node N is **insulated** from the set of ports A by the set of ports B iff for all input ports p of N

1 - $p \not\in A$

2 - N covered by B

3 - Either $p \in B$ or $\forall L \in nodes: p \in out(L) \Rightarrow L$ is insulated from A by B

Note that insulation implies coverage. The intended property of insulation is embodied in the following lemma, which states that if in a color preserving net a node that is covered by a set of ports fires then a token of the same color must have passed through one of those ports.

Lemma B1:

For each execution path $EP = <M_1, \ldots, M_m>$ and set of ports S in a color preserving net with

- M_1 an input marking

- $<M_{m-1}, M_m>$ a firing with color C of a node insulated by S from *entry*

$\exists M_i \in EP$ and a $p \in S$, such that $M_i(p)$ contains a token of color C.

Proof outline:

For each node K it is proven that for some $p \in S$, $M_i(p)$ contains a token with color C by induction on $dist(S,K)$.

The basis of the induction is trivial.

In the induction step nodes K with $dist(S,K) = n+1$ are considered. It is shown that K must have received a token of color C from some node L. Since $dist(S,L) \leqslant n$ the induction hypothesis implies the correctness of the lemma.□

For the description of conditional nets the concept of a *multiswitch* is introduced, which is a set of switching nodes controlled by a common node. The intuitive function of such a multiswitch is to send its input tokens to one of two sets of ports (called *destination sets*), depending on the output of the controlling node. A knot is *synchronized* by such a multiswitch, if it is an output port of two nodes, each of which is insulated by one of the two destination sets. The theorem then states that nets with only such synchronized knots and which also satisfy restrictions 2 and 3 of theorem A are safe. Theorem A of course follows directly from this second theorem.

$MS \subset nodes$ is called a **multiswitch** with **control node** CN and **destination sets** A and B, iff

1 - ports in the destinations sets can receive tokens only through the multiswitch:

$\forall p \in (A \cup B)$ and $L \in nodes$:

$p \in out(L) \Rightarrow (L \in MS$ or L insulated from *entry* by A or $B)$

2 - All tokens produced by a firing of the control node are identical:

$\forall f \in prog(CN), \exists t \in tokens$ such that $\forall p \in outel(f)$: $O(p) = \{t\}$

3 - Each node of the multiswitch has a control port which only receives tokens from the control node. The tokens on these control ports determine the destination set to which all output tokens are sent:

$\exists$ **true** $\subseteq tokens$, such that $\forall N \in MS$ and $f \in prog(N), \exists p \in inel(f)$, such that

$$1 - outel(f) = \left| \begin{array}{l} \subseteq A, \text{ if } I(p) \in \textbf{true} \\ \subseteq B, \text{ otherwise} \end{array} \right.$$

2 - $\forall K \in nodes$: $p \in out(K)$ iff $N = CN$

Note that A or B may be empty.

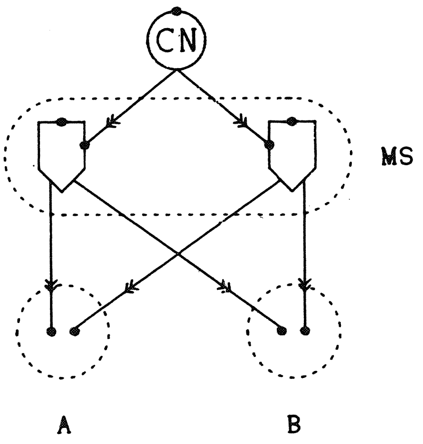

Figure 2
A Multiswitch

An example of a multiswitch is depicted in figure 2.

A knot p, which is output port for exactly two nodes K and L, is called **synchronized** by MS iff
1 - MS is a multiswitch with destination sets A and B
2 - (K insulated by A from *entry*) or ($K \in MS$)
3 - (L insulated by B from *entry*) or ($L \in MS$)

The following lemma is a relaxed form of lemma A2. It states that if all knots are synchronized than each node will still fire each color only once.

Lemma B2:
If *NET* is such that
1 - each knot is synchronized by a multiswitch
2 - all nodes are unitary, strict and color preserving
3 - $\forall K \in nodes: out(K) \cap entry = \emptyset$
then $ONCE(K)$ holds for all nodes K.
Proof outline:
The proof is identical to that of lemma A2 with the addition of the special case that an input port of node K is a knot. Since this knot is synchronized by a multiswitch it must be an output port for two different nodes that are insulated by the destination sets of the multiswitch. According to lemma B1 tokens of the same color must have passed through both destination sets. This implies that the control node of the multiswitch must have fired one color twice, contrary to the induction hypothesis.□

Theorem B:
If *NET* is such that
1 - each knot is synchronized by a multiswitch
2 - all nodes are unitary, strict and color preserving
3 - $\forall K \in nodes: out(K) \cap entry = \emptyset$
then *NET* is safe.
Proof:
The proof is identical to that of theorem A with the addition of the proof for the case that q is a knot synchronized by a multiswitch with control node *CN*. The same argument as used in the previous proof implies that non-safety of q would imply that $ONCE(CN)$ does not hold. This would contradict lemma B1.□

5. TRANSLATION

As an illustration of the use of the model we will describe in this section the translation of a conventional program into a data flow net and we will prove that it will only produce safe nets. For the sake of brevity the description is kept slightly less formal than in the previous chapters. The language considered is a very simple version of an expression oriented programming language with variables and (multiple) assignments. The basic algorithm is described in section 5.1 and proved to be safe in 5.3. An extension of the algorithm that can handle conditional constructs is treated in the last two sections.

Since we are not interested here in the process of parsing we simply assume that the program is available in the form of a parse tree. Each node of the parse tree corresponds to a (sub)expression of the program and the root corresponds to the complete program. The translation algorithm defines a function T, which associates a data flow net with each node of the parse tree and thus with each (sub)expression of the program. The net associated with the root of the parse tree is thus the translation of the program. Each (sub)expression in the program delivers a value, which will be available in the *exit* port of the corresponding data flow net.

The algorithm defines two more functions *DEF* and *USE*, which are concerned with multiple assignment. They keep track of the places where variables are defined and used, such that each use can be correctly connected to the corresponding definition:

DEF associates with an expression the output bindings that are defined after the evaluation of the expression. An output binding is an association between a variable and the port that will hold the value of that identifier. An output binding corresponds to an assignment in the program.

USE makes similar associations, but this time for input bindings: associations between a variable and the ports that need to receive the value of the variable. An input binding corresponds to the occurrence of a variable on the right hand side of an assignment.

5.1. Basic Translation Algorithm

Let us first consider the translation of a very simple language into data flow nets. The syntax of the language is:

*SYNTAX*1:
```
    <expr> ::= <identifier>                 |
               <identifier> := <expr>       |
               <expr> <op> <expr>           |
               ( <expr> )                   .
    <op>   ::= ; | + | * | - | / | =        .
```

The semantics of the binary operators are irrelevant in the present context since our prime concern is with the safety of the generated nets. We assume an expression oriented language so we can treat the semicolon as an ordinary binary operator. We assume that for each program in this language a parse tree is available with three types of nodes:

(1) Leaves that represent a reference to a variable on the right hand side of an assignment (a *use*).

(2) Nodes that represent an assignment (a *definition*). Such nodes have only one descendant, viz. the node that corresponds to the right hand side of the assignment.

(3) Nodes that represent a *binary operation*. Such nodes have two descendants, corresponding to the two subexpressions that are connected by the binary operator.

An example of a parse tree is shown in figure 3. The type of each node is indicated to its right.

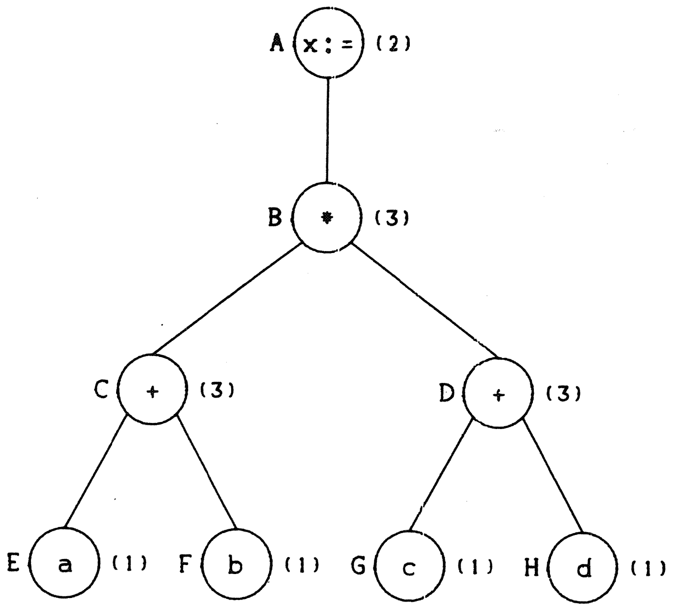

Figure 3
Parse tree of x := (a+b) * (c+d)

The sets *tokens*, *states* and *color* are identical for all nets and will not be defined any further. For each net the set *entry* is the image of the function *USE*: each entry port corresponds to a use of a variable that is not yet linked to its appropriate definition. The translation function T and the bookkeeping functions *DEF* and *USE* are defined with the aid of so called **templates**. There is one template for each type of parse tree node:

(1) In the template for the use of a variable (figure 4), T delivers a net with no nodes and only one port. This port is meant to receive the value of the variable and is bound in *USE* to the variable name to indicate that later a data path is to be created from the most recent definition. This port is also the *exit* port of the expression.

TEMPLATE (X):

$$T(N).exit = T(N).ports = \{p\}$$

$$T(N).nodes = \varnothing$$

$$T(N).prog = DEF(N) = \text{undefined}$$

$$USE(N)(Y) = \begin{cases} \{p\}, \text{ if } X = Y \\ \varnothing \text{ , otherwise} \end{cases}$$

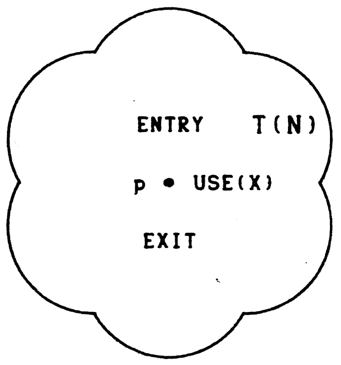

Figure 4
Net generated by template (1)

(2) The template for the definition of a variable is illustrated in figure 5. *T* delivers a net that is the net corresponding to its only descendant plus one new node and two new ports. The new node copies the value delivered by the right hand side to the two new ports. One of these is the *exit* port of the expression. The other port is bound in *DEF* to the variable name and will serve as the input port of a node that is to be created later to deliver the value to subsequent uses of the variable. These definitions and uses will be linked by the next template. The reason that two ports have to be created is that input ports cannot be shared between nodes.

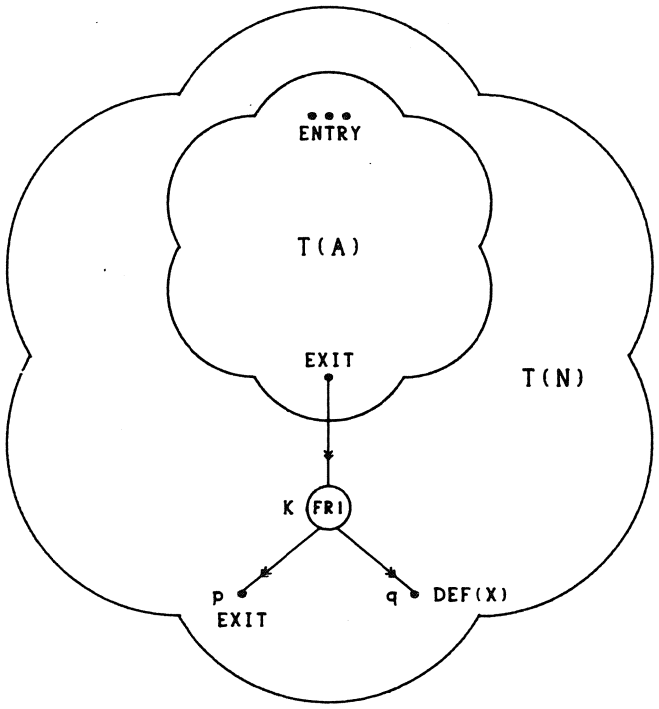

Figure 5
Net generated by template (2)

TEMPLATE $(X:=)$ with descendant node A:

$T(N).ports \;=\; T(A).ports \;\cup\; \{p, q\} \qquad p, q \not\in T(A).ports$

$T(N).nodes \;=\; T(A).nodes \;\cup\; \{K\} \qquad K \not\in T(A).nodes$

$T(N).prog \;=\; T(A).prog \;\cup\; (K \rightarrow FR\,1)$

where $FR\,1$ = [COPY FROM $T(A).exit$ TO $\{p, q\}$]

$T(N).exit \;=\; \{p\}$

$USE(N) \;=\; USE(A)$

$$DEF(N)(Y) \;=\; \begin{cases} \{q\} & \text{, if } X=Y \\ DEF(A)(Y) & \text{, otherwise} \end{cases}$$

(3) Figure 6 illustrates the binary operator template. Here T delivers a net that is the union of the nets corresponding to the two descendant nodes plus a series of new nodes and ports. One new node (K) implements the binary operator. It has the exit ports of the two descendants as input ports and one new port (p) as output port. This last port is the exit port of the expression. Since within the scope of this paper, we are not interested in the semantics of the translated program, we simply program K with a COPY firing rule. Programming it with a more realistic firing rule would not affect the safety of the net. The sets LN, LP and LF denote the nodes, ports and firing rules that are created to link corresponding definitions and uses of a variable in the two descendants. These sets will be specified later.

TEMPLATE $(<op>)$ with descendant nodes A and B:

$T(N).ports \;=\; T(A).ports \;\cup\; T(B).ports \;\cup\; LP \;\cup\; \{p\}$ (these four sets are disjoint)

$T(N).nodes \;=\; T(A).nodes \;\cup\; T(B).nodes \;\cup\; LN \;\cup\; \{K\}$ (these four sets are disjoint)

$T(N).prog \;=\; T(A).prog \;\cup\; T(B).prog \;\cup\; (LN \rightarrow LF) \;\cup\; (K \rightarrow FR\,2)$

where $FR\,2$ = [COPY FROM $(T(A).exit \;\cup\; T(B).exit)$ TO $\{p\}$]

$T(N).exit \;=\; \{p\}$

Since we are dealing with an expression oriented language, definitions and uses of variables can occur in both descendants and the semicolon can be treated the same as any other binary operator. The major part of this template is therefore concerned with the creation of the links between corresponding definitions in the left operand and uses in the right operand. Readers not accustomed to expression oriented languages might prefer to think of the expression "A ; B". For each link to be established a node in LN and a port in LP is created. The functions P and M associate port and node with the corresponding variable name. The linkage node has as input port the binding in DEF and as output ports the bindings in USE and the newly created port. The latter is now bound to the variable name in DEF instead of the old binding to prevent that a port will be used as input port to more than one node. The corresponding bindings in USE are removed.

LN is a set of nodes

LP is a set of ports

LF is a set of firing rules

such that,

1 - there is a partial function P from identifiers to LP

for each identifier X, $P(X)$ is defined iff $DEF(A)(X)$ and $USE(B)(X)$ are defined

2 - there is an injective function M from LN to identifiers

$\forall L \in LN$

1 - $P(M(L))$ is defined

2 - $prog(L) \;=\; FR\,3 \in LF$

where $FR\,3$ = [COPY FROM $DEF(A)(M(L))$ TO $(USE(B)(M(L)) \;\cup\; \{P(M(L))\})$]

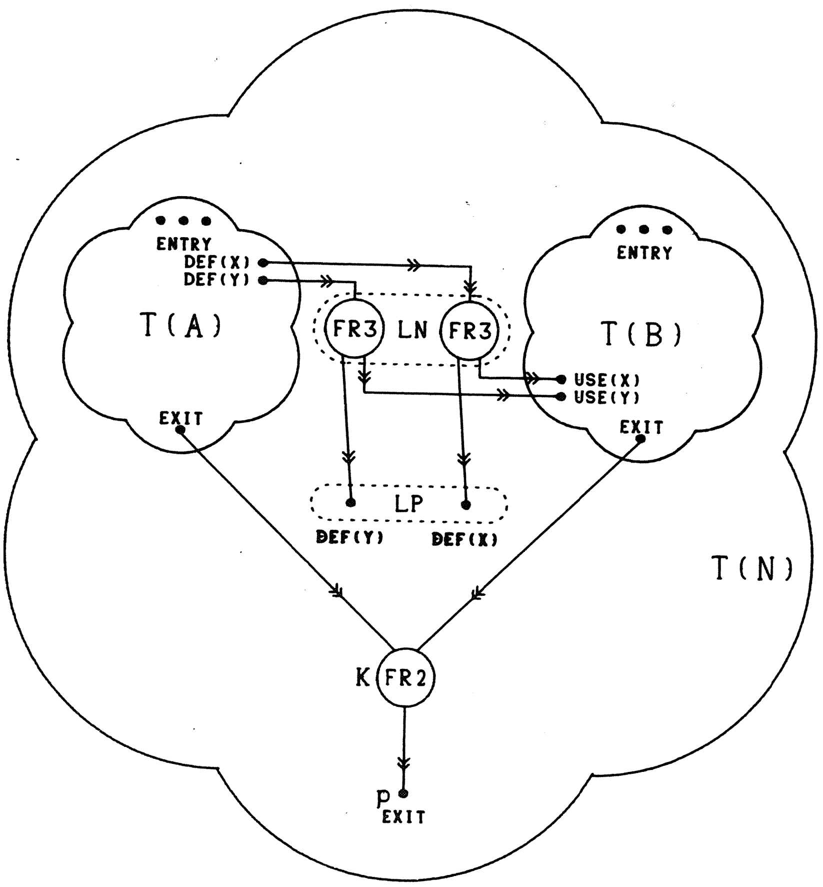

Figure 6
Net generated by template (3)

$DEF(N)$ and $USE(N)$ are such that for all identifiers X

$$1 - DEF(N)(X) = \begin{cases} P(X) & \text{, if } P(X) \text{ defined and } DEF(B)(X) = \varnothing \\ DEF(B)(X) & \text{, if } DEF(B)(X) \neq \varnothing \\ DEF(A)(X) & \text{, otherwise} \end{cases}$$

$$2 - USE(N)(X) = \begin{cases} USE(A)(X) & \text{, if } P(X) \text{ defined} \\ USE(A)(X) \cup USE(B)(X) & \text{, otherwise} \end{cases}$$

5.2. Example

SIMPLENET, depicted in figure 1, is a translation of the program $(a+b)\times(c+d)$, which corresponds to node B in figure 3. The parse tree contains four leaves corresponding to the uses of the variables and three nodes for the three binary operators. At the leaves template (1) is applied, which leads to the creation of ports p_1, p_2, p_3 and p_4. At parse tree node C template (3) is applied, which creates node N_1 with p_1 and p_2 as input port and p_5 as output port. At parse tree node D node N_2 and port p_6 are created. At node B finally N_3 and exit port p_7 are added.

5.3. Proof of Safety

Theorem C:

For each program P generated by $SYNTAX1$, $T(P)$ is safe.

Proof:

The proof is a straightforward application of Theorem 1. It is easy to see that for each node N of the parse tree, $T(N)$ satisfies the three groups of conditions of the theorem:

1 - $T(N)$ is free of knots. $FR\,1$ and $FR\,2$ define only new ports as output ports so they cannot produce a knot. The output ports defined by $FR\,3$ are either in the image of P, which is LP and consists only of new ports, or in USE. To prove that the latter case does not produce a knot we prove that each template leaves the following properties invariant:

I1 - $USE(N)$ contains only bindings to ports that do not serve as output port to any node.

I2 - in $USE(N)$ no two identifiers are bound to the same port.

2 - The only firing rules mentioned in the templates are those of unitary, strict, color preserving nodes.

3 - The only entry ports that are created are ports that appear in USE and, according to invariant I1, USE contains only bindings to ports that are not output port to any node. □

5.4. Translation of Conditionals

We will extend the translation function T to one that can handle a language that includes conditional statements. The syntax for this new language is:

$SYNTAX2$:

```
<expr> ::= <identifier> | <identifier> := <expr> |
        <expr> <op> <expr> | ( <expr> ) |
        if <expr> then <expr> fi     .
<op>  ::= ; | + | * | - | / | = .
```

An expression of the form '**if** A **then** B **fi**' is equivalent with the expression '*value-of-condition* $:= A$; **if** *value-of-condition* **then** B **fi**' where *value-of-condition* is a reserved identifier. Without loss of generality we can therefore assume that the parse tree contains in addition to the three types of nodes mentioned in the previous section only one more type:

(4) A node that represents the simplified condition (**if** *value-of-condition* **then**). This node has as only descendant the node that corresponds to the **then** branch.

The templates for the first three types of nodes remain the same and we only have to define the template for the conditional node.

(4) In this template two sets of ports DP and CP and a set of nodes SN are created. Each such node acts as a switch node with an element of CP as the control port and an element of DP as the data input port. For each identifier that is used or defined in the **then** branch of the condition one such switch node is created. The functions SDP, SCP and S define the correspondence between ports and nodes on the one hand and identifiers on the other hand. The new nodes are programmed with a SWITCH type firing rule such that if *value-of-condition* is **true** the tokens arriving at ports in DP will be send to the USE ports of the **then** branch and otherwise to the DEF port. At this latter port a knot is formed that is synchronized by the switch node.

We will use the notation

[SWITCH CONTROLLED BY *control-port* FROM *data-in-port*
IF TRUE TO *true-out-ports* IF FALSE TO *false-out-ports*]

to indicate a firing rule of a dynamic node in which for each firing element f:

1 - $inel(f) = \{control\text{-}port, data\text{-}in\text{-}port\}$

2 - $outel(f) = \begin{cases} true\text{-}out\text{-}ports & \text{, if } I(control\text{-}port) \in \textbf{true} \\ false\text{-}out\text{-}ports & \text{, otherwise} \end{cases}$ (a specific subset of *tokens*)

3 - Each output token is a copy of $I(data\text{-}in\text{-}port)$

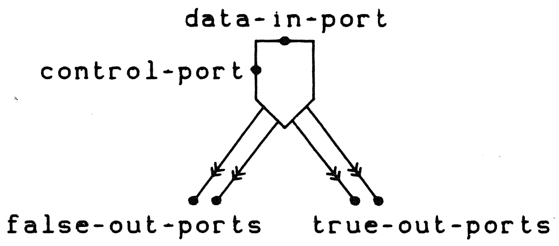

Figure 7
SWITCH firing rule

TEMPLATE (if *value-of-condition* **then**) with descendant node A:

$T(N).ports = T(A).ports \cup DP \cup CP$ (these three sets are disjoint)
$T(N).nodes = T(A).nodes \cup SN$ (these two sets are disjoint)
$T(N).prog = T(A).prog \cup (SN \to SF)$
$T(N).exit = T(A).exit$

DP and CP are sets of ports
SN is a set of nodes
SF is a set of firing rules
such that,
1 - there are two partial functions SDP and SCP from identifiers to DP and CP respectively
For each identifier X:
$SDP(X)$ and $SCP(X)$ are defined iff $USE(A)(X)$ or $DEF(A)(X)$ is defined
2 - There is an injective function S from SN to identifiers
$\forall N \in SN$
1 - $SDP(S(N))$ and $SCP(S(N))$ are defined
2 - $prog(N) = FR4 \in SF$
where
$FR4 =$ [SWITCH CONTROLLED BY $SCP(S(N))$ FROM $SDP(S(N))$
IF TRUE TO $USE(A)(S(N))$ IF FALSE TO $DEF(A)(S(N))$]
Note that $USE(A)(S(N))$ or $DEF(A)(S(N))$ can be empty.

$DEF(N) = DEF(A)$

$$USE(N)(X) = \begin{cases} \{SDP(X)\} & \text{, if } SDP(X) \text{ defined} \\ CP & \text{, if } X = value\text{-}of\text{-}condition \\ \emptyset & \text{, otherwise} \end{cases}$$

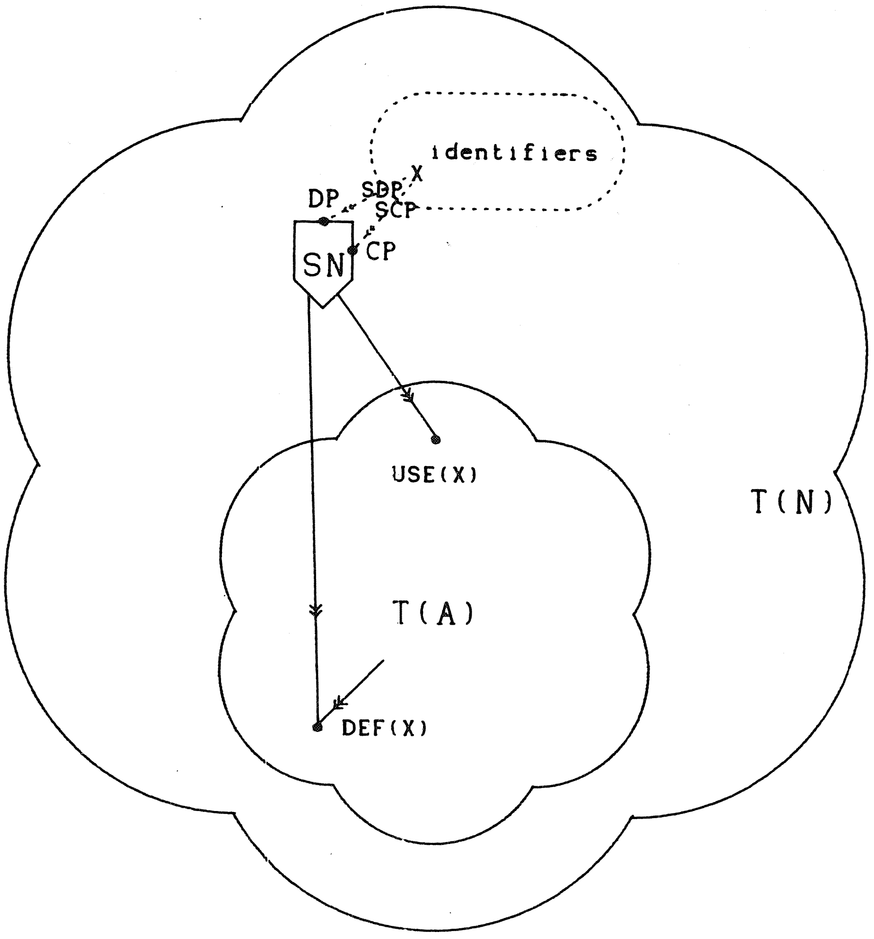

Figure 8
Net generated by template (4)

5.5. Proof of Safety

Theorem D:

For each program P generated by $SYNTAX2$, $T(p)$ is safe.

Proof:

This proof is a straightforward application of theorem B.

1 - Each knot is synchronized by a multiswitch. In the proof of theorem C we already showed that firing rules $FR\,1$, $FR\,2$ and $FR\,3$ produce no knots and that the templates (1), (2) and (3) leave properties I1 and I2 invariant. It is easy to see that the new template (4) also leaves I1 and I2 invariant. $FR\,4$ however may produce a knot on port $DEF(A)(S(N))$. We will prove that the set of nodes SN is a multiswitch that synchronizes this knot:

Consider template (4). We can prove that each node in $T(A)$ is insulated by $USE(A)$ from $USE(N)$ and eventually from *entry*. This follows directly from a third property

I3 - each node in $T(N)$ is covered by $USE(N)$

which can easily be seen to be left invariant by all four templates.

The direct ancestor of a conditional node is a ; binary operator node which creates the linkage nodes between DEFs in the condition part and USE in the **then** branch. It can be seen that the linkage node VOC which links definition and uses of *value-of-condition* turns SN into a multiswitch with destination sets $USE(A)$ and $DEF(A)$ and control node VOC, since it fulfills the three properties of a multiswitch:

1 - Ports in $USE(A)$ are output port only of nodes in SN. Ports in $DEF(A)$ are output ports of nodes in SN or of nodes in $T(A)$ which are all insulated from *entry* by $USE(A)$ as we have seen.

2 - The control node VOC is programmed with a $COPY$ firing rule.

3 - Ports in CP are output port only for the control port. The other components of this property follow directly from the definition of the $SWITCH$ firing rule.

Say $p = DEF(A)(S(N))$ is a knot since it is output port for some node $K \in SN$ and some node $L \in T(A)$. Since L is insulated by $USE(A)$ from *entry* it follows that p is synchronized by SN.

2 - All firing rules are unitary, strict and color preserving.

3 - *entry* ports are always in USE and these are not output port for any node. □

6. DISCUSSION

We have formulated a general model for data flow programs that can serve as a vehicle for the study of unresolved issues concerning token coloring and dynamically created data paths. We have shown that rigorous proofs can be given for simple theorems. Although the proofs are often cumbersome they have helped to gain some insight in fundamental properties of safe data flow graphs. We have used the model to validate a non-trivial translation algorithm.

The main value of the work reported in this paper lies in the formulation of the model, since equivalent versions of the theorems that we have proven so far have been given elsewhere. This work is intended to lay a sound foundation for more interesting analysis. We mention a few directions into which the work can be fruitfully extended.

- The types of graphs that are proven to be safe need to be extended. So far we have limited ourselves to color preserving graphs, with a severely limited type of dynamic arcs. Restrictions need to be formulated on the way a net manipulates colors and creates dynamic arcs. These restrictions should be simple enough to make a rigorous proof feasible, but they should not be so stringent as to exclude common and sound translation schemes.
- The only property that has been considered so far is safety. Other properties of nets such as freedom from deadlock, cleanliness and deterministic execution also merit attention. We expect the analysis for these properties to proceed basically along the same lines as that for safety.
- For deadlock free and deterministic nets the function that maps input markings to output markings can be called the meaning of the net. If a meaning function for source programs is also available, the translation of a source program into a net is proven semantically correct if it is proven that the net is deadlock free and deterministic and that the meaning of the net matches that of the program.
- The model can be tailored to a particular machine by stating a number of restrictions. It could then be used to compare different machines by comparing the restrictions, which would give an abstract but clear view of the functional differences between these machines, without being obscured by implementation details or differences in terminology.

ACKNOWLEDGEMENTS

Wim Böhm, Jan Heering, Paul Klint and Marleen Sint struggled their way through earlier versions of this paper and provided many helpful suggestions. I am also grateful to Arie de Bruin, without whose help and encouragement my first steps in this alien territory would have been my last.

REFERENCES

[1] Brock, J. D., "Operational Semantics of a Data Flow Language," Technical Memorandum 120, MIT/Laboratory for Computer Science (Dec 1978).

[2] Glauert, J. R. W., "A Single Assignment Language for Data Flow Computing," Dissertation, Dept. of Computer Science - Victoria University of Manchester (Jan 1978).

[3] Gurd, John and Ian Watson, "A Data Driven System for High Speed Parallel Computing," *Computer Design* 9(6 & 7), pp.91-100 & 97-106 (June & July 1980).

[4] Jaffe, J., "The Use of Queues in the Parallel Data Flow Evaluation of "if-then-while" Programs," Technical Memorandum 104, MIT/Laboratory for Computer Science (May 1978).

[5] Manna, Z., *Lecture Notes on the Logic of Computer Programming*, 1980.

[6] Rumbaugh, J., "A Parallel Asynchronous Computer Architecture for Data Flow Programs," Technical Report 150, MIT/Project Mac (May 1975).

[7] Veen, Arthur, "Reconciling Data Flow Machines and Conventional Languages," *Proceedings of CONPAR81, Conference on Analysing Problem Classes and Programming for Parallel Computing* (June 1981).

[8] Whitelock, P. J., "A Conventional Language for Data Flow Computing," Dissertation, Dept. of Computer Science - Victoria University of Manchester (Oct 1978).

APPENDIX I

Equivalence of Parallel and Sequential Firings

The definition of the reachability relation is based on a sequence of firings of single nodes. The question arises whether a definition which would allow *simultaneous* firings would lead to different reachability sets. A general definition of a simultaneous firing is problematic since it is not obvious how to treat simultaneous updates of the global memory. If we restrict ourselves however to firings which do not affect the global memory the definition becomes analogous to that of a simple firing:

A **functional** *parallel firing* of a set of nodes $\{N_1, \ldots, N_n\}$ is a pair $<M, M'>$ where M and M' are markings of *NET*, such that there is a set of firing elements $\{f_1, \ldots, f_n\}$ with
1 - $M_{state} = M'_{state}$
2 - $\forall_{1 \leq i \leq n}\ f_i = <(I_i, M_{state}), (O_i, M_{state})> \in prog(N_i)$
3 - $\forall p \in ports$
- $\bigcup_{1 \leq i \leq n} R_i(p) \subset M(p)$ (the enabling condition)
- $M'(p) = M(p) \cup \bigcup_{1 \leq i \leq n} O_i(p) - \bigcup_{1 \leq i \leq n} R_i(p)$

$$\text{where } R_i(p) = \begin{vmatrix} \emptyset & \text{, if } I_i(p) = \text{NIL} \\ \{I_i(p)\} & \text{, otherwise} \end{vmatrix}$$

The following lemma states that functional parallel firings are elements of the reachability relation:

Lemma S1:
If $<M, M'>$ is a functional parallel firing of $\{N_1, \ldots, N_n\}$ then M' is reachable from M.
Proof:
We will proof that the markings that would be produced if the nodes fire sequential are all reachable and that the last marking thus reached is M'.
Let f_i, I_i, O_i and R_i be defined as above and let $E = <M_0, \ldots, M_n>$ be a series of markings with
- $M_0 = M$
- for $0 \leq i \leq n$: $M_{i+1}(p) = M_i(p) \cup O_i(p) - R_i(p)$
It follows from the enabling condition for a parallel firing that $\forall_{1 \leq i \leq n}\ R_i(p) \subseteq M(p) = M_0(p)$. Since in the sequence $M_0(p), \ldots, M_j(p)$ only elements of $R_i(p)$ for $1 \leq i \leq j$ have been removed it follows that $R_i(p) \subseteq M_{i-1}(p)$. So the enabling condition for normal firing has been satisfied, $<M_{i-1}, M_i>$ is a firing of N_i and E is an execution path. By induction on i it is easy to see that $M_n(p) = M_0(p) \cup \bigcup_{1 \leq i \leq n} O_i(p) - \bigcup_{1 \leq i \leq n} R_i(p)$
So $M' = M_n$.□

We now define an alternative reachability relation based on parallel firings and we proof that it is a subset of the sequentialized reachability relation:

A marking M of *NET* is **parallel reachable** from a marking M' iff
there is a series of markings $<M_1, \ldots, M_n>$ with
1 - $M_1 = M'$
2 - $M_n = M$
3 - $\forall_{1 \leq i \leq n}$ $<M_i, M_{i+1}>$ is a parallel firing of
some subset of *nodes*

The following theorem states that this notion of parallel reachability is equivalent with the notion of reachability as used in the rest of this article.

Sequentialization theorem:
If a marking M' is parallel reachable from M then M' is reachable from M.
Proof:
The proof is a straightforward application of lemma S1 and the associative property of the reachability relation. □

Note that the enabling condition is essential to this result.

APPENDIX II

Proofs of Lemmas

Proof of lemma A1:
Let $<M_1, \ldots, M_m>$ be an execution path with $<M_i, M_{i+1}>$ a firing of node N_i and $S = \{p \mid M_1(p) \neq \varnothing\}$.
We will prove by induction on i the following properties
A - $dist(S, N_i) < i$
B - $\forall K \in nodes: \; dist(S,K) > i \Rightarrow \exists p \in in(K)$ with $M_{i+1}(p) = \varnothing$.
Induction basis $i = 1$:

$<M_1, M_2>$ is a firing of N_1. Since N_1 is strict and M_1 is $\varnothing$ outside of S property A holds for $i = 1$:

$dist(S, N_1) = 0$

It follows from the definition of *dist* that each node K that has all its input ports loaded in M_2 must have $dist(S,K) \leqslant 1$ which implies property B for $i = 1$:

$\forall K \in nodes: \; dist(S,K) > 1 \Rightarrow \exists p \in in(K)$ with $M_2(p) = \varnothing$

Induction step:

Assume properties A and B hold for $i \leqslant n$. Consider the firing $<M_{n+1}, M_{n+2}>$ of node N_{n+1}. It follows from induction hypothesis B that

$dist(S, N_{n+1}) > n \Rightarrow \exists p \in in(N_{n+1})$ with $M_{n+1}(p) = \varnothing$

Because N_{n+1} is strict property A holds for $i = n+1$:

$dist(S, N_{n+1}) < n+1$

Let K be a node such that $\forall p \in in(K)$:

1 - $M_{n+2}(p) \neq \varnothing$

2 - $p \in (S \cup \bigcup_{1 \leqslant j \leqslant n+1} out(N_j))$

Since property A holds for $j \leqslant n+1$ it follows that $dist(S,K) \leqslant n+1$ which proves property B for $i = n+1$.

Property A implies that all N_i are covered by S. □

Proof of lemma A2:
Lemma A1 implies that a node K that is not covered by *entry* will never fire. So $ONCE(K)$ holds for these nodes. We will prove that $ONCE(K)$ holds for each node K that is covered by *entry*, by induction on $dist(entry, K)$:
Basis of induction:

To prove: $dist(entry, K) = 0$ implies $ONCE(K)$.
Say there is an input marking M_I, an execution path $<M_I, \ldots, M_n>$ and a color C, such that $<M_j, M_{j+1}>$ and $<M_k, M_{k+1}>$ are firings of K with color C and $k > j$. We will prove that M_I cannot be a safe marking:
$\forall q \in in(K)$

1 - since q is not an output port for any node

$$M_I(q) \supseteq M_j(q) \supseteq M_{j+1}(q) \supseteq M_k(q)$$

2 - since K is strict and the firings are both of color C

$(M_j(q) - M_{j+1}(q))$ and $(M_k(q) - M_{k+1}(q))$ both contain a token of color C

So $M_I(q)$ holds at least two tokens of color C.

Induction step:

Induction hypothesis: $ONCE(L)$ holds for all nodes L with $dist(entry, L) \leqslant n$.
To prove: $ONCE(K)$ holds for all nodes K with $dist(entry, K) = n+1$.
Say there is an input marking M_I, an execution path $<M_I, \ldots, M_n>$ and a color C, such that $<M_j, M_{j+1}>$ and $<M_k, M_{k+1}>$ are firings of K with color C and $k > j$.
$\forall q \in in(K)$:
- since K is strict and the firings are both of color C: $\exists t_1 \in (M_j(q) - M_{j+1}(q))$ and $t_2 \in (M_k(q) - M_{k+1}(q))$ both of color C

- since *NET* contains no knot there is exactly one node L such that $q \in out(L)$. L is unitary so there must have been two separate firings of L that have put t_1 and t_2 on q. L is color preserving so both firings must have been of color C. This contradicts the induction hypothesis so $ONCE(K)$ holds. □

Proof of lemma B1:
Let EP be an execution path $<M_1, \ldots, M_m>$ with M_1 an input marking and $<M_{m-1}, M_m>$ a firing with color C of a node K insulated by S from *entry*. We will prove that there is a M_i and a $p \in S$ such that $M_i(p)$ contains a token with color C by induction on $dist(S, K)$.
Basis of induction: $dist(S, K) = 0$.

Because $<M_{m-1}, M_m>$ is a firing of K with color C there must be a $p \in in(K)$ such that $M_{m-1}(p)$ contains a token of color C. Since $dist(S, K) = 0$ it follows that $p \in S$.

Induction step:

Assume that the lemma holds for all nodes N with $dist(S, N) \leqslant n$. Let $dist(S, K) = n+1$. Since $<M_{m-1}, M_m>$ is a firing of K with color C there must be a $p \in in(K)$ such that $M_{m-1}(p)$ contains a token t of color C. The net is color preserving and from the definition of insulation it follows that $p \notin entry$, so there must be markings M_k and M_{k+1} in EP such that $<M_k, M_{k+1}>$ is a firing with color C of some node L producing token t. From $p \in out(L)$ and $dist(S, K) = n+1$ it follows that $dist(S, L) \leqslant n$.
It follows from the induction hypothesis that the lemma holds for K. □

Proof of lemma B2:
The proof is identical to that of lemma A2 with the addition of the special case that the input port q of node K is a knot. Because q is synchronized there must be two nodes L and M with $q \in (out(L) \cap out(M))$ and a multiswitch MS with destination sets A and B, such that L and M are either $\in MS$ or insulated from *entry* by A and B respectively.
Since $dist(entry, L) < dist(entry, K) = n$
and $dist(entry, M) < dist(entry, K) = n$
and L and M are color preserving
it follows that L and M have both fired with color C.
If $L \in MS$ then $q \in A$ and there is at least one marking M_a in the execution path such that $M_a(q)$ contains a token of color C.
If $L \notin MS$ then lemma B1 implies that there must have been a port $a \in A$ and a marking M_a such that $M_a(a)$ contains a token of color C.
The same holds for a port $b \in B$ and a marking M_b.
From the definition of a multiswitch it then follows that the control node CN of MS must have fired twice with color C. □