The Semantics of Data Flow Diagrams

P.D. Bruza
Th.P. van der Weide

Dept. of Information Systems , University of Nijmegen
Toernooiveld, NL-6525 ED Nijmegen, The Netherlands

July 26, 1993

Abstract

In this article we provide insight as to how semantics can be attached to Data Flow diagrams. We first present a method for transforming a Data Flow Diagram(DFD) to a Petri-Net (PT-net) which specifies the synchronization aspects of a DFD. Secondly, we sketch how a Data Flow diagram may be transformed to expressions whose semantics are described in terms of finite automata.

Published as: P.D. Bruza and Th.P. van der Weide. The Semantics of Data Flow Diagrams. In N. Prakash, editor, *Proceedings of the International Conference on Management of Data*, Hyderabad, India, 1989.

1 Introduction

In this document we present a way in which semantics can be attached to Data Flow diagrams (see [4] or [9]). DFD's often form an important role in the design of information systems. Their intention is to model the process aspects of an information system (IS). They are often used in the first phases of information analysis to establish a global model of an information system which can be further refined.

In this document we use a rigid formalism to describe the (dynamic) behaviour of the concepts that are used to specify processes and their relations. Usually this dynamic behaviour is considered to be self explaining. When building information systems, however, there is a need to be very precise about these aspects. Without precision, the modeller can have no clear idea of the meaning of the specification under development. The intention of this paper is to formalise the intuition behind process interaction in DFD's with the end that these can

be used more precisely by the designer. For the implementor, a precise specification forms a more secure platform on which to base an implementation.

Figure 1: External entity

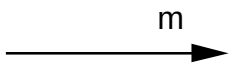

Figure 2: Data Flow

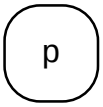

Figure 3: Process

We will now give a brief introduction to the structure of a DFD. A DFD is a diagram built up from elementary building blocks. These are denoted in figures 1 to 4. An example of how these elementary components can be connected to form a DFD is given in figure 5. A DFD can be viewed as directed graph wherein the nodes are *external entities*, *processes* or *data stores* and the edges are *data flows*. There exist rules for the construction for DFD's. For example, there can be no data flow from a data store directly to another data store. We will not deal further with the structure of a DFD. A formal specification of DFD's is given in [3].

The intention of a DFD is to describe an aspect of the information system at a particular abstraction level. A useful feature of DFD's is that a process can be further refined by another DFD. This DFD is a more detailed description of the process at a lower level of abstraction.

DFD's are a useful description mechanism for modelling process architecture, but they do not allow sufficient information to give a clear interpretation of what is being described. This problem is already identified in [1]. We present the problem on the basis of the DFD given in figure 6. This DFD has multiple interpretations. For example,

1. If process P receives m_1 and m_2, then P is activated and produces both m_3 and m_4.
2. If P receives m_1 alone, it is activated and produces m_3.
3. If P receives m_2 alone, it is activated and produces m_4.

The above are not the only interpretations possible. It is obvious that the DFD's must be enriched with further information so that they can clearly be interpreted. This is the topic of the following section.

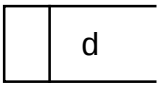

Figure 4: Data store

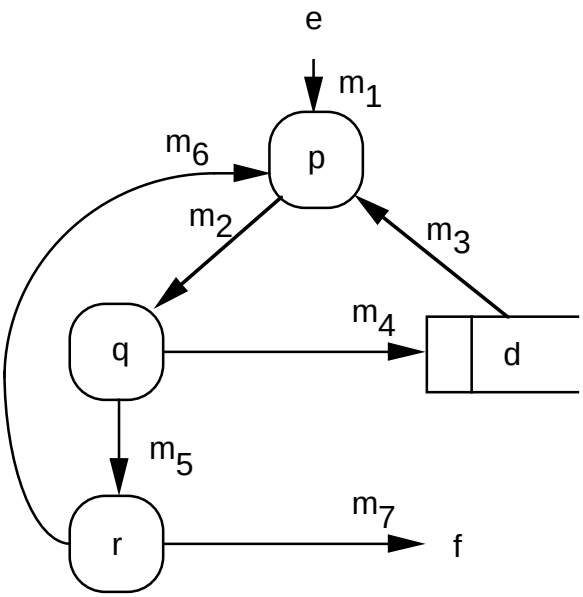

Figure 5: Example of DFD

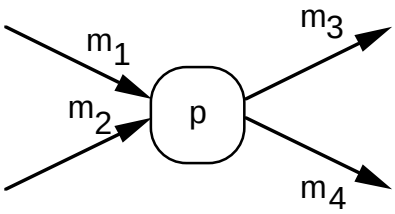

Figure 6: Problem DFD

2 Enhancement of DFDs

In this section we present a number of extensions to DFDs to increase their information content. This is important because it is necessary to have a clear interpretation of a DFD before we are in position to specify its semantics.

The first enhancement we present is the transformation of DFDs to a variant of Petri-Nets ([8]). We follow [1]. We have chosen Petri-Nets because they allow synchronization aspects to be straightforwardly modelled. They also are supported with an easy to understand graphical notation that can easily be incorporated into a DFD. The transformation of a DFD to a Petri-Net can be accomplished by transforming every data flow as in figure 7. The place holder allows the modelling of message flow along the data flow. For example,

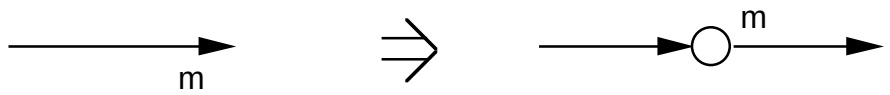

Figure 7: Transformation of a Data Flow

the possible interpretations cited with the DFD of figure 6 are depicted in figure 8. The resulting Petri-Net is an ad hoc variant of Petri-Nets. This is because in pure Petri-Nets all input places must have tokens before the transition can fire. From the above we see that this is not always the only way in which a process can fire. To solve this we adopt the view of [1]. In this solution we attach to each process its *activation possibilities*. In the example above we can denote this as follows:

$$p : (\ m_1 \wedge m_2 \ \rightarrow m_3 \wedge m_4)$$
$$p : (\ m_1 \ \rightarrow m_3)$$
$$p : (\ m_2 \ \rightarrow m_4)$$

For the example of figure 5 this leads to the PT-net of figure 9, which we also enriched with its activation possibilities.

We make some further enhancements to the Petri-Nets. In many cases we wish to model that a data flow can support a number of messages at the same time. This is called the *capacity* of the data flow and is denoted as in figure 10. A process may 'read' more that one message at a time from an incoming data flow. We term this as the the *demand* of the process on that data flow, and is denoted in figure 12. This figure depicts the demand of process p on dataflow m as being x messages. Conversely, we introduce the notion of the *yield* of a process onto a particular data flow. This represents the number of messages 'written' at the same time onto a data flow by a given process.

Note that if process P_1 sends information to process P_2 via data flow m and further the *yield* of P_1, $(yield(P_1, m))$ or $demand(P_2, m)$ is less that the capacity of m, then this is a depiction of the producer-consumer problem. The data flow m is actually a buffer.

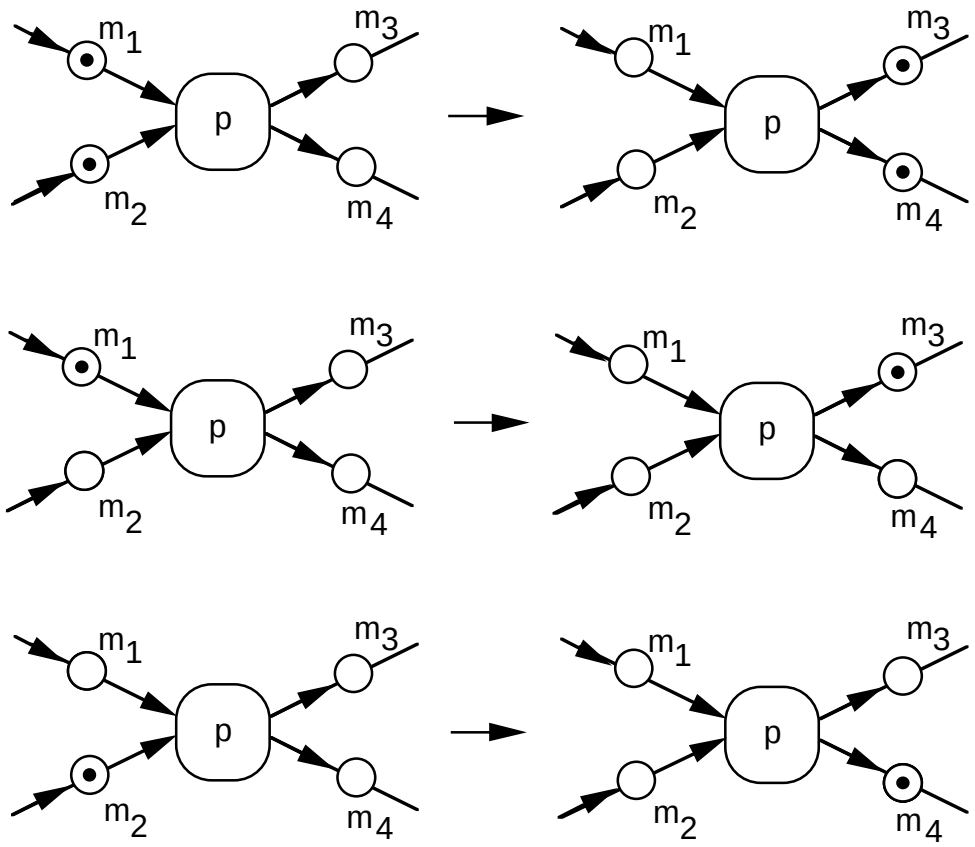

Figure 8: Activation Possibilities

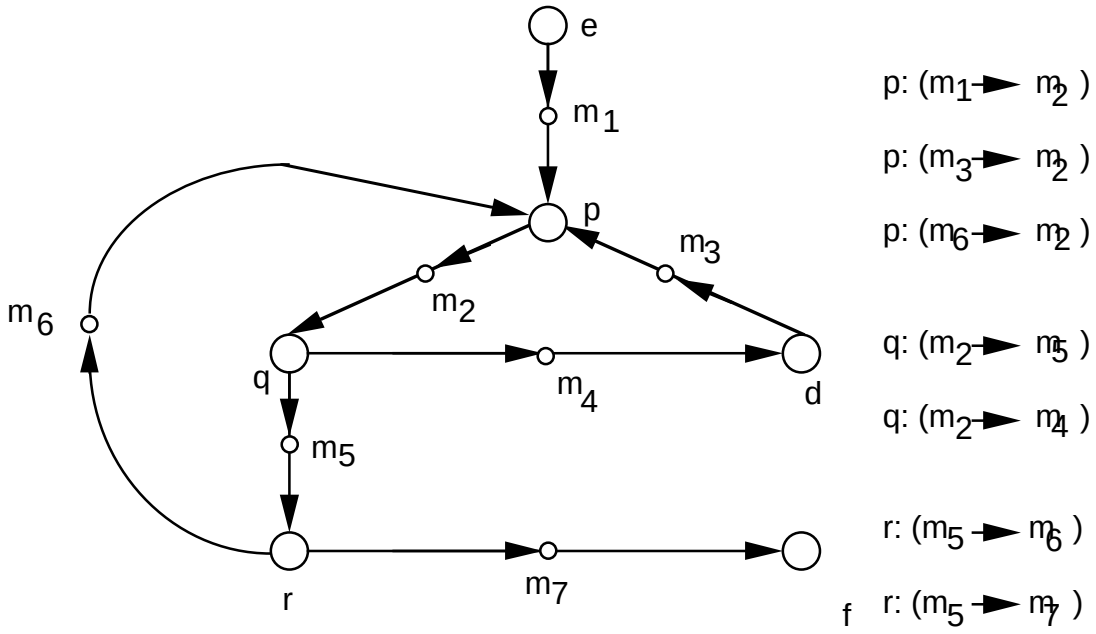

Figure 9: Example of DFD transformed into PT-net

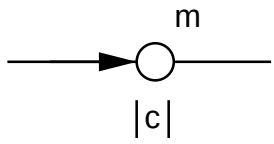

Figure 10: Capacity

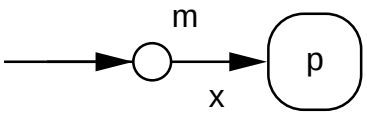

Figure 11: Activation Possibilities

Figure 12: Demand

Sometimes it useful to specify an ordering on the buffer. For example, FIFO in which case the buffer is a stack, or LIFO, in which case the buffer acts like a queue.

Finally, we introduce the notion of the *load* of a data flow. This is the number of messages currently on the data flow.

The *activation* of a process can now be defined as follows:

Definition

> If $m_k \wedge .. \wedge m_l \rightarrow z_p \wedge .. \wedge z_q$ is an activation possibility of process P, where P is not active, and further, for all input data flows m_i $(k \leq i \leq l)$ the condition $load(m_i) \geq demand(P, m_i)$ holds, then process P is termed *activated* and $load(m_i) := load(m_i) - demand(P, m_i)$.

This definition corresponds to the idea of the triggering of a process. The trigger condition is that there is sufficient input stimulus to wake the process.

The de-activation of a process can be conceived of as the process having done its work sending its outputs and going to sleep.

Definition

> If a process P has been activated by the activation possibility $m_k \wedge .. \wedge m_l \rightarrow z_p \wedge .. \wedge z_q$ and for all output data flows z_j $(p \leq j \leq q)$ the condition $load(z_j) + yield(P, z_j) \leq capacity(z_j)$ holds, then process P *de-activates* and $load(z_j) := load(z_j) + yield(P, z_j)$.

Note: If a process P is activated it must be de-activated before it can be re-activated.

This completes the list of enhancements to a DFD. The result basically is a PT-net but with a different firing mechanism. From this point on we will refer to this net as a process structure (PS).

3 DFDs as Path Expressions

In this section we present a method to convert a process structure (the PT-net form of a DFD) to an algebraic expression. This is a convenient representation of process structures (including its semantics). An advantage of these expressions is that they can be manipulated algebraically. This is helpful for example when establishing process structure equivalence. This representation will also be helpful for the transformation of a process structure to implementation in terms of a set of programs.

The syntax of the expressions is based on the language COSY ([6]) which was developed for specifying systems of concurrent processes sharing distributed systems of resources. The intention of this language was an implementation independent vehicle with which concurrent and distributed algorithms for operating system functions could be defined. For this reason the language is capable of specifying complex process interaction, but the full power of the language is not used in connection with DFDs because these are only capable of specifying rather simple process interaction.

The semantics of COSY is given in terms of PT-nets. In [5] it is shown how every COSY construct can be expressed as a PT-net. In this article we investigate the other direction, namely to acquire a COSY-like expression from the process structure as defined in the previous section.

We start with a short introduction to the syntax and semantics of the kind of expressions that will be generated. The work is based on [2], [5] and [6]. We distinguish four types of process synchronization: *sequencing, selection, repetition and concurrency.* All of these can be expressed by a path expression as follows:

sequencing A sequence of processes means that the processes must execute in the order given. Given that p, q, r are processes, then we denote sequentialization with a ';'. For example, path expression $p; q; r$ means that p, q, r are to be executed one after the other in the order given.

selection Selection from a set of processes permits only one to activate. Suppose the executions of processes p, q and r are to be selectively synchronized then this is denoted by the path expression p, q, r.

The above two synchronization schemes are fundamental and can be combined to specify complexer synchronization schemes. For example, $p; (q, r); s$. This path expression denotes two possible scenarios: Either p executes before q before s, or p before r before s. Next we introduce the concepts *repetition* and *concurrency*:

repetition This allows the specification of path expressions that repeat. This is denoted as P^*, if process P can be repeated a zero or more times, or P^+ if P can be repeated one or more times. If the process never stops then P^∞ is used. Repetition is used to specify loops in the PS.

concurrency A path expression describes a sequential system. Concurrency is modelled by concurrent sequential systems denoted $P \parallel Q$, where P and Q are path expressions. For example, $p; r \parallel q; r$ specifies that the processes p and q execute concurrently before process r is invoked.

3.1 The Semantics of Path Expressions

A meaning can be attached to each path expression in the following sense: A path expression can be viewed as a finite automaton. In the case of parallel path expressions, this can be considered as finite automatons that are operating concurrently.

In short, the semantics can be equated with the language defined by the automaton in question. We denote this as follows given that p is a process and P, Q are path expressions:

$$\begin{array}{lcl} \mathcal{L}(p) & = & \{p\} \\ \mathcal{L}(P,Q) & = & \mathcal{L}(P) \cup \mathcal{L}(Q) \\ \mathcal{L}(P;Q) & = & \mathcal{L}(P)\mathcal{L}(Q) \\ \mathcal{L}(P^*) & = & \mathcal{L}(P)^* \\ \mathcal{L}(P^+) & = & \mathcal{L}(P)^+ \\ \mathcal{L}(P^\infty) & = & \text{all infinite strings over } \mathcal{L}(P) \end{array}$$

There is a problem with denoting the semantics of parallel path expressions. We adopt the convention of [6], namely we write the denotations of the individual path expressions on top of each other. For example, $\mathcal{L}(p; r \parallel q; r)$ is denoted as:

$$\begin{array}{l} p \\ q \end{array} r$$

The advantage of specifying the semantics in terms of the languages produced by the automatons is that a particular string from the language can be viewed as a history. We can therefore discuss about the behaviour of DFDs in terms of such histories. We therefore must be able to reduce a given DFD to a path expression. This is the topic of the following section.

3.2 Translation of Process Structures to Path Expressions

In this section we show how the four types of synchronization can be identified in process structures.

sequencing This corresponds with processes that are connected via a data flow. If process p sends messages to process q via data flow m, then this can be represented by the path expression $p; q$.

selection This can occur in two ways in a PS: Selection can arise because a process may fire in more than one way. This often occurs as a selection between processes that may follow a given process depending on which way this process fires.

The second way in which selection can occur corresponds with conflict situations (in the PT-net sense). These situations are generalized by the PS of figure 13. The

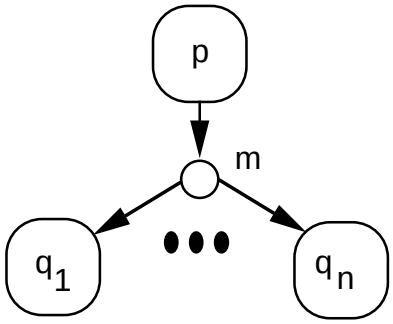

Figure 13: Conflict Situation

conflict situation is determined by the condition $capacity(m) < demand(q_1, m) + .. + demand(q_n, m)$

parallelism All processes that are not connected directly or indirectly are considered to be in parallel.

repetition Repetition corresponds with loops in the PS. Loops disrupt the sequencing by introducing parallelism between processes that are directly or indirectly related. We demonstrate this with the example given figure 14. We consider two cases:

1. The activation possibilities of q are:

   $q : (\ m_1\ \rightarrow m_2)$
   $q : (\ m_1\ \rightarrow m_3)$

   Then the corresponding path expression is:

   $$(p; q)^+; r$$

2. Process q has activation possibility:

   $q : (\ m_1\ \rightarrow m_2 \wedge m_3)$

   Then the corresponding path expression is:

   $$p; q; ((p \parallel r); q)^+$$

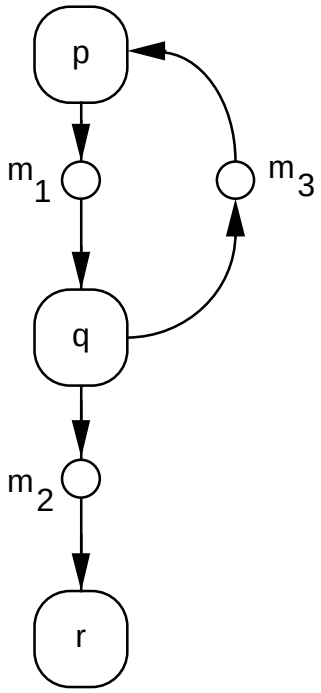

Figure 14: Example of a loop

4 A Basis for PS Equivalence

The path expressions introduced in the previous section also have the advantage that they can be manipulated algebraically. This can be used in establishing process structure equivalence. We now give the properties on the path expression algebra. The reader should keep in mind that the priorities of the path operators are as follows (in increasing order):

1. ,
2. ; ||
3. +

This means that + is most binding, followed by (; ||), followed by ,. We use parentheses to establish priority in the usual way. Then we have the following equivalence rules for path

expressions.

$$
\begin{array}{lcl}
P \parallel Q & = & Q \parallel P \ (Commutativity) \\
P, Q & = & Q, P \\
 & & \\
(P; Q); R & = & P; (Q; R) \ (Associativity) \\
(P, Q), R & = & P, (Q, R) \\
(\mathrm{P} \parallel \mathrm{Q}) \parallel R & = & P \parallel (\mathrm{Q} \parallel \mathrm{R}) \\
 & & \\
P; (Q, R) & = & P; Q \ , \ P; R \ (Distributivity) \\
P \parallel (Q,R) & = & P \parallel Q \ , \ P \parallel R \\
P \parallel (Q;R) & = & (P \parallel Q); R
\end{array}
$$

We now give an example of how the above algebra can be applied. Consider the PS of Figure 14 with the following activation possibility for process q:

$$q : (\ m_1 \ \rightarrow m_2 \wedge m_3)$$

We can define a set of path expression equations based on the DFD. We then solve these equations which results in a path expression for the DFD. The setting up of the equations can be done in a systematic way. For example, the DFD begins with the invocation of process p followed by the rest. We model this in the equation

$$P = p; Q$$

where P is the path expression which describes the DFD and Q is the path expression which models the part of the DFD following process p. Such an equation can be considered as a grammar rule for the language $\mathcal{L}(P)$.

We can now further define Q. This begins with process q and due to the activation possibility of q parallelism occurs between P and the rest of the DFD which we model as R. Q is thus defined as:

$$Q = q; (P \parallel R)$$

Note that if process q had the activation possibilities $q : (\ m_1 \ \rightarrow m_2)$ and $q : (\ m_1 \ \rightarrow m_3)$, then selection would have occurred. The equation for Q would in this case be:

$$Q = q; (P, R)$$

We now define equation R. This is simply r. On the basis of the equation P, Q and R, we can substitute and get an equation for P which we solve in the following way:

$$
\begin{array}{rcl}
P & = & p; q; (P \parallel r) \\
 & = & p; q; (p;q;(P \parallel \mathrm{r}) \parallel r) \\
 & = & p; q; ((p \parallel r); q; (P \parallel r)) \ \ Distributivity \\
 & = & p; q; ((p \parallel r); q; (p;q;(P \parallel \mathrm{r}) \parallel r)) \\
 & = & p; q; ((p \parallel r); q; (p \parallel r); q; (P \parallel r)) \\
 & = & p; q; ((p \parallel r); q)^{\infty}
\end{array}
$$

Note how the process structure described by P never halts.

We now wish to introduce the following definition:

Definition

Two process structures are defined equivalent if they can be reduced to path expressions with the same semantics.

Consider the PS of Figure 9. This has the followind set of path equations:

$$\begin{aligned} P &= e;Q \\ Q &= p;R \\ R &= q;(S,T) \\ S &= d;P \\ T &= r;(P,U) \\ U &= f \end{aligned}$$

This results in the following path expression:

$$P = e;(p;q;(d;p;q)^*;r)^+;f$$

If process q in this figure were to have only the following activation possibility

$$q : (\ m_2\ \rightarrow m_4 \wedge m_5)$$

then the process structure has the following path expression:

$$P = p;q;(p \parallel r);((p \parallel f);q;(d \parallel r))^\infty$$

Thus the two process two process structures are not equivalent, although there underlying DFD is the same. This demonstrates that a DFD by itself is not powerful enough to precisely model the behaviour of an information system.

5 Conclusions and Further Research

In this paper we have given some guidelines as to how semantics can be attached to DFD's. The importance of such research is clear. Current information system design too often involves formalisms without a clearly defined meaning. This leads to misunderstandings both in the use of the formalism and the interpretation of the results. The transformation of the DFD to a PT-net variant plus the associated path expressions with a clear meaning and algebraic properties provide a useful framework for the investigation of DFD's. In this way we can come to a better understanding of them.

In this paper we have only sketched how a DFD may be transformed to a path expression. In general this is quite a difficult task. Further research is needed to provide a general algorithm for this transformation. Also the algebra is not complete. Research must be done to determine what extra transformations are necessary.

Finally, as DFD's are not that different from the A-graphs and I-graphs of ISAC ([7]). We believe that the semantics of these graphs can also be established in an analagous way to that presented in this document for DFD's.

References

[1] J.A. Bergstra and G.P.A.J. Delen. Van dataflowdiagrammen via petrinetten naar systeemmatrixnotatie. Technical report, Mathematical Centre, Amsterdam, The Netherlands, 1982.

[2] R.H. Campbell and A.N. Habermann. *The Specification of Process Synchronization by Path Expressions*, volume 16 of *Lecture Notes in Computer Science*. Springer Verlag, 1974.

[3] E.D. Falkenberg, R. van der Pols, and Th. P. van der Weide. Understanding process structure diagrams. *Information Systems*, 16(4):417–428, Sept 1991.

[4] C. Gane and T. Sarson. *Structured System Analysis: Tools and techniques*. IST Databooks. MacDonald Douglas Corporation, St. Louis, 1986.

[5] P.E. Lauer and R.H. Campbell. Formal semantics of a class of high-level primitives for coordinating concurrent processes. *Acta Informatica*, 5:247–332, 1975.

[6] P.E. Lauer, P.R. Torrigiani, and M.W Shields. Cosy - a system specification language based on paths and processes. *Acta Informatica*, 12:109–158, 1979.

[7] M. Lundeberg, G. Goldkuhl, and A. Nilsson. *Information Systems Development - A Systematic Approach*. Prentice Hall, 1981.

[8] Peterson. *Petri Net Theory*. Prentice Hall, 1982.

[9] R. van der Pols. The SSAD handbook. Technical report, Digital Equipment BV, Utrecht, 1988.