Abstract

A framework is presented for constructing formal specifications of systems in a way that matches the use of Data Flow Diagrams to represent their structures, and which is therefore particularly applicable to parallel and distributed systems. The basis of this framework is that the behaviour of a system may be defined in terms of the possible patterns of data flows from one process to another via the channels that connect them. These patterns are specified by means of regular expressions that are structured as production rules, and that form a data flow algebra whose properties are described. The principal operations in this algebra are sequencing, alternation, repetition, interleaving, intersection and synchronisation, and examples are given to illustrate the use of these constructs. The grammars generated by these productions can be extended to attribute grammars, in which the attributes constitute specifications of the semantics of components of the system, which can be defined using any of a variety of existing specification methodologies.

Key Words and Phrases

Formal specifications, data flow diagrams, dataflow algebra, process algebra, parallel systems, distributed systems.

Introduction

Data Flow Diagrams (DFDs from now on) are a fundamental technique for representing the structure of systems. They are an essential part of most systems analysis methodologies, both the "structured" ones such as SSADM [1], Yourdon [2] and their derivatives, and the object-oriented approaches. In one form or another they also appear in most design methodologies: for instance, the structure diagrams used by Yourdon and others can be regarded as essentially DFDs that have been constrained to a tree form. Similarly, the object structure diagrams that are used in various object-oriented methodologies, such as Booch's Rational method [3], are effectively DFDs, as are the diagrams that are used to describe the hardware and communications structure for systems that are to be implemented on a distributed architecture. Furthermore, the process diagrams used to illustrate the design of message-passing parallel systems [4] are also forms of DFD.

Since DFDs are so widely used, it is important that they should be related easily to formal specifications of systems, particularly in situations where there are safety-critical aspects to the system design that require formal reasoning about the behaviour of the system in order to establish sufficient confidence in its correctness. So far, however, the problem of formalising the concepts of DFDs to the point where specifications can be integrated into them in this way has proved to be a difficult one. Tse [5] has shown how the rules that govern the topology of DFDs in any particular methodology can be formalised, and in a sense this defines a syntax for DFDs: what is required is a way of defining the semantics of DFDs, with the particular aim of providing a framework into which can be embedded formal specifications of the systems being described by the diagrams. The purpose of this paper is therefore to outline a formal model of DFDs which provides just such a framework.

Most approaches so far to this problem have focussed on specifying the processing that is carried out within the individual components, as for instance the methods described by Randell [6], Semmens & Allen [7] or Polack, Whiston & Hitchcock [8], but the problem with these approaches is that while they can work for simple cases where the flows of data occur in a way which makes the sequence of processing predictable, they do not scale up readily to situations where the possible sequences of processing are more complex. Some attempts have been made to address this sequencing problem, either by using non-deterministic models such as Petri nets (as described by Kung [9]), or by trying to define a form of "activation logic" to enable the possible sequences to be determined from the possible patterns of data flows, an idea which was also suggested originally by Kung and which has been developed further by Chantatub [10]. In each case, though, the root cause of the problem is that on their own the DFDs for a system do not contain enough information about it to provide an adequate framework, since they do not embody any idea of time. That is, the representation of a data flow between two processes or other elements in a particular DFD simply means that within the system it is possible for that data flow to take
place between those two elements, but in the general case it does not imply anything about the frequency with which items of data flow, or the conditions under which they flow. Consequently, the diagram does not imply anything either about the way in which the processes will need to be activated in order to handle input data flows or to generate output data flows.

As an illustration of this, within a DFD such as the one shown in figure 1 (which uses a simplified version of the SSADM notation, with processes shown simply as rectangles) there is nothing to indicate the sequencing of particular data flows, and so there is no assumption in this example as to how the individual inputs to the process and the outputs from it will be related. In particular, it is certainly not implied that the arrival of one data in message will then generate one data out message, although under some conditions this might be a natural interpretation, and indeed some methodologies make at least partial attempts to impose this sort of interpretation in some situations. They can not impose it in all cases, however, and it is this lack of information about the sequencing of data flows in the general case which makes the problem referred to above of inferring the sequence of processing for a complex diagram such a difficult one to solve.

![Figure 1: A Simple Data Flow Diagram.](image)

In practice, however, these sort of relationships between flows are a fundamental part of the behaviour of any system, and the approach described here is based on the observation that a description of the permitted sequences in which data flows can occur ought to be an integral part of any specification of a complete system. Indeed, during the course of analysing a system initially the DFDs will usually be built up essentially by determining these sequences of flows, by tracing particular items of data through the system, and observing what processing takes place and what other data flows are generated as a result. The results of these observations are often referred to as "use cases", and are recognised as an important element in at least one major object-oriented analysis and design methodology [11]. Thus, even though this sequencing information may then not be represented within the DFD, it is still important in that it determines how the components of the system will interact. Hence, the framework that is developed here is based on modelling both the structure of the data flows that is shown explicitly on the DFDs, and the relationships between flows that are implicit in the way in which they are sequenced.

So far, the formal models that have been suggested for the sort of parallel or distributed systems that DFDs represent have ignored this aspect of the overall sequencing of data flows, and have focussed instead on trying to specify sequences of actions on a process-by-process basis. That is, fundamental to all the process algebra models such as CSP [12] or CCS [13] is the notion that individual processes are explicitly composed from sequences of actions, and that some of these actions can involve two (or more) processes in communicating in some synchronised way. Such a formulation then implies that particular sequences of data flows must occur between particular sets of processes. The approach that is described here is, however, the dual of this, in that it aims to model explicitly the patterns of data flows across the whole set of processes forming a system, to form what we term a dataflow algebra. The behaviour of individual processes can then be implied from these patterns of data flows. (We recognise that the term "dataflow algebra" could be confused with the use of the adjective "dataflow" to refer both to parallel architectures based on graph reduction methods [14], and also to the graph structures which these architectures use to represent programs [15], but we are unable to find an acceptable substitute for use in this context.)

Thus, there are effectively two orthogonal but related views that can be taken of the behaviour of such a system, with one view or dimension primarily observing the set of processes and the other view observing the set of communications channels. In this context the process algebra models can then be understood as viewing systems along the process dimension, while the dataflow algebra model to be described here views them along the communications channel dimension. The duality or relationship between the two views is then emphasised by the fact that the actual formalism used in this approach is to specify the sequences of communications within a system by means of grammars that define regular expressions for the sequences. As such, it has much in common with the COSY approach [16] to process algebras, which uses regular expressions to define the sequences of actions that constitute the paths or individual processes within a concurrent system. At the same time, the fact that the basic elements in this formalism are communications between processes rather than actions of one process means that there are significant differences between the two approaches.
### Actions and Sequences

The basic aim of this dataflow algebra approach is, therefore, to allow the behaviour of a system to be defined in terms of the set of possible sequences of data items or messages which can flow between the different processes within that system, where the intention is that the sequences should be described in a way which reflects the causality relationships between the arrival of one message at a process, and the consequential generation by that process of other messages. Thus, these sequences will be composed ultimately of primitive actions, each corresponding to one message flowing from one process (its source) to another process (its destination) via some channel. Such a primitive action will be written in the form `source ! channel ? destination`. Here the different channels represent the different kinds of message which can flow between particular processes, although it is not necessarily to distinguish whether the messages actually flow via separate parallel channels (as would normally be shown in a DFD at the analysis stage) or via a single multiplexed channel (as with variant protocols in occam2 [17]). Thus, for the simple system illustrated in figure 1, the two possible actions would be denoted as `Source ! data in ? Process` and `Process ! data out ? Destination`.

The basic sequential composition of two actions (or sequences of actions) `a1` and `a2` is then written as `a₁ ; a₂`, so that if in figure 1 the operation of the system were to consist of two `data in` messages followed by one `data out` message, this could be written as `Source ! data in ? Process ; Source ! data in ? Process ; Process ! data out ? Destination`. Alternatively, one could write the grammar as a set of expressions, with the basic actions as terminal symbols and introducing non-terminal symbols as names for the various sub-expressions. This grammar will generate the set of (possibly infinite) strings of these basic actions, and so define the set of allowable behaviours of the system. For instance, the grammar for this example might be written as

\[
\begin{align*}
   a₁ &= \text{Source ! data in ? Process} \\
   a₂ &= \text{Process ! data out ? Destination} \\
   \text{System} &= a₁ ; a₁ ; a₂
\end{align*}
\]

To allow more complex systems to be handled, it is necessary to allow processes and channels to be identified by indexed sets of names as well as just single names, as in the example of a three-stage pipeline illustrated in figure 2. Here the indexes for the sets of names are denoted by square brackets, as in most programming languages, but in writing the specifications either this convention or the normal subscript notation could be used. Where specifying a sequence of actions involves successive elements of an indexed set of either processes or channels, or both, then what is required is a convenient notation for the concatenation of a set of elements to form a string. This can be regarded as analogous to adding successive elements to the string (although, unlike addition, the sequencing operator `;` is not commutative), and this analogy suggests that the sigma notation be used. Thus, for a set of actions `ai` with index `i` the ascending sequence `a₁ ; a₂ ; ... ; aᵢ ; ... ; aₙ-₁ ; aₙ` is denoted `∑_{i=1}^{n} a_i`, and the descending sequence `aₙ ; aₙ-₁ ; ... ; a₁ ; ... ; a₂ ; a₁` is denoted `∑_{i=1}^{n} a_i`

Hence, the passage of a single item of data through this pipeline could be specified by the single expression

\[
\text{Oneltem} = \text{Start ! In} ; P₁ ; \sum_{i=1}^{3} Pᵢ ! \text{To}ᵢ ; Pᵢ+₁ ; P₃ ! \text{To}₃ ? \text{End}
\]

**Figure 2: A Three-Stage Pipeline.**

Alternatively, since it would be convenient not to have to treat the last stage of the pipeline as a special case, in situations where actions depend solely on the value of the index for a set the `IF ... THEN ... ELSE ... FI` notation can be used, to effectively provide a macro facility that will generate an indexed set of productions from a single expression. Thus, for the pipeline one could construct the indexed expression

\[
\text{move}ᵢ = Pᵢ ! \text{To}ᵢ ? \text{IF } i = 3 \text{ THEN } \text{End ELSE } Pᵢ+₁ \text{ FI}
\]

and then write the above specification as

\[
\text{Oneltem} = \text{Start ! In} ; P₁ ; \sum_{i=1}^{3} \text{move}ᵢ
\]
Alternatives

As well as straightforward sequences of messages, in systems of any complexity there will be situations where one message arriving at a process may cause that process to generate any one of several different possible sequences, and so the notation needs to be able to express the overall pattern of a choice between the different alternatives. As an example, figure 3 illustrates a transputer farm with four workers. This farm is a common structure for exploiting data parallelism in problems, in that the farmer process generates sub-problems that can be represented by sub-sets of the data known as work packets; it then distributes the work packets to the workers, which actually solve the individual sub-problems, and it collects and integrates the solutions that they return. For principle the work packets could be sent directly to each worker, and the results returned directly, but in practice it is usual in transputer implementations to organise the workers as shown here, with pipelines of input and output processes, so as to minimise the number of channels connected directly to the farmer in order to match the limits on the number of links provided in the transputer hardware.

The way in which these pipelines of input and output processes operate is that a work packet is sent out from the farmer to a particular worker, so that each input process in turn either passes it on to its own worker, or to the next stage in the pipeline; similarly, each output process both returns its own result packets to the farmer, and passes on result packets from the previous stages in the pipeline. Thus, for any of the intermediate input processes there are two alternative patterns: one is to receive a work packet from the previous stage and send it to its worker, and the other is to receive a packet and send it to the next stage. The choice between two alternative sequences $a$ and $b$ is denoted $a | b$, and so the operation of an intermediate input process for a single work packet can be denoted by the expressions

$$
in_i \cdot \text{inputPassTo } \text{ToWork } \text{PassTo } \text{ToIn } = + (a | b)$$

Since the semantics of these expressions are defined in terms of sets of strings over the alphabet of events, the semantics of this alternation operator $|$ will be defined in terms of the union of the sets of strings generated by each component expression. From this it follows that, like set union, the alternation operator is idempotent, commutative and associative, expressed by the axioms $a | a = a$, $a | b = b | a$ and $(a | b) | c = a | (b | c)$ for any expressions $a$, $b$ and $c$ that denote actions or sequences of actions.

Another basic property is that alternation distributes over sequencing, which can be expressed by the axiom $a ; (b | c) = (a ; b) | (a ; c)$. Since the semantics of these expressions are defined in terms of sets of strings, the significance of this property is that any sequence can be expressed in a fully distributed form (ie as on the right-hand side of this axiom), where each component of the choice contains no alternations. This form is actually a canonical form for an expression, and there are proofs of properties of some of the operators introduced below which rely on the existence of such canonical forms. To avoid confusion, therefore, from now on the term sequence will be used generally, and the term strings will be used when it is necessary to refer specifically to sequences which contain no alternations, as when discussing canonical forms. Although these canonical forms are useful, however, they are not generally very convenient ones for actually writing specifications: for instance, the above expression for the input stage of the farm could also be written more compactly, and therefore more naturally, as

$$\text{input}_i = \text{PassTo}_i ; (\text{ToWork}_i ; \text{PassTo}_{i+1})$$

Similarly, given the expressions

\[\text{input}_i = \text{PassTo}_i ; (\text{ToWork}_i ; \text{PassTo}_{i+1})\]
PassFrom$_i$ = Out$_i$! From$_i$? IF $i=1$ THEN Farmer ELSE Out$_{i-1}$FI
FromWork$_i$ = Work$_i$! Res$_i$? Out$_i$

the operation of an intermediate output process could be defined by either of the expressions

$$\text{output}_i = (\text{FromWork}_i ; \text{PassFrom}_i) \lor (\text{PassFrom}_{i-1} ; \text{PassFrom}_i)$$
or, more naturally,

$$\text{output}_i = (\text{FromWork}_i \lor \text{PassFrom}_{i-1}) ; \text{PassFrom}_i$$

In the same way as it necessary to be able to define sequences over an indexed set of actions, it is also necessary to be able to denote alternatives that cover such an indexed set too. For instance, in the case of the transputer farm one can define the sequence of actions necessary to process one packet by worker $i$ as

$$\text{onepacket}_i = \sum_{j=1}^i \text{PassTo}_j; \text{ToWork}_j; \text{FromWork}_j; \sum_{j=1}^i \text{PassFrom}_j$$

and one then wishes to be able to denote that processing one packet involves the choice of this sequence for any suitable value of $i$. Since the semantics of this alternation is defined in terms of the union of sets of strings, the obvious way to denote the choice over an indexed set of actions $a_i$ is as $\bigcup_i a_i$. Thus, with four workers the operation of the farm for one packet can be specified as

$$\text{DoOnePacket} = 4 \bigcup_{i=1}^4 \text{onepacket}_i$$

Silent and Forbidden Actions

As well as alternations between different sequences of actions, there are situations in which one needs to be able to express that a particular subsequence of actions may or may not occur. For instance, figure 4 illustrates a process $P$ that during its operation needs to read data from a data store $D$, so that whenever it needs to fetch an item of data there will be the sequence of actions $\text{OneFetch} = P ! \text{request} ? D ; D ! \text{dataitem} ? P$. This subsequence will therefore occur in any of the sequences that define the processing that $P$ undertakes, so that a typical operation of the system might be specified as $\text{OneOp} = x_1 ! \text{in1} ? P ; \text{OneFetch} ; P ! \text{out1} ? x_3$. If, however, $P$ were assumed to contain a cache for some of these items of data, then any sequence that might have involved accessing the data store may on some occasions not need to include the subsequence $\text{OneFetch}$, as the required item may already be in the cache. Thus, instead of the subsequence $\text{OneFetch}$ one needs to be able to define a choice of that or no action.

To make this possible, the silent action is introduced to represent the situation where no action takes place, and is denoted by the symbol $\varepsilon$, so that the required choice can be written as $\text{PossibleFetch} = \text{OneFetch} \lor \varepsilon$, and then the typical operation given above can be written as $\text{OneOp} = x_1 ! \text{in1} ? P ; \text{PossibleFetch} ; P ! \text{out1} ? x_3$. As well as denoting an empty action, $\varepsilon$ can also be interpreted as an empty sequence of actions, which implies that it forms the identity element for the sequencing operator, as defined by the axioms $\varepsilon ; s = s = s ; \varepsilon$ for any sequence of actions $s$.

![Diagram](image-url)

*Figure 4: A process that reads from a data store.*
As well as an identity element for the sequencing operator, it was found during the development of the methodology that in certain situations an element was required that would act as an identity for the alternation operator. Initially the significance of such an element for the process of writing specifications was not clear, since it effectively represented the situation where a choice between some sequence of actions and this identity element simply reduced to that sequence of actions. Thus, the other choice could never occur in practice, and so one would never actually want to write this identity as a legal choice in the grammar defining the behaviour of a DFD. The conclusion which was therefore reached eventually was that this identity element must signify an action which is forbidden.

This forbidden action is denoted by \( \phi \), and its properties as the identity for the alternation operator are defined by the axioms \( \phi; s = s = s; \phi \) for all sequences of actions \( s \). In terms of the semantics of the productions as strings of actions, any string that contains \( \phi \) is one which is not actually a legal sequence, and which therefore needs to be removed from the set of generated strings. Furthermore, since any sequence which contains \( \phi \) has to be removed, a sequence consisting just of the single action \( \phi \) has to be removed as well, so that semantically any sequence containing \( \phi \) is equivalent just to \( \phi \) itself, and so can be reduced to it. This property is defined by the axioms \( \phi; s = \phi = s; \phi \) for all sequences of actions \( s \).

**Repetitions**

Unlike some of the process algebra approaches, where repetitions of sequences of actions may be explicit or implicit, in this methodology they must be represented explicitly. There are two possible cases: one is where a specific number of repetitions is required of a particular action or sequence of actions, and the other is where any arbitrary number of repetitions can occur. In the first case, for any sequence of actions \( s \), the occurrence of \( n \) repetitions of \( s \) is denoted \( s^n \), which can be defined recursively by \( s^0 = \varepsilon \), \( s^1 = s \) and \( s^n = s; (s^{n-1}) \) \( \forall n > 1 \). Thus, in the examples given above, the consecutive passage of three items down the pipeline could be specified as OneItem\(^3\), and the consecutive processing of four packets by the farm could be specified as DoOnePacket\(^4\).

Where an arbitrary number of repetitions of some sequence \( s \) can occur, so that one wants to specify \( s^n \) for any arbitrary \( n \), then it is normal to distinguish between the cases where \( n > 0 \) (so that at least one repetition must occur) and the case where \( n \geq 0 \) (so that zero occurrences is permitted). The usual notation from formal grammars is used here, viz \( s^+ \) to denote one or more repetitions of \( s \) (ie the case \( n > 0 \)), and \( s^* \) to denote zero or more repetitions of \( s \) (ie the case \( n \geq 0 \)). Effectively each of these represents a choice between the different sequences \( s^n \) for possible values of \( n \), and the difference between the two is that \( s^* \) also allows \( \varepsilon \) as a valid choice. Since the operation of either the farm or the pipeline would normally require that at least one item of data or work packet be processed, these would naturally be written as OneItem\(^+\) and DoOnePacket\(^+\) respectively. Similarly, in the example in figure 4 the possibility that the process \( P \) might need to fetch several data items from the store \( D \), or none if they were all already in the cache, could be expressed as PossibleFetch = OneFetch\(^4\).

**Parallel Composition**

While the notation for repetitions describes sequences of actions that are repeated consecutively, in parallel or distributed systems there will by definition be a number of activities going on simultaneously. To reflect this, the notation has to be able to define that there may be several different sequences occurring in parallel, and that (as in the pipeline and the transputer farm) repetitions of the same sequences may be taking place in parallel. In terms of the underlying semantics there are in principle two approaches to interpreting such parallel compositions: one is to allow genuine parallelism, in which different actions of the system are permitted to occur simultaneously, and the other is to treat the parallelism as non-deterministic interleaving, so that the different simultaneous actions are actually treated as occurring in some arbitrary order rather than taking place exactly together.

As far as the principles of DFDs are concerned either interpretation would be equally acceptable, since one of the weaknesses of DFDs is that (because they do not embody any concept of time) they do not incorporate any notion of the data flows having a specific duration. Thus, it would not be inconsistent with these principles to assume that one could somehow associate with each action in a sequence a time instant at which that data flow takes place (which in practice might be a start or a finish time), and to derive the ordering or simultaneity of actions in an actual sequence from the associated time instances. Within this framework, however, it has to be recognised that there are genuine philosophical problems raised by attempting to determine whether two events in different parts of a distributed system have occurred at exactly the same time instant, where time instants are measured on the line of real number quantities. Consequently, to create a way of writing a specification that two events must occur at the same time would be to run into the danger of allowing specifications to be written that would not be meaningful. Furthermore, if genuine parallelism were permitted, then the semantics of the model would become more complicated, as the underlying strings would not just be of individual actions but of groups of possibly concurrent actions.
At this stage, therefore, the approach that has been adopted is to ignore the possibility of genuine concurrency of actions, and to work instead in terms of an interleaving semantics. Thus, given two sequences of actions $s_1$ and $s_2$, the parallel composition of the two sequences is defined to be the choice between all possible interleavings of their actions. This parallel composition is denoted $s_1 \parallel s_2$, and it is obviously a commutative operation, so that $s_1 \parallel s_2 = s_2 \parallel s_1$. The algebraic definition of this operator is a recursive one, which uses a structural induction over the sequencing and alternation operators. The definitions are symmetrical, and so only the left-hand recursions are given here: they are

\[
(s_1 \parallel s_2) \parallel s_3 = (s_1 \parallel s_3) \parallel s_2 \\
(a_1 ; s_1) \parallel (a_2 ; s_2) = (a_1 ; (s_1 \parallel (a_2 ; s_2))) \parallel (a_2 ; ((a_1 ; s_1) \parallel s_2))
\]

where $a_1$ and $a_2$ denote any individual actions and $s_1$ etc denote actions or sequences of actions. The base case of the recursion is $\varepsilon \parallel s_1 = s_1 \parallel \varepsilon = s_1$, so that the silent action is an identity element for the parallel composition operator as well as for the sequencing operator.

In most parallel systems, of course, the significant parallelism will come from having multiple versions of the same activity occurring at once, and so it is also necessary to be able to define parallel compositions over an indexed set of processes or channels, which in one sense is a form of product operator. The parallel composition over an indexed set of sequences $s_i$ is therefore denoted by $\prod_i s_i$, and products of unindexed sets can also be constructed. Thus, to represent the fact that the three-stage pipeline in figure 2 would normally be processing sequences of data items, with three data items (one in each stage) being handled in parallel, one could write the specification as

\[
\text{PipeLine} = \prod_{i=1}^{3} (\text{OneItem}^*)
\]

Similarly, the fact that the farm in figure 3 would normally be processing a sequence of packets in each worker, with the four workers operating in parallel, could possibly be specified as

\[
\text{Farm} = \prod_{i=1}^{4} (\text{DoOnePacket}^*)
\]

although since each complete repetition of a packet being processed by one particular worker should then be followed by another packet being processed by the same worker, it would be more accurate to specify the farm as

\[
\text{Farm2} = \prod_{i=1}^{4} (\text{onepacket}^*)
\]

Restriction

An important aspect of DFDs is their hierarchical structure, which means that in modelling them one may need to represent the relationships between different levels in a hierarchy. Several operations are needed to express this, of which the first focusses on the individual processes within a system, and extracts from the specification of the complete system the subsequences that consist of just the actions involving those processes. This operation is termed restriction, and for any sequence of actions $s$ and set of processes $p$ it is denoted by $\text{res}_p s$. It can be defined formally in terms of a function which maps any action that has an element of $p$ as its source or destination into itself, and maps any other action into $\varepsilon$. Thus, the behaviour of the single stage $p_i$ in the pipeline could be denoted as $\text{PipeLine} \setminus p_i$.

In practice, when applying such a restriction it is often also convenient to allow the actions to omit the names of the external entities (ie the ones which are not elements of $p$). For instance, in discussing the behaviour of the system illustrated in figure 4 the names of the external entities such as $x1$ do not contribute any significant meaning to the descriptions of the actions, which could more conveniently be written as $\text{in1} ? p, p ! \text{out1}$, etc. As a syntactic convenience this can be allowed, provided that the names of the channels connected to the process in question are unique: if they are not, then some form of notation for renaming channels is required. While in principle this would also be just a syntactic convenience, in practice it would be an absolute necessity for specifying any large system (as might a notation for renaming processes). Such a renaming notation is not needed for the examples being considered here, but the one that is suggested is that the renaming of a channel or process $\text{oldname}$ to $\text{newname}$ in any sequence of actions $s$ be denoted by $s.(\text{oldname/newname})$.  

Inclusion

The second operation needed for describing the relationship between levels in a hierarchy of DFDs then extends the notion of renaming so as to collapse all the processes or data stores of a lower-level DFD into a single process, that is intended to correspond to the appropriate element of the higher-level DFD. This operation consists essentially of renaming all the lower-level elements, and then mapping any action which now has the same element as both source and destination into the silent action. In the case where one is considering DFDs that form a single hierarchy, the result should then have the same specification as the corresponding single process in the higher-level DFD. In many situations, however, such a hierarchy of DFDs will reflect the fact that components are being assembled to form a system, and it may often be the case that the component represented by the lower-level DFD will actually be more robust than is needed for the role that it is to play in the total system, in the sense that even if it is presented with a sequence of inputs that are not supposed to be valid for the whole system it will behave reliably, in that it will generate a corresponding set of outputs that are valid for the rest of the system.

To represent this, the relationship of inclusion can be defined between sequences of actions, which for any two sequences \( s_1 \) and \( s_2 \) means that any string of actions generated by \( s_1 \) (ie as one of the choices in the canonical form for \( s_1 \)) will also be a legal string of actions generated by \( s_2 \). This relation is denoted by \( s_1 \subseteq s_2 \), and its algebraic definition is a recursive one, in which the principal axioms are

\[
\begin{align*}
  a & \subseteq a, \\
  s_1 & \subseteq s_1 | s_2, \\
  s_1 & \subseteq s_2 \land s_3 \subseteq s_4 \Rightarrow (s_1 ; s_3) \subseteq (s_2 ; s_4) \text{ and} \\
  s_1 & \subseteq s_2 \land s_3 \subseteq s_4 \Rightarrow (s_1 | s_3) \subseteq (s_2 | s_4),
\end{align*}
\]

for all actions \( a \) (including \( \epsilon \)) and sequences \( s_1, s_2, s_3 \) and \( s_4 \).

Applying this to a single stage in the pipeline example, one would expect to be able to produce a specification for the sequences of actions that it is intended to be able to process, and if the stage is assumed not to contain any unnecessary buffering this might be written \( \text{IdealStage}_{i} = (\text{To}_{i-1} ? P_{i} ; P_{i} ! \text{To}_{i}) \). Then, given a potential implementation of a pipeline stage with a specification \( \text{RealStage}_{i} \), one would want to demonstrate that it was sufficiently robust to be used in this situation, and hence would need to show that \( \text{IdealStage}_{i} \subseteq \text{RealStage}_{i} \).

Intersection

The inclusion relationship can also be applied to the pipeline example in another way, for if the specification given above for \( \text{PipeLine}_{1} \) is accurate, then the specifications of the individual stages should be consistent with it. To demonstrate this, one would need to show that the restriction of the overall specification to an individual stage is included within the specification of the stage, ie that \( (\text{PipeLine}_{1} \setminus P_{i}) \subseteq \text{IdealStage}_{i} \). Unfortunately, it turns out for this example that this is not the case, as the canonical form of the specification \( \text{PipeLine}_{1} \) contains subsequences such as \( \text{move}_{1} ; \text{move}_{1} ; \text{move}_{2} ; \text{move}_{2} \), which are certainly not consistent with (ie included within) the specification for \( \text{IdealStage}_{i} \). This is because of the way in which the parallel composition operator is defined, in that it imposes no constraints at all on the relative orderings of the sequences being interleaved, so that when it is used in the specification of \( \text{PipeLine}_{1} \) there is effectively nothing to indicate the constraint that one data item is not allowed to catch up with or overtake another.

In part this is a consequence of the basic level of abstraction of the model, in that the causality within different sequences is indicated purely by the resultant ordering of the actions within those sequences, as there is no representation of the content of the data being transferred in different actions. Thus, a subsequence such as \( \text{move}_{1} ; \text{move}_{1} ; \text{move}_{2} ; \text{move}_{2} \) simply describes stage 2 receiving two data items and then passing them on (presumably after some processing which is not defined), but it does not prescribe whether the two items are processed in series or in parallel, and whether or not they are passed on in the same order as they were received. Within this framework, therefore, the most that can be done is to prescribe whether or not it is acceptable for a stage to receive another item of data for processing before it has passed on the first one.

It appears that there are two possible approaches to this. One approach would be to introduce more complicated versions of the parallel composition operator, that would incorporate implicit constraints on the order in which elements from the component sequences may be interleaved. The other approach would be to augment the basic specification of the behaviour of a system (as in \( \text{PipeLine}_{1} \)) by explicit constraints that specify the limitations of the individual components of the system. Since it is not yet clear quite what form the implicit constraints demanded by the first approach might take,
this paper is concerned primarily with the second of these two approaches, in the expectation that this may help to identify issues that will then be relevant in investigating the viability of the first approach.

In principle, therefore, the specification of the pipeline given by Pipeline1 needs to be augmented to indicate that it must also be consistent with the specified behaviour of each stage: ie that the relation \( \text{Pipeline1} \setminus \text{IdealStage} \) must hold for each stage. Unfortunately, it would not be possible to simply express the constraints in this form without abandoning the notion of the specifications being represented simply by expressions that form production rules for a grammar, which raises an issue that is discussed further below. For the moment, therefore, the aim is to keep to this form of specification, and so the approach that is adopted is to introduce the notion of the intersection of two sequences, whose semantics is the set of strings which is common to the sets generated by the two sequences. Then, the specifications of the individual components of a system can be composed to form an alternative specification of the behaviour of the complete system, and the final specification will be the intersection of the two alternatives.

For any two sequences \( s_1 \) and \( s_2 \) the intersection will be denoted by \( s_1 \cap s_2 \). It can be defined in two ways: one, which represents the semantics directly, is effectively a definition in terms of the canonical forms of the sequences, and can be written as \( s_1 \cap s_2 = \bigcup\{s : (s \subseteq s_1 \land s \subseteq s_2)\} \). The other is an algebraic definition, which again is recursive, and for which the principal axioms are

\[
\begin{align*}
    s_1 \cap s_1 &= s_1, \\
    (s_1 ; s_2) \cap (s_1 ; s_2) &= (s_1 ; s_2), \\
    (s_1 | s_2) \cap (s_1 | s_2) &= (s_1 | s_2) \text{ and} \\
    (s_1 | s_2) \cap s_1 &= s_1,
\end{align*}
\]

where \( s_1 \) and \( s_2 \) are any sequences of actions. Thus, to apply this approach to the pipeline one would in principle define

\[
\text{Stage}_i = (\text{IF } i = 1 \text{ THEN Start } ! ? \text{ ELSE move}_{i-1} \text{ Fl } ; \text{ move}_i)^\star
\]

Pipeline2 = \( \prod_{i=1}^{3} \text{Stage}_i \)

and then specify the overall behaviour of the pipeline as

Pipeline3 = Pipeline1 \cap Pipeline2

\section*{Synchronisation}

Unfortunately, it turns out in practice that the above definition of Pipeline2 is incorrect. If the component of the parallel composition \( \text{Stage}_2 \parallel \text{Stage}_3 \) is considered for a single repetition of the operation of each stage, it will be found to include strings such as \( \text{move}_1 ; \text{move}_2 ; \text{move}_3 \) (ie two actions of stage 2 followed by two actions of stage 3), whereas the correct description should be the sequence \( \text{move}_1 ; \text{move}_2 ; \text{move}_3 \). The problem is that the definition of the parallel composition operator assumes that the sequences being composed are independent, whereas in this case it is plain that they are not. Instead, because the individual sequences represent separate components of the system, when they are composed they should be synchronised, so that common actions (in this case \( \text{move}_2 \)) should be forced to occur simultaneously, and in the resultant string should only appear once (as a common action) rather than twice.

It may appear that trying to solve this problem by specifying the behaviour of individual components and then composing them in a synchronised fashion blurs the distinction between the dataflow algebra methodology and the process algebra methodologies such as COSY, but there is still a significant difference. In particular, in COSY there are no restrictions on the number of processes that may incorporate a particular action, and hence which may need to synchronise on it. Here, however, because the actions represent messages passing down individual channels, and each channel connects exactly two processes in a unidirectional fashion, where synchronisation is required on a particular action it can only be between exactly two processes, and this greatly simplifies the treatment.

More generally, it is arguable that whether one considers a system from the process viewpoint or the communications viewpoint, it should still be possible to express both the overall system-wide behaviour and the behaviour of individual components. From this perspective, therefore, it could be argued that the fact that the dataflow algebra approach integrates these two aspects of the behaviour of a system is not a weakness: rather, it would be a weakness if it did not provide clear links between the two. Indeed, if anything this approach integrates these two aspects more naturally through the definition of synchronised composition than do many of the process algebra approaches, where the emphasis on constructing the overall behaviour as a synthesis of the individual components means that they do not adequately describe the behaviour at the level of abstraction of the overall system. Nevertheless, the fact of having to revert to process-style specification in
this way does appear to be a weakness in this mode of use of the dataflow algebra approach, and alternatives are outlined later in the paper.

To represent the way in which two components synchronise, the common alphabet \( C(s_1, s_2) \) can be defined for any two sequences \( s_1 \) and \( s_2 \) as the set of actions which are contained in both sequences (ie in the sets of strings generated by them), so that these will be the actions over which the two sequences will need to synchronise. If the common alphabet is empty, then of course there will be no synchronisation between the two sequences, and so the normal interleaving will describe their composition. Otherwise, there will be actions on which the two sequences do need to synchronise, but it is possible that in the canonical forms of the sequences there may still be some strings which do not contain any of the common actions, and so these will still need to composed by normal interleaving.

This can be formalised by defining that for a set of actions \( A \), a sequence may be either fully dependent on \( A \) (if all the strings that it generates contain one or more of the actions in \( A \)), partly dependent on \( A \) (if some of the strings contain actions in \( A \)), or independent of \( A \) (if none of the strings contain actions of \( A \)). For a sequence that is partly dependent on \( A \) two components can then be defined, namely the dependent part and the independent part: for a fully dependent sequence the independent part will be defined to be \( \phi \), as will the dependent part of an independent sequence. From these, an operation of merging two sequences can then be defined, which will compose their dependent parts in a fashion that respects the synchronisation of the actions in the common alphabet, and will form the normal parallel composition of the independent parts.

The dependency relation and the operations to construct the dependent and independent parts of a sequence can all be defined algebraically via structural recursion. For a set of actions \( A \) the basic axioms that define whether a sequence is fully or partly dependent on \( A \) or independent of it are as follows:

\[
\begin{align*}
    a; s &\text{ is fully dependent on } A \text{ if } a \in A, \\
    &\text{otherwise it is fully dependent, partly dependent or independent if } s \text{ is}, \\
    s_1 | s_2 &\text{ is fully dependent on } A \text{ if both } s_1 \text{ and } s_2 \text{ are}, \\
    &\text{independent of } A \text{ if both } s_1 \text{ and } s_2 \text{ are, and partly dependent otherwise;} \text{ and} \\
    \varepsilon &\text{ is independent of } A.
\end{align*}
\]

The operator that generates the part of a sequence \( s \) that is dependent on \( A \) is then denoted \( \text{Dep}_A(s) \), and if \( s \) is fully or partially dependent on \( A \) then it is defined by the following axioms:

\[
\begin{align*}
    \text{Dep}_A(a) &= a \text{ for all actions } a \text{ (including } \varepsilon \text{ and } \phi), \\
    \text{Dep}_A(s_1 \mid s_2) &= \text{Dep}_A(s_1) \mid \text{Dep}_A(s_2) \text{ for any sequences } s_1 \text{ and } s_2, \\
    \text{Dep}_A(s_1; s_2) &= \\
    &\text{if } s_1 \text{ is fully or partly dependent on } A \text{ and } s_2 \text{ is fully or partly dependent on } A \text{ then } \text{Dep}_A(s_1); \text{Dep}_A(s_2), \\
    &\text{if } s_1 \text{ is fully or partly dependent on } A \text{ and } s_2 \text{ is independent of } A \text{ then } \text{Dep}_A(s_1); s_2, \text{ and} \\
    &\text{if } s_1 \text{ is independent of } A \text{ then } s_1; \text{Dep}_A(s_2).
\end{align*}
\]

Similarly, the operator that generates the part of a sequence \( s \) that is independent of \( A \) is denoted \( \text{Ind}_A(s) \), and if \( s \) is partly dependent on \( A \) then it is defined by the following axioms:

\[
\begin{align*}
    \text{Ind}_A(a) &= a \text{ for any action } a \text{ that is not in } A, \text{ and} \\
    \text{Ind}_A(s_1 \mid s_2) &= \text{Ind}_A(s_1) \mid \text{Ind}_A(s_2) \text{ and} \\
    \text{Ind}_A(s_1; s_2) &= \text{Ind}_A(s_1) \mid \text{Ind}_A(s_2) \text{ for any sequences } s_1 \text{ and } s_2.
\end{align*}
\]

Given these definitions, the operation that merges two sequences \( s_1 \) and \( s_2 \) over a common alphabet of actions \( C \), where \( s_1 \) and \( s_2 \) are both supposed to be fully dependent on \( C \), is denoted by \( s_1 M_C s_2 \). For the cases where \( s_1 = \phi \) or \( s_2 = \phi \) or \( s_1 \) or \( s_2 \) are independent of \( C \) it is defined that \( s_1 M_C s_2 = \phi \). Otherwise it is defined for alternation by a pair of symmetrical axioms, of which the left-hand version is:

\[
(s_1; (s_2 \mid s_3); s_4) M_C s_5 = ((s_1; s_2; s_4) M_C s_5) \mid ((s_1; s_3; s_4) M_C s_5)
\]

where \( s_1 \) is independent of \( C \) (which applies in the case where \( s_1 = \varepsilon \), since \( \varepsilon \) is independent of any set of actions). For sequential composition there is an axiom to define \((s_1; a_1; s_3) M_C (s_2; a_2; s_4)\), where the sequences \( s_1 \) and \( s_2 \) are independent of \( C \) and the actions \( a_1 \) and \( a_2 \) are in \( C \), as the following series of cases:
if \(a_1 \neq a_2\) then \(\phi\),
if \(a_1 = a_2\) and \(s_3\) and \(s_4\) are both independent of \(C\) then \((s_1 || s_2); a_1; (s_3 || s_4)\),
if \(a_1 = a_2\) and \(s_3\) and \(s_4\) are both dependent on \(C\) (ie fully dependent) then \((s_1 || s_2); a_1; (s_3 \mathcal{M}_C s_4)\),
otherwise \(\phi\).

Finally, the general operation that performs the merge of two sequences \(s_1\) and \(s_2\), synchronised where necessary, is then denoted \(s_1 \mathcal{M} s_2\), and it is defined by the single axiom

\[
s_1 \mathcal{M} s_2 = (\mathcal{D}_{\mathcal{C}}(s_1) \mathcal{M}_C \mathcal{D}_{\mathcal{C}}(s_2)) \parallel (\mathcal{I}_{\mathcal{C}}(s_1) \parallel \mathcal{I}_{\mathcal{C}}(s_2)).
\]

It is an important property of these definitions that where two components whose behaviours (represented by sequences \(s_1\) and \(s_2\)) are such that they would deadlock if composed together in a synchronised fashion to form a subsystem, then the term \(\mathcal{D}_{\mathcal{C}}(s_1) \mathcal{M}_C \mathcal{D}_{\mathcal{C}}(s_2)\) that represents the allowable behaviour for the synchronised parts will reduce to the forbidden action \(\phi\). What is not clear yet is whether the methodology can also predict whether such partial deadlock would then cause the rest of the system to deadlock as well, or whether the other parts of the behaviour represented by the term \(\mathcal{I}_{\mathcal{C}}(s_1) \parallel \mathcal{I}_{\mathcal{C}}(s_2)\) would still be able to proceed. This is an aspect of the methodology that requires further work, as does the question of whether this model can predict the possibility of livelock (although it is hypothesised that it can not), and the issue of the relationship between this model of deadlock and models based on the existence of client-server relationships between components of systems [18].

Two other important properties of this synchronised composition operation are that (like ordinary parallel composition) it is commutative and associative, so that \(s_1 \mathcal{M} s_2 = s_2 \mathcal{M} s_1\) and \((s_1 \mathcal{M} s_2) \mathcal{M} s_3 = s_1 \mathcal{M} (s_2 \mathcal{M} s_3)\). Consequently, synchronised composition can also be defined over a set of sequences, and for an indexed set \(a_i\) it is denoted \(\mathcal{M}_{a_i}\). Thus, in the case of the pipeline, the specification that is required is

\[
\text{Pipeline}_4 = \bigcap_{i=1}^{4} \text{Stage}_i
\]

\[
\text{Pipeline}_5 = \text{Pipeline}_1 \cap \text{Pipeline}_4
\]

Similarly, in the case of the farm, to specify the individual workers the definitions of \(\text{input}_{\text{ker}}\) and \(\text{output}_{\text{ker}}\) need to be modified so as to be valid for \(\text{Worker}_{4}\) as well as for the other three, and then the specification that is required is

\[
\text{DoWork}_i = (\text{ToWork}_i; \text{FromWork}_i)^+
\]

\[
\text{OneWorker}_i = (\text{input}_i^+ \mathcal{M} \text{DoWork}_i \mathcal{M} \text{output}_i^+)
\]

\[
\text{Farmer} = \prod_{j=1}^{4} (\text{PassTo}_j; \text{PassFrom}_j)^+
\]

\[
\text{Farm3} = \text{Farmer} \mathcal{M} (\bigcap_{i=1}^{4} \text{OneWorker}_i)
\]

\[
\text{Farm4} = \text{Farm2} \cap \text{Farm3}
\]

Specifying the Semantics of Actions

It has already been noted that at the basic level of abstraction of this framework as presented so far, there is no way in which semantic aspects of the processing can be described, so that the causality between the messages that a process receives as inputs and those that it generates as outputs can only be described in terms of the ordering of the sequences of actions that define the behaviour of that process. At the same time, if this framework is to facilitate the construction of formal specifications of complete systems, which is one of its goals, then it has to be possible to define these semantics within the framework.

It is to make this possible that the framework has incorporated the notion that the expressions written in the data flow algebra can also be regarded as productions in a grammar, for it allows the basic context-free grammars that have been presented so far to be extended to attribute grammars, where the attributes that are attached to the symbols in the productions will be the relevant components of the appropriate formal specifications of the semantics. Thus, given that actions have the form \(\text{source} ! \text{channel} ? \text{destination}\), it is envisaged that the specification of the data type of the message would form an attribute of the \text{channel} element, the specification of the operation which produces the data contained in the message would form an attribute of the \text{source} element, and the specification of the operation which
utilises the data when it is received would form an attribute of the destination element. Thus, within the framework the sequences of actions will simply constitute the top level of a specification, which we may describe as the syntactic level, and the attributes provide a more detailed level, which we may describe as the semantic level.

It then appears that this principle of attaching the specifications as attributes to the productions ought to be applicable to any formal specification methodology which allows the construction of abstract type and operation specifications. Work is currently in progress to investigate the application of this framework to specifications written in OBJ3 [19], and both the B method [20] and Z [21] are other approaches to specification that need to be fitted into this framework as well. The current work is aimed at determining the composition rules which are needed to apply to the operators of sequencing, alternation etc to enable the attributes for complete productions to be synthesised from the attributes of individual actions. One problem which has already been encountered is that there needs to be a way of defining the initial state of a process before it has received any messages, but it appears that this can be solved by utilising the technique sometimes adopted within formal grammars, of augmenting the grammar with special start and stop symbols that bracket the non-terminal representing the top-level production, so that the specification of the initial state of the system can then be attached as an attribute to the start symbol.

The other aspect of using attribute grammars in this way that needs to be investigated is the question of whether they can be used to impose the sort of restrictions on a specification that were being discussed above in the context of the pipeline example. This could be done, for instance, by introducing into the grammar an attribute to correspond to the number of items of data in each stage of the pipeline, and defining that for each action the value of this attribute for the source element is decreased by one while the value for the destination element is increased by one, so that the restriction can then be imposed that a sequence is only legal if at no point in it does the value of this attribute exceed the limits on the capacities of the stages. Such an approach would be very similar in style to the way in which attribute grammars for programming language impose restrictions on the language such as limits on the permissible lengths of identifiers, so it is not as unnatural an approach as might at first sight appear. The issue of whether it would provide a "better" model for a system than either the more process-based approach which was considered above, or than an approach in which the constraints are defined implicitly in a specialised version of the parallel composition operator, is one which still requires further investigation.

Conclusions

It has been demonstrated that this dataflow algebra approach makes it possible to write specifications of the behaviour of parallel or distributed systems in a way that matches naturally the representation of the system structures as data flow diagrams. Using the operations of the algebra, the method allows both high-level specifications of the overall behaviour of a system to be written, and also more detailed specifications of individual components within it. Furthermore, the method allows these two aspects to be integrated in a natural fashion, and it allows the relationships to be defined between the different levels of detail that are implicit in the hierarchical structure of data flow diagrams.

While the basic syntactic structure of the method has been established, there is of course much scope for developing the semantic aspects further. As well as trying to apply it to larger case-studies than those presented here, more work is needed to establish completely that the extension to attribute grammars does fully accommodate formal specifications of the semantics of a system in the way that has been suggested, and also to establish the validity of using attributes to embody semantic restrictions on the specifications. Also, since the dataflow algebra approach is intended to be complementary to process algebra descriptions of systems, the relationships between the two types of model need to be analysed much further than just the demonstration given above that basic synchronisation of processes can be defined within the dataflow algebra. The issues relating to the detection of deadlock also require much further investigation. These aspects are particularly important, as there is a need for formal results from process algebra models to be transferrable to methods of system design that are based on more informal representations such as data flow diagrams. Thus, as well as being a valuable specification method in their own right, dataflow algebras are also likely to be an essential tool in making these formal results more applicable to the process of system design.

References