Dataflow Algebra Specifications of Pipeline Structures

A. J. Cowling & M. C. Nike

Verification and Testing Research Group,
Department of Computer Science,
University of Sheffield,
Sheffield, S10 2TN, United Kingdom

Email: A.Cowling@dcs.shef.ac.uk
Telephone: +44 114 222 1823   Fax: +44 114 278 0972

Abstract

The original description of the dataflow algebra approach to specifying concurrent systems included some discussion of various pipeline structures, and of several alternative ways for dealing with the problems that these presented. This report outlines the strengths and weaknesses of these different alternatives, and proposes a more general approach for dealing with the main problems that arise in modelling pipeline structures. This approach has led to a more rigorous definition of the syntax for dataflow algebra specifications, and this definition is summarised here, along with an identification of the aspects that still need further work before a complete definition can be produced.

Key Words and Phrases

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

Introduction

An earlier report [1] (which will be referred to as "the original report" from now on) introduced dataflow algebras as a technique for specifying concurrent systems. This technique was based on the idea of trying to define a formal semantics for dataflow diagrams (DFDs from now on), as these diagrams are used in most systems analysis methodologies (e.g. SSADM [2], Yourdon [3], Booch's Rational method [4], etc). As well as achieving this, dataflow algebras also provide a formalisation of the concept of "use case", which is a fundamental concept in other methodologies, such as Jacobson's Objectory [5].

Levels of Specification

Underlying much of the material in the original report was the idea that a concurrent system can be described at three different levels of detail. The top level is concerned with identifying the processes and the interconnections between them, which in dataflow algebra are assumed to behave like channels in occam [6], in that they are unidirectional connections between a single source process and a single destination process. In describing this level an analogy was drawn with the idea of defining the lexemes to be used in a grammar, and the term "lexical level of specification" has been used to describe it. In terms of DFD models for systems, however, a more accurate statement of the role of this level would be that it defines the topology of the diagrams, and so it will be referred to as the topological level in what follows.

The next level down is the one at which the dataflow algebra is used to define the sequences of actions that can occur within a system. This is done by writing what are effectively productions in a grammar, which will then generate all the legal sequences, and so this level of specification will be referred to as the syntactic level. Most of the original report was concerned with describing this level of specification, by defining the operations of the algebra and the axioms which govern their behaviour. As will be seen below, though, some of the definitions that were given were not as rigorous as they might have been, and in the light of subsequent work it has become apparent that better approaches could be found for tackling some of the problems that arose in the examples being considered.

Much of the rest of this report is therefore concerned with discussing these points and describing some of these alternative approaches, and in particular with making the basic definitions more rigorous. In doing this, however, the intention is that this report should be capable of standing on its own as a presentation of the current state of the development of the dataflow algebra approach. It should therefore only be necessary for a reader to refer to the original report if they actually wish to study in more detail how changes have taken place since that report was written.

The third level was only discussed briefly in the original report, and this is the level that is concerned with specifying the values that are communicated between processes, when the actions that are defined in the syntactic level occur. It thus
seems natural to refer to this level as the semantic level, and it is defined in terms of embedding into the actions of the grammar specifications of the operations that need to be invoked within the relevant processes, in order to both generate the data values that will be passed, and to handle those data values when they are received. The effect of this embedding process is to turn an ordinary grammar produced at the syntactic level into an attribute grammar [7], where the attributes associated with individual actions will be actual attributes and those associated with sequences of actions will be synthesised attributes.

Most of the work that has been done so far on this semantic level of specification has required the values of these attributes to be expressed in OBJ-3 [8], and in particular this approach has been used in building a case study of the alternating-bit protocol [9]. There is also a need to investigate how other approaches to specification, such as Z [10], the B method [11] or COLD [12], can be fitted into this level, but that issue is not addressed in this report. Consequently, this report is primarily concerned with the topological and syntactic levels of specification, and the semantic level is only considered briefly in the final section, where the issue of a concrete syntax is dealt with.

**Pipeline Structures**

The main concern in this report is to improve on the treatment of pipeline structures from the one suggested in the original report. The basic example that was considered there was a structure like the one illustrated in figure 1, in which there are three intermediate stages (P[1], P[2] and P[3]), together with input and output processes (Start and End respectively). These are then linked by channels (To[1] to To[4]), down which data items pass along the pipeline: in principle it is assumed that these may be different types of items at each stage, as a consequence of the processing done in each stage, but in practice this can be ignored in considering the topological and syntactic levels of specification.

![Figure 1: A Three-Stage Pipeline.](image)

The basic passage of one item down this pipeline can be specified as:

\[
\text{move}_i = \text{IF } i = 1 \text{ THEN Start ELSE } P_{i-1} \text{ FI } ! \text{ To}_i \text{ ? IF } i = 4 \text{ THEN End ELSE } P_i \text{ FI}
\]

\[
\text{OneItem} = \sum_{i=1}^{4} \text{move}_i
\]

This illustrates four features of the notation. The first is that the basic units are individual actions, where in general any action is written in the form \(<\text{source process}> ! <\text{channel}> ? <\text{destination process}>\). The second feature is that there can be families of actions, distinguished by subscripts, so that \(\text{move}_i\) denotes a single action within such a family: the intention is that subscripts should be literal values (ie fixed at the point where they are used) rather than varying dynamically. The third feature is that sequences of actions can be written down, where \(;\) denotes the sequencing operator, and the fourth feature is that for families of actions (or, in some cases, repeated actions) the \(\Sigma\) notation can be used as a shorthand form, so that the expression for \(\text{OneItem}\) could also be written as

\[
\text{OneItem} = \text{move}_1 ; \text{move}_2 ; \text{move}_3 ; \text{move}_4
\]

Since it was assumed in the original example that each stage is only capable of dealing with one item at a time, there can only be three items in the whole pipeline, and so it was suggested initially that the specification for the whole pipeline could possibly be written as

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

This illustrates three further features of the notation, one of which is the use of \(s^*\) and \(s^+\) to denote respectively zero or more and one or more repetitions of the sequence \(s\). The second feature is the notation of parallel composition of sequences of actions, where \(||\) denotes the parallel composition operator, so that \(s1 || s2\) denotes the sequence consisting of all possible interleavings of the actions of \(s1\) and \(s2\) (This in turn requires the notion of a sequence which may have several variants, which is referred to as alternation, where \(|\) denotes the alternation operator, so that \(s1 | s2\) denotes the sequence which consists of either \(s1\) or \(s2\).) The third feature is then the use of the \(\Pi\) notation as a shorthand for multiple
parallel compositions, where (as in this case) the sequences being composed do not necessarily have to depend on the index for the $\Pi$. For completeness, it should be added that there is also shorthand operator for multiple alternations, namely $U$.

Returning to the example being discussed above, the original report then went on to point out that the specification $\text{PipeLine}1$ would not in fact be correct, because as well as valid sequences of actions it would also generate sequences such as

$$\text{move}_1 ; \text{move}_1 ; \text{move}_1 ; \ldots$$

which would not be valid, given the constraint that each stage can only hold one item.

Two main approaches were discussed in that report for dealing with this problem. The first of these approaches was to introduce the idea of the intersection operator, so that a second specification could be constructed to represent what is effectively the process algebra approach to modelling the system. This approach would require a specification to be built for each of the three stages of the pipeline, in the form

$$\text{Stage}_i = (\text{move}_i ; \text{move}_{i+1})^*$$

and then the whole pipeline would be composed of these three stages running in synchronisation, which could be specified as

$$\text{PipeLine}2 = \bigcap_{i=1}^{3} \text{Stage}_i$$

where $M$ denotes the synchronous merge operation, which is not a primitive operation in the dataflow algebra, although it is in process algebras. A definition for this operation was given in the original report, but needs to be discussed further, and this comes later on in this report. The intersection of the two specifications would then be denoted as

$$\text{PipeLine}3 = \text{PipeLine}1 \cap \text{PipeLine}2$$

From the theoretical point of view this is probably a perfectly legitimate way of writing a specification, but from the practical point of view it seems to be very sloppy, and hence rather unsatisfactory. Of course, in the early stages of requirements analysis it might be perfectly acceptable to describe the behaviour of some component from several different points of view, and then suggest that the required behaviour is to be found in the common subset of these different viewpoints, and this is an aspect of the practical use of dataflow algebras which still needs to be investigated.

By the time that precise specifications are being produced, however, one would expect that it ought to be possible to capture a precise model of that common subset from each of the viewpoints, so that then the specifier should be able to demonstrate the accuracy of the models by formally showing that some suitable equivalence relationship holds between them. In this particular case there are two such views: a system-wide view of the operation of the pipeline, which is represented by $\text{PipeLine}1$, and a process-oriented view which is represented by $\text{PipeLine}2$. Thus, the implication here is that one would need to be able to refine the model expressed in $\text{PipeLine}1$ independently of that expressed in $\text{PipeLine}2$.

**Constrained Parallel Composition**

The other approach that was suggested in the original report for refining a specification like $\text{PipeLine}1$ was to extend the use of the attribute grammar concept, by using attributes to express constraints on the system such as the limitation on the capacity of each stage. This has been investigated, and it would require two features to be added to the model. One feature would be to provide attributes to express the effect on the capacity of each action, and this could be done in a straightforward fashion within the algebra as it had been defined. The other feature would be to provide some way of expressing the fact that the interleaving of the individual sequences $\text{Oneltem}^*$ was to be limited by the constraints implicit in these attributes. This would not be so straightforward, as it would involve the creation of some new construction within the algebra in order to denote a parallel composition that was to be subject to such constraints.

If such a new construction were to be added to the algebra, however, then potentially other possibilities might be opened up as well, apart from expressing the constraints on the interleaving in terms of attributes. In this particular case, the use of attributes in this way would have the effect of limiting the behaviour of any one stage, and in this case would mean that it could only behave as represented by the specification $\text{Stage}_i$. Given that the effect produced by this constraint could be expressed more directly in terms of this specification, it was therefore concluded that it would be better to pursue such a direct approach, rather than trying to use the attribute mechanism in this way.
Consequently, the approach that has now been adopted is to introduce a constrained form of the parallel composition operator, in which only the only interleavings permitted of the sequences that are being composed in parallel are those that satisfy the constraints. These constraints should be expressed directly in terms of sequences of actions if possible, and we have established that this certainly is possible for the most common form of constraint. This form is called a buffering constraint, because in many of the cases where it is needed it will be used to express limitations on the number of items that can be buffered within a particular process.

Buffering Constraints

The constrained parallel composition operator requires two kinds of operands: the sequences to be composed in parallel, and the set of sequences representing the constraints that are to be applied to the composition. If the first of these kinds of sequence is denoted by \( s_1, s_2, \ldots \), and the second by \( c_1, c_2, \ldots \), then constrained parallel compositions can be written in either of the two forms

\[
cpc = s_1||c s_2 / \{c_1, c_2, \ldots\}
\]

or

\[
cpc = \prod c_i / \{c_1c_2, \ldots\}
\]

In each case, the meaning of this is that if \( cpc \) is restricted over the actions occurring in constraint \( c_i \), then the result will be consistent with the specification \( c_i \). In the original report the operation of restriction was only defined over a set of processes, but the extension to restricting over a set of actions is straightforward, and the same notation will be used for it, so that if \( s \) is a sequence and \( as \) a set of actions then the restriction will be denoted \( s \setminus as \). A notation is also needed for referring to the set of actions which occur in a sequence, which in CSP [13] would be known as its alphabet, and so for any sequence \( s \) this will be denoted by \( \text{Alpha}(s) \). Thus, for any constrained parallel composition \( cpc \) and constraint \( c_i \) the effect of the constraint can be written as requiring the condition \( cpc \setminus \text{Alpha}(c_i) \subseteq c_i \) to hold, and if the components of \( cpc \) are \( s_1 \) and \( s_2 \) then also the condition \( cpc \subseteq s_1 || s_2 \) must hold, and equivalently for the \( \Pi \) form.

In the case of the pipeline example illustrated in figure 1, therefore, the specification that is required would be written as a constrained parallel composition, as

\[
\text{PipeLine4} = \prod_{i=1}^{3} (\text{OnelItem}^+) / \{\text{Stage}_1, \text{Stage}_2, \text{Stage}_3\}
\]

Other Forms of Constraint

While the buffering constraint is the most common sort, it is not the only form of constraint, and in particular a situation was encountered in the case study of the alternating-bit protocol referred to above, where a form of constraint was required that could best be described as a no-overtaking constraint. The kind of situation where this sort of constraint might occur can be illustrated by the system shown in figure 2, where two separate streams of data (which will be referred to as \( A \) and \( B \) in what follows) are being multiplexed through a single pipeline.

![Figure 2: A Pipeline Carrying Two Streams of Data.](image)

The basic dataflow algebra specification of this system can be written as follows.

\[
\text{One}_A\_\text{Item} = \text{Start} ! \text{Ain} ? \text{In} ; \text{In} ! \text{X} ? \text{Out} ; \text{Out} ! \text{Aout} ? \text{End}
\]

\[
\text{One}_B\_\text{Item} = \text{Start} ! \text{Bin} ? \text{In} ; \text{In} ! \text{X} ? \text{Out} ; \text{Out} ! \text{Bout} ? \text{End}
\]
The idea of the no-overtaking constraint then arises from the possibility that there might be a requirement for such a system that an A item may not overtake a B item (or vice-versa) if they are both passing through the pipeline at the same time. Such a requirement would arise in situations where it was important for the pipeline to respect the order in which items were input to it by the Start process, so as to ensure that they arrived at the End process in the same order.

Ways of formulating such constraints have been investigated, and it appears that it should be possible to define them in terms of identifying sets of channels within the system that would be intersected by what can be called "cut lines", such as the three lines that are shown dotted in figure 2. For each of the cut lines, the definition of the constraint then needs to focus on the actions that involve channels that intersect that cut line.

The effect of the interleaving can be described for these sets of actions in terms of the order in which they are selected from the various parallel sequences: in this case, A and B. For instance, for the cut line through the channels Ain and Bin, a particular interleaving of A and B would give rise to some ordering for occurrences of the actions Start ! Ain ? In and Start ! Bin ? In, depending at each point in the overall sequence whether an action was selected from A or B. The basic constraint would then simply require that this ordering of the sequences should be the same for each of the cut lines, although it also appears that there might be situations where more complex relationships between these orderings were required.

While it therefore appears that such a formulation should in principle be possible for this sort of constraint, what is not yet clear is whether in practice it is necessary. In particular, if the stages in the pipeline only have a capacity of one item each (which was the situation in the case study of the alternating-bit protocol) then it would anyway be impossible for one item to overtake another, since this would require the stage in which the overtaking was to occur to be buffering at least two items. Consequently, in this sort of case there would be no need for a separate form of constraint to express the situation, as the required property could have been expressed in terms of a buffering constraint. It could well be the case that this would be true more generally, and if so this would suggest that the no-overtaking constraint should be treated as a less fundamental construction than the buffering constraint.

Having said that, though, the no-overtaking constraint does appear to represent a qualitatively different sort of property from the buffering constraint, so that there could be an argument for claiming that the modelling language of dataflow algebra specifications should provide a natural means of expressing it. At this stage in the work this has to be regarded as an issue that requires further investigation, by means of suitable case studies, as indeed do some of the other concepts proposed here. In particular, it also appears that there could be significant applications for the notions of representing cut lines through systems in terms of collections of channels, and also the notion of defining the interleavings generated by parallel compositions of sequences in terms of the order in which elements are selected from one sequence or the other. These issues will therefore be deferred until they can be investigated further, and so for the purposes of the rest of this report only buffering constraints will be considered, and no-overtaking constraints will be ignored.

The Abstract Syntax of Sequences

In the course of investigating the idea of constrained parallel composition, it was realised that there was a need for a more rigorous definition of the abstract structure of sequences than had been provided in the original report, in order to ensure that definitions constructed by structural recursion were in fact well founded. An abstract syntax was therefore developed for the structure of expressions, or more precisely a number of versions of such a syntax were developed.

The first of these attempted to define structures that involved sequencing (using the ; operator), alternation (using the | operator) and parallel composition (using the || operator), and it defined these structures using repetition rather than recursion. Thus, given the basic productions

\[
\begin{align*}
<\text{seq-sym}> & ::= \text{","} \\
<\text{alt-sym}> & ::= \text{"|"} \\
<\text{par-sym}> & ::= \text{"||"}
\end{align*}
\]

in which double quotes are used to denote terminal symbols, then the sentence symbol <sequence> is defined as follows, using the normal meta-symbols of extended BNF, and treating <action> as though it were a terminal symbol.

\[
\begin{align*}
<\text{sequence}> & ::= <\text{par}> \\
<\text{par}> & ::= <\text{alt}> \{ <\text{par-sym}> <\text{alt}> \}^*
\end{align*}
\]
In attempting to use this syntax, it was quickly recognised that there were various problems with it. One of these was concealed by the use of repetitions, and so these needed to be replaced by the equivalent recursions, and this point is discussed further below. A more fundamental issue, however, was that there are actually at least two different levels of abstraction involved, and the syntax needs to reflect these. At the most basic level, sequences of actions can be defined entirely in terms of sequencing and alternation, and so there was a need for a syntax to define this level of abstraction, which will be referred to as the basic abstract syntax. Then, "syntactic sugar" such as repetition constructions can be added to this, although without requiring any modification to the underlying abstract syntax. Finally, an extended abstract syntax needs to be defined, which recognises the importance of parallel compositions by including them as though they were basic constructions.

The initial attempt at producing the basic syntax was therefore as follows, where the unchanged productions for the terminal symbols have not been repeated.

\[
\begin{align*}
<sequence> & ::= <alt> \\
<alt> & ::= <alt-elem> | <alt-elem><alt-tail> \\
<alt-tail> & ::= <alt-sym><alt> \\
<alt-elem> & ::= <seq> \\
<seq> & ::= <seq-elem> | <seq-elem><seq-tail> \\
<seq-tail> & ::= <seq-sym><seq> \\
<seq-elem> & ::= <action> | <par>
\end{align*}
\]

Unfortunately, while this syntax is abstract in the sense of ignoring issues such as bracketing or operator priorities, it is not abstract enough, in that it does not incorporate some of the reduction rules that result from the properties of the operators. Consequently, this syntax is not abstract enough to be conveniently usable for defining structural recursions.

Specifically, the problem that was identified in trying to produce a rigorous axiomatic definition of the parallel composition operation was that, for the construction that might be written \( s_1 ; (s_2 ; s_3) ; s_4 \), this syntax would permit the part of the production tree corresponding to \( (s_2 ; s_3) \) to take the form

\[
\begin{align*}
\text{.... } & <seq-elem> \text{....} \\
\downarrow & <seq> \\
\downarrow & <seq-tail> \\
\downarrow & <seq-sym><seq>
\end{align*}
\]

whereas the associativity property of \( ; \) would mean that one would actually want this part of the tree to have to be flattened out, so that the whole tree would instead correspond to the equivalent construction \( s_1 ; s_2 ; s_3 ; s_4 \). Thus, a more abstract version of the syntax was developed, in which what is now called a \(<\text{proper-alt}\>\) must genuinely be an alternation (ie it must include at least one \(<\text{alt-sym}\>\), rather than being allowed to be something that is actually a \(<\text{seq}\>\).

Of course, the possibility still has to be left open that a \(<\text{seq}\>\) may genuinely consist of only one action, and so may not necessarily contain a \(<\text{seq-sym}\>\). On the other hand, the same argument about the flattening of structures to correspond to the associativity property also applies to the \( ; \) operator, and so where a \(<\text{seq}\>\) occurs as part of an \(<\text{alt}\>\) one also wants to guarantee that this is not just going to be a single \(<\text{alt}\>\). Thus, it is also necessary to introduce another construction in the form of a \(<\text{proper-seq}\>\), meaning one that does not just consist of a single element, and then define the situation where an \(<\text{alt-elem}\>\) consists of a single action as a separate base case. Thus, the resultant version of the basic abstract syntax is as follows.

\[
\begin{align*}
<sequence> & ::= <alt> \\
<alt> & ::= <alt-elem> | <proper-alt> \\
<proper-alt> & ::= <alt-elem><alt-tail> \\
<alt-tail> & ::= <alt-sym><alt> \\
<alt-elem> & ::= <action> | <proper-seq> \\
<proper-seq> & ::= <seq-elem><seq-tail> \\
<seq-tail> & ::= <seq-sym><seq> \\
<seq> & ::= <seq-elem> | <proper-seq>
\end{align*}
\]
<seq-elem> ::= <action> | <proper-alt>

The significance of this syntax is that it makes it clear that any definition by structural induction needs to consider three levels of structuring. At the first level there are two cases, corresponding to the two productions <alt> ::= <alt-elem> and <alt> ::= <proper-alt>. Similarly, at the second level there will be two cases, one corresponding to the production <alt-elem> ::= <action> (which acts as one of the base cases for any such structural induction) and the other corresponding to the production <alt-elem> ::= <proper-seq>. Finally, at the third level there will again be two cases, this time corresponding to the productions <seq-elem> ::= <action> (which will be the other base case for a structural induction) and <seq-elem> ::= <proper-alt>.

The Treatment of Repetition

The next stage in developing this abstract syntax is to extend it to provide the elements of "syntactic sugar" referred to earlier, in the form of the repetition constructions s* and s+. These are not fundamental constructions, in that both of them can be defined in terms of elementary recursions, but from the point of view of practical modelling they are very useful, and so it is important that they should be fitted rigorously into the framework provided by the abstract syntax. There are several different formulations which could be given for these two constructions, but the most natural one is that for any sequence s, the sequence s* should be defined directly by the recursion

\[ s^* = \varepsilon | s ; s^* \]

where \( \varepsilon \) denotes what is called the silent action, although it actually means that no action occurs at all, so that \( \varepsilon \) is a left and right identity for the ; operator. It therefore has a slightly different significance to the silent action in CCS [14], which is the action produced by synchronising an output action from one process with an input action to another process. Indeed, there is a very real sense in which any action in the dataflow algebra model of a system corresponds to what would be a silent action in a CCS model of that system.

Returning to the definition of repetition, given the above definition for s* the sequence s+ can then either be defined directly by the recursion

\[ s^+ = s | s ; s^+ \]

or indirectly as

\[ s^+ = s ; s^* \]

Alternatively, s* could be defined indirectly as

\[ s^* = \varepsilon | s^+ \]

and this would be valid irrespective of which form of definition was used for s+, although combining it with the indirect one would lead to a double recursion which would be inherently more complex than the single recursions implicit in the other forms of the definitions.

In terms of the abstract syntax, the implication of these definitions is that any occurrence of s* can be treated as a <proper-alt>, irrespective of which of the two definitions is used. By contrast, an occurrence of s+ can be treated as either a <proper-alt> or as a <proper-seq>, depending on which of the two forms of definition one chooses to adopt. In principle it might appear that treating it as a <proper-seq> could in some cases simplify the parse tree generated for a particular sequence, but in practice this is not the case, because it is equivalent to rewriting an occurrence of s+ as the sequence s ; s*, and the s* term in this will still need to be parsed as a <proper-alt>. Therefore, either definition of s+ will lead to an occurrence of a <proper-alt> in the abstract syntax, and so sequences of both the forms s* and s+ will therefore be treated in what follows as occurrences of a <proper-alt>.

The Definition of Parallel Composition

Having shown how the repetition constructions can be accommodated within the basic abstract syntax, the next stage in developing it is to use it to construct revised versions for the definitions that were presented in the original report of various non-primitive operators of the algebra. The most fundamental of these is the parallel composition operator, since although this is not a primitive operation within the algebra, its importance in the modelling language means that it will need to be treated as though it were a primitive in the extended abstract syntax which is to be developed below.
At the first level of syntactic structure the revised definition of the operation $s_1 \parallel s_2$ requires axioms for four different cases, which are generated by the various combinations of the possible forms for $s_1$ and $s_2$, where each can either have the form $\texttt{<alt-elem>}$ or $\texttt{<proper-alt>}$. In practice, though, these cases are simplified by the fact that the $\texttt{<proper-alt>}$ cases are symmetrical, so that the following two axioms cover all of them except the case where both $s_1$ and $s_2$ are $\texttt{<alt-elem>}$.

Where $s_1$ is a $\texttt{<proper-alt>}$, \quad $(s_1 \mid s_1b) \parallel s_2 = (s_1 \mid s_2) \mid (s_1b \parallel s_2)$

Where $s_2$ is a $\texttt{<proper-alt>}$, \quad $s_1 \parallel (s_2a \mid s_2b) = (s_1 \parallel s_2a) \mid (s_1 \parallel s_2b)$

Where $s_1$ and $s_2$ are both $\texttt{<alt-elem>}$, then the second level of structuring again gives rise to four cases corresponding to the various combinations of $s_1$ and $s_2$ having either of the forms $\texttt{<action>}$ or $\texttt{<proper-seq>}$.

The case where both have the form $\texttt{<action>}$ gives rise to one of the base cases, for which the axiom is:

$$a_1 \parallel a_2 = (a_1 ; a_2) \mid (a_2 ; a_1)$$

but for the cases where either $s_1$ or $s_2$ have the form $\texttt{<proper-seq>}$ it is necessary to go down to the third level of structuring, where on each side there are two sub-cases, depending on whether the head element of the sequence has the form $\texttt{<action>}$ or $\texttt{<proper-alt>}$.

The case where both head elements are actions gives rise to what is effectively a base case for the structural induction, although it is actually defined recursively, and the axiom for it is as follows.

$$(a_1 ; s_1) \parallel (a_2 ; s_2) = (a_1 ; (s_1 || (a_2 ; s_2))) \mid (a_2 ; ((a_1 ; s_1) || s_2))$$

For the other cases, which represent the recursions in the structural inductions, the axioms are symmetric (as with the first level), and so it is only necessary to give two forms, one for the construction of each side. These axioms are therefore as follows.

Where $s_1$ is a $\texttt{<proper-seq>}$ and its head is a $\texttt{<proper-alt>}$
$$((s_1a \mid s_1b) ; s_1t) \parallel s_2 = ((s_1a ; s_1t) || s_2) \mid ((s_1b ; s_1t) || s_2)$$

Where $s_2$ is a $\texttt{<proper-seq>}$ and its head is a $\texttt{<proper-alt>}$
$$s_1 \parallel ((s_2a \mid s_2b) ; s_2t) = (s_1 \parallel (s_2a ; s_2t)) \mid (s_1 \parallel (s_2b ; s_2t))$$

This therefore completes the definition of ordinary parallel composition: the equivalent for constrained parallel composition will be discussed below, along with the other operations that it relies on.

The Definition Of Synchronisation

Before dealing with this, it is appropriate to consider the definitions of the auxiliary operators involved in modelling the synchronised merge of two processes, since these also need to be revised. There are three such operators: a pair of what will be referred to as dependency operators, which are called $\texttt{Dep}$ and $\texttt{Ind}$, and then a synchronised merge operator $\texttt{MC}$ which is applied to some of the components produced by these dependency operators.

Here, the significance of the name "dependency operators" for the first two is that they effectively separate the dataflow algebra specification of an individual process into two components. One of these components will contain all the parts of the specification that are dependent on the set of common actions: that is, the set of actions that are common to the process in question and some other process with which it must synchronise. The other component will contain all the parts of the specification that are independent of these synchronising actions. Thus, $\texttt{Dep}_A(s)$ extracts the parts of a sequence $s$ that are dependent on a set of actions $A$, while $\texttt{Ind}_A(s)$ extracts the parts of $s$ that are independent of $A$. As with the definition above of parallel composition, there are a number of cases for each that need to be considered, depending on the structure of $s$ in terms of the abstract syntax.

The Dependent Component of a Sequence

To define $\texttt{Dep}_A(s)$, where $A$ is a set of actions and $s$ is any sequence, the first level of structuring is the case where $s$ is an $\texttt{<all>}$, which can have the form of either a $\texttt{<proper-alt>}$ or an $\texttt{<alt-elem>}$, and so leads to two sub-cases. The first of these, where $s$ is a $\texttt{<proper-alt>}$, is a straightforward one, and is defined by the axiom

$$\texttt{Dep}_A(s_1 \mid s_2) = \texttt{Dep}_A(s_1) \mid \texttt{Dep}_A(s_2)$$

If $s$ is an $\texttt{<alt-elem>}$ then the second level of structuring gives rise to two further cases, as the $\texttt{<alt-elem>}$ can be either an $\texttt{<action>}$ or a $\texttt{<proper-seq>}$. The case where $s$ is an $\texttt{<action>}$ gives rise to one of the base cases of the recursive definition, for which the axiom is
\[ \text{Dep}_A(a) = \text{if } a \text{ is in } A \text{ then } a \text{ else } \phi \]

Here, \( \phi \) denotes what is called the forbidden action, which is the action that forms the left and right identity for the alternation operator |, although (as will become apparent from the discussion below of the semantics of sequences) it also has another significance as well, in that because the forbidden action can not actually occur, it may also represent a situation in which a system deadlocks, or otherwise terminates abnormally.

The existence of the two special actions \( \phi \) and \( \varepsilon \) means that there are also two special cases that need to be defined, but these follow from the fact that neither \( \phi \) and \( \varepsilon \) can appear as common actions for the purpose of synchronising two sequences, and so they can not appear in any valid set \( A \) for the purpose of defining \( \text{Dep}_A \). Consequently, these two special cases can be stated as

\[ \begin{align*}
\text{Dep}_A(\varepsilon) &= \phi \\
\text{Dep}_A(\phi) &= \phi 
\end{align*} \]

Where \( s \) is not an \(<\text{action}>\), the other case in the definition of \( \text{Dep}_A(s) \) is that it may be a \(<\text{proper-seq}>\), and here the third level of structure again gives rise to sub-cases, depending on whether the head of the \(<\text{proper-seq}>\) is an \(<\text{action}>\) or a \(<\text{proper-alt}>\). If it is an \(<\text{action}>\), and this action is dependent on \( A \), then this means that the whole sequence must be dependent on \( A \). This is also true if the tail of the sequence \( s \) is fully dependent. Both these two conditions give rise to base cases, and the axiom for them is

\[ \text{Dep}_A(a ; st) = \text{if } a \text{ is in } A \text{ or } st \text{ is fully dependent then } a ; st \]

However, if the head action is not in \( A \), then the result is dependent on the tail of the sequence. Therefore the following two axioms are required.

\[ \begin{align*}
\text{Dep}_A(a ; st) &= \text{if } a \text{ is not in } A \text{ and } st \text{ is independent then } \phi \\
\text{Dep}_A(a ; st) &= \text{if } a \text{ is not in } A \text{ and } st \text{ is partially dependent then } a ; \text{Dep}_A(st)
\end{align*} \]

Where the head of the \(<\text{proper-seq}>\) is a \(<\text{proper-alt}>\), then there are a number of cases depending on whether or not the tail of the \(<\text{proper-seq}>\) is dependent on \( A \). Where the tail is fully dependent, the following axiom applies, which is also a base case for the recursion.

\[ \text{Dep}_A( (s1 | s2) ; st ) = \text{if } st \text{ is fully dependent then } (s1 | s2) ; st \]

If the tail of the \(<\text{proper-alt}>\) is partially dependent, then it requires further decomposition to extract the dependent part. As part of this decomposition, the \(<\text{proper-alt}>\) may need to be ‘flattened’ out so as to remove the second level of structuring, and to produce an alternation of two structures that both have the form of a \(<\text{proper-seq}>\). Consequently, the axioms for this case are

\[ \begin{align*}
\text{Dep}_A( (s1 | s2) ; st ) &= \text{if } st \text{ is partly dependent then} \\
&= \text{if } st \text{ is partly dependent then} \\
&\text{Dep}_A( (s1 | s2) ; st ) = \\
&\text{Dep}_A(s1 | s2) ; \text{Dep}_A(st) = \\
&\text{if } st \text{ is independent then} \\
&\text{Dep}_A(s1 | s2) ; st = \\
&\text{Dep}_A(s1) ; st ) | ( \text{Dep}_A(s2) ; \text{Dep}_A(st) ) \\
&\text{if } st \text{ is independent then} \\
&\text{Dep}_A(s1 | s2) ; st = \\
&\text{Dep}_A(s1) ; st ) | ( \text{Dep}_A(s2) ; st ) \\
\end{align*} \]

These axioms then complete the definition of the operator \( \text{Dep} \).

The Independent Component of a Sequence

The other dependency operator \( \text{Ind}_A(s) \) extracts the parts of a sequence \( s \) that are independent of the set of actions \( A \), and as with the operator \( \text{Dep} \) different possible forms of the sequence \( s \) lead to different cases. As previously, the first of these is where \( s \) is a \(<\text{proper-alt}>\), for which the following axiom is required:

\[ \text{Ind}_A(s1 | s2) = \text{Ind}_A(s1) | \text{Ind}_A(s2) \]

Where \( s \) is an \(<\text{alt-elem}>\) then again two different cases arises, since it can either be an \(<\text{action}>\) or a \(<\text{proper-seq}>\). Where \( s \) is an \(<\text{action}>\) then the following axiom is required, and as before this is a base case for the recursion,

\[ \text{Ind}_A(a) = \text{if } a \text{ is not in the set } A \text{ then } a \text{ else } \phi \]
As with $\text{Dep}$, there are two special cases of this definition for the actions $\phi$ and $\epsilon$, since again neither of them can appear in any valid set $A$, and so the two cases required are

$$
\text{Ind}_A(\epsilon) = \epsilon \\
\text{Ind}_A(\phi) = \phi
$$

Where $s$ is a $<\text{proper-seq}>$ its head can either be an $<\text{action}>$ or a $<\text{proper-alt}>$, and if the head is an action then the following axioms are required:

$$
\text{Ind}_A(a ; st) =
\begin{cases}
  \phi & \text{if } a \text{ is in } A \text{ or } st \text{ is fully dependent on } A \\
  a ; \text{Ind}_A(st) & \text{if } a \text{ is not in } A \text{ and } st \text{ is partially dependent on } A \\
  a ; \text{Ind}_A(st) & \text{if } a \text{ is not in } A \text{ and } st \text{ is independent}
\end{cases}
$$

Where $s$ is a $<\text{proper-seq}>$ with a $<\text{proper-alt}>$ as its head element, then the following axiom is required, and again there is the possibility that the resultant structure may need to be flattened to as to produce an alternation of structures that both have the form of a $<\text{proper-seq}>$:

$$
\text{Ind}_A( (s1 | s2) ; st) =
\begin{cases}
  \text{Ind}_A(s1 | s2) ; \text{Ind}_A(st) = & \text{if } st \text{ is partially dependent on } A \\
  (\text{Ind}_A(s1) ; \text{Ind}_A(st)) | (\text{Ind}_A(s2) ; \text{Ind}_A(st)) & \text{if } st \text{ is independent} \\
  (\text{Ind}_A(s1) ; st) | (\text{Ind}_A(s2) ; st) & \text{if } st \text{ is fully dependent}
\end{cases}
$$

Finally, before leaving the definitions of these two operations it is worth noting an invariant that was implicit in the arguments used to derive them, both in the original report and here, and which ought to be made explicit. This invariant is that, for any sequence $s$ and set of actions $A$, the equation should hold

$$
s = \text{Dep}_A(s) | \text{Ind}_A(s)
$$

Attempting to provide this invariant from the definitions is outside the scope of this report.

The Synchronisation Operator

The operator that performs the synchronous merge of two sequences is $M$, and its definition is as follows, which in form is unchanged from the original report. In it, $C$ has the same meaning as in the original report, in that it denotes the set of actions that are common to $s1$ and $s2$.

$$
s1 M s2 = (\text{Dep}_C(s1) \ M_C \text{Dep}_C(s2)) | (\text{Ind}_C(s1) || \text{Ind}_C(s2))
$$

The parallel operator has already been defined in terms of the structures used in the abstract syntax, but the definition of the operator $M_C$, which merges together the dependent parts of $s1$ and $s2$, now needs to be defined in these terms as well. As before, the first level of structure for $s1 M_C s2$ arises from the different combinations of cases generated when $s1$ and $s2$ are either a $<\text{proper-alt}>$ or $<\text{alt-elem}>$, and the following pair of symmetrical axioms are required for the $<\text{proper-alt}>$ cases.

Where $s1$ is a $<\text{proper-alt}>$,

$$
(s1a | s1b) M_C s2 = (s1a M_C s2) | (s1b M_C s2)
$$

Where $s2$ is a $<\text{proper-alt}>$,

$$
s1 M_C (s2a | s2b) = (s1 M_C s2a) | (s1 M_C s2b)
$$

Where $s1$ and $s2$ are both $<\text{alt-elem}>$ there are again four cases, due to the second level of structuring giving rise to the various combinations of $<\text{action}>$ and $<\text{proper-seq}>$. The base case is the case where both $s1$ and $s2$ have the form $<\text{action}>$, and the following axiom applies for this.

$$
a1 M_C a2 =
\begin{cases}
  \phi & \text{if } a1 = a2 \\
  a1 = a2 & \text{if } a1 \neq a2
\end{cases}
$$

For the case of a $<\text{proper-seq}>$ there are a number of different situations. The $<\text{seq-elem}>$ components that make up the $<\text{proper-seq}>$ can each either be an $<\text{action}>$ or a $<\text{proper-alt}>$. It is perfectly feasible for there to be, somewhere in the sequence, a $<\text{proper-alt}>$ sub-tree that is dependent on $C$. Hence, an axiom is required for when the situation arises where $s1$ or $s2$ contains such a construction. This gives a recursion for the structural induction,

$$
( s1h ; (s1a | s1b) ; s1t ) M_C s2 = ( (s1h ; s1a ; s1t) M_C s2 ) | ( (s1h ; s1b ; s1t) M_C s2)
$$
where \((s1a \mid s1b)\) is the dependent sub-tree, and \(s1h\) is independent of \(C\).

The final case to be considered is where both \(s1\) and \(s2\) have the form <proper-seq>, but neither of them contains a dependent <proper-alt> subtree. If both \(s1\) and \(s2\) have the form <proper-seq> then there must be at least one <action> making up the <proper-seq> in both of them that is dependent on \(C\), since otherwise the structure would not be occurring in something that had been produced by the operator \(\text{Dep}\). Therefore, this case can be written as

\[
s1 = s1a \mid a1 \mid s1b \text{ and } s2 = s2a \mid a2 \mid s2b
\]

where \(s1a\) and \(s2a\) are independent sequences, and \(a1\) and \(a2\) are actions in the common set \(C\). The axiom for this situation then contains both the base case for the induction (where \(s1b\) and \(s2b\) are both independent) and also a recursive part.

\[
(s1a ; a1 ; s1b) \quad \text{MC} \quad (s2a ; a2 ; s2b) =
\]

- If \(a1 \neq a2\) then \(\phi\)
- If \(s1b\) and \(s2b\) are both independent, then \((s1a \parallel s2a) ; a1 ; (s1b \parallel s2b)\)
- If \(s1b\) and \(s2b\) are both dependent, then \((s1a \parallel s2a) ; a1 ; (s1b \text{ MC } s2b)\)
- Otherwise \(\phi\)

This then completes the definition of \(\text{MC}\), and hence also of \(\text{M}\).

### The Restriction Operator

Having revised the definition of synchronisation, the last stage before defining the extended abstract syntax is to extend the definition for the ordinary parallel composition operator, so as to produce a comparable definition for constrained parallel composition. Since this will depend in part on the definitions of the restriction operator \(\setminus\) and the containment relation \(\subseteq\), revised definitions of these need to be given first.

For the restriction operator \(s \setminus \text{as}\), where \(\text{as}\) denotes some set of actions, the definition simply consists of the various cases for the different possible structures of \(s\). The first of these cases is where \(s\) is a <proper-alt>, and the axiom for this is

\[
(s1 \mid s2) \setminus \text{as} = (s1 \setminus \text{as}) \mid (s2 \setminus \text{as})
\]

The second case, for when \(s\) is an <alt-elem>, splits into two sub-cases, depending on whether the <alt-elem> is an <action> or a <proper-seq>. For the case of an <action> there are three axioms required, which are

- \(\epsilon \setminus \text{as} = \epsilon\)
- \(\phi \setminus \text{as} = \phi\)
- \(a \setminus \text{as} = \text{if } a \in \text{as } \text{then } a \text{ else } \epsilon\)

Where an <alt-elem> is a <proper-seq>, then the required axiom is

\[
(s1 ; s2) \setminus \text{as} = (s1 \setminus \text{as}) ; (s2 \setminus \text{as})
\]

and this axiom applies irrespective of the construction of the <seq-elem> which forms the head of the <proper-seq>. Consequently, this axiom completes the definition, since the two possible cases for the construction of this <seq-elem> are <proper-alt> and <action>, and both of these have already been defined by the previous axioms.

### The Containment Relationship

The definition of the containment relationship \(\subseteq\) is rather more complex, and in particular there is an issue which arises from trying to define it in terms of the abstract syntax, and which probably requires further investigation. This issue is that it is only practical to construct a definition based on the abstract syntax for sequences if the definition is restricted mainly to cases where the sequences being considered have the same syntactic structure, whereas one in principle one could have sequences that ought to be recognised as related, even though they have different syntactic structures.

A simple illustration of how this situation might arise is provided by the two sequences

\[
s1 = a1 ; (a2 \mid a3)
\]

and

\[
s2 = (a1 ; a2) \mid (a1 ; a3)
\]
where a₁, a₂ and a₃ are any actions. It will be apparent from the definitions given in the original report that algebraically these two sequences are the same, so that one would have s₁ = s₂ and hence for the containment relationship would also expect to have s₁ ⊆ s₂, as indeed one would for any other reflexive relationship. In terms of their abstract syntactic structure, though, the two sequences are not the same: s₁ has the form of a <proper-alt>, whereas s₂ has the form of a <proper-seq>. Consequently, any definition of ⊆ that is based on only considering sequences which have the same syntactic structure would have to treat these two as being not comparable, and so would clearly be incomplete.

To achieve a complete definition of the relationship, the following rules that are given for sequences having similar syntactic structures would need to be embedded into a framework that also allowed the axioms governing the equalities between sequences with different syntactic structures to be incorporated as well. The formulation of this embedding will require further work, and so it is not considered here, and the following definitions therefore deal only with the relationship between sequences that have the same abstract syntactic structure.

As with the other definitions, the starting point for defining s₁ ⊆ s₂ is the case where both of the sequences s₁ and s₂ are structured as a <proper-alt>. Here, the basic property that containment is trying to express is that each of the alternatives in s₁ is also an alternative in s₂, or (more precisely) is contained within one of the alternatives in s₂. This then carries the implication that s₂ could legitimately have more alternatives in it than s₁ does, in which case the extra alternatives in s₂ (ie those which do not contain one of the alternatives in s₁) are simply ignored. Consequently, while the definition is mainly constructed for sequences with the same syntactic structure, it does also need to cover the case where s₁ is an <alt-elem> and s₂ is a <proper-alt>, and in this case the definition should express the property that s₁ is one of the alternatives in s₂, or rather is contained within it. The axioms that express this are therefore as follows.

Where s₁ is a <proper-alt> and s₂ is a <proper-alt>,

\[(s₁a | s₁b) ⊆ (s₂a | s₂b) = (s₁a ⊆ (s₂a | s₂b)) \text{ and } (s₁b ⊆ (s₂a | s₂b))\]

In this axiom the first term on the right-hand side (ie the term before the and) involves comparing an <alt-elem> with a <proper-alt>, while the second term may also involve either a similar comparison, or another comparison of one <proper-alt> with another, depending on the structure of s₁b. For the case where an <alt-elem> is being compared with a <proper-alt>, the comparison is defined by

\[s₁ ⊆ (s₂a | s₂b) = (s₁ ⊆ s₂a) \text{ or } (s₁ ⊆ s₂b)\]

Here, the first term on the right-hand side will involve the comparison of one <alt-elem> with another, while again the second term will either involve the same kind of comparison, or the comparison of an <alt-elem> with a <proper-alt>, depending this time on the structure of s₂b. The comparison of one <alt-elem> with another will depend on their structures, since each could be an <action> or a <proper-seq>, but for the two to be comparable each must have the same structure. The case where one <action> is being compared with another is simple, and is defined as

\[a₁ ⊆ a₂ = (a₁ = a₂)\]

The comparison of one <proper-seq> with another then essentially expands on this, by requiring both their heads to be comparable and their tails, so that the axiom for this is

\[(s₁a ; s₁b) ⊆ (s₂a ; s₂b) = (s₁a ⊆ s₂a) \text{ and } (s₁b ⊆ s₂b)\]

This axiom applies irrespective of whether the heads have the same structure or not, and the cases where they do have the same structures (either both being <action>s or both being <proper-alt>s) have already been defined by the axioms above. In addition, though, it could well be the case here that s₁a was structured as an <action> while s₂a was a <proper-alt> in which this action appeared as an alternative, so that s₁a should be regarded as being contained in s₂a. To define this therefore requires one last axiom, similar to the one given above for an <alt-elem> and a <proper-alt>, and this is

\[a ⊆ (s₂a | s₂b) = (a ⊆ s₂a) \text{ or } (a ⊆ s₂b)\]

The first term of the right-hand side here could, of course, give rise to the case where an <action> is being compared with an <alt-elem> that is structured as another <action>, but the axiom for that case has already been given above. If s₂a was not structured as an <action>, however, but was a <proper-seq> instead, then these two structures would not be comparable, and so a definition does not need to be given for this case. Thus, the definition of the containment operator has now been completed, at least insofar as it can be stated in terms of the syntactic structure of sequences.
Constrained Parallel Composition

Now that restriction and containment have been defined, it is possible to give a definition of the constrained parallel composition operator. As might be expected, this broadly follows the structure of the definition of ordinary parallel composition, both in terms of the cases that need to be defined and the axioms that define them. There is, however, one complication that does not occur in ordinary parallel composition, and this arises from the way in which the effects of the constraints are incorporated into the definition.

Basically, the effect that is required is that, whenever the interleaving of sequences could give rise to alternations, one wishes to eliminate those components of the alternations that are not consistent with the constraints. This is done by introducing an auxiliary operation that is denoted SubjectTo, which incorporates this consistency check. The consistency check is formulated in terms of the containment operator, but the problem that arises is that many of the invocations of SubjectTo will need to be made at points where part of the interleaved sequence has already been checked for validity, but still needs to be incorporated into the containment check.

The way in which this is resolved is by extending the notation of a constraint, so that what is propagated down through the various layers of containment checking is not just a set of constraints, but a set of constraints and the "prefix" sequence which has already been established to be valid. For the purposes of this definition, this notion of "constraint plus prefix" will be denoted cons, and will be written in the form cs after sp, where cs denotes a set of constraints (each of which is assumed to be a buffering constraint, that therefore has the syntactic form <sequence>), and sp denotes the prefix sequence. An ordinary invocation of the operation s1 || c s2 / cs can therefore be treated as an invocation of the form s1 ||c s2 / (cs after ε), and it is this extended form which is defined by the following axioms.

Since all the issues to do with the structure of constraints will be dealt with in the definition of SubjectTo, for the constrained parallel composition operator itself the only levels of syntactic structure that need to be considered are those for s1 and s2. At the top level, therefore, there are in principle four cases, since each of them can either have the form of an <alt-elem> or a <proper-alt>, but in practice the <proper-alt> cases are symmetrical, and can be defined as follows.

Where s1 is a <proper-alt>, (s1a | s1b) ||c s2 / cons = (s1a ||c s2 / cons) | (s1b ||c s2 / cons)
Where s2 is a <proper-alt>, s1 ||c (s2a | s2b) / cons = (s1 ||c s2a / cons) | (s1 ||c s2b / cons)

This leaves the case where s1 and s2 are both <alt-elem>, and for this the second level of structuring gives rise to four further cases, since both s1 and s2 can have either of the forms <action> or <proper-seq>. The case where both have the form <action> gives rise to one of the base cases, which therefore requires an invocation of SubjectTo, and the axiom for this is

\[ a1 ||c a2 / cons = ((a1 ; a2) SubjectTo cons) | ((a2 ; a1) SubjectTo cons) \]

For the cases where either s1 or s2 have the form <proper-seq> it is necessary to go down to the third level of structuring, where on each side there are two sub-cases, depending on whether the head element of the sequence has the form <action> or <proper-alt>. As with ordinary parallel composition, the case where both head elements are actions gives rise to what is actually a base case for the structural induction, despite being defined recursively, where the recursion involves both the sequences being interleaved and also the prefix sequence for the constraint. The axiom for it is therefore as follows, where cons = (cs after sp).

\[ (a1 ; s1) ||c (a2 ; s2) / cons = ((a1 ; (s1 ||c (a2 ; s2) / (cs after (sp ; a1))) SubjectTo cons) | ((a2 ; ((a1 ; s1) ||c s2 / (cs after (sp ; a2)))) SubjectTo cons) \]

For the other cases, which represent the recursions in the structural inductions, the axioms are again symmetric, and are defined by two forms, one for the construction of each side. These axioms are as follows.

Where s1 is a <proper-seq> and its head is a <proper-alt> ((s1a | s1b) ; s1t) ||c s2 / cons = ((s1a ; s1t) ||c s2 / cons) | ((s1b ; s1t) ||c s2 / cons)
Where s2 is a <proper-seq> and its head is a <proper-alt> s1 ||c ((s2a | s2b) ; s2t) / cons = (s1 ||c (s2a ; s2t) / cons) | (s1 ||c (s2b ; s2t) / cons)

Finally, the operator SubjectTo needs to be defined, and this requires a single axiom, as follows

\[ s SubjectTo (cs after sp) = if ( \forall c \in cs, ( (sp ; s) \setminus Alpha (c) ) \subseteq c ) then s else \phi \]

This then completes the definition of constrained parallel composition.
Semantics

Now that the basic abstract syntax has been used to establish rigorous definitions for the more elaborate operations of parallel composition and constrained parallel composition, the only aspect of the original report that has not been revised to match this abstract syntax is the notion that the dataflow algebra itself has a semantics. In that report this semantics was defined rather informally, in terms of associating with each expression in the algebra a set of strings over the alphabet of actions.

Unfortunately, this formulation does not adequately express the properties of the forbidden action, and in particular the discussion of the forbidden action that led up to it was in error in suggesting that "any sequence containing \( \phi \) is equivalent just to \( \phi \) itself", since the later work in that report made it clear that any sequence of the form \( \alpha ; \phi \) actually represents a situation where the system deadlocks after the occurrence of the action \( \alpha \). Consequently, a sequence of the form \( \alpha ; \phi \) does not reduce to \( \phi \), although a sequence of the form \( \phi ; \alpha \) should reduce to \( \phi \), as no further actions can occur after the \( \phi \).

To correct this error requires a more elaborate formulation of the semantics, which recognises that such deadlock situations can legitimately occur within dataflow algebra expressions, and that the meaning of them is that any such sequence of actions that includes \( \phi \) is a sequence that does not terminate normally. This notion of abnormal termination is clearly related to the notion of failures in CSP, and since these are an important element in the specification of a process in CSP it is perhaps not surprising that a proper semantics for the dataflow algebra needs to include the distinction between normal and abnormal termination. Indeed, CSP does include a symbol \( \triangledown \), which represents a form of action that corresponds to normal termination, and initial investigation of how the algebra could be extended to allow for the dynamic creation and deletion of processes has suggested that this might require the incorporation of this concept into the algebra. This, however, is an issue that still requires further investigation, as does the precise relationship between the semantics proposed here and concepts of CSP such as failures and refusals.

Thus, the revised version of the semantics for a dataflow algebra expression is that it consists of an ordered pair of sets, known as the valid set and the invalid set. Each set is a set of strings of actions, where the actions are drawn solely from the alphabet of the system, and so do not include either \( \varepsilon \) or \( \phi \). These ordered pairs of sets can be treated as though they form an abstract type, where \( \text{valid}(\text{sem}) \) and \( \text{invalid}(\text{sem}) \) are the two observer operations for the type, and the constructor operation for it is denoted \( \langle \text{v}, \iota \rangle \), and is defined by the basic axiom

\[ \forall \text{sem}, \langle \text{valid}(\text{sem}), \text{invalid}(\text{sem}) \rangle = \text{sem} \]

The significance of these two sets is that each element of \( \text{valid}(\text{sem}) \) denotes a trace of the system behaviour which terminates normally, while each element of \( \text{invalid}(\text{sem}) \) denotes a trace which terminates abnormally, in the sense that after the specified sequence of actions has occurred the system is incapable of further action (probably, but not necessarily, because it has deadlocked). The assumption which is made is that any given sequence can not be in both sets, since this would correspond to the system having reached a point where it has a choice between either performing the forbidden action \( \phi \), or continuing with some other actions. Given such a choice, the property that \( \phi \) is the identity for the alternation operator would imply that always the option of continuing with other actions would be taken, although this does raise issues of causality within the algebra which require further investigation. For the time being, however, this assumption will be taken as reasonable, and it leads to an invariant for the semantics that the two sets must be disjoint, which can be expressed by the axiom

\[ \forall \text{sem}, \text{valid}(\text{sem}) \cap \text{invalid}(\text{sem}) = \emptyset \]

In principle the semantics of an expression will then be given by a semantic function, which will be denoted \( \text{Sem} \), but in practice the treatment of the forbidden action complicates the situation slightly, because in some cases its semantic interpretation will be as the identity for the alternation operator, and in others it will be as the forbidden action, and these two turn out to require different formulations. While this could be represented by incorporating the relevant cases into the definition of the one semantic function \( \text{Sem} \), it turns out to be simpler to formulate it in terms of two separate semantic functions, which will be denoted \( \text{SemS} \) (for sequencing, which is the one that requires the deadlock properties for \( \phi \)) and \( \text{SemA} \) (for alternation, which is the one that requires the identity properties for it).

The definitions of the semantic functions then follow the structure of the abstract syntax, in that the first level requires the semantics to be defined for a \( \langle \text{proper}-\text{alt} \rangle \), and this requires the introduction of an auxiliary function to combine the semantics of the components of the alternation. This function is called \( \text{SemAlt} \), and it is used in the definition as follows.

\[
\begin{align*}
\text{SemS}(s1 \mid s2) &= \text{SemA}(s1 \mid s2) = \text{Sem}(s1 \mid s2) \\
\text{Sem}(s1 \mid s2) &= \text{SemAlt}(\text{SemA}(s1), \text{SemA}(s2))
\end{align*}
\]
The definition of $\text{SemAlt}$ is then essentially that the sets of valid and invalid strings for the components are united, and any duplicates removed from the resultant set of invalid strings. This is expressed as

$$\text{SemAlt} (x, y) = <v, \ i - (v \cap i)>$$

where

$$v = \text{valid} (x) \cup \text{valid} (y)$$

$$i = \text{invalid} (x) \cup \text{invalid} (y)$$

The next level of the abstract syntax then requires the definition of the semantics for the case of an $<$alt-elem$>$ that is structured as an $<$action$>$, and this is straightforward, except for the fact that $\text{SemS}$ and $\text{SemA}$ will be defined differently for $\phi$. The required definitions are therefore

$$\text{SemS} (\phi) = \text{Sem} (\phi) = <\emptyset, \ {\lambda}>$$

$$\text{SemA} (\phi) = <\emptyset, \emptyset>$$

$$\text{SemS} (c) = \text{SemA} (c) = \text{Sem} (c) = <{\lambda}, \emptyset>$$

$$\text{SemA} (a) = \text{SemA} (a) = \text{Sem} (a) = <a, \emptyset>$$

where $\lambda$ is used to denote the empty string, so that a set containing the single element $\lambda$ is not the same as the empty set (ie it has cardinality 1 rather than 0). The last level of the abstract syntax for which the semantics need to be defined is the level at which an $<$alt-elem$>$ is structured as a $<$proper-seq$>$, since the definitions given above cover all the cases of the construction of a $<$seq-elem$>$. As with the $<$proper-alt$>$, this definition requires the introduction of another auxiliary function, which is called $\text{SemSeq}$, and the required definitions are as follows.

$$\text{SemS} (s1 ; s2) = \text{SemA} (s1 ; s2) = \text{Sem} (s1 ; s2)$$

$$\text{Sem} (s1 ; s2) = \text{SemSeq} (\text{SemS} (s1), \text{SemS} (s2))$$

where

$$v = \text{valid} (x) \times \text{valid} (y)$$

$$i = \text{invalid} (x) \cup (\text{valid} (x) \times \text{invalid} (y))$$

Here, the operator $\times$ between any two sets of strings $A$ and $B$ is defined by the axiom

$$A \times B = \{a.b \text{ where } a \in A, b \in B\}$$

where $a.b$ denotes the string formed by concatenating the two strings $a$ and $b$. It should therefore be apparent that the empty set of strings is the zero element for this operation, and the set $\{\lambda\}$ is the identity element for it.

The operation of these definitions may not be completely obvious, and so it is worth giving some simple examples. For some action $a$ the semantics of $a ; \phi$ should give an empty valid set and an invalid set that just contains the single action $a$. Formally, this would be computed as

$$\text{Sem} (a ; \phi) = \text{SemSeq} (<a, \emptyset>, <\emptyset, \{\lambda\}>)$$

giving

$$v = \{a\} \times \emptyset = \emptyset$$

$$i = \emptyset \cup \{a\} = \{a\}$$

and so

$$\text{Sem} (a ; \phi) = <\emptyset, \{a\}>$$

Similarly, for some other action $b$ the semantics of $b | (a ; \phi)$ should then give a result that has the same invalid set of actions, but a valid set containing the single action $b$. The formal computation in this case is

$$\text{Sem} (b | (a ; \phi)) = \text{SemAlt} (\{b, \ emptyset\}, <\emptyset, \{a\}>) = <\{b\} \cup \emptyset, (\emptyset \cup \{a\})> = <\{b\}, \{a\}>$$

Note, however, that for $\text{Sem} (a | (a ; \phi))$ the result would not be $<\{a\}, \{a\}>$ but $<\{a\}, \emptyset>$, because the maintenance of the invariant would result in the element $a$ being deleted from the invalid set. This is consistent with the algebraic identities given in the original report, which mean that $a | (a ; \phi)$ can be rewritten as $\phi | (a ; \epsilon)$ (since $\epsilon$ is the identity for sequencing), and then as $\phi ; (\epsilon | \phi)$ (using the distributive property), which then reduces to $\phi ; \epsilon$ and hence to $a$.

As a final example, although a sequence of the form $a ; \phi$ does not reduce to $\phi$, a sequence of the form $\phi ; a$ should, and this can be established by computing its semantics. The formal calculation is
Sem (φ ; a) = SemSeq ( ⟨∅, {λ}⟩, ⟨{a}, ∅⟩ )
giving
\[ v = ∅ \times \{a\} = ∅ \]
\[ i = \{λ\} \cup ( ∅ \times ∅ ) = \{λ\} \cup ∅ = \{λ\} \]
and so
Sem (φ ; a) = ⟨∅, {λ}⟩ = Sem (φ)
which is the expected result.

An Extended Abstract Syntax

This definition of the semantics of the expressions completes the revisions to the material of the original report that have been necessitated by the introduction of the basic abstract syntax. From the point of view of the practical modelling language, perhaps the most important of these revisions has been to give rigorous definitions of the various forms of parallel composition (ie ordinary and constrained), and in particular these make it possible to go on to define an extended abstract syntax, in which both forms of parallel composition are included as constructions in their own right.

The syntax for these constructions follows the general form of the basic abstract syntax, in that again it is desirable to only have <par> elements appearing where they are genuinely required, meaning where there is an occurrence of a parallel composition operator. Consequently, it is necessary to introduce constructions such as <proper-par> for ordinary parallel compositions and <proper-c-par> for constrained parallel compositions, in much the same way as the basic syntax used <proper-alt> and <proper-seq>. Moreover, these have to be defined in such a way that, where there are several parallel components, the two types of parallel composition can not be mixed up. Consequently, while the syntax should allow constructions of the form s1 || s2 || s3 (as a <proper-par>) and s1 ||c s2 ||c s3 / {cs} (as a <proper-c-par>), it should not allow something like s1 || s2 ||c s3 / {cs}.

At the same time, though, the fact that there are now three main levels of structuring has its own effects, in that it can not necessarily be assumed that a <seq-elem> which is not an <action> is going to be either a <proper-par> or a <proper-c-par>, since it might not be a <par> at all, but an <alt> instead. On the other hand, such a <seq-elem> does have to be one of these forms, and whichever it is it does have to be proper. Thus, the production in the new syntax for <seq-elem> has to include both <proper-par>, <proper-c-par> and <proper-alt> as possible constructions, and the requirement for these to be proper then ensures that a circularity could not arise through a <seq-elem> being constructed as an improper <par>.

One consequence of this more elaborate structure for <seq-elem> is that <action> should not be introduced as an alternative in the production for <par-elem>, in the way that it is in the production for <alt-elem> in the basic syntax. The reason for this is that it would cause an ambiguity if an <action> could be parsed directly as either a <par> or an <alt>, whereas it will not generate any ambiguities if a single action has to be parsed as an (improper) <par> which is actually an (also improper) <alt>. This ambiguity, however, could not arise in the basic syntax, since there it will always be clear which of the three possible cases can apply for the parsing of a single <action>: that is, whether it is part of a <proper-seq> (ie it is a <seq-elem>); or is not part of a <proper-seq> but part of a <proper-alt> (ie it is an <alt-elem>); or it is not part of either a <proper-seq> or a <proper-alt> (in which case it is treated as an <alt-elem>). Thus, the required syntax is as follows.

\[
\begin{align*}
<c-par-sym> & ::= "|| c" \\
<const-sym> & ::= "/c" \\
<s-open-sym> & ::= "{" \\
<s-close-sym> & ::= "}" \\
<s-in-sym> & ::= "," \\
<sequence> & ::= <par> \\
<par> & ::= <par-elem> | <proper-par> | <proper-c-par> \\
<par-elem> & ::= <alt> \\
<proper-par> & ::= <par-elem><par-tail> \\
<par-tail> & ::= <par-sym><par-rest> \\
<par-rest> & ::= <par-elem> | <proper-par> \\
<proper-c-par> & ::= <par-elem><c-par-tail> \\
<c-par-tail> & ::= <par-sym><c-par-rest> \\
<c-par-rest> & ::= <par-elem><const> | <proper-c-par> \\
<const> & ::= <const-sym><s-open-sym><par-set><s-close-sym> \\
<par-set> & ::= <par> | <par><s-in-sym><par-set> 
\end{align*}
\]
Given this extended syntax, it is then possible to work in terms of structural definitions in which parallel compositions are treated as though they were primitive, and hence defined in their own right, rather than always having to be defined in terms of the axioms that construct them using sequencing and alternations. This may be particularly significant where properties are being proved for more complex constructions, since the proofs for parallel compositions may need to be done separately from those for sequencing or alternation, as the axiomatic definition of parallel composition does not lend itself readily to producing proofs by induction.

Concrete Syntax

As well as an abstract syntax for the structure of sequences, and the (informal) syntax for what might be described as the "publication language" for specifications of systems, there is also a need for a more formal concrete syntax that is suitable for machine processing. At this stage this is still very much under development, and so the concrete syntax that is described here is only an initial version, in which the key decisions that have been made are documented, so as to provide a basic concrete syntax that can be used as a starting point for experimentation. Because of this the syntax is defined more by giving examples of individual productions, rather than as a complete BNF grammar.

Other issues in the design of this concrete syntax will still require further investigation, and these are flagged here so as to indicate areas where these investigations need to be carried out. To simplify cross-reference to aspects of this syntax, it is divided into numbered sections and subsections.

1. General Principles.

(i) The basis of this concrete syntax is the notion that a machine processable version of a specification will effectively constitute a program in a form of functional programming language. As such, it will consist of two parts:

(a) a set of declarations, that constitute the specification itself, and

(b) a set of expressions, that are to be evaluated in the context provided by this set of declarations.

(ii) The syntax is defined here on the basis that the whole of such a functional program would be organised as a file, and input to a suitable language processor. It is recognised that, in practice, many such processors expect to accept input interactively, at least for the expressions that are to be evaluated, if not for the declarations as well. The modifications to the syntax that would be needed to facilitate such interactive operation have not been investigated yet.

(iii) Equally, in practice it would also be convenient to allow the text of such a program to be spread across a number of files, and any such facilities would also affect whatever provision was to be made for interactive input. In particular, given the kind of structures that are to be represented by the declarations, it seems appropriate that any such interactive facilities should be structured so that complete files of declarations have to be processed, rather than individual declarations. Some aspects of this are defined below, but these issues too need to be investigated further.

(iv) The set of declarations will need to be split into three groups, corresponding to the three levels at which a system can be specified. These three groups will be bracketed by keywords, which will therefore be reserved words. Similarly, the expressions to be evaluated will form a fourth group, and will also be bracketed by keywords.

(v) Ordinary identifiers will follow the usual convention of being composed of letters or digits, but starting with a letter: they will be case sensitive, and all the reserved words will be written in capitals. White space (ie spaces, tabs or newline symbols) may not occur within identifiers, but must occur to separate adjacent identifiers, reserved words, etc.

(vi) The keywords bracketing the sections will be as follows: for the declarations forming the topological level, TOPOLOGY and END; for the declarations forming the syntactic level, SYNTAX and END; for the declarations forming the semantic level, SEMANTICS and END; and for the expressions to be evaluated, PROGRAM and END.

(vii) The ordering of the sections will be determined solely by the requirement that names be declared before they are used. It should be possible for a particular level to be spread across a number of sections, for instance to reflect the modular structure of a system being specified. The issue of whether to allow sections to be given a name (which would
follow the opening keyword, and then be repeated after the closing END, as with the Modula-2 conventions for modules) is one that still needs to be investigated, since it would obviously need to interact with the way in which the hierarchical structure of a large system was defined at the topological level.

(viii) The primitive data types that need to be provided within such a program will need to include as a minimum the following: processes (denoted PROC), channels (denoted CHAN), actions (denoted ACT), sequences (denoted SEQ), attributes for sequences (denoted ATT), attribute names (denoted ANAME), semantic specifications (denoted SPEC), integer numbers (denoted INT) and booleans (denoted BOOL). There will also be a need to represent file names, although it is not clear whether there would ever be a situation in which these would need to be declared, and so no type name is defined for them here. The same might also be true of characters and character strings, if only to allow the equivalent of comments to be output, but this possibility is ignored here, and requires further investigation.

(ix) A file name is written in the form

FILE <file name in conventional operating system format> END

and the operating system filenames must include any extensions: no assumptions can be made about defaults for them. It is suggested that either .DFA or .dfa (depending on what assumptions about cases are made by the underlying operating system) should be the standard file name extension for any file containing part or all of a specification.

(x) A file must contain complete sections of a specification, and the last END in any file must be followed by a full stop, as in Modula-2. The construction

INCLUDE <filename>

can occur anywhere where a section would be legal, and indicates that the sections contained in the specified file are to be included in the specification at this point.

(xi) Most declarations will be preceded by the appropriate type names (as defined in (viii) above), and so they will follow the Algol pattern of

<type> <name> = <value>

rather than the Pascal patterns (which would be <CONST> <name> = <value> or <name> : <type>).

(xii) The main compound data types that will be required will be arrays and sets. Arrays will be used for processes, channels, actions and sequences, and where the type needs to be written it will be denoted as ARRAY OF <type>. Subscripts will be written enclosed in square brackets, and in this version of the syntax all array subscripts will be integers: the issue of whether facilities should be provided for defining enumerated types and using them as subscripts still requires investigation. The Algol pattern will also be used for defining array subscripts, so that subscript ranges will be denoted lo : high rather than lo .. high.

(xiii) The Algol pattern will also be used in writing conditional expressions, which will have the general form:

IF <condition> THEN <value> ELSE <value> FI

The THEN <value> part may be followed by as many occurrences as required of:

ELSF <condition> THEN <value>

It is not clear whether there would be a requirement for a CASE clause: if so then an Algol-like pattern should be followed for that too, expect that END should be used rather than ESAC as the closing keyword.

(xiv) Comments may be included in a specification at any point where white space would be legal, and may have either of two forms. A short comment may appear at the end of a line, preceded by white space and two hyphens (as in occam): such a comment is terminated by the end of the line. A long comment follows the Modula-2 convention of being bracketed by (* and *), and such comments may spread over several lines. Long comments may be nested, so as to allow the commenting out of sections of code (which may themselves include long comments), and so to ensure that the brackets (* and *) balance, they must be treated as significant when parsing the text contained in long comments, and it is probably desirable that they are not allowed to occur in short comments.
2. The Topology Section.

(i) Many of the declarations in this section simply enumerate the objects of the various types required (viz processes and channels), and so will have the form

\[ \text{<type> <list of items> <full-stop-sym>} \]

where the individual items in the list will be separated by commas, and will have either of the forms

\[ \text{<name>} \quad \text{or} \quad \text{<name> [ <subscript range(s)> ] } \]

There are arguments to be made both for and against requiring the declaration of a channel to specify its source and destination processes, and the issue requires further investigation. At this stage the syntax does not require such specifications, as this information can be inferred from the structure of the actions that use a channel.

(ii) Action declarations have a similar form, but need to define the values of the actions concerned: action values follow the syntax of the publication language (ie \( \text{<process> ? <channel> ! <process>} \)). Thus, these declarations are written in the form

\[ \text{ACT <list of action declarations> <full-stop-sym>} \]

where, as before, the individual declarations in the list are separated by commas, and have the form

\[ \text{<action-name> = <action-value>} \]

or, for an array of actions

\[ \text{<action-array-name> [ <subscript-declarations> ] = <action-values>} \]

where a \(<subscript-declaration>\) has the form

\[ \text{<integer-variable-name> = <subscript-range>} \]

and if the array has more than one subscript then the declarations of them are separated by commas.

3. The Syntax Section.

(i) This section consists entirely of declarations of sequences, and so in principle there is no need for individual declarations to be preceded by the type \( \text{SEQ} \). In practice, though, to avoid confusion it is probably better to require it to appear, so as to be uniform with the other sections. Thus, this section will consist of a set of declarations that have similar forms to those given in 2(ii), viz:

\[ \text{SEQ <list of sequence declarations> <full-stop-sym>} \]
\[ \text{<sequence-name> = <sequence-value>} \]
\[ \text{<sequence-array-name> [ <subscript-declarations> ] = <sequence-values>} \]

(ii) The basic forms of sequence value (ie constructed from sequencing, alternation and ordinary parallel composition) will be as in the publication language, with suitable definition of operator priorities and bracketing rules to match the abstract syntax given earlier.

(iii) Representations will be needed for the empty action \( \varepsilon \) and the forbidden action \( \phi \), and it is proposed that \( \text{EPS} \) and \( \text{PHI} \) respectively be used for these.

(iv) The symbols \( \Sigma \), \( U \) and \( \Pi \) in the publication language will need to be represented by suitable expressions, that allow specification of the variable, the range of values that it is to take and the expression over which the sequence or choice is constructed. The proposed forms for this are as follows:

\[ \text{SIGMA <subscript-declaration> OVER <sequence-values> END} \]
\[ \text{UNION <subscript-declaration> OVER <sequence-values> END} \]
\[ \text{PI <subscript-declaration> OVER <sequence-values> END} \]
Representations will be needed for repetitions, and the proposed forms for these are as follows.

- `DO <sequence-values> FOR <integer-value> OD`
- `DO <sequence-values> BY + OD`
- `DO <sequence-values> BY * OD`

Forms are also needed for constrained parallel composition. For a simple constrained parallel composition the publication language symbol $||_c$ can be written as $||$, although in some respects it would be desirable if the symbol $||_c$ could be used either as well as $||$ or instead of it. Such a symbol could, however, give rise to ambiguities, although these could possibly be eliminated by requiring the symbol to be followed by white space. This is an issue that requires further investigation. By contrast, defining an equivalent for the $\Pi_c$ form is straightforward, since this can be written in a similar fashion to the $\Pi$ form, as follows:

- `PIC <subscript-declaration> OVER <sequence-values> / <constraint-sequence-set> END`

Set values are always written in the same form, irrespective of the kind of values: this form is:

- `{<list-of-values>}`

where the individual values are, of course, separated by commas.

4. The Semantics Section

(i) This syntax assumes that semantics will be defined in terms of OBJ-3, although in future it may be necessary to extend it to allow other semantic specification methodologies to be employed as well, of which Z is perhaps the most obvious. This could be done by having each of the various keywords needed in this section include the name of the methodology in use (e.g., `OBJ3SPEC`, `ZSPEC`, and so on), but it seems simpler to insist that all the specifications within one semantics section will be in the same methodology, and that the initial keyword `SEMANTICS` be followed by one keyword giving the name of that methodology, which in this case will be `OBJ3`.

(ii) Essentially this section of a specification will need to define two things. One will be the underlying process specifications, which will be referred to as whole specifications, and these will be incorporated as they stand. The other will be the attributes that are to be attached to the sequences, and these will be referred to as partial specifications. To avoid any parser for a dataflow algebra specification having to be able to parse as well the complete underlying specification language, it is proposed that whole specifications will be required to exist in files, and the declarations will simply refer to these. The partial specifications that form the attribute values will, however, have to form part of the dataflow algebra language, but since they will essentially just consist of fairly simple expressions, that will refer to more elaborate constructions occurring in the whole specifications, they can be assumed to be expressed in a very restricted subset of the underlying specification language. Defining that subset for each specification language will require further investigation.

(iii) Since partial specifications need to be associated with actions as attributes, or with processes as specifications of them, a notation is required for representing this association. In the case study of the alternating bit protocol references to these associations as attributes are written in the form:

- `<action-name><full-stop-sym><attribute-name>`

but there is a danger that this could be confused with the use of full stops to terminate declarations, and so it is proposed that instead they be written in the form:

- `<action-or-process-name> $ <attribute-name>`

(iv) The example given in that case study suggests that the following attribute names should be regarded as predeclared (which can be interpreted as reserved for this purpose, except that they will not follow the usual convention for reserved words of being all in capital letters): `Src` (for the expression representing the operation in the source process of an action that produces a data item to flow), `Dest` (for the expression representing the operation in the destination process of an action that handles the data item that flows), and `States` (for the expression representing the state of a system affected by an action). Although it was not used explicitly in that example, there also appears to be a need for an attribute name `DataType` (for the expression representing the kind of data that flows during an action), and in order to express the fact that particular whole specifications are to be associated with particular processes, there seems to be a need for one called `ProcSpec`.
(v) In principle it may be necessary to allow other attribute names to be declared as well, in which case their declarations would have the form of process or channel declarations, but with the type ANAME. The issue of how to define the significance of such attributes is one that still needs to be investigated.

(vi) The declarations of whole process specifications will have the form of action or sequence declarations, with the type SPEC (at present there does not seem to be any need to require this to be qualified by a keyword like WHOLE, so as to distinguish them from partial specifications, but further investigation might reveal a need for such qualification). The values will be written in the form:

\[
\text{FROM } \text{<filename>}
\]

where <filename> has the form defined in 1(ix) above.

(vii) The association of a named specification with a process will be written as an attribute declaration, in the usual form, with the type ATT. As an example, to associate a process p with a (partial) specification sp the attribute declaration would be written as:

\[
\text{ATT } p \, $ \, \text{ProcSpec} = \text{sp}
\]

(viii) Partial specifications to be used in the declarations of attribute values will need to be written in a subset of the specification language (which, as noted in (i) above, is assumed here to be OBJ-3), but including also references to other attributes, which will be written in the form defined in (iii) above. These values will also have the type SPEC (where again there does not currently seem to be need for this to be qualified by a keyword like PARTIAL, but further investigation might reveal such a need). They will be written in the form:

\[
\text{BEGIN } \text{<fragment of modified OBJ-3 subset> END}
\]

As noted in (ii) above, the precise definition of the underlying subset of OBJ-3 (or any other specification language to be used instead) needs further investigation.

(ix) The association of an attribute value with an action will be written as an attribute declaration. Thus, the equivalent of the set of attribute declarations given in the alternating-bit protocol case study for the process R2 (which will be called R2 in what follows) could be represented by declarations along the lines of the following example.

\[
\text{SPEC } \text{R2Source} = \text{BEGIN } M2.out(M2 $ States) \text{ END},
\]
\[
\text{R2Dest} = \text{BEGIN } \text{Sender.in2(R2 $ Src, Sender $ States END},
\]
\[
\text{R2NewState} = \text{BEGIN } \text{Sender $ States} = \text{R2 $ Dest END}
\]
\[
\text{ATT } R2 $ Src = \text{R2Source}, R2 $ Dest = \text{R2Dest}, R2 $ States = \text{R2NewState}.
\]

5. The Program Section.

(i) This consists of one or more expressions, separated by commas. In executing this program, each of the expressions will be evaluated, and the results output (ie to the screen). In practice it would also be useful to be able to direct the output to a file as well, and it is probably desirable that this be done from within the program itself, rather than relying on operating system facilities for redirection of screen output. To allow this, it is therefore proposed that an expression (or, more precisely, an \texttt{<expression-list>}, but in practice such an expression list can be regarded as a single expression) can also have the form

\[
\text{TO } \text{<filename> OUT <expression-list> END}
\]

which indicates that the values of each of the expressions are to be output to the named file as well as to the screen. Also, to enable output to the screen to be suppressed, the form

\[
\text{SILENT <expression>}
\]

is proposed.

(ii) Many expressions will involve the boolean operators that are written as \texttt{< , > , \leq or \geq} in the publication language: they are represented in this syntax as \texttt{< , > , \leq or \geq} respectively.
(iii) Where expressions involve semantic specifications in some underlying specification language, then the keyword
PROGRAM must be followed by the name of the specification methodology, as defined in 4(i) above. This would also
apply where expressions are being translated into some alternative notation, such as CCS (or, more precisely, the machine-
processable dialect of it used by the Concurrency Workbench [15]), which for this purpose can be regarded as a
specification methodology.

(iv) It is clear that there will be a need for various translation functions to be provided, that will generate code in
formats required by other specification methodologies. These would therefore have to be regarded as predeclared, in
the same way as the attribute names referred to in 4(iv) above, but as with those attribute names, the issue of how to define the
significance of particular translations requires further investigation. It is also likely that there will be a need to allow the
same names to be used for functions that relate to translations into different specification methodologies, but this effective
overloading of the names will not be ambiguous, because of the requirement in (iii) to name the specification methodology
being referred to in any expression.

Summary and Conclusions

This report has put the method of describing systems in terms of dataflow algebra specifications onto a much more solid
footing than the original report was able to do. There, many of the ideas were still being formulated, and a lot of ends had
been left very loose. Most of these have now been tied up, and in particular the introduction of constrained parallel
composition has enabled most of the problems raised in attempting to specify pipeline structures to be solved in a fashion
which has been rigorously defined, and which will facilitate the construction of rigorous models. Furthermore, the
definition of the abstract syntax has made it possible to create definitions in terms of structural inductions in such a way
that those definitions can be guaranteed to be complete and rigorous, as has been illustrated by the definitions of the
various forms of parallel composition. This report has therefore provided a much more solid foundation for the future
theoretical work, such as the various topics which have been noted as requiring further investigation.

Finally, the definition of at least the basic requirements for a concrete syntax has provided a foundation on which work
may be carried out to investigate the various issues that arise in trying to automate the processing of models of systems,
and the construction of translations into other specification methodologies. Such automated processing should enable the
methods for analysing the complete behaviour of systems to be applied to larger examples than the ones which have been
considered here. These methods were worked by hand in the case study of the alternating bit protocol, but for examples
with more than just three or four processes the complexity of the algebra would mean that hand analysis would at least be
unwieldy, if not actually completely impracticable.

References

1 A. J. Cowling, Dataflow Algebras as Formal Specifications of Data Flows, Department of Computer Science
5 I. Jacobson, M. Christerston, P. Jonsson & G. Övergaard, Object-Oriented Software Engineering: A Use Case
8 J. A. Goguen & T. Winkler, Introducing OBJ3, Computer Science Laboratory Technical Report SRI-CSL-88-9,
9 A. J. Cowling & M. C. Nike, Using Dataflow Algebra to Analyse the Alternating Bit Protocol, in Software
15 R. Cleaveland, J. Parrow & B. Steffen, The Concurrency Workbench: A Semantics-Based Verification Tool for