LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

Denotational and Axiomatic Definitiuons for Path Expressions

Computation Structures Group Memo 153-1
November 1977

Valdis Berzins
Deepak Kapur

This research was supported by the National Science Foundation under grant MCS74-21892 A01 and by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under contract N00014-75-C-0661.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Denotational and Axiomatic Definitions for Path Expressions

Abstract

Path expressions are a well known means for specifying synchronization constraints among concurrent programs. A denotational semantics for path expressions is presented, based on a model of parallel processing where computations are described in terms of indivisible events. A method for deriving a formula in the first order predicate calculus that captures the synchronization constraints imposed by a path expression is given for a restricted class of path expressions. An example of a proof based on this formula is presented.

1. Introduction

We present two formal methods for defining the meaning of path expressions. The methods for defining synchronization primitives for interacting concurrent processes that have appeared in the literature vary greatly in their degree of formality. Semaphores were introduced informally in [4], and an attempt was made to formalize them using invariant relations in [8]. Path expressions were introduced informally in [9] and an implementation in terms of semaphores was sketched. Path expressions have since evolved [2, 10, 5, 17], and the semantics of a version of the formalism has been defined in terms of the Petri net formalism [21] by Lauer and Campbell [16].[1] Monitors were introduced informally in [12, 14]. Serializers were introduced in [13], with an axiomatic definition, using a model of computation based on message passing [6].

An informal definition of the semantics of a formalism is not sufficient, because it admits several conflicting interpretations. A definition in terms of a specific implementation introduces irrelevant details and accidental properties, which complicate the definition, and which may make it more restrictive than intended. A formal specification of a system is precise, and can serve as a basis for proofs of properties of the system. A formal specification can also be used to establish whether a given implementation satisfies the desired behavior. Some progress has been made in developing formalisms for defining synchronization primitives, but the area is still not well understood.

Our approach is based on a model of parallel processing where computations are described in terms of indivisible events. We identify the meaning of a synchronization primitive with the set of all legal computation histories. In the denotational version of our method, we construct this set explicitly. A related approach can be found in [15]. Our definition method is to be contrasted with the Oxford definition method [23], where a program denotes a particular function on a suitably chosen domain. In the axiomatic version of our method, we characterize the set of legal histories by a first order theory [18]. Properties of the synchronization scheme can be proven within the theory. The set of all finite models satisfying the theory is precisely the set of all legal histories.

Section 2 contains a description of the underlying model of computation. In Section 3 we illustrate the denotational method by presenting a definition of path expressions, while in Section 4 we give an axiomatic definition of path expressions. In Section 5 we show how to construct proofs

---

1. The class of path expressions we define is strictly larger than that considered in [16].

based on our axiomatic definition method and present an example of a proof that a path expression maintains an invariant assertion. Section 6 contains our conclusions.

2. Events and Computation Histories

We view the computation performed by a parallel program as a finite partially ordered set of indivisible events, denoted $\langle E, \leq \rangle$, where E is the set of events that make up the computation, and where $\leq \subseteq E^2$ is a partial ordering relation on the set E. $\leq$ represents the history of the computation: the relation $x \leq y$ holds precisely when either the event x has occurred strictly before the event y, or when x and y denote the same event. Since $\leq$ is a partial order, it is reflexive, transitive, and antisymmetric.[2]

The restriction that the set of events E must be finite is justified since any real machine performs computations at a finite rate, and operates only for a finite length of time (eg. since it was built). Consequently, at any given time, the execution of any given program must have a finite computation history. We say nothing about whether the program in question has terminated, and it may well be the case that at some later time the program will have run some more, and have a larger but still finite computation history.

The events in E are defined to be indivisible, so that they must correspond to the most primitive operations performed by the hardware. Since we are interested in discussing synchronization at a level of abstraction appropriate to a high level programming language, where the *actions* we wish to synchronize will usually consist of many machine operations, we define an action to be a set of events, and we require distinct actions to be disjoint. For example, in the path expression formalism, actions are identified with particular invocations of procedures, and it is assumed that the set of events associated with a particular procedure call is unambiguously defined, at least for those procedures subject to synchronization constraints.

We want to talk about the order in which actions occur in the computation history. We will say that the action a_1 precedes the action a_2 (written $a_1 \leq a_2$) precisely when either every event of a_1 precedes every event of a_2, or a_1 and a_2 denote the same action.[3] It is not hard to show that

---

2. $(\forall x \in E)[\, x \leq x \,]$,
   $(\forall x, y, z \in E)[\, x \leq y \wedge y \leq z \Longrightarrow x \leq z \,]$, and
   $(\forall x, y \in E)[\, x \leq y \wedge y \leq x \Longrightarrow x = y \,]$.
3. $(\forall a_1, a_2 \in A)[\, a_1 \leq a_2 \iff (a_1 = a_2 \vee (\forall x \in a_1)(\forall y \in a_2)[\, x \leq y \,])\,]$

this defines a partial ordering on actions. <A, ≤> is a computation history which describes the computation in terms of the component actions. The ordering on actions has been defined so that it is consistent with the underlying ordering on the component events, and so that it captures all of the information about the underlying ordering that can be observed without examining the substructure of the actions.

If neither $x \leq y$ nor $y \leq x$ then the events x and y are unordered, and we say that they are *concurrent*. If two events are concurrent in a computation history, then the order in which they occurred is not determined, and need not be observable - the events may have been executed in some unpredictable order by a single processing unit, or they may have been executed at the same time by some parallel hardware. If two actions are concurrent in a computation history, then either the relative order of the component events of the two actions was not observed, or the component events of the two actions were interleaved.

3. The Semantics of Path Expressions

In this Section, we define the semantics of two classes of path expressions. A path expression of Type 1 allows at most a fixed number of processes (equal to the number of parallel paths) to be actively executing code constrained by the path expression at any one time. The class of path expressions defined by Lauer and Campbell in [16] is a proper subset of our first class. Path expressions of Type 2 allow an unbounded number of processes to be active at the same time, by means of Habermann's braces ("{}") construct [9].

3.1 Regular Path Expressions

The abstract syntax [19] for path expressions of Type 1 is shown in Figure 1. Each operator expression is taken to be a labeled n-tuple. We denote an n-tuple x by $\langle s_1: v_1, \ldots, s_n: v_n \rangle$, where the s_i are the labels or selector names, and where the v_i are the component values. The s component of a labeled n-tuple x is denoted by x.s, so that for the example shown above, $x.s_1 = v_1$.

The abstract syntax can be embodied by a wide variety of concrete syntax schemes. We do not want to commit ourselves to any particular scheme, so we will formulate our definitions in terms of the abstract syntax. To improve readability, we will use a simplified concrete syntax in our examples, where we drop everything except for the component values of the n-tuples, introducing parentheses as necessary to show the association of operators. For instance, the abstract notation

Figure 1. Abstract Syntax for Path Expressions of Type 1

path-expression(p) <==> parallel-path(p) ∨ connected-path(p) ∨ path(p)
parallel-path(p) <==> p = <a1: x, op: "!", a2: y> ∧ path-expression(x) ∧ path-expression(y)
connected-path(p) <==> p = <a1: x, op: "&", a2: y> ∧ (connected-path(x) ∨ path(x))
∧ (connected-path(y) ∨ path(y))

path(p) <==> p = <beg: "path", a1: x, end: "end"> ∧ pattern(x)
pattern(p) <==> sequence(p) ∨ choice(p) ∨ star(p) ∨ name(p)
sequence(p) <==> p = <a1: x, op: ";", a2: y> ∧ pattern(x) ∧ pattern(y)
choice(p) <==> p = <a1: x, op: ",", a2: y> ∧ pattern(x) ∧ pattern(y)
star(p) <==> p = <a1: x, op: "*"> ∧ pattern(x)
name(p) <==> p ∈ P

where P is the set of all procedure names.

<a1: x, op: ";", a2: y> simplifies to just "x ; y".

A path expression is associated with a data type. Each object of the type has its own instance of the path expression, constraining the order in which the operations (procedures) of the data type can be applied to that particular object. Operations on disjoint sets of objects are not constrained, and can always proceed concurrently. The constraints imposed by a path expression can be summarized as follows. A single path requires that the procedures mentioned in it be executed in an order consistent with the pattern contained in it, where the specified pattern may be repeated indefinitely. Any two invocations of procedures mentioned in the same path are required to be mutually exclusive. A sequence expression "a ; b" requires an action of type "a" to immediately precede every action of type "b". A choice expression "a , b" matches an action of type "a" or of type "b", but not both. A star expression "a*" means that the pattern "a" may be repeated an arbitrary number of times, including none at all. Parentheses are used in the usual way to indicate the association of operators.

A path expression with parallel paths (!) requires the constraints imposed by each of the components to be met. A connected path (&) also imposes the constraints due to its components, and furthermore it requires all procedures mentioned to be mutually exclusive.

We first define the meaning of a path expression p containing only a single path, and we then define the meaning of a compound path expression in terms of the meanings of its component paths.

A path expression restricts the order in which the procedures named in it may be called,

and it does not restrict the procedures that are not mentioned. Let S(p) denote the set of procedures constrained by the path expression p. The function S is defined formally in the Appendix (definition A1). The constraints on the order of invocation for the procedures in a single path are defined in terms of a regular language over the alphabet S(p), which is derived from the path in a straightforward way: if p is a path, then the associated language L(p) is the language generated by the regular expression R(p), where the function R is defined recursively as follows:

R(p) = if path(p) then (R(p.a1))*
else if sequence(p) then R(p.a1) R(p.a2)
else if choice(p) then R(p.a1) | R(p.a2)
else if star(p) then (R(p.a1))*
else if name(p) then p

where | denotes disjunction and where * is the Kleene star. These concepts are illustrated in Example 1.

Example 1. p = path a ; (b , c) end,
S(p) = {a, b, c},
R(p) = (a (b | c))*, and
L(p) = {λ, ab, ac, abab, abac, acab, acac, ... }, where λ denotes the empty string.

To formalize the correspondence between the language and the set of legal computation histories, we introduce the concept of a *restricted computation history*. As discussed in the previous Section, the set of events E is partitioned into disjoint subsets which we identify with actions, denoting the set of all actions by A. We associate each action with a unique procedure, by introducing the set of all procedure names P and a function **type**: A --> P such that an action a is an invocation of the procedure named **type**(a).

To identify the data objects participating in an action, we introduce the **uses** relation, where **uses**(a, d) is true whenever the action a uses the data object d. Now we can define the set of actions constrained by the instance of the path p associated with the data object d to be $A_{p,d} = \{ a \in A \mid \mathbf{type}(a) \in S(p) \wedge \mathbf{uses}(a, d) \}$, which is just the set of invocations of the procedures in the set S(p) that involve the data object d. We define the partial order $\leq_{p,d} = \leq \mid A_{p,d}$ to be the restriction of ≤ to the subset of actions $A_{p,d}$. $\langle A_{p,d} , \leq_{p,d} \rangle$ is then a restricted history of the computation, describing only the set of actions constrained by p and using the data object d.

A computation history <A, ≤> is defined to be *consistent* with a single path p if and only if for every object d in the data type associated with p, there is an enumeration of the actions in the restricted history $A_{p,d} = \{ a_1, a_2, \ldots, a_n \}$ such that $a_1 < a_2 < \ldots < a_n$ and the word $w = \mathbf{type}(a_1) \cdot \mathbf{type}(a_2) \cdot \ldots \cdot \mathbf{type}(a_n)$ is a prefix of some word in the language L(p), where · denotes concatenation.

A computation history is consistent with a parallel path p if it is consistent with each component of p, and a computation history is consistent with a connected path p' if it is both consistent with each component and if all of the procedures mentioned in p' are pairwise mutually exclusive. More precisely, if parallel-path(p), then a computation history <A, ≤> is consistent with p if and only if it is consistent with p.a1 and it is consistent with p.a2. If connected-path(p), then a computation history <A, ≤> is consistent with p if and only if it is consistent with p.a1 as well as p.a2, and in addition $\leq_{p,d}$ is a total order for each data object d.

Parallel and connected paths are illustrated in Example 2.

Example 2. $p = p_1 ! p_2$

p_1 = path a ; b end

p_2 = path a ; c end

$L(p_1) = (ab)^*$

$L(p_2) = (ac)^*$

$A_{p,d} = \{ A, B, C \}$

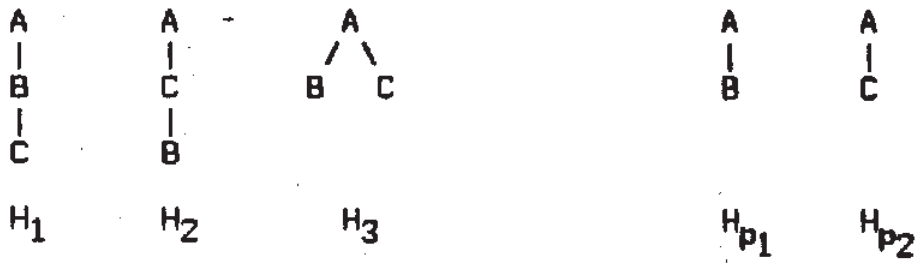
type(A) = a

type(B) = b

type(C) = c

In this Example as well as in the following discussion, we discuss the instance of the path expression p associated with some particular data object d. In the particular history shown in the Example, there is only one action of each type that uses d. The legal orderings for these actions are illustrated in Figure 2. Time flows from top to bottom: for two distinct nodes x and y, x ≤ y if and only if there is a downwards path from node x to node y. Note that the histories H_1, H_2, and H_3 all have the same restricted histories H_{p_1} and H_{p_2}. H_1 and H_2 are also legal histories for the path expression $p' = p_1 \,\&\, p_2$. H_3 is not consistent with p', because A and C are concurrent, but p' requires them to be mutually exclusive.

Figure 2. Some Legal Computation Histories for p



3.2 Braces

The abstract syntax for a path expression of Type 2 is shown in Figure 3. Our main concern in this Section is the formalization of Habermann's braces [9]. We have omitted the & and * operators, which present no essential difficulties, in order to simplify our discussion. A path expression { a } requires at least one action of type a to occur before it can be passed over, and it allows an arbitrary number of actions of type a to occur concurrently.

We first define the constraints imposed by a single path, in terms of a generalized language. In the previous Section, we used the conventional definition of a language as a set of words over some alphabet. In this Section we will take a language to be a set of acyclic directed graphs, with vertices bearing labels from the alphabet S(p). The formal definition of function S for path expressions of Type 2 is given in the Appendix (definition A2). A single path expression free of braces generates a language containing only graphs that are chains (strict total orders), which is isomorphic to the regular language defined in the previous Section.

Figure 3. Abstract Syntax for Path Expressions of Type 2

path-expression(p) <==> parallel-path(p) ∨ path(p)
parallel-path(p) <==> p = <a1: x, op: "|", a2: y> ∧ path-expression(x) ∧ path-expression(y)

path(p) <==> p = <beg: "path", a1: x, end: "end"> ∧ pattern(x)
pattern(p) <==> sequence(p) ∨ choice(p) ∨ braces(p) ∨ name(p)
sequence(p) <==> p = <a1: x, op: ";", a2: y> ∧ pattern(x) ∧ pattern(y)
choice(p) <==> p = <a1: x, op: ",", a2: y> ∧ pattern(x) ∧ pattern(y)
braces(p) <==> p = <beg: "{", a1: x, end: "}"> ∧ pattern(x)
name(p) <==> p ∈ P

where P is the set of all procedure names.

We will take a labeled graph G to be an ordered triple <v: G.v, e: G.e, t: G.t>, where G.v $\subseteq$ N is the set of vertices of G, G.e $\subseteq$ G.v X G.v is the set of edges of G, and where G.t: G.v --> P is a function associating type labels from the set of procedure names P with the vertices of G. We define the language L(p) as follows:

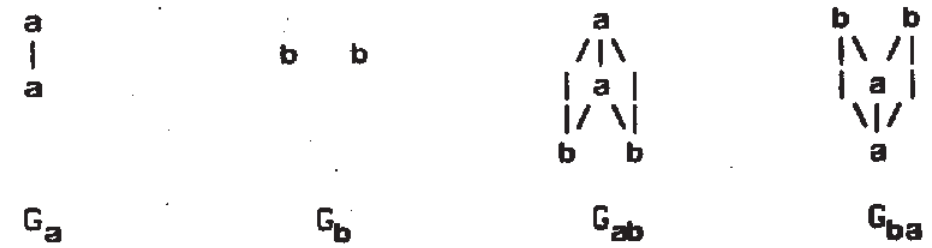
```
L(p) = if path(p) then (L(p.a1))*
       else if sequence(p) then L(p.a1) << L(p.a2)
       else if choice(p) then L(p.a1) U L(p.a2)
       else if braces(p) then (L(p.a1))‖
       else if name(p) then single(p)
```

where single(p) constructs a graph with one vertex labeled by p and with no edges, and where << is a concatenation operator on graphs, * is analogous to the Kleene star, and ‖ denotes parallel closure. These operations are defined so that the set of edges of any graph in L(p) is a strict partial ordering relation.

The concatenation of two graphs is a graph with two disjoint copies of the original graphs, with additional edges from every vertex of the first graph to every vertex of the second graph. The vertices all bear their original labels. For example, in Figure 4 we have $G_{ab} = G_a << G_b$ and $G_{ba} = G_b << G_a$, where all of the arcs are assumed to be directed from top to bottom. The concatenation of two sets of graphs is the set of all pairwise concatenations of the elements of the given sets.

If S is a set of graphs, S^* is the set of all possible repeated concatenations of elements of S. The * operation can be defined in terms of << as follows:

Figure 4. Examples of the Graph Concatenation Operator <<



$S^* = \bigcup_{i \geq 0} S_i$, where

$S_0 = \{ < \{\}, \{\}, \{\} > \}$ (the set containing only the empty graph), and where

$S_n = S << S_{n-1}$ for $n > 0$.

The parallel closure of a graph g is the set containing a graph consisting of n disjoint copies of g, for every natural number $n > 0$. For example, the graphs G_1, G_2, G_3, etc. shown in Figure 5 are all members of the set of graphs $G_1^{\|}$. The parallel closure of a set of graphs is the union of the parallel closures of the elements of the given set. The formal definitions of << and || are given by A3 and A4 in the Appendix.

A computation history <A, ≤> is consistent with a single path p of Type 2 if and only if for each object d in the data type associated with p there is an enumeration of the actions in the restricted history $A_{p,d} = \{ a_1, a_2, \ldots, a_n \}$ and a graph G with vertices $\{ v_1, v_2, \ldots, v_n \}$ such that the following conditions hold:

1. The type label $G.t(v_i)$ is equal to $type(a_i)$ for each node v_i.
2. For each edge from v_i to v_j in the graph G, $a_i \leq a_j$ in the computation history.
3. G is a prefix[4] of some graph in $L(p)$.

The restricted history $A_{p,d}$ is defined as in the previous Section. If parallel-path(p) (cf. Figure 3) then a computation history is consistent with p if and only if it is consistent with p.a1 and it is consistent with p.a2.

Example 3 treats the readers-writers problem [3], a well known synchronization task. Many variations have been discussed in the literature [7], but the basic problem is to synchronize many processes wishing to read and write a shared data base. Only one writer is allowed to modify the

---

Figure 5. Example of the Parallel Closure Operator ||

---

4. The graph X is a prefix of the graph G iff $X.v \subseteq G.v \wedge X.e \subseteq G.e \wedge X.t \subseteq G.t \wedge (\forall n \in X.v)(\forall m \in G.v)[<m, n> \in G.e \Longrightarrow m \in X.v \wedge <m, n> \in X.e]$

data base at any one time, and no readers may examine the data base while a writer is modifying it. However, readers do not interfere with each other, so that any number of readers may examine the data base concurrently. The path expression p_{rw} describes the above synchronization requirement.

Example 3. p_{rw} = path write , { read } end

Some of the graphs in the language $L(p_{rw})$ are shown in Figure 6. The arcs that can be deduced from the transitivity property have been omitted to simplify the figure. The subset of $L(p_{rw})$ shown corresponds to computation histories with exactly two distinct write actions, separated by some number of read actions on the same data base. Any computation whose restricted history is a prefix of a graph in this set is consistent with the path expression p_{rw}.

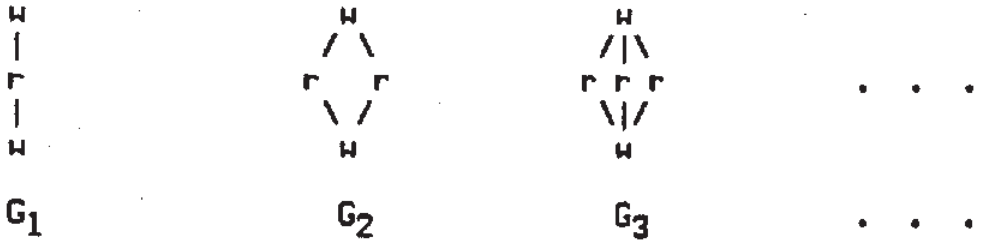
Braces can also be used in less obvious ways, as illustrated in Example 4 by the shared stack problem. A stack is shared among many processes that can all perform push and pop operations. Only one operation of either type may be performed at the same time, and furthermore any attempt to pop an empty stack is to be suspended until some other process makes the stack non-empty by performing a push operation. The path expression p_{stack} describes the above synchronization constraint.

Example 4. p_{stack} = p_{stack1} ! p_{stack2}
p_{stack1} = path { push ; pop } end
p_{stack2} = path push , pop end.

p_{stack1} guarantees that every pop is preceded by a corresponding push on the same stack object, so that the stack is never empty when a pop is performed, while p_{stack2} establishes the mutual

---

Figure 6. Some Members of $L(p_{rw})$



exclusion constraint. Note that all of the legal histories are total orders. However, the language over the alphabet $S(p_{stack})$ corresponding to the set of legal histories is not regular, so that this synchronization constraint cannot be described by any path expression of Type 1. A graph G in $L(p_{stack1})$ is shown in Figure 7, as well as three histories H_1, H_2 and H_3. These three histories, together with the three more that are obtained by interchanging the subscripts, exhaust the set of histories with two push actions and two pop actions that are consistent with G and with p_{stack2}. Note that the graph G has two disjoint subgraphs.

Our definition of braces differs from Habermann's definition [9, 2] in that we do not require the simultaneous executions of procedures in braces to overlap, while he does. Every history that is legal with respect to Habermann's definition is legal with respect to our definition (the implementation sketched in [2] is consistent with our definition), but our definition admits some histories as legal that Habermann's definition does not. For example, the history H shown in Figure 8 is not consistent with the path expression p_{ab} = path a ; { b } end according to the informal description of braces in [2], but it is a legal history according to our definition, since the graph G1 corresponding to the history H is a prefix of G2 $\in L(p_{ab})$. We have formulated our definitions to exhibit the histories with the greatest amount of parallelism that satisfy the synchronization constraints. An implementation must guarantee that the ordering constraints we define are met, although it may introduce additional constraints if that is convenient. However, these additional constraints may vary from installation to installation and from time to time, so that programs should not rely on them.

Figure 7. Some Legal Computation Histories for p_{stack}

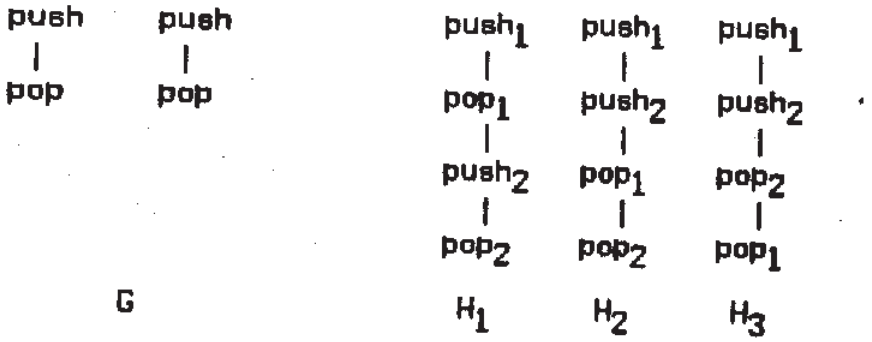
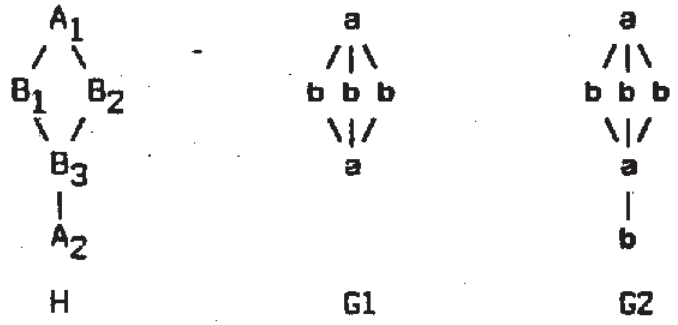


Figure 8. A Disputed History



4. Axiomatics

In this Section we define the semantics of a path expression p associated with a data object d in terms of a first order theory [18]. We derive a sentence F(p, d) from the path expression p which serves as a non-logical axiom of the theory. Each model of the theory contains a relation structure (A, ≤) where A is the set of actions and ≤ is the partial order defined over A, which is essentially a computation history, as defined above. The set of all finite models satisfying the theory associated with a path expression is precisely the set of histories of all computations allowed by the path expression.

The above formalism is chosen because it has been extensively used for proving properties of well known synchronization problems [6, 8, 12] as well as for proving properties of sequential programs. This Section describes a method for deriving the predicates from a path expression specification. In the next Section, the predicate is used to prove synchronization properties.

For simplicity, we first consider a subset of the path expressions of Type I, as defined in Section 3.1, such that a procedure name appears at most once in a single path, where a single path is defined to be a path expression satisfying the predicate *path* of Figure 1. We describe a method for deriving a predicate F(p, d) corresponding to a single path p, and then we define the predicate corresponding to a compound path expression in terms of the predicates corresponding to its component paths.

We know that the constraints on the order of invocation for the procedures in a single path p can be defined in terms of a regular language L(p) generated by the associated regular expression R(p), over the alphabet S(p). There is a well known effective method [1] for

constructing a finite state acceptor which accepts the strings in the language defined by a regular expression. The predicate F(p, d) for a single path p is constructed from M(p), the finite state acceptor for the *prefix* language of L(p), the language containing all the prefixes of the strings in L(p). It can be shown that F(p, d) associated with a path expression p completely captures the synchronization constraints due to p, in the sense that the set of computation histories for which F(p, d) holds is the same as the set of computation histories consistent with p, so that the above two semantic definitions of a path expression are equivalent.

We will define a finite state acceptor (FSA) M as a quadruple $\langle M_Q, M_\Sigma, M_T, M_{q_0} \rangle$ where

M_Q = a finite set of *states*,
M_Σ = a finite set of symbols called the *input alphabet*,
M_T = a relation describing the transitions between states in M_Q
($M_T \subseteq M_\Sigma \times M_Q \times M_Q$) called the *next-state* (or *transition*) relation, and
M_{q_0} = the distinguished element of M_Q called the *starting state*.

In order to construct a FSA accepting all the prefixes of L(p), we construct the FSA accepting L(p) and make every state in its state set a final state. Thus the set of final states in M(p) is the state set M_Q.

There is a distinct state in M(p) corresponding to every occurrence of a procedure name in p. As we assume that a procedure name can appear at most once in a single path, there is a unique state q_t in M(p) corresponding to each procedure name t in S(p). M(p) also has a distinguished state, labelled M_{q_0}, distinct from the states corresponding to the procedure names in p, which is the starting state of M(p), so that $|M_Q| = |S(p)| + 1$. M_Σ, the input alphabet of M(p), is S(p), the set of procedure names in p. The details of the construction are given in definition A5 of the Appendix.

The FSA for the path expression in Example 1 is shown below:

Example 1 *Continued.* p = path a; (b, c) end

$M(p) = (M_Q, M_\Sigma, M_T, M_{q_0})$ where
$M_\Sigma = S(p) = \{ a, b, c \}$,
$M_Q = \{ q_0, q_a, q_b, q_c \}$,
$M_T = \{ \langle a, q_0, q_a \rangle, \langle a, q_b, q_a \rangle, \langle a, q_c, q_a \rangle, \langle b, q_a, q_b \rangle, \langle c, q_a, q_c \rangle \}$.
$M_{q_0} = q_0$

The state diagram of the FSA M(p) is given in Figure 9 below.

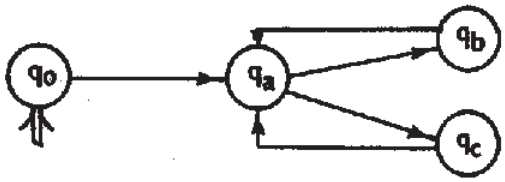
4.1 Predicates for Single Paths

In this Subsection, we construct the predicate F(p, d) from the finite state acceptor M(p) for a single path p. A state in M(p) can be identified with an equivalence class of legal histories of p, where two histories are equivalent if and only if the corresponding restricted histories are equivalent. Two restricted histories are equivalent whenever their last actions have the same type.

The starting state M_{q_0} represents the equivalence class containing computation histories in which none of the actions is of a type in S(p) (i.e. their restricted histories are empty), while any other state q_t represents the equivalence class containing computation histories in which the last action in the corresponding restricted histories is of type t. A computation history in the equivalence class corresponding to $q_{t'}$ can be extended by adding an action of type t if and only if there is a transition labelled t from $q_{t'}$ to q_t in the state diagram, i.e. if $\langle t, q_{t'}, q_t \rangle \in M_T$. A legal history thus drives the FSA to the state representing the equivalence class of which it is a member.

The predicate F(p, d) states that every history in the equivalence class q_t must be a *simple extension* of some history in an equivalence class $q_{t'}$, such that there is a transition from $q_{t'}$ to q_t in M(p). A history $\langle A, \leq \rangle$ is a simple extension of the history $\langle A', \leq' \rangle$ if $A' \subseteq A$, $\leq' \subseteq \leq$, there is a unique action a such that $a \in A - A'$ and type(a) $\in S(p)$, and the action a occurs strictly after every action in A'. This means that any (finite) legal history can be constructed from a null history by a

Figure 9. State Diagram of M(p)



series of simple extensions, which corresponds to the sequence of transitions M(p) undergoes in accepting the history.

In order to simplify our definition of F(p, d), we will assume that quantified variables range over the set of actions restricted by the path expression associated with the data object d throughout the remainder of Section 4. The treatment below can be made rigorously correct by replacing all predicates of the form $(\forall a)[\ P(a)\ ]$ by $(\forall a)[\ \text{constrained}(a, p, d) \Longrightarrow P(a)\ ]$ and replacing all predicates of the form $(\exists a)[\ P(a)\ ]$ by $(\exists a)[\ \text{constrained}(a, p, d) \wedge P(a)\ ]$, where $\text{constrained}(a, p, d) \equiv \text{type}(a) \in S(p) \wedge \text{uses}(a, d)$.

F(p, d) is expressed as the conjunction of predicates $f_{q_t}(a)$ corresponding to every non-starting state q_t of M_Q and the predicate Mutex(S(p)) stating that the actions of the types in S(p) are mutually exclusive.

$$F(p, d) \equiv \text{Mutex}(S(p)) \wedge (\forall a)[\ \bigwedge_{q_t \in M_Q - \{ M_{q_0} \}} f_{q_t}(a)\ ]$$

$$\text{Mutex}(A) \equiv (\forall a1, a2)[\ (\ \text{type}(a1) \in A \wedge \text{type}(a2) \in A) \Longrightarrow (\ a1 \le a2 \vee a2 \le a1)\ ]$$

$\bigwedge_{e \in A} f_e$ denotes the conjunction of a family of formulas $\{ f_e \mid e \in A \}$ indexed by the set A.

The predicate $f_{q_t}(a)$ partially specifies the ordering constraints on an action a of type t. This is done by enumerating the possible types of the action immediately preceding a in the restricted history corresponding to a history in the equivalence class represented by q_t. The predicate is the disjunction of the predicates corresponding to every transition from some $q_{t'}$ into q_t. If q_t has a transition from the starting state M_{q_0}, an action of type t can be the first action in the restricted history. In such a case, f_{q_t} is defined by

$$f_{q_t}(a1) \equiv \text{First}_p(a1, t) \vee (\ \bigvee_{\langle t, q_{t'}, q_t \rangle \in M_T} \text{Precedes}_p(t', a1, t)\ )$$

$$\text{First}_p(a1, t) \equiv \neg (\exists a2)[\ a2 < a1\ ]$$

$$\text{Precedes}_p(t', a1, t) \equiv \text{type}(a1) = t \Longrightarrow (\exists a2)[\ a2 \rightarrow_p a1 \wedge \text{type}(a2) = t'\ ]$$

$$a2 \rightarrow_p a1 \equiv (a2 < a1) \wedge \neg (\exists a3)\ [\ a2 < a3 \wedge a3 < a1\ ]$$

and otherwise

$$f_{q_t}(a1) \equiv \Big(\bigvee_{\langle t, q_{t'}, q_t \rangle \in M_T} \textbf{Precedes}_p(t', a1, t) \Big)$$

$\bigvee_{e \in A} f_e$ denotes the disjunction of a family of formulas $\{ f_e \mid e \in A \}$. The predicate $\textbf{Precedes}_p(t', a1, t)$ states that if a1 is of type t, an action of type t' immediately precedes a1 in the restricted history ($\rightarrow_p$ relates an action to its immediate successor in the restricted history).

It can be easily shown that F(p, d) holds for a history in which none of the actions is of a type in S(p) (i.e., the restricted history is 'null'). The predicate for the path expression of Example 1 is shown below.

Example 1 *Continued.* p = path a; (b, c) end (Please refer to Figure 9)

$f_{q_a}(a1) \equiv \textbf{First}_p(a1, a) \vee \textbf{Precedes}_p(b, a1, a) \vee \textbf{Precedes}_p(c, a1, a)$

$f_{q_b}(a1) \equiv \textbf{Precedes}_p(a, a1, b)$

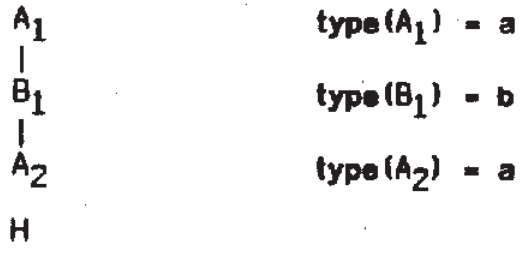
$f_{q_c}(a1) \equiv \textbf{Precedes}_p(a, a1, c)$

$F(p, d) \equiv \textbf{Mutex}(\{ a, b, c \}) \wedge (\forall a1) [f_{q_a}(a1) \wedge f_{q_b}(a1) \wedge f_{q_c}(a1)]$

It can be easily shown that the history corresponding to a string in $L = \{ \lambda, a, ab, ac, aba, aca, abab, \ldots \}$ satisfies F(p, d), or equivalently that the history corresponding to a string in L is a model of the theory generated by F(p, d). To illustrate, we show that the history corresponding to the string aba in Figure 10 is a model. Clearly, Mutex({ a, b, c }) holds as all the actions are totally ordered. $f_{q_a}(A_1)$ holds as A_1 is the first action in H. $f_{q_a}(B_1)$ also holds as B_1 is not of type a. $f_{q_a}(A_2)$ holds as A_2 is of type a and an action of type b immediately precedes A_2. So $f_{q_a}(a1)$ holds for each a1 in the history. Similarly it can be shown that $f_{q_b}(a1)$ as well as $f_{q_c}(a1)$ holds for every a1. Thus F(p, d) holds establishing that H is a model of the theory generated by F(p, d). Any computation history with its restricted history corresponding to a string

---

Figure 10. A Model of F(p, d)



in L satisfies F(p, d).

4.2 Predicates for Compound Path Expressions

The predicate for a compound path expression can be defined in terms of the predicates associated with each of the simple paths in it. The predicate F(p, d) for a path expression p with parallel paths (!) is simply the conjunction of predicates associated with each of its components.

If parallel-path(p) then F(p, d) = F(p.a1, d) ∧ F(p.a2, d)

The predicate associated with a connected path p (&) is also the conjunction of predicates associated with each of the components with an additional restriction that all the actions of type in S(p) are mutually exclusive (i.e. totally ordered).

If connected-path(p)
then F(p, d) = F(p.a1, d) ∧ F(p.a2, d) ∧ **Mutex**(S(p.a1) ∪ S(p.a2))

Example 2 *Continued.* $p = p_1 \,!\, p_2$
p_1 = path a; b end
p_2 = path a; c end

The state diagrams of $M(p_1)$ and $M(p_2)$ are given in Figure 11 below. The predicates corresponding to $M(p_1)$ are:

$f_{q_a^1}(a1) \equiv \textbf{First}_{p_1}(a1, a) \vee \textbf{Precedes}_{p_1}(b, a1, a)$

$f_{q_b^1}(a1) \equiv \textbf{Precedes}_{p_1}(a, a1, b)$

$F(p_1) \equiv \textbf{Mutex}(\{ a, b \}) \wedge (\forall a1)\, [\, f_{q_a^1}(a1) \wedge f_{q_b^1}(a1) \,]$

The predicates corresponding to $M(p_2)$ are:

$f_{q_a^2}(a1) \equiv \textbf{First}_{p_2}(a1, a) \vee \textbf{Precedes}_{p_2}(c, a1, a)$

$f_{q_c^2}(a1) \equiv \textbf{Precedes}_{p_2}(a, a1, c)$

Figure 11. State Diagrams of $M(p_1)$ and $M(p_2)$

$F(p_2) \equiv \text{Mutex}(\{ a, c \}) \wedge (\forall\, al)\, [\, f_{q_a^2}(al) \wedge f_{q_c^2}(al)\,]$

$$\begin{aligned} F(p, d) &\equiv F(p_1) \wedge F(p_2) \\ &\equiv \text{Mutex}(\{ a, b \}) \wedge \text{Mutex}(\{ a, c \}) \\ &\quad \wedge (\forall\, al)\, [\, (\, \text{First}_{p_1}(al, a) \vee \text{Precedes}_{p_1}(b, al, a)\,) \wedge (\text{First}_{p_2}(al, a) \vee \text{Precedes}_{p_2}(c, al, a)\,) \\ &\qquad \wedge \text{Precedes}_{p_1}(a, al, b) \wedge \text{Precedes}_{p_2}(a, al, c)\,] \end{aligned}$$

It can be easily shown that the histories H_1, H_2, and H_3 in Figure 2 satisfy F(p, d). If instead we had $p' = p_1 \,\&\, p_2$ then the restricted histories should satisfy an additional constraint. The predicate F(p') corresponding to p' is

$F(p, d) \wedge \text{Mutex}(\{ a, b, c \})$

H_1 and H_2 satisfy F(p') while H_3 does not as the actions B and C are not mutually exclusive.

4.3 Extensions

In Sections 4.1 and 4.2 we showed how to construct the predicate F(p, d) characterizing the histories consistent with a path expression p, assuming that each procedure name occurs at most once in the path expression p. A similar approach can be used for axiomatizing general path expressions of Type 1, as outlined below. Handling *braces* (i.e. path expressions of Type 2) requires substantial modifications to the method which are beyond the scope of this paper.

The construction of M(p) presented in the Appendix will work for arbitrary path expression of Type 1, provided that we associate a distinct state of M(p) with each occurrence of a procedure name in the path expression p. Procedures that are mentioned more than once will therefore correspond to several different states. The construction is illustrated in Example 5 below.

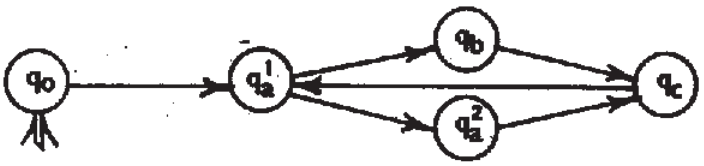
Example 5. p = path a; (b, a); c end

The corresponding FSA M(p) is
$(M_Q, M_\Sigma, M_T, M_{q_0})$ where

$M_\Sigma = S(p) = \{ a, b, c \}$,
$M_Q = \{ q_0, q_a^1, q_a^2, q_b, q_c \}$,
$M_T = \{ \langle a, q_0, q_a^1\rangle, \langle a, q_c, q_a^1\rangle, \langle a, q_a^1, q_a^2\rangle, \langle b, q_a^1, q_b\rangle, \langle c, q_a^2, q_c\rangle, \langle c, q_b, q_c\rangle \}$.
$M_{q_0} = q_0$

The state diagram of M(p) is given below in Figure 12. Notice that there are two states, q_a^1 and q_a^2, corresponding to the procedure name a.

Figure 12. State Diagram for M(p)



The construction of F(p, d) is more difficult than in the restricted case because we can no longer uniquely characterize a state of the FSA M(p) by the associated procedure name. For a single path p, F(p, d) is supposed to be true for a given computation history (model) whenever there is some sequence of state transitions corresponding to the sequence of actions in the history. We proceed by associating a predicate $f_{q_t^i}(a)$ with each state q_t^i, which states the constraints a history must satisfy if the action a of type t drives the FSA M(p) into the state q_t^i.

F(p, d) is the conjunction of **Mutex**(S(p)), the predicate expressing total ordering on the actions of the type in S(p), and the predicates corresponding to each procedure name in S(p). The predicate corresponding to a procedure name t is the disjunction of the predicates associated with the states corresponding to t, because a history ending in an action of type t will drive the FSA into one of the states corresponding to t.

The predicate $f_{q_t^i}$ is the disjunction of the predicates associated with each transition into the state q_t^i. Let $\textbf{Last-state}_p(q_{t'}, a1, t)$ be the predicate stating that if a1 is of type t, an action of type t' immediately precedes a1 in the restricted history and drives M(p) into the state $q_{t'}$.

$$\textbf{Last-state}_p(q_{t'}, a1, t) \equiv \textbf{type}(a1) = t \Longrightarrow (\exists\, a2)\,[\,\textbf{type}(a2) = t' \wedge a2 \rightarrow_p a1 \wedge f_{q_{t'}}(a2)\,]$$

If $q_{t'}$ is the unique state corresponding to t', it can be completely characterized by the associated type. In that case, the predicate $\textbf{Precedes}_p(t', a1, t)$ defined in Section 4.1 can be used instead of $\textbf{Last-state}_p(q_{t'}, a1, t)$. If each cycle in the machine is broken by a state which is the only one of its type, this recursive definition will terminate, as illustrated for the path expression of Example 5 below.

Example 5 *Continued.* p = path a ; (b , a) ; c end

$f_{q_a^1}(a1) \equiv \textbf{First}_p(a1, a) \vee \textbf{Precedes}_p(c, a1, a)$

$f_{q_a^2}(a1) \equiv \textbf{Last-state}_p(q_a^1, a1, a)$

$f_{q_b}(a1) \equiv \textbf{Last-state}_p(q_a^1, a1, b)$

$f_{q_c}(al) \equiv \textbf{Last-state}_p(q_a^2, al, c) \vee \textbf{Precedes}_p(b, al, c)$

$F(p, d) \equiv \textbf{Mutex}(\{ a, b, c \}) \wedge (\forall al)\ [\ f_a(al) \wedge f_b(al) \wedge f_c(al)\ ]$

where $f_a(al) \equiv f_{q_a^1}(al) \vee f_{q_a^2}(al)$

$f_b(al) \equiv f_{q_b}(al)$

$f_c(al) \equiv f_{q_c}(al)$

Note that in this Example, there is a unique state associated with the procedure c, and that this state cuts every cycle of M(p). Expanding all of the definitions we used in constructing M(p) will lead to a finite formula in terms of actions and their types.

M(p) can also have cycles in which no state can be uniquely characterized by a procedure type. Such a path expression is shown in Example 6, and the corresponding FSA is shown in Figure 13.

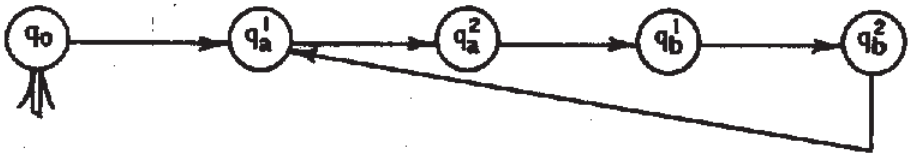
Example 6. p = path a ; a ; b ; b end

In such a case, we can no longer consider the formulas $f_{q_t^i}$ to be abbreviations for formulas involving only $\leq$ because the process of expanding the abbreviations will not terminate. However, we can throw in the definitions of the $f_{q_t^i}$ as additional non-logical axioms. If this is done, these axioms can be used in a proof to expand the definition of F(p, d) as far as required. We have found this to be a convenient framework for doing proofs.

5. Proving Properties of Path Expressions

One of the motivations for constructing a formal semantics for path expressions is the desire to prove that the synchronization scheme specified by a path expression satisfies certain properties. We will be concerned with two kinds of properties: the invariance of a given assertion, and liveness.

Figure 13. State Diagram for M(p)



5.1 Invariants

One way to increase the likelihood that a synchronization scheme meets a user's needs is to formulate a predicate expressing some user requirement, and to prove that it is always satisfied. Such predicates, known as *invariant conditions* [22, 20], are often easier to understand than process oriented specifications, because they describe static properties of the system. Therefore an inappropriate invariant is more likely to be recognized as an error than an "almost correct" process oriented specification.

We will say that a property is an *invariant* if it holds for all (finite) legal histories. The rest of this Subsection is devoted to establishing a method for proving that a property is an invariant.

We define the **prefix** relation on computation histories as follows: h **prefix** h' if and only if h is a proper subset of h' and none of the new actions in h' occur strictly before any of the actions in h.[5] It is easy to show that **prefix** is a strict partial ordering relation on computation histories. The empty history is the least element of the set of all finite histories FH with respect to the **prefix** ordering, and in general any subset of FH has at least one minimal element with respect to **prefix**, as is easily shown by noting that <A, ≤> **prefix** <A', ≤'> implies |A| $<_N$ |A'|, since $<_N$ is a well ordering relation on the natural numbers N. This justifies the following (strong) induction rule: from a proof of the statement (∀h) [(∀h') [h' **prefix** h ==> P(h')] ==> P(h)] we can conclude that (∀h) [P(h)], for any predicate P.

A typical proof of the invariance of a property using this rule breaks up into cases. First the property is shown to hold for the empty history, and then for non-empty histories, a maximal action a_{max} is chosen, and a case analysis on the type of a_{max} completes the proof. This approach is illustrated in Example 7 below.

Example 7. One slot message buffer.

Consider the one slot message buffer data type presented in [10]. A message buffer is used as a communication link between several processes, all of which have access to the buffer. The data type provides two operations: *deposit*, which inserts a message into the buffer, and *remove*, which removes (and returns) a message from the buffer. The buffer can hold at most one message

---

5. Let h = <A, ≤> and h' = <A', ≤'>. Then h **prefix** h' iff
A ⊆ A' ∧ ¬(A = A') ∧ ≤ = (≤' | A) ∧ (∀a1 ∈ A)(∀a2 ∈ A') [a2 <' a1 ==> a2 ∈ A]

at a time.

The path expression p associated with the one slot message buffer data type is shown in Figure 14, together with the predicate F(p, b), constructed by the method of Section 4. We would like to prove that in any legal computation, no attempt will be made to remove a message from an empty buffer, or to deposit a message into a full buffer. An assertion expressing this requirement is

inv(b, h) ≡ legal(h) ==> 0 ≤ #(deposit, b, h) - #(remove, b, h) ≤ 1,

where b is a message buffer, h is a computation history, and #(t, b, h) is the number of actions of type t using the data object b in the history h. The predicate legal(h) means that the computation history h is consistent with the path expression p, so that h is a finite model of F(p, b). The invariant inv(b, h) says that no more messages have been removed than have been deposited, and that at most one more message has been deposited than has been removed. Note that inv(b, h) is expressed as a property of a legal computation history, and that it is completely independent of how the message buffer is implemented. The proof of the invariant is shown below.

Theorem: (∀h, b) [inv(b, h)]
Proof: By strong induction. Let h = <A, ≤>.
Assume: (∀h') [h' **prefix** h ==> inv(b, h')]
Show: inv(b, h)

Case 1: |A| = 0.
then #(deposit, b, h) = 0 = #(remove, b, h)

Figure 14. Synchronization Constraints for the One Slot Message Buffer

path deposit ; remove end

F(p, b) ≡ **Mutex**({ deposit, remove }) ∧ ∀a [constrained(a, p, b) ==> $f_{deposit}$(a) ∧ f_{remove}(a)]
$f_{deposit}$(a) ≡ $\mathbf{First}_p$(a, deposit) ∨ $\mathbf{Precedes}_p$(remove, a, deposit)
f_{remove}(a) ≡ $\mathbf{Precedes}_p$(deposit, a, remove)

Mutex(S) ≡ (∀ a1, a2) [(**type**(a1) ∈ S ∧ **uses**(a1, b) ∧ **type**(a2) ∈ S ∧ **uses**(a2, b))
==> (a1 ≤ a2 ∨ a2 ≤ a1)]
$\mathbf{First}_p$(a1, t) ≡ ¬ (∃ a2) [constrained(a2, p, b) ∧ a2 < a1]
$\mathbf{Precedes}_p$(t', a1, t) ≡ **type**(a1) = t ==> (∃ a2) [**type**(a2) = t' ∧ **uses**(a2, b) ∧ a2 $\dashrightarrow_p$ a1]
a2 $\dashrightarrow_p$ a1 ≡ (a2 < a1) ∧ ¬ (∃ a3) [constrained(a2, p, b) ∧ a2 < a3 ∧ a3 < a1]
constrained(a, p, b) ≡ **type**(a) ∈ S(p) ∧ **uses**(a, b)
S(p) = { deposit, remove }.

Since $0 \le 0 \le 1$, we conclude that
$0 \le$ #(deposit, b, h) - #(remove, b, h) ≤ 1,
and hence inv(b, h) holds.

Case 2: $|A| > 0$.

Assume: legal(h)
Show: $0 \le$ #(deposit, b, h) - #(remove, b, h) ≤ 1
Since A is nonempty and finite, there must be a maximal element with respect to $\le$.
Call it a.
Then $\neg (\exists a1 \in A)[\, a < a1 \,]$
Since legal(h), h is a model of F(p, b).

Case 2.1: type(a) = deposit $\wedge$ uses(a, b)
From $f_{deposit}$ we get $First_p$(a, deposit) $\vee$ $Precedes_p$(remove, a, deposit).

Case 2.1.1: $First_p$(a, deposit)
Since type(a) = deposit,
$\neg (\exists a2)[$ type(a2) $\in S(p) \wedge$ uses(a2, b) $\wedge\ a2 < a \,]$
But $\neg (\exists a1 \in A)[\, a < a1 \,]$ and Mutex({ deposit, remove }),
so a is the only action of type deposit using b,
and there are no actions of type remove using b.
#(deposit, b, h) = 1, and #(remove, b, h) = 0.
Since $0 \le 1 \le 1$, we conclude that
$0 \le$ #(deposit, b, h) - #(remove, b, h) ≤ 1.

Case 2.1.2: $Precedes_p$(remove, a, deposit)
Since type(a) = deposit,
$\exists a2 [\, a2 \dashrightarrow_p a \wedge$ uses(a2, b) $\wedge$ type(a2) = remove $\,]$
Choose such an a2,
let B = $\{ x \mid x \in A \wedge a2 \le x \}$, and
let h' = $\langle A - B, \le \mid (A - B) \rangle$.
By construction, h' is a prefix of h.
Since also $a2 \dashrightarrow_p a$, a and a2 are the only actions
of type deposit or remove in B that use b,
so #(deposit, b, h) = 1 + #(deposit, b, h'), and
#(remove, b, h) = 1 + #(remove, b, h').
inv(b, h') holds by the induction hypothesis,
and legal(h') since both legal(h) and h' prefix h,
so $0 \le$ #(deposit, b, h') - #(remove, b, h') ≤ 1. Since
#(deposit, b, h) - #(remove, b, h) = #(deposit, b, h') - #(remove, b, h'),
we conclude that
$0 \le$ #(deposit, b, h) - #(remove, b, h) ≤ 1.

Case 2.2: type(a) = remove $\wedge$ uses(a, b)
From f_{remove} we get $\exists a2 [\, a2 \dashrightarrow_p a \wedge$ type(a2) = deposit $\,]$
By the same argument as in case 2.1.2, we conclude that
$0 \le$ #(deposit, b, h) - #(remove, b, h) ≤ 1.

Case 2.3: ¬ (type(a) ∈ { deposit, remove } ∧ uses(a, b))
Let h' = <A - { a }, ≤ | (A - { a })>.
Since ¬ (∃ a1 ∈ A) [a < a1],
h' is a prefix of h, legal(h'), and inv(b, h'), yielding
0 ≤ #(deposit, b, h') - #(remove, b, h') ≤ 1.
#(deposit, b, h) = #(deposit, b, h'), and
#(remove, b, h) = #(remove, b, h'), so that
0 ≤ #(deposit, b, h) - #(remove, b, h) ≤ 1.

From cases 2.1, 2.2, and 2.3 we conclude that
0 ≤ #(deposit, b, h) - #(remove, b, h) ≤ 1,
and hence inv(b, h) holds.

From cases 1 and 2 we conclude that
∀h, b [∀h' [h' prefix h ==> inv(b, h')] ==> inv(b, h)],
So by induction ∀h, b [inv(b, h)], as was to be shown.

5.2 Liveness

We will say that a path expression is *live* [cf. 11] if given any computation history <A, ≤> consistent with the path expression p, there is a history <A', ≤'> that is an extension of <A, ≤>, [6] and which contains an action a_t of type t for each t in S(p), such that a <' a_t for each action a in A. If a process constrained by a path expression is free from deadlocks, the path expression must be live. It is still possible for an uncooperative set of processes to deadlock a live path expression, but there is no stronger condition depending only on the path expression and not on the processes invoking the operations constrained by the path expression that guarantees freedom from deadlock. For a live path expression, there is always some legal path out of any potential deadlock situation, but a deadlock can occur if no process actually invokes the next operation on that legal path. Liveness is clearly a desirable property for any synchronization scheme; however, it is possible to write path expressions that are not live, as shown by Example 8 below.

Example 8. path chicken ; egg end ! path egg ; chicken end

Since every chicken event must be preceded by an egg event, and vice versa, there can be no first chicken (or egg) event, and hence there can be no non-empty finite histories.

A procedure for deciding whether a given path expression of type I is live is outlined

6. A ⊆ A' ∧ (≤ = ≤' | A)

below. Lauer and Campbell [16] have proved that all single paths are representable as live Petri nets. Their result can be readily extended, yielding that all single paths of our Type 1 path expressions are live, as defined above. So we can limit our attention to path expressions with multiple paths.

The basic idea is to construct a cross product machine that will accept a language corresponding to the set of all restricted histories consistent with each path in the path expression p. Then p is live if and only if a cycle containing at least one transition of each type in S(p) is reachable from every state of the cross product machine that is reachable from the starting state. This characterization of liveness depends on the fact that we require actions to be disjoint, so that a procedure constrained by a path expression p may not call another procedure that is also constrained by p.[7] If actions are not required to be disjoint, then detecting deadlocks becomes more difficult, because there will in general be additional ordering constraints due to the fact that some events belong to several different actions. The above criterion can still be applied if the procedures are broken up into sequences of smaller actions that are in fact disjoint, and if the path expressions are rewritten in terms of these finer grained action types.

We construct the cross product machine for the path expression p from the finite state acceptors for the single paths of p. Let single-paths(p) = { $p_1, \ldots, p_n$ }, where the function single-paths is defined as follows:

```
single-paths(p) = if path(p) then { p }
                  else if connected-path(p) ∨ parallel-path(p)
                  then single-paths(p.a1) ∪ single-paths(p.a2)
```

Recall that $M(p_i)$ denotes the FSA for the single path p_i (for $1 \le i \le n$), as described in Section 4.1. The cross product machine M(p) is constructed as follows.

The set of states $M(p)_Q$ is just the cross product $M(p_1)_Q \times \ldots \times M(p_n)_Q$. The input alphabet $M(p)_\Sigma$ is $M(p_1)_\Sigma \times \ldots \times M(p_n)_\Sigma$, which is the same as $S(p_1) \times \ldots \times S(p_n)$. The transition relation $M(p)_T$ is defined by

---

7. Habermann does not impose this constraint, and one of the examples in [9] violates it. Lauer and Campbell [16] also do not mention this constraint, but it is implicit in the Petri net model they use.

$\langle t, q1, q2\rangle \in M(p)_T$ <==>
$\forall i\,[\,1 \le i \le n \Rightarrow ((t.i \in S(p_i) \wedge \langle t.i, q1.i, q2.i\rangle \in M(p_i)_T) \vee (t.i = \lambda \wedge q1.i = q2.i))$
$\wedge\ \forall j\,[\,1 \le j \le n \wedge t.i \in S(p_j) \Rightarrow t.j = t.i\,]\,]$

In other words, any component that changes must change via a transition of the FSA for that component, and if one component has a transition of type t, then all other components that have the type t in their input alphabets must also undergo a transition of type t. Components that do not change have λ transitions. The starting state $M(p)_{q_0}$ is just $M(p_1)_{q_0} \times \ldots \times M(p_n)_{q_0}$.

Some of the states of M(p) may not be reachable from the starting state. If this construction is to be mechanized, it may be desirable to save time and space by constructing only the states reachable from the starting state. Also note that the above definition includes λ transitions from each state to itself; these transitions do not affect the language accepted by the machine, and hence can be deleted if that is convenient.

8. Conclusions

The most important contribution of this paper is the formalization of synchronization constraints as a set of legal computation histories, which are viewed as partially ordered sets. We have shown how the set of legal histories can be characterized by a set of axioms, and how the finiteness of all legal histories can be described by introducing an induction principle. We have also clarified the connection between an event based description of a synchronization constraint, and a description based on invariant assertions: an assertion is an invariant if it is true for all legal computation histories.

We feel that our approach can be readily extended to cover other, more expressive formalisms for specifying synchronization constraints. For example, for a proper description of priority requirements, a distinction between a *request* for an action and the set of events constituting the *execution* of that action should be introduced.

Our definitions also provide insights into the properties of path expressions that are not as readily apparent from the informal descriptions in [2, 9, 10] or from the Petri net descriptions in [16]. Our definitions are consistent with and extend those of [16]. It is difficult to compare our definitions to Habermann's informal descriptions, because they are not precisely stated, but some differences are noted in Sections 3.2 and 5.2.

We have presented a well founded method for proving properties of path expressions,

which can be used to establish partial correctness results for programs containing path expressions. Positive total correctness results cannot be established without additional assumptions about the fairness of the scheduler, which cannot be expressed within the path expression formalism. We have presented an effective method for deciding whether a given path expression is live, which can be used to show that a path expression has a potential deadlock.

7. Acknowledgements

We wish to thank Professor Carl Hewitt and Professor Barbara Liskov for encouraging our research and providing many helpful suggestions. We also wish to thank Toby Bloom, Lawrence Flon, Eliot Moss and Mandayam Srivas for their critical readings of the drafts of this paper.

Appendix - Technical Definitions

Definition A1: S(p) for path expressions of type 1.

S(p) = if parallel-path(p) ∨ connected-path(p) ∨ sequence(p) ∨ choice(p) then S(p.a1) U S(p.a2)
else if path(p) ∨ star(p) then S(p.a1)
else if name(p) then { p }

Definition A2: S(p) for path expressions of type 2.

S(p) = if parallel-path(p) ∨ sequence(p) ∨ choice(p) then S(p.a1) U S(p.a2)
else if path(p) ∨ braces(p) then S(p.a1)
else if name(p) then { p }

Definition A3: The graph concatenation operator <<.

In order to formalize the notion of disjoint copies of a graph, we introduce the bijections b_1: N --> N_{even} and b_2: N --> N_{odd} defined by $b_1(x) = 2x$ and $b_2(x) = 2x + 1$. The essential property of these bijections is that their ranges are disjoint, so that $b_1(S_1)$ and $b_2(S_2)$ are disjoint sets, for any two sets of vertices (natural numbers) S_1 and S_2. We extend these bijections to edges (pairs of vertices) by the rule $b_i(\langle x, y\rangle) = \langle b_i(x), b_i(y)\rangle$, for i = 1, 2. The << operator is defined as follows:

g << h = G
G.v = b_1(g.v) U b_2(h.v),
G.e = b_1(g.e) U b_2(h.e) U (b_1(g.v) X b_2(h.v)),
G.t = (g.t • b_1^{-1}) U (h.t • b_2^{-1}).
S_1 << S_2 = { z | (∃ x ∈ S_1) (∃ y ∈ S_2) [z = x << y] }

where g and h are graphs, S_1 and S_2 are sets of graphs, and where X denotes cartesian product. Note that we are treating functions as sets of pairs, and that our convention for functional composition is (f • g) (x) = f(g(x)).

Definition A4: The parallel closure operator ‖

$g^{\|} = \{ g_i \mid i \geq 1 \}^{-}$
$g_1 = g$,
$g_n = \langle b_1(g.v) \cup b_2((g_{n-1}).v),\ b_1(g.e) \cup b_2((g_{n-1}).e),\ (g.t \cdot b_1^{-1}) \cup ((g_{n-1}).t \cdot b_2^{-1}) \rangle$
for $n > 1$.
$S^{\|} = \cup \{ S' \mid (\exists g \in S)[S' = g^{\|}] \}$.

where g is a graph and where b_1 and b_2 are defined as in A3.

Definition A5: M(p) for path expressions of type I.

$M(p) = \langle M_Q, M_\Sigma, M_T, M_{q_0} \rangle$
$M_Q = \{ q_t \mid t \in S(p) \} \cup \{ M_{q_0} \}$
$M_\Sigma = S(p)$

In order to construct the next-state relation M_T, we define below two functions *First* and *Last* for a single path with the power set of S(p) ∪ { λ } as their range, using the abstract syntax as defined in Figure 1. First(p) gives the set of procedure names appearing as the first element of the strings in the regular language L(p) corresponding to p (λ is assumed to be the first element of λ, the *empty* string.). Similarly, Last(p) is the set of procedure names appearing as the last element of the strings in L(p) (λ is also assumed to be the last element of λ.).

```
First(p) = if path(p) then First(<a1:p.a1, op:"*">)
           else if sequence(p) then
              if λ ∈ First(p.a1) then (First(p.a1) - { λ }) ∪ First(p.a2) else First(p.a1)
           else if choice(p) then First(p.a1) ∪ First(p.a2)
           else if star(p) then First(p.a1) ∪ { λ }
           else if name(p) then { p }
```

and

```
Last(p) = if path(p) then Last(<a1:p.a1, op:"*">)
          else if sequence(p) then
             if λ ∈ Last(p.a2) then Last(p.a1) ∪ (Last(p.a2) - { λ }) else Last(p.a2)
          else if choice(p) then Last(p.a1) ∪ Last(p.a2)
          else if star(p) then Last(p.a1) ∪ { λ }
          else if name(p) then { p }
```

The transition relation M_T is constructed as follows:

$$M_T = \tau(p) \cup \{ \langle t, M_{q_0}, q_t \rangle \mid t \in (First(p) - \{ \lambda \}) \}$$

where $\langle t2, q_{t1}, q_{t2} \rangle \in M_T$ means that there is transition labelled t2 from the state q_{t1} to the state q_{t2} in the state diagram of M(p). τ is defined as follows:

$\tau(p)$ = if path(p) then $\tau(\langle$a1:p.a1, op:"*"$\rangle)$
else if sequence(p) then
$\tau(p.a1) \cup \tau(p.a2) \cup \{ \langle t2, q_{t1}, q_{t2} \rangle \mid t1 \in (Last(p.a1) - \{ \lambda \}), t2 \in (First(p.a2) - \{ \lambda \}) \}$
else if choice(p) then $\tau(p.a1) \cup \tau(p.a2)$
else if star(p) then $\tau(p.a1) \cup \{ \langle t2, q_{t1}, q_{t2} \rangle \mid t1 \in (Last(p.a1) - \{ \lambda \}), t2 \in (First(p.a1) - \{ \lambda \}) \}$
else if name(p) then { }

Notice that M_T only contains triples of the form $\langle t2, q_{t1}, q_{t2} \rangle$, where the input symbol is always the index for the next state.

References

1. Aho, A.V., Ullman, J.D.: The Theory of Parsing, Translation, and Compiling: Vol. I: Parsing. Englewood Cliffs, N. J.: Prentice-Hall Inc. 1972.

2. Campbell, R.H., Habermann, A.N.: The Specification of Process Synchronization by Path Expressions. Lecture Notes on Computer Science, Vol. 16. Heidelberg-Berlin-New York: Springer Verlag 1974.

3. Courtois, P.J., Heymans, F., Parnas, D.L.: Concurrent Control with 'Readers' and 'Writers'. Comm. ACM 14, 10 pp. 667 - 668 (Oct. 1971).

4. Dijkstra, E.W.: Co-operating Sequential Processes. In: Programming Languages (F. Genuys, ed.). New York: Academic Press 1968.

5. Flon, L., Habermann, A.N.: Towards the Construction of Verifyable Software. Proc. Conf. on Data: Abstractions, definition, and Structure, SIGPLAN Notices 8, 2 (1976).

6. Greif, I.: Semantics of Communicating Parallel Processes. M.I.T., Laboratory for Computer Science, MAC TR-154, Ph. D. Thesis, Sept. 1975.

7. Greif, I.: Formal Problem Specifications for Readers and Writers Scheduling. University of Washington, Dept. of Computer Science, April 1976.

8. Habermann, A.N.: Synchronization of Communicating Processes. Comm. ACM 15, 3 pp. 171-176 (March 1972).

9. Habermann, A.N.: Operations on Shared Data Controlled by Function Modules in Type Definitions. Carnegie-Mellon University, Sept. 1973.

10. Habermann, A.N.: Path Expressions. Carnegie-Mellon University, June 1975.

11. Hack, M.H.T.: Analysis of Production Schemata by Petri Nets. M. I. T., Laboratory for Computer Science, Masters Thesis, MAC TR-94, Feb. 1972.

12. Brinch Hansen, P.: Operating System Principles. N.J.: Prentice-Hall 1973.

13. Hewitt, C., Atkinson, R.: Synchronization in Actor Systems. In: 24th SIGPLAN-SIGACT Symposium on Principles of Programming Languages Jan. 1977, pp. 267-280 (1977).

14. Hoare, C.A.R.: Monitors: An Operating System Structuring Concept. Comm. ACM 17 10, pp. 549-557 (Oct. 1974).

15. Keller, R.M.: Denotational Models for Parallel Programs with Indeterminate Operators. Proc. IFIP Working Conf. on Formal Description of Programming

Concepts, August 1977.

16. Lauer, P.E., Campbell, R.H.: Formal Semantics of a Class of High Level Primitives for Coordinating Concurrent Processes. Acta Informatica 5, 4 pp. 297-332 (1975).

17. Lauer, P.E., Best, E., Shields, M.W.: On The Problem of Achieving Adequacy of Concurrent Programs. In: Proc. of IFIP Working Conf. on Formal Description of Programming Concepts. Saint Andrews, New Brunswick, Aug. 1977.

18. Margaris, A.: First Order Mathematical Logic. Mass.: Blaisdell Publishing Company, 1967.

19. McCarthy, J.: Towards a Mathematical Science of Computation. In: Information Processing 1962 (Proc. IFIP Congress 1962), pp. 21-28.

20. Owicki, S.S.: Axiomatic Proof Techniques for Parallel Programs. Cornell University, Ph.D. Thesis, TR75-251, July 1975.

21. Petri, C.A.: Kommunikation mit Automaten (Communication with Automata). Technische Hochschule, Ph.D. Thesis, Darmstadt, 1962.

22. Robinson, L., Holt, R.C.: Formal Specifications for Solutions to Synchronization Problems. Stanford Research Institute, Computer Science Group.

23. Scott, D.: Data Types as Lattices. SIAM Journal on Computing 5, 3 pp. 522 - 587 (Sept. 1976).