International Conference on
Computer System and Education, IISc, 1994

Maintaining Global Assertions on Distributed Systems *

Alexander I. Tomlinson Vijay K. Garg

Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, Texas 78712

Abstract

This paper develops a method for maintaining global assertions on a network of distributed processes. The global assertion has the form $(X_1^1 X_2^1 \cdots X_{N_1}^1) + \ldots + (X_1^M X_2^M \cdots X_{N_M}^M) \geq K$, where X_i^j is a variable which is local to one process in a distributed system and K is a constant. It is assumed that the initial values of all local variables are known and that the global assertion initially holds. This form is more general than the summation form considered in earlier work.

This research has applications in distributed software development, and as a general synchronization mechanism. Many classical synchronization problems (mutual exclusion, dining philosophers, readers/writers) can be solved with the results of this work.

1 Introduction

A *global assertion* is an expression whose value depends on the state of multiple processes in a distributed system.[1] Given an initially true global assertion,

$$\sum_{j=1}^{N}(X_1^j X_2^j \cdots X_{N_j}^j) \geq K$$

where X_i^j is a variable which is local to one process in a distributed system and K is a constant, the goal is to approve changes to the values of X_i^j such that the truth of the global assertion is invariant.

Note that although X_i^j is a variable, it can represent any function local to a process. For example, it might represent a function such as y^3, or $log(y)$; the assertion is concerned with the value of the function.

Without the ability to continuously monitor the global state, maintaining global assertions becomes a difficult problem. Given the global state, the assertion could be easily evaluated. However, it is impossible to

*Research supported in part by NSF Grant CCR 9110605, Navy Grant N00039-88-C-0082, TRW faculty assistantship award, IBM Agreement 153, and a Microelectronics Computer Development Fellowship.

[1] Message exchange is assumed to be ordered and reliable.

know the exact global state without halting the execution of all processes[3]. An alternative is to use consistent global states using a global snapshot algorithm[3]. However consistent global states are not sufficient for maintaining global assertions; the the assertion may be temporarily invalidated between snapshots.

Maintaining global assertions has applications in distributed software engineering as a general synchronization mechanism and as a debugging tool. The synchronization constraints for many classic problems such as producer/consumer, readers/writers, mutual exclusion, and dining philosophers can be characterized by a global assertion. The ability to maintain global assertions can simplify the development of distributed programs; as demonstrated in the following example.

Example 1 Suppose the US government has the power to set monetary exchange rates between the dollar and foreign currencies. Let C_m, C_f, C_y denote cost (in dollars) of the mark, franc and yen. Thus m marks is equivalent to mC_m dollars. The government is required to maintain a minimum investment of D dollars in foreign currency. The exchange rates are determined by different computers, and the total investment in any one foreign currency is distributed among several banks. For example one bank has m_1 marks, and another bank has m_2 marks. The following global assertion models this scenario.

$$m_1C_m+f_1C_f+y_1C_y+m_2C_m+f_2C_f+y_2C_y \geq D$$

Each quantity is a non-constant local variable: m_i, f_i, and y_i are maintained by the various banks, and C_m, C_f, and C_y are maintained by the federal government. The above assertion holds if and only if the total investment in foreign currency is at least D dollars. The ability to maintain the global assertion greatly simplifies the task of solving this problem.

■

The problem of maintaining global assertions is similar to managing global resources[1]. Consider the assertion $X_1 + X_2 + X_3 \geq 0$, where each X_i has arbitrary behavior, and initially $X_1+X_2+X_3 = 75$. In this example the total slack is 75. If the slack is treated as a resource and managed so that $slack \geq 0$, then the assertion holds. Increasing (decreasing) X_i corresponds to producing (consuming) resources.

However, there are significant differences between maintaining global assertions (MGA) and global resource management (GRM):

- The total amount of resources in GRM is bounded; this is not necessarily true in MGA.
- GRM assumes a user eventually produces as much as it consumes (user's *borrow* resources); MGA makes no such assumption (users *arbitrarily* produce and consume resources).
- GRM manages one type of resource; MGA manages two types (product slack and sum slack, which are convertible with each other).
- The goal of GRM is to resolve deadlock (through avoidance or recovery); the goal of MGA is to ensure that the assertion holds and to not introduce any possibility of deadlock that did not previously exist.[5]

One of the implications of these differences is the method of dealing with deadlock and fairness.

Since the variables have arbitrary behavior, there is no way to avoid deadlock. For example, if the total slack at a given instant is 10, and every process requests 20 units, a state of deadlock results. This scenario cannot be avoided due to the arbitrary behavior of the variables. Thus the policy with respect to deadlock is to not introduce any possibility of deadlock that did not previously exist.

Starvation cannot be avoided. Consider the assertion $X_1 + X_2 + X_3 \geq 0$, and suppose the system slack is currently 10. Suppose X_1 and X_2 make repeated requests to decrease their value by 20 and 10 respectively. Let X_3 be an adversary that increases its value by 10 if and only if X_2's request is satisfied. In this scenario X_1 will always starve regardless of the allocation policy. If the algorithm waits for enough slack to satisfy X_1, it will wait forever. If it satisfy X_2's request, then X_2 will request 10 more units and X_3 will produce 10 units, which results in the same initial scenario outlined above.

The term "SOP" is used to refer to the class of global assertions that can be expressed as a sum of products. This paper uses the SOP class of global assertions for several reasons.

First, it is more general than previous work [5, 6, 7] which only addresses global assertions of the form $\sum_{i=1}^{N} X_i \geq K$, (denoted by "SUM"). Second, there is a one to one correspondence between the set of polynomials and the assertions in the SOP class. Polynomial functions are widely used for analysis and modeling in scientific disciplines, hence the ability to represent them with a global assertion is valuable. Third, the SOP class is a superset of the product of sum (POS) class. And finally, boolean expressions can be transformed into global assertions whose variables take on values from the set $\{0, 1\}$. This can be done by a simple substitution of variables and operators.

Section 2 reviews related work such as distributed synchronization and global assertions. Section 3 presents most of the theoretical work of the paper. The concept of critical values and critical points are defined and subsequently used to decompose the global assertion into local assertions. It is proven that the conjunction of all the local assertions implies the global assertion. Methods for adjusting the local assertions are also presented in Section 3. Section 4 presents the algorithm and provides proof that it does maintain the global assertion. Section 5 discusses algorithm performance and alternative algorithms. Section 6 presents example applications and section 7 concludes the paper.

2 Related Work

Global assertions can be characterized by their expressive power. The most general form defines the global assertion's domain to be the set of global states. This allows any set of global states to be expressed as an assertion. These forms have been studied although no algorithms have been developed due to the complexity of the problem. Less general forms typically use process variables to define the assertion. The work of this paper falls into this category.

Carvalho and Roucairol [2] define a global assertion as a mapping from the set of global states to { TRUE, FALSE}. This allows any combination of global states to be characterized with a single global assertion. They decompose the global assertion into local and communication assertions using lattice theory and Galois connections. The authors acknowledge that finding the protocols necessary to maintain the communication assertions a much more difficult task, and only indicate some principles that can be used in their design. Although this paper use a less general global assertion, it allows the development of simple closed form expressions for assertion decomposition and the development of an algorithm.

Raynal [5, 6, 7] describe algorithms for implementing global assertions of the form $\sum_{i=1}^{N} \alpha_i x_i \leq k$ where $\alpha_i = \pm 1$, k is an integer constant, and each x_i is a local variable. Several variations of the algorithm are presented depending on the type of variables and the assumptions regarding communication (ie, reliability, ordering). The algorithms use upper and lower bounds on the local variables to maintain the assertion. This paper uses a more general form than theirs and avoids the problem of maintaining multiple copies of each upper and lower bound. The function of the bounds is replaced by critical values, which represent the aggregate effect of all the bounds.

Herman [4] presents a high level language abstraction for implementing global synchronization. The language consists of building expressions out of counter variables. The abstraction is equivalent in generality to maintaining assertions in a summation form where all variables are either monotonically increasing or decreasing.

In summary, previous work in global assertions has focused on algorithms for maintaining assertions of the form $\sum \alpha_i X_i \leq 0$, or on the decomposition of an assertion into local and communication assertions. The main contributions of this paper are:

1. To consider a global assertion more general than the summation form.
2. To develop and prove a decomposition into local assertions.
3. To develop and prove an algorithm for maintaining the global assertion.

3 Decomposing the Assertion

This section presents the decomposition of the global assertion into local assertions, provides closed form expressions for determining the local assertions, proves that the conjunction of all local assertions imply the global assertion, and presents methods for adjusting the local assertions to allow the system to evolve (ie, to reach states where the global assertion holds, but would not be reachable without adjusting the local assertions).

For notational convenience, define $P^j \triangleq \prod_{i=1}^{N} X_i^j$ and $S \triangleq \sum_{j=1}^{M} P^j$. The initial values of all local and derived variables are assumed to be known, and are denoted by $X0_i^j$, $P0^j$, and $S0$. It is assumed that the global assertion is initially true, i.e., $S0 \geq K$.

The global assertion is maintained by setting constraints on the values of X_i^j, denoted by `constraint`(X_i^j). If the constraints on all variables are respected, then the global assertion will be maintained.

The constraints on X_i^j are determined in two steps. The first step defines *product term constraints* for each P^j such that their conjunction implies $S \geq K$. The second step defines *local variable constraints* for each product term constraint such that the conjunction local variable constraints imply the product term constraint. The end result is that if all local variable constraints are maintained, the global assertion is satisfied.

3.1 Product Term Constraints

Consider the global assertion: $\sum_{j=1}^{M} P^j \geq K$. The global state space is an M dimensional space with coordinates $P^1, \ldots, P^M$. Figure 1 shows the state space for $P^1 + P^2 \geq 5$. Initially the global assertion holds, thus $P0^1 + P0^2 \geq 5$. Given the initial coordinates ($P0^1$,$P0^2$), the *critical point*, (PC^1, PC^2), is defined to be any point such that $\sum_{j=1}^{M} PC^j = K$ and $\forall j : PC^j \leq P0^j$. Figure 1 shows the initial point and a valid critical point.

The critical point is used to define constraints on the values of P^j. The constraint is defined to be: $P^j \geq PC^j$. The shaded area in figure 1 shows the states reachable under these constraints. If P^1 requested a new value less than PC^1, then it would have to agree with P^2 to exchange "resources". This corresponds to sliding the critical point along the line $P^1 + P^2 = K$ as shown in figure 1.

Formally, the constraint is defined in equation 2 and the critical point is defined as any point such that equation 1 holds and $\texttt{constraint}(P0^j)$ holds for all j.

$$\begin{aligned} \sum_{j=1}^{M} PC^j &= K & (1) \\ \texttt{constraint}(P^j) &\triangleq P^j \geq PC^j & (2) \end{aligned}$$

An example of a closed form solution for valid critical values is: $PC^j \triangleq P0^j - (S0 - K)/M$. Lemma 1 proves that if these constraints are maintained, then the global assertion is maintained. It is obvious from the definition of PC^j, that the constraints initially hold.

Lemma 1 $[\forall j : \texttt{constraint}(P^j)] \Rightarrow S \geq K$

Proof: By equation 2, $\forall j$: $\texttt{constraint}(P^j)$ is equivalent to $\forall j : P^j \geq PC^j$. This implies that $\sum_{j=1}^{M} P^j \geq \sum_{j=1}^{M} PC^j$. By definition of S, the left hand side equals S, and by equation 1, the right hand side equals K. Thus $S \geq K$. ∎

3.2 Local Variable Constraints

The product term constraints specify that $P^j \geq PC^j$. In terms of local variables, the constraints are: $\prod_{i=1}^{N} X_i^j \geq PC^j$. For each j, the constraints on the local variables, $\texttt{constraint}(X_i^j)$, must imply $\texttt{constraint}(P^j)$. Let the superscript j be understood for the remainder of this section.

The goal is to develop constraints on X_i^j such that

$$\left[\forall \mathrm{i} : \texttt{constraint}(\mathrm{X_i^j})\right] \Rightarrow \prod_{\mathrm{i=1}}^{\mathrm{N}} \mathrm{X_i} \geq \mathrm{PC^j} \quad (3)$$

The problem is broken into three cases according to the value of PC. Figure 2 shows the three cases ($PC < 0$, $PC = 0$, $PC > 0$) for $N = 2$. The shaded areas represent the valid state space (those which satisfy equation 3).

3.2.1 Case 1: $PC = 0$

The constraint is defined in equation 5 and the critical point is defined as any point such that equation 4 holds and $\texttt{constraint}(X0_i)$ holds for all i.

$$\begin{aligned} \prod_{i=1}^{N} XC_i &> 0 & (4) \\ \texttt{constraint}(X_i) &\triangleq X_i/XC_i \geq 0 & (5) \end{aligned}$$

Lemma 2 proves that these constraints ensure the product term constraint (equation 3) holds. Initially the constraints are satisfied; this is apparent from the definition of XC_i.

Lemma 2
$[\forall i : \texttt{constraint}(X_i)] \Rightarrow \prod_{i=1}^{N} X_i \geq PC$

Proof: If any $X_i = 0$, then $\prod_{i=1}^{N} X_i = 0$, and the lemma holds. Otherwise, the constraints (equation 5) imply that $\prod_{i=1}^{N} X_i$ has the same polarity as $\prod_{i=1}^{N} XC_i$. And since $\prod_{i=1}^{N} XC_i^j > 0$ (by equation 4), then $\prod_{i=1}^{N} X_i \geq PC = 0$. ∎

3.2.2 Case 2: $PC > 0$

The constraint is defined in equation 7 and the critical point is defined as any point such that equation 6 holds and $\texttt{constraint}(X0_i)$ holds for all i.

$$\begin{aligned} \prod_{i=1}^{N} XC_i &= PC & (6) \\ \texttt{constraint}(X_i) &\triangleq X_i/XC_i \geq 1 & (7) \end{aligned}$$

This definition is valid since $XC_i \neq 0$. This is due to the fact that, in this case, $\prod_{i=1}^{N} XC_i = PC \neq 0$. This defines the critical point to be a point on the border between the valid and invalid state spaces[2] such

[2] The border is a curve for $N = 2$, a surface for $N = 3$, etc.

that each coordinate XC_i is closer to the origin than the corresponding initial coordinate $X0_i$. One closed form solution for valid critical values is: $XC_i \triangleq X0_i(PC/P0)^{1/N}$.

Lemma 3 proves that these constraints ensure the product term constraint (equation 3) holds. From the definition of XC_i, it is obvious that initially the constraints hold.

Lemma 3
$[\forall i : \mathtt{constraint}(X_i)] \Rightarrow \prod_{i=1}^{N} X_i \geq PC$

Proof:
By equation 7, $\forall i$ $\mathtt{constraint}(X_i)$ is equivalent to $\forall i : X_i/XC_i \geq 1$. This implies $\forall i : |X_i| \geq |XC_i|$. Thus $\left|\prod_{i=1}^{N} X_i\right| \geq \left|\prod_{i=1}^{N} XC_i\right|$. By equation 6, and since $PC \geq 0$, $\left|\prod_{i=1}^{N} XC_i\right| = PC$. Thus $\left|\prod_{i=1}^{N} X_i\right| \geq PC$. Since $\forall i : X_i/XC_i \geq 1$, the polarity of $\prod_{i=1}^{N} X_i$ and $\prod_{i=1}^{N} XC_i$ must be the same. And since $\prod_{i=1}^{N} XC_i = PC > 0$, then $\prod_{i=1}^{N} X_i \geq 0$, which implies that $\left|\prod_{i=1}^{N} X_i\right| = \prod_{i=1}^{N} X_i$. Thus, $\prod_{i=1}^{N} X_i \geq PC$. ■

3.2.3 Case 3: $PC < 0$

This case is subdivided into two overlapping cases depending on the initial value $P0 = \prod_{i=1}^{N} X0_i$. The two cases are shown in figure 3.

Case 3A: $PC < 0$ **and** $P0 \geq 0$: The constraint is defined in equation 9 and the critical point is defined as any point such that equation 8 holds and the $\mathtt{constraint}(X0_i)$ holds for all i.

$$\prod_{i=1}^{N} XC_i > 0 \tag{8}$$

$$\mathtt{constraint}(X_i) \triangleq X_i/XC_i \geq 0 \tag{9}$$

These definitions are identical to that of case 1 ($PC = 0$). In this case, $P0 \geq 0$, thus lemma 2 proves that the constraints satisfy the product term constraint for this case too.

Case 3B: $PC < 0$ **and** $|P0| \leq |PC|$ **:** The constraint is defined in equation 11 and the critical point is defined as any point such that equation 10 holds and $\mathtt{constraint}(X0_i)$ holds for all i.

$$\left|\prod_{i=1}^{N} XC_i\right| = |PC| \tag{10}$$

$$\mathrm{constraint}(X_i) \triangleq |X_i/XC_i| \leq 1 \tag{11}$$

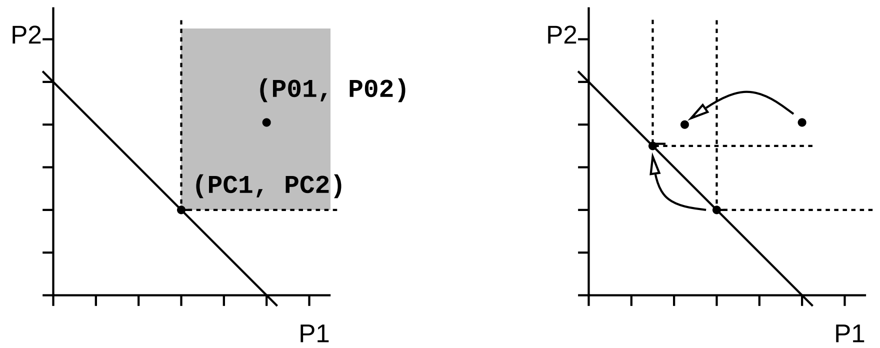
The critical point is a point on the border (or an image of the border) such that each $X0_i$ is closer to the origin than XC_i. A valid closed form solution is: $XC_i \triangleq |X0_i|\,|PC/P0|^{1/N}$.

Lemma 4 proves that the constraints imply the product term constraints. The proof is very straight forward and is left to the reader.

Lemma 4
$[\forall i : \mathtt{constraint}(X_i)] \Rightarrow \prod_{i=1}^{N} X_i \geq PC$

Table 1 summarizes the case analysis. Recall that case 3A is treated identically to case 1, therefore it appears in the table with case 1. The algorithm is based on this table and the "Mode" entry shows the term used to refer to the case in the algorithm. Note that in all three modes, the constraint is determined by the ratio X/XC, which is defined as the *slack* of variable X. The slack of X represents the amount of leeway X has before invalidating the constraint. The concept of slack is used in the algorithm.

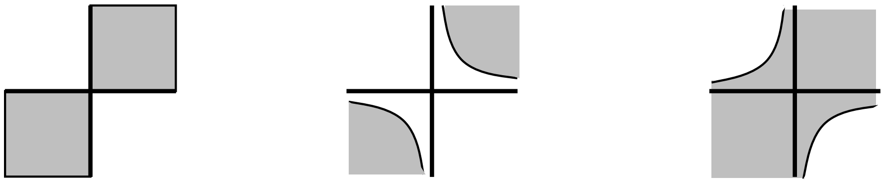
The constraints were defined to take a variable X and return a value TRUE or FALSE depending on the current mode and value of XC. From table 1 it is apparent that the constraint depends on the slack of X. Thus, at this point the constraint function is redefined to take the mode and slack as an argument and return TRUE or FALSE. The new definition is shown below:



Product Critical Point and the resulting restricted state space.

Adjusting Product Critical Point

Figure 1: Product Term Critical Values

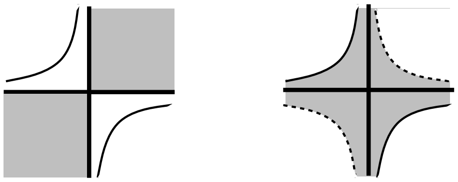


PC = 0

PC > 0

PC < 0

Figure 2: Three cases for local variable constraints.



Case 3A

Case 3B

Figure 3: Two cases for $PC < 0$, based on initial value $P0$

Case	Value of PC	Mode	Constraint
2	$PC > 0$	GTZ	$X/XC \geq 1$
3B	$PC < 0$	LTZ	$\lvert X/XC \rvert \leq 1$
1,3A	$PC = 0$	EQZ	$X/XC \geq 0$

Table 1: Summary of case analysis.

```
Boolean constraint(mode,$\alpha$)

    switch (mode) {
        case GTZ : return ( $\alpha \geq 1$ );
        case LTZ : return ( $|\alpha| \leq 1$ );
        case EQZ : return ( $\alpha \geq 0$ );
    }
```

3.3 Adjusting the Constraints

The constraints are built around critical values, which must be adjusted in order to allow the system to reach all valid states. Figure 1 shows a critical point being adjusted. It is apparent that if critical points could not be adjusted then much of the valid state space would be unreachable. In this section the procedure for adjusting the critical values is described.

Suppose X_i^j wants to change its value to $X_i^{j\prime}$. It is known that `constraint(`$Mode_j, X_i^j/XC_i^j$`)` holds since it holds initially and is invariant. If `constraint(`$Mode_j, X_i^{j\prime}/XC_i^j$`)` holds then the change can be made with no change to XC_i^j. If the constraint does not hold then XC_i^j must be changed to some value $\alpha_i^j XC_i^j$ such that `constraint(`$Mode_j, X_i^{j\prime}/(\alpha_i^j XC_i^j)$ `)` holds before the change can take place. In this situation the process in which X_i^j resides is blocked until an adjustment to the critical point $(XC_1^j, XC_2^j, \ldots, XC_N^j)$ can be made in which the new value of XC_i^j is $\alpha_i^j XC_i^j$. Let α_i^j be defined as $\alpha_i^j \triangleq X_i^{j\prime}/XC_i^j$.

Lemma 5 *If* $\alpha_i^j = X_i^{j\prime}/XC_i^j$, *then* `constraint(` $Mode_j, X_i^{j\prime}/(\alpha_i^j XC_i^j)$ `)` *holds regardless of* $Mode_j$.

Proof: By definition of α_i^j, $X_i^{j\prime}/(\alpha_i^j XC_i^j) = 1$. And also by definition, `constraint(`$Mode_j$`,1)` is true regardless of $Mode_j$. Hence the lemma is proven. ■

Since the local variables are being changed concurrently, there may exist several blocked processes waiting for an adjustment to the critical point $(XC_1^j, XC_2^j, \ldots, XC_N^j)$, which is initially assumed to both valid and respected. The value of the needed change to XC_i^j is indicated by α_i^j. If, after the adjustment, all critical points are still valid and respected, the the global assertion will continue to hold.

The procedures for adjusting the critical points such that they remain valid and respected are presented in the next two sections.

One way to adjust $(XC_1^j, XC_2^j, \ldots, XC_N^j)$ is to redistribute the slack among each local variable. Another way is to obtain slack from a different product term P^k, $k \neq j$. This is accomplished by an adjustment to the critical point $(PC^1, PC^2, \ldots PC^M)$.

3.3.1 Adjusting Critical Point: $(XC_1^j, XC_2^j, \ldots, XC_N^j)$

Adjusting this critical point corresponds to moving the critical point along the border[3] defined by $\prod_{i=1}^N XC_i^j = PC^j$ with no change to the value of PC^j. This redistributes the slack among the local variables in product term P^j. No other product terms are affected. If the critical point can be adjusted so that the product of the critical values remains constant, then the critical point remains valid. The following lemma proves this.

Lemma 6 *Given a valid critical point* $(XC_1^j, XC_2^j, \ldots, XC_N^j)$ *and* α_i^j, $1 \leq i \leq N$, *such that* $\prod_{i=1}^N \alpha_i^j = 1$,
then $(\alpha_1^j XC_1^j, \alpha_2^j XC_2^j, \ldots, \alpha_N^j XC_N^j)$ *is a valid critical point.*

Proof: By definition, the validity of a sum critical point depends only upon the product of its coordinates. Since $\prod_{i=1}^N \alpha_i^j = 1$, then

$$\prod_{i=1}^N XC_i^j = \prod_{i=1}^N \alpha_i^j \prod_{i=1}^N XC_i^j = \prod_{i=1}^N (\alpha_i^j XC_i^j)$$

[3] This visualization is not accurate for EQZ mode.

Therefore, $(\alpha_1^j XC_1^j, \alpha_2^j XC_2^j, \ldots, \alpha_N^j XC_N^j)$ is a valid critical point. ■

3.3.2 Adjusting Critical Point: $(PC^1, PC^2, \ldots PC^M)$

Adjusting this critical point transfers slack from one product term to another. If adjusting $(XC_1^j, XC_2^j, \ldots XC_N^j)$ failed to satisfy all requests in P^j, then slack can be transferred from another product term in an attempt to satisfy those requests. Only binary transfers are considered between product terms P^0 and P^j since the algorithm uses only that technique.

Let $X_0^j \triangleq 1$ for all j, and $P^0 \triangleq 0$. Then the global assertion may be rewritten as

$$P^0 + \sum_{j=1}^{M} \prod_{i=0}^{N} X_i^j \geq K$$

The variables X_0^j, P^0 have critical values XC_0^j, PC^0. They are used as intermediaries when transferring slack from one variable to another.

The algorithm assumes that $Mode_j$ is constant,[4] thus changes of mode will not be considered.

The following lemma demonstrates how to adjust the product critical point.

Lemma 7 *Assuming all sum and product critical points are initially valid and respected, then after executing the following sequence of commands, they will all remain valid and respected — assuming that $Mode_j$ remains constant.*

$$\begin{aligned}
\Delta &= P^0 - PC^0 \\
\alpha_0^j &= (PC^j - \Delta)/PC^j \\
PC^0 &= PC^0 + \Delta \\
PC^j &= PC^j - \Delta \\
XC_0^j &= \alpha_0^j XC_0^j
\end{aligned}$$

Proof: We need only to consider the critical points $(PC^0, PC^1, \ldots PC^M)$ and $(XC_0^j, XC_1^j, \ldots XC_N^j)$ since no others are affected. Let XC_i^j, P^j refer to the initial values and $XC_i^{j\prime}, P^{j\prime}$ refer values after execution of the commands. It is assumed that $Mode_j$ is constant, thus this sequence of commands cannot be executed if $Mode_j$ or $Mode_0$ is EQZ since it would cause the mode to change. Thus only modes GTZ and LTZ are considered. The proof is in four parts.

1) Product critical point remains valid if $\sum_{j=0}^{M} PC^{j\prime} = K$.

It is evident from the proposed sequence of commands that the sum of the coordinates of the product critical point remain unchanged. Thus $\sum_{j=0}^{M} PC^{j\prime} = \sum_{j=0}^{M} PC^j = K$.

2) Product critical point remains respected if $P^{0\prime} \geq PC^{0\prime}$ and $P^{j\prime} \geq PC^{j\prime}$.

It is obvious that $PC^{0\prime} = P^0$, and since $P^{0\prime} = P^0$, then $P^{0\prime} \geq PC^{0\prime}$. Since $P^0 \geq PC^0$ (initially respected), then $\Delta \geq 0$. Thus $PC^{j\prime} = PC^j - \Delta \leq PC^j$. And since $PC^j \leq P^j$ (initially respected), then $PC^{j\prime} \leq P^j$. And since $P^{j\prime} = P^j$, then $PC^{j\prime} \leq P^{j\prime}$.

3) Sum critical point remains valid if $\prod_{i=0}^{N} XC_i^{j\prime} = PC^{j\prime}$.

The critical point is initially valid, thus $\prod_{i=0}^{N} XC_i^j = PC^j$. By algebraic manipulation, $\prod_{i=0}^{N} XC_i^{j\prime} = \alpha_0^j \prod_{i=0}^{N} XC_i^j$ $= \alpha_0^j PC^j = ((PC^j - \Delta)/PC^j)PC^j = PC^j - \Delta = PC^{j\prime}$

4) Sum critical point remains respected if `constraint(` $Mode_j, X_0^{j\prime}/XC_0^{j\prime}$ `)` is true.

Recall that $\Delta \geq 0$ (from part 2).

Consider $Mode_j =$ GTZ, Since $Mode_j$ is constant, $\Delta \leq PC^j$. This fact, in conjunction with $\Delta \geq 0$ and $PC^j \geq$

[4] This assumption keeps the algorithm simple and efficient.

0, implies that $0 < \alpha_0^j \le 1$. Since `constraint`(GTZ,X_0^j/XC_0^j) holds (initially respected), then $X_0^j/XC_0^j \ge 1$. And since $0 < \alpha_0^j \le 1$, then $X_0^{j\prime}/XC_0^{j\prime} = X_0^j/(\alpha_0^j XC_0) \ge 1$. Hence `constraint`(GTZ, $X_0^{j\prime}/XC_0^{j\prime}$) holds.

Consider $Mode_j =$ LTZ, then $PC^j \le 0$ and $\Delta \ge 0$. This implies $\alpha_0^j \ge 1$. Since `constraint`(LTZ,X_0^j/XC_0^j) holds, then $\left|X_0^j/XC_0^j\right| \le 1$. Thus $1 \ge \left|X_0^j/XC_0^j\right| \ge \left|X_0^j/(\alpha_0^j XC_0^j)\right| = \left|X_0^{j\prime}/XC_0^{j\prime}\right| \le 1$.

Thus `constraint`(LTZ, $X_0^{j\prime}/XC_0^{j\prime}$) holds.

■

4 Algorithm

In this section we present an algorithm for the case when $Mode_j$ is constant for all j, $1 \le j \le M$. Recall that the slack of X_i^j is X_i^j/XC_i^j. Define the slack of P^j to be $P^j - PC^j$. The slack indicates how much leeway a variable has before invalidating its constraint.

There are two types of processes: Product Term Managers (PTM) and Local Variable Managers (LVM). Each LVM manages one local variable X_i^j, and each PTM manages one product term P^j. The communication structure is shown in figure 4. Each LVM has one *parent* (a PTM) and each PTM has multiple *children* (LVMs) who make up its product term. These terms are used in presenting the algorithm. When PTM is referred to in the context of a specific LVM, then the PTM referred to is the parent PTM. The subscripts and superscripts will be omitted for most of this section except to avoid ambiguity.

LVMs communicate with their parent PTM through direct message exchange. PTMs communicate with other PTMs via a token. The function of an LVM is to approve requests for changing the value of its local variable. An application process requests to change X to X' with the function call `change`(X, X'). The local assertion is invariant outside of this function. This ensures that the applications view of X is one which satisfies the local assertion. The local assertion is assumed to hold upon entry to `change` and guaranteed to hold upon return.

When `change`(X, X') is invoked, LVM evaluates the local assertion with the new value. If it holds then the change is made and control is returned to the application. If it does not hold then a message is sent to PTM *requesting* an adjustment to XC, and the process is blocked until PTM acknowledges the request. The acknowledgment indicates that the adjustment has been approved.

There is one omission in the above description: before LVM returns control to the application, excess slack is *donated* to PTM. This simplifies the algorithm by avoiding the need for the PTM to explicitly collect excess slack when servicing requests. In fact donations and requests are lumped into one message: < "SLACK", *id*, α >, where *id* identifies the sender and $\alpha = X'/XC$. The meaning depends on the value of α. If α satisfies the local assertion then it is a donation, otherwise it is a request. The code for LVM is shown below.

```
LVM: Change($X, X'$)
    $XC$ : critical value, initially $X0$;
    $id$ : unique process identifier;
    $\alpha = X'/XC$;
    send < "SLACK", $id$, $\alpha$ > to parent PTM;
    if not constraint($Mode, \alpha$)
        then Wait for < "ACK" >;
    $XC = \alpha XC$;
    $X = X'$;
```

The PTM accepts "SLACK" messages from its children and redistributes it as necessary. The redistribution takes the form of adjustments to $(XC_0, XC_1, \ldots XC_N)$ and

$(PC^0, PC^1, \ldots PC^M)$. Adjustments to $(XC_0, XC_1, \ldots XC_N)$ are implemented directly by PTM. Adjustments to $(PC^0, PC^1, \ldots PC^M)$ are accomplished by transferring slack between PTMs via the token.

Each PTM has a local variable, X_0, with a constant value 1. This variable appears in the global assertion as follows:

$$\sum_{j=1}^{M} X_0^j \prod_{i=1}^{N} X_i^j = \sum_{j=1}^{M} \prod_{i=0}^{N} X_i^j \geq K$$

Note that the truth of the assertion is not affected. As with any other local variable, X_0 has a critical value XC_0. X_0 is used as an intermediary when transferring slack from one local variable to another.

Analogous to XC_0, the token has a critical point PC^0 which carries slack between the product terms. The corresponding product term is $P^0 \triangleq 0$. PC^0 must follow the same set rules that other PC^j critical values follow. Initially, all the slack in the system $(S0 - K)$ is given to PC^0.

An array α_i, $0 \leq i \leq N$ is maintained which indicates the amount of slack requested or donated by child i (which contains variable X_i). If $\alpha_i = 1$, then no request or donation from X_i is outstanding.

PTM: Data Structures

```
$\alpha_i, {}_{0 \leq i \leq N}$ : slack from $X_i$;
$PC$ : product term critical value, initially $P0$;
$X_0$ : local variable, constant value 1;
$XC_0$ : critical value, initially $X_0$;
```

When PTM receives slack from one of its children, it records the event in its slack array and calls `AdjustXC`.

PTM: To rcv < "SLACK", i, α > from child

```
$\alpha_i = \alpha$;
AdjustXC();
```

PTM: AdjustXC()

```
while ( $\exists\, I \subseteq \{1, 2, \ldots N\}$ such that
        $(\forall i \in I : \alpha_i \neq 1) \;\wedge\; (I \neq \emptyset) \;\wedge$
        constraint$(\mathit{Mode}, (X_0/XC_0) \prod_{i \in I} \alpha_i)$ )
begin
    $\alpha_0 = 1/(\prod_{i \in I} \alpha_i)$
    $XC_0 = \alpha_0 XC_0$
    for each $i \in I$
    begin
        if not constraint$(\mathit{Mode}, \alpha_i) then$
            begin
                Send < "ACK" > to child LVM $i$
            endif
        $\alpha_i = 1$
    endfor
endwile
```

The algorithm for `adjust()` is a direct implementation of lemma 6. If β_i is defined as follows:

$$\beta_i = \begin{cases} \alpha_i & i \in I \\ \alpha_0 & i = 0 \\ 1 & \text{otherwise} \end{cases}$$

then $\prod_{i=1}^{N} \beta_i = \alpha_0 \prod_{i \in I} \alpha_i = 1$. Thus by lemma 6, the new critical point, $(\beta_0 XC_0, \beta_1 XC_1, \ldots \beta_N XC_N)$, is valid.

The new critical point is respected if `constraint`$(\mathit{Mode}, X_i'/(\alpha_i XC_i))$ is true for all $i \in I$ and for $i = 0$, where X_i' is the new value of X_i. Consider $i \in I$: By definition, $\alpha_i = X_i'/XC_i$. Thus, as a result of lemma 5, `constraint(`$\mathit{Mode}, X_i'/(\alpha_i XC_i)$`)` holds. Consider $i = 0$: from the code it can be seen that $\alpha_0 = 1/\prod_{i \in I} \alpha_i$. Thus

$$(\mathrm{X_0}/\mathrm{XC_0}) \prod_{\mathrm{i \in I}} \alpha_{\mathrm{i}} = \mathrm{X_0}/(\alpha_0 \mathrm{XC_0}) = \mathrm{X}_0'/(\alpha_0 \mathrm{XC_0})$$

The left hand side of the above equation is the argument to `constraint()` in the test of the `while` loop in the algorithm. Thus the value of `constraint(`$\mathit{Mode}, X_0'/(\alpha_0 XC_0)$`)` is true. Therefore the new critical point is respected. And since it is also valid (as shown above), the global assertion holds.

If adjusting $(XC^j_1, XC^j_2, \dots XC^j_N)$, can not satisfy all requests, then slack may be transferred from another product term P^k. The token carries slack between PTMs, allowing each PTM to use the slack it needs to satisfy its requests. The token represents a predefined product term, P^0, where $P^0 \triangleq 0$. Associated with P^0 is a critical value PC^0.

When the token visits the PTM representing product term P^j, its slack is transferred from PC^0 to XC^j_0 in the procedure `Transfer_From_Token` After the transfer a call to `AdjustXC` is made, the left over slack is transferred back to the token with a call to `Transfer_To_Token`

```
PTM: To rcv < "TOKEN", $P^0$, $PC^0$ >

    Transfer_From_Token($P^0$, $PC^0$);
    AdjustXC();
    Transfer_To_Token($P^0$, $PC^0$);
    Pass Token to next PTM;
```

```
PTM:Transfer_From_Token($P^0$, $PC^0$)

    $\Delta = P^0 - PC^0$
    $\alpha = (PC^j - \Delta)/PC^j$
    $PC^0 = PC^0 + \Delta$
    $PC^j = PC^j - \Delta$
    $XC_0 = \alpha XC_0$
```

```
PTM:Transfer_To_Token($P^0$, $PC^0$)

    $\alpha = X_0/XC_0$
    $\Delta = PC^j(1 - \alpha)$
    $PC^0 = PC^0 - \Delta$
    $PC^j = PC^j + \Delta$
    $XC_0 = \alpha XC_0$
```

The text of the procedure `Transfer_From_Token` matches exactly the code that appears in lemma 7. Thus, by lemma 7 `Transfer_From_Token` maintains the validity and respectedness of all critical points. The same is true of `Transfer_To_Token`; since it is the inverse of `Transfer_From_Token`, it merely undoes the effect of `Transfer_To_Token`. It has already been shown that `AdjustXC` maintains the validity and respectedness of all critical points.

It has been shown that all procedures used in this algorithm maintain the validity and respectedness of all critical points. In addition, we have shown that if all critical points are valid and respected, then the global assertion holds. Therefore the algorithm presented here maintains the global assertion.

5 Algorithm Analysis

In this section different algorithms for maintaining the global assertion are analyzed in terms performance. The approaches can be broadly categorized by their software structure: centralized, client/server, or token ring. Centralized algorithms are not considered because of the obvious bottleneck that would result.

There are two kinds of slack: product slack and sum slack. There are three levels of slack management: local, product term, and global. The local managers and product term managers exchange product slack, while the product term managers and the global manager exchange sum slack.

The goal is to distribute both kinds of slack throughout the system in order to satisfy consume operations. The slack can be distributed according to one of two policies: demand driven or supply driven. In demand driven distribution, slack remains where it is produced until it is demanded by a consumer. In supply driven distribution, slack is given to the manager when it is produced, and consumers request slack from the manager. The distribution mechanisms considered are: token ring, and client/server.

Note that the management of the two types of slack need not use the same techniques. Sum slack is used (produced and consumed) by the PTMs and managed by the

GM, and product slack is used by the LMs and managed by the PTMs. The algorithm presented in this paper uses a supply driven client/server for product slack, and a supply driven token ring for sum slack.

5.1 Comparison of Different Approaches

The approaches are compared on the basis of the worst case cost and delay per operation. An *operation* is a consume or produce event. The *cost* is defined to be the number of messages per operation. The *delay* is defined to be the maximum length sequence of causally related messages. The unit of cost is MSG, and the unit of delay is HOP. For example, a request/acknowledge exchange has a cost of 2 MSGS and a delay of 2 HOPS. A broadcast to n processes followed by acknowledgments has a cost of $2n$ MSGS and a delay of 2 HOPS.

For slack consumption, it is assumed that the amount of slack distributed throughout the system is enough to satisfy the request. The performance is given in terms of n, which is defined (in this section only) to be the number of slack users. The delay of produce operations are always zero since a producer of slack never blocks.

5.1.1 Client/Server

In this approach the slack manager is the server and the users are the clients.

With a supply driven policy, when slack is produced it is immediately forwarded to the manager (server). This takes one message, thus has a cost of 1 MSG. When slack is consumed, the consumer must request slack from the manager and the manager replies when the requested amount is available. Thus a consume event costs 2 MSGS and incurs a delay of 2 HOPS.

With a demand driven policy, slack remains where it was produced until it is demanded by another process. Thus no messages are required and the cost is zero. A consume event can cost up to $2n$ MSGS and have a delay of 4 HOPS. This situation arises when the consumer sends a request to the manager, who in turn broadcasts to the other users, collects the slack, and forwards it to the consumer.

5.1.2 Token Ring

Only the delay is determined in this case. Cost can only be analyzed in terms of average cost since the token circulates independently of operations. However, the average cost depends ratio between the number of operations and the number of token circulations, which is an unknown quantity.

With a supply driven policy, when slack is produced it is given to the token which carries it to other users that may need it. A consume operation can have a delay of n HOPS, which occurs when the user requests slack immediately after the token was forwarded. The token must circulate once, visiting all n users, before returning to the consumer.

With a demand driven policy, In this approach the token carries *requests* for slack. The slack itself remains at the site of production until a request on the token needs it. A consume operation has a worst case delay of $2n$ HOPS. This occurs when the request is made just after the token leaves. It must travel around the ring once before the consumer gets it, then the consumer places it's request on the token and it travels around the ring again.

5.2 Overall Performance

It is apparent that the supply driven client/server approach has the best local performance, ie, performance on one level of slack management only. This approach is used for product slack management, however sum slack is managed with the supply driven token ring approach. The reasoning for this is based on the global performance of the algorithm.

Given that many requests will likely be satisfied via exchange of product slack without having to resort to exchange of sum slack, it makes sense to use the most efficient algorithm for product slack management. Hence, the supply driven client/server approach is used for managing product slack. The alternatives for managing sum slack are considered next.

With a supply driven client/server the overall algorithm would in effect be centralized. Each user would forward product slack to the product term manager, which would then forward it to the global manager. Thus all slack would be distributed through a single point, the global manager. This design would create a bottleneck at the global manager and therefore it is not used.

With a demand driven client/server a produce would have cost/delay of 1/0 and a consume would have cost/delay of $2 + 2m/6$, where m is the number of product terms. With a supply driven token ring a produce would have cost/delay of */0 and a consume would have cost/delay of $0/2 + m$. With a demand driven token ring a produce would have cost/delay of */0 and a consume would have cost/delay of $0/2 + 2m$.

These values are obtained by adding the appropriate entries from table 2. It is easy to see that the supply driven token ring offers the best performance, and that is the approach used for management of sum slack.

6 Example Applications

The problem of mutual exclusion between M processes can be solved with the assertion $\sum_{j=1}^{M} X_i \geq -1$, where initially $X_i = 0$ for all i. The entry and exit protocols would be requests to change X_i to -1 or 0. Thus the users code for process i would appear as follows:

```
change($X_i$,-1);
*program is in critical section*
change($X_i$,0);
```

A call to change(X_i,-1) will block if another process is already in the critical region. When the call returns, then entry has been granted. Upon exiting the critical region a process calls change(X_i,0) which will never block because it is producing slack.

One solution to "five dining philosophers" is limiting the number of eating philosophers to four, and implementing mutual exclusion on the forks. We can define multiple global assertions: one for the number of eaters, and one each for the forks. The assertion $\sum_{j=1}^{5} \texttt{eat}_i \geq -4$ restricts the number of eaters; and $\forall i$: $\texttt{left\_fork}_i + \texttt{right\_fork}_{(i+1)mod4} \geq -1$ implements mutual exclusion on the forks. The code for philosopher i is:

```
change(eat$_i$,-1);
change(left_fork$_i$,-1);
change(right_fork$_i$,-1);
eat();
change(right_fork$_i$,0);
change(left_fork$_i$,0);
change(eat$_i$,0);
```

7 Conclusion

A global assertion in a sum of products form is more general than the summation form used in earlier work. This paper developed a decomposition of the global assertion and an algorithm for maintaining the global assertion based on the decomposition. Proofs of the correctness of the decomposition and the algorithm were presented.

The global assertion was decomposed into a set of local assertions, and it was proven that the conjunction of these local assertions imply that the global assertion holds. The decomposition was based on the concept of critical points. The notions of valid and respected critical points were developed and it was shown that if all critical points are both valid and respected, then the global assertion holds. This result was used in the proof of the correctness of the algorithm. Closed form expressions were developed for generating initial critical points that are valid and respected.

This work has applications as a general synchronization mechanism, and in distributed debugging. Many classical synchronization problems can be trivially written as a global assertion. Mutual exclusion was given as an example. The family of Boolean logic equations can also be expressed as a special case of the global assertions.

The algorithm presented is valid for all global assertions, however it does not allow the system to evolve into all valid states due to the assumption that the "mode" is constant.

The algorithm presented here has been implemented. Future work includes looking into fairness and deadlock issues, which demand consideration from a novel view point. An interesting algorithmic issue is determining the optimal distribution of slack. Ideally it would be best to keep the slack more distributed than in the algorithm. This greatly complicates the algorithm, causes higher message complexity, and makes fairness issues even harder to work out. Future work also includes developing a complementary algorithm for detecting global predicates of the same form.

References

[1] Y. Afek, B. Awerbuch, S. A. Plotkin, and M. Saks. Local management of a global resource in a communication network. In *Proc. of the Conference on the Foundations of Computer Science*, pages 347–357. IEEE, August 1987.

[2] O. Carvalho and G. Roucairol. On the distribution of an assertion. In *Principles of Distributed Computing*, pages 121–131, 1982.

[3] K.M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[4] D. Herman. *Distributed Computing Systems*, page 51. Academic Press, 1983. (Parker-Verjus Ed.).

[5] M. Raynal. *Networks and Distributed Computation: Concepts, Tools and Algorithms.* The MIT Press Series in Computer Systems, Cambridge, MA, 1988.

[6] M. Raynal and J.M. Helary. *Synchronization and Control of Distributed Systems and Programs.* John Wiley and Sons Ltd, Chichester, England, 1990.

[7] M. Raynal. *Distributed Algorithms and Protocols.* John Wiley and Sons Ltd, Chichester, England, 1988.

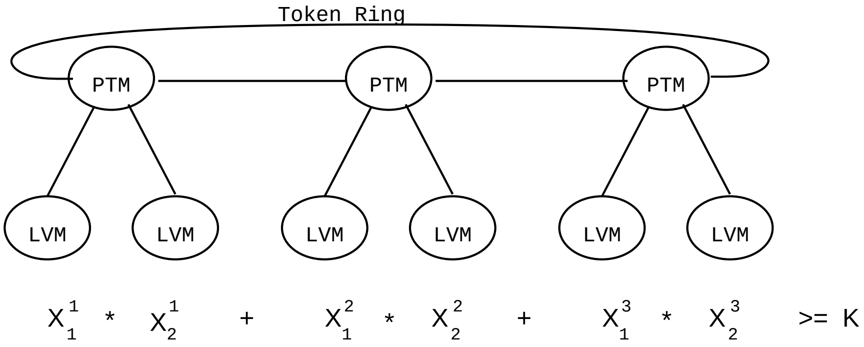


Figure 4: Communication structure of the algorithm.

		Produce		Consume	
		cost	delay	cost	delay
client/server	supply driven	1	0	2	2
	demand driven	0	0	2n	4
token ring	supply driven	*	0	*	n
	demand driven	*	0	*	2n

Table 2: Performance summary of algorithm alternatives with n users.