Event-based Composition of Concurrent Programs[⋆]

Raju Pandey James C. Browne

Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712
{raju, browne}@cs.utexas.edu

Abstract. This paper presents a model for concurrent programming, where programs (concurrent program structures) are represented as composition expressions over component programs with suitably defined operators. In this model of programming, called Composition bY Event Specification (C-YES), a composition of programs specifies that all events (individual occurrences of named operations, called actions) of component programs can execute in parallel, except for a set of events that interact with each other. Interactions among such events can be specified by establishing execution orderings among them. This paper presents a mechanism, where such interactions are specified by constructing algebraic expressions from a set of primitive interaction expressions and interaction operators. The primitive expressions model interactions at the fundamental level of computations, namely, events. The interaction operators model nondeterministic interactions and interaction over sets of events.
A mechanism for the representation of concurrent programs, called *interacting blocks*, is also given. The interface of an interacting block contains, in addition to the conventional parameters, references to events that may interact with events of other interacting blocks. The implementation of an interacting block contains definitions of its computations and mappings between internal events, and references to these events. Examples illustrating the properties of the C-YES model are presented in the paper.

1 Introduction

A concurrent program is a composition of a set of component programs. The components represent certain aspects of a problem that the composite program models. During an execution of the composite program, the components execute in parallel and contribute to the overall result by solving sub-problems. They cooperate and share information and resources with each other. Such cooperation, called *interaction*, among the component programs arises due to semantic

[⋆] This work was supported by Texas Advanced Research Program under grant 003658-445.

dependencies such as data dependency, resource consistency, resource fairness, etc., among certain operations of the component programs. The dependencies impose constraints on how such operations can be executed. A correct execution of a concurrent program is one where executions of these operations satisfy any semantic constraints associated with the interactions of the component programs.

Concurrent programming is difficult because asynchronous and concurrent executions of component programs introduce nondeterminism in the way interacting operations can affect each other. It is possible that semantic constraints associated with interactions are not preserved over different executions of a concurrent program. One of the outcomes is that different executions may produce different results. Mechanisms are needed to ensure that semantic constraints are preserved during the executions of the component programs. Such mechanisms, called *interaction specification mechanisms*, are very important aspects of a concurrent programming model. They influence how programs are designed, implemented, and reasoned with (during verification and debugging).

Many models for concurrent programming have been proposed [And91]. In most of these models, the components of a concurrent program contain two kinds of operations: i) simple operations denoting certain computations, and ii) synchronization operations of the form

```
if <condition> <operation> else <delay>
```

The execution of a synchronization operation delays the executing program if the `<condition>` is not true. The program starts to execute again once the `<condition>` becomes true. A concurrent program execution, thus, consists of parallel executions of the component programs, where each component repeatedly executes a simple operation or a synchronization operation. Interaction among the components is realized by a set of synchronization operations. The following characterize the interaction specification mechanisms in these models:

– Indirect: Semantic dependencies among operations are specified indirectly through a set of synchronization operations. Conditions in synchronization operations determine if a component program might violate any semantic constraints by executing a synchronization operation. If it does so, the program is delayed until other component programs execute a set of operations in order to ensure that the delayed program may proceed to execute the operation without violating the semantic constraints. Programs, therefore, affect each other *indirectly* by accessing and modifying elements that affect the validity of synchronization conditions.
– Operational: Synchronization operations are embedded within the definitions of component programs. One must reason with the logic associated with the implementation of a program in order to determine how synchronization operations are used in the program and how they affect the executions of other programs.
– Noncompositional: The interaction specification mechanisms identify a set of synchronization operations and their semantics. All interactions must

only be realized through these synchronization operations only. There is no separate mechanism for combining the synchronization operations order to construct more powerful and abstract synchronization operations. Most languages use sequential compositional mechanisms (such as if–then–else, procedure or function composition operators, etc.,) to combine synchronization operations.

All of these characteristics make it difficult to design, implement, and reason with concurrent programs.

This paper presents a model of concurrent programming, called Composition bY Event Specification (C-YES). The C-YES model is based on the observation that concurrency and interaction represent specific kinds of semantic relationships among operations of programs. The relationships underline semantic independence or dependence among these operations. Semantically independent operations can execute in parallel. Executions of dependent operations should occur in an order that preserves any semantic constraints associated with the interaction.

The C-YES model identifies individual occurrences of operations, called *events*, as the basis for specifying concurrency and interaction relationships. Concurrent events are not ordered and, therefore, can execute in parallel. Our interaction specification mechanism identifies the interaction between two events as the fundamental interaction. It models such an interaction by specifying an execution ordering between the two events. The interaction specification mechanism represents other interactions by combining fundamental interactions among different events with suitable operators, called *interaction operators*. The interaction operators model nondeterministic interactions as well as interactions over sets of events. The following are the characteristics of the interaction specification mechanism:

- Interactions are specified directly and precisely by establishing ordering relations among events.
- The interaction specification mechanism is compositional: it defines mechanisms for specifying fundamental interactions, and for combining interaction expressions in order to represent more powerful and abstract interactions.
- The interaction specification mechanism is algebraic. It is possible to reason with interaction expressions in order to determine how programs affect each other.
- The interaction specification mechanism captures only the abstractions associated with interactions. The primitives and the operators do not depend on the semantics of their operands (events); it is, therefore, possible to use the interaction specification mechanism for both shared and distributed memory programming models.

This paper also presents a mechanism, called an *interacting block*, for representing concurrent programs. A program represented using this mechanism contains two elements: the first is the specification of its computations, and the second is the identification of certain execution points, called *interaction* *points*, where the program may interact with other programs. Interaction points are abstractions of execution points of a program: they capture those execution points that may have semantic relationships with the interaction points of other programs. A concurrent program, therefore, is defined by specifying its component programs and an interaction expression. The latter determines execution orderings among the interaction points of the component programs.

Our model separates interaction specifications from program specifications. It is, therefore, possible to reuse a program definition in many concurrent program compositions. Also, the model separates how interaction points are used and how they are defined. This allows one to localize changes in either program implementations or interaction behaviors without affecting each other in certain cases.

The paper is organized as follows: Section 2 contains definitions of basic terms, and their notations. In Section 3, we present the C-YES model. It first describes how programs can be composed and how their interactions can be specified. The representation mechanism of concurrent programs is then presented. We develop the classical producer and consumer problem, and a problem from numerical analysis (Gaussian elimination algorithm) to illustrate the C-YES model. We briefly survey the different synchronization mechanisms in section 4. Section 5 contains final remarks and ongoing work.

2 Basic Definitions

We define two sets of terms we used in this paper. The first denotes terms that are used while specifying the representation of programs, whereas the second contains terms used during the execution of programs.

2.1 Structures

Definition 2.1 (Program) *A program is a composition of operations and other programs.* $\triangleleft$

Definition 2.2 (Action) *An action labels operations that will be performed during the execution of a program.* $\triangleleft$

Actions represent the alphabet that a programmer can use to compose a program. At a certain level of abstraction, the notions of program and action coincide. We will not distinguish between the two and will use them interchangeably.

2.2 Executions

Definition 2.3 (Computation) *A computation denotes a specific execution of a program.* $\triangleleft$

Each program has a set of computations associated with it.

Notation: Let terms $\xi_s(C)$ and $\xi(C)$ respectively denote the set of computations and some computation of program C.

Definition 2.4 (Event) *An event is a specific occurrence of an action during a computation.* $\triangleleft$

The relationship between actions and events can be described by the following program:

```
for (i = 0; i < 5; i = i+1)
        sum();
```

In the above, `sum()` denotes an action. It is executed five times when the above program is executed. Each execution of `sum()` denotes a unique event occurrence during the above computation.

Definition 2.5 (Event Ordering) *The event ordering relation $\otimes$ between two events is an asymmetric, nonreflexive, and transitive relation. The relation $e_1 \otimes e_2$ specifies that e_1 occurs before e_2.* $\triangleleft$

Relation $\otimes$ models the execution ordering between events.

Definition 2.6 (Concurrent Event) *Events e_1 and e_2 are concurrent if there is no ordering relation between the two events.* $\triangleleft$

Notation: We use $(e_1 \parallel_e e_2)$ to denote concurrency between events e_1 and e_2. Concurrency is modeled by the lack of any execution orderings.

A concurrent computation is a set of events such that some of the events are not ordered with respect to each other, and hence, can be executed in parallel. We will use two mathematical entities, pomsets [Pra86] and event dependency graphs, to represent a concurrent computation.

Definition 2.7 (Pomset) *A labeled partial order is a 4-tuple $(V_c, \Sigma, \otimes, \mu)$ consisting of* i) *a set V_c of events,* ii) *an alphabet Σ of actions,* iii) *a partial order $\otimes$ on set V_c, and* iv) *a labeling function $\mu : V_c \rightarrow \Sigma$ assigning symbols to events, each labeled event representing an occurrence of the action labeling it.* $\triangleleft$

Notation: We will write $(V_c, \otimes)$ for pomsets when Σ and μ can be derived from the context.

Notation: We will use $e \in V_c$ to mean that e is an event in computation $(V_c, \otimes)$.

Definition 2.8 (Event Dependency Graph) *An event dependency graph $G = (V_c, E_c)$ is a directed graph such that i) V_c is a set of events, and ii) $(e_1, e_2) \in E_c$ iff $e_1 \otimes e_2$.* $\triangleleft$

3 C-YES Model

3.1 Model of Concurrent Programming

In most conventional concurrent programming models, a concurrent program is composed from a set of component programs, each containing a set of both simple as well as synchronization operations. During the execution of the composite program, the components form separate threads of executions, and execute in parallel. A thread of execution is delayed if it attempts to execute a synchronization operation and the condition associated with the operation is not true. It remains blocked until the condition becomes true. A correct execution of the composite program is one where the executions of the components satisfy all semantic constraints associated with the interactions of the components.

We take an alternate, and direct, view of the composition of concurrent programs: here, the role of a concurrent composition mechanism is to establish two kinds of relationships among operations of programs. The first, called *concurrency*, represents semantic independence among these operations. Such operations can execute in parallel. The second, called *interaction*, underlines semantic dependencies among operations. Semantic dependencies arise because an operation may rely on information produced by another operation (data dependency), operations must access a resource in an ordered manner (resource consistency and fairness), or operations must satisfy other application specific semantic considerations. Unlike existing programming models, where interaction and concurrency are represented indirectly through the executions of concurrent threads and synchronization constraints, our compositional model represents the two relationships directly.

Our model is based on the Pomset model [Pra86] in that concurrency and interaction among operations are represented by specifying the presence or absence of execution orderings among the operations. Concurrency is specified by the lack of execution orderings, whereas interaction is represented by specifying the order in which interacting operations must be executed. For instance, if an operation e_1 utilizes information produced by an operation e_2, their interaction is specified by the relation

$$e_2 \olessthan e_1$$

The relation states that e_1 executes only after e_2 has terminated. Other semantic dependencies such as consistency and fairness can also be represented by establishing proper execution orderings among operations.

3.2 Event-based Concurrent Program Composition

We now present our concurrent composition mechanism. We choose "event" as the basis for specifying concurrency and interaction relations: concurrency is specified by the lack of execution orderings among events, whereas interaction is represented by specifying constraints on the execution orderings of interacting events. The composition mechanism is defined below:

Definition 3.1 (Constrained Concurrent Composition) *The constrained concurrent composition*

$$(C_1 \parallel C_2 \parallel \ldots \parallel C_n) \textit{ where } \mathcal{E}$$

of programs C_1, C_2, ..., C_n denotes a program C such that during an execution $\xi(C)$ of C,

$$\begin{aligned}&\langle\, \forall\, i,j : i,j \in \{1 \ldots n\} \wedge i \neq j : \\ &\quad \langle\, \forall\, e_k, e_l : e_k \in \xi(C_i) \wedge e_l \in \xi(C_j) \wedge \neg(Ordered(\mathcal{E}, e_k, e_l)) : \\ &\qquad\qquad e_k \parallel_e e_l \rangle\rangle\end{aligned}$$

where $Ordered(\mathcal{E}, e_k, e_l)$ is true if the event ordering constraint expression $\mathcal{E}$ specifies an ordering relation between e_k and e_l. Computations $\xi(C_i)$ and $\xi(C_j)$ respectively are specific executions of program C_i and C_j and occur during the computation $\xi(C)$. $\triangleleft$

Intuitively, the above composition expression defines a program C such that during its execution, all events in the computations of C_1, C_2, ..., and C_n occur in parallel with respect to each other, except for a small set of events (compared to the number of events in the composite program). Events in this set must be executed in an order that satisfies the ordering constraints imposed by the event ordering constraint expression $\mathcal{E}$. The focus of a composition, therefore, is on the identification of interacting events, and on the derivation of an event ordering constraint expression.

3.2.1 Interaction specification. Interaction among programs in our model is specified by an algebraic expression, called *event ordering constraint expression*. It is used to represent semantic dependencies among the events of component programs by specifying execution orderings — deterministic or nondeterministic — among the events. An event ordering constraint expression (eoce) is constructed from a set of *primitive ordering constraint expressions* and a set of *interaction operators*. A primitive ordering constraint expression captures the interaction between two events, whereas the interaction operators are used to represent nondeterministic interactions as well as interactions among sets of events.

Syntax: The BNF expression for generating an event ordering constraint expression (eoce) is shown in figure 1.

In the BNF definition, the term `<event_id>` denotes a unique event. Since most programming languages do not define any syntactic mechanisms for differentiating the different occurrences of actions, we use the notation `(<ActionId>, <Selector>)` to denote events in this paper. Here `<ActionId>` denotes an action, and `<Selector>` is one of the following:

– Occurrence number: We associate a number with each occurrence of an action, starting with the number 1. A unique event can be selected by identifying its occurrence number.

<eoce> ::= event_id $\olessthan$ event_id
 | <eoce> $\ovee$ <eoce>
 | <eoce> $\owedge$ <eoce>
 | forall <iterator> in <set>
  <eoce>
 | exists <iterator> in <set>
  <eoce>
 | (<eoce>)

Fig. 1. BNF expression for event ordering constraint expressions

– Boolean condition: The boolean condition selector specifies a condition that must be true during the execution of an action. Note that it is the programmer's responsibility to specify a boolean condition that uniquely identifies an event.

The term `<iterator>` in figure 1 is a mechanism used to iterate over a set denoted by the term `<set>`.

Semantics: We develop the semantics associated with event ordering constraint expressions by defining two terms, *Ordering Constraint Set* and *Satisfy.*

Definition 3.2 (Ordering Constraint Set) *An ordering constraint set S is a set of sets of ordered pairs (a, b) such that a and b are events and $a \olessthan b$.* ◁

Notation: We use $\mathcal{O}(E)$ to denote the ordering constraint set associated with the event ordering constraint expression E.

Definition 3.3 (Sat) *For a computation $(V, \olessthan_v)$ and an event ordering constraint expression E, the term $Sat((V, \olessthan_v), E)$ is true if*

$$\begin{array}{l} \langle \exists\ s : s \in \mathcal{O}(E) : \\ \quad \langle \forall\ (e_i, e_j) : (e_i, e_j) \in s : \\ \quad\quad (e_i \in V) \wedge (e_j \in V) \wedge (e_i \olessthan_v e_j)\ \rangle\ \rangle \end{array}$$

We say that $(V, \olessthan_v)$ satisfies E. ◁

Intuitively, if $\mathcal{O}(E) = \{\wr_1, \wr_2 \ldots \wr_n\}$, a computation satisfies E if all orderings specified in at least one of $\wr_j, j \in \{1 \ldots n\}$ are preserved in the computation.

The constrained concurrent composition

$$C = (C_1 \parallel C_2 \parallel \ \ldots\ \parallel C_n) \text{ where } \mathcal{E}$$

of programs C_1, C_2, …, C_n, therefore, denotes a program C such that

$$\langle\ \forall\ (V, \olessthan_v) : (V, \olessthan_v) \in \mathcal{X}_s(C) : Sat((V, \olessthan_v), \mathcal{E})\ \rangle$$

We now define the semantics of the primitive event ordering constraint expression and the interaction operators by specifying the corresponding ordering constraint sets:
1. For expression `<eoce>::= event_id1` $\olessthan$ `event_id2`

$$\mathcal{O}(\texttt{<eoce>}) = \{\{\texttt{event\_id1, event\_id2}\}\}$$

The above expression models the interaction relationship between two events. The order of execution between the events is determined by issues such as data dependency, safety, and progress properties. Primitive ordering constraint expressions allow one to capture such interactions when they are translated to the most primitive level of computations, that is, events.

2) And constraint operator ($\owedge$): The operator $\owedge$ is used for combining a set of event ordering expressions in order to represent interactions among sets of events. For expression `<eoce>::=` $E_1 \owedge E_2$

$$\mathcal{O}(E_1 \owedge E_2) = \bigcup_{s_i \in \mathcal{O}(E_1)} \left(\bigcup_{s_j \in \mathcal{O}(E_2)} s_i \cup s_j \right)$$

Intuitively, the above expression specifies that a computation satisfies $(E_1 \owedge E_2)$ if it satisfies both E_1 and E_2.

Example 3.1 *($\owedge$ operator)* For expression $E = (a \olessthan b) \ \owedge \ (c \olessthan d)$

$$\mathcal{O}(E) = \{\{(a,b),(c,d)\}\}$$

A computation satisfies E if event a occurs before event b *and* event c occurs before event d in the computation. $\triangleleft$

3) Or Constraint Operator ($\ovee$): The operator $\ovee$ is used to incorporate nondeterminism in the orderings of events. For expression `<eoce>::=` $E_1 \ovee E_2$

$$\mathcal{O}(E_1 \ovee E_2) = \mathcal{O}(E_1) \cup \mathcal{O}(E_2)$$

Intuitively, the above expression specifies that a computation satisfies $(E_1 \ovee E_2)$ if it satisfies *at least one* of E_1 or E_2.

Example 3.2 *($\ovee$ operator)* For expression $E = (a \olessthan b) \ovee (b \olessthan a)$

$$\mathcal{O}(E) = \{\{(a,b)\},\{(b,a)\}\}$$

A computation satisfies E if a occurs before b or b occurs before a in the computation. Expression E specifies mutual exclusion between the events a and b. $\triangleleft$

Assume that F and G denote ordered (with respect to an index) sets of events $\{f_1, f_2, \ldots, f_n\}$ and $\{g_1, g_2, \ldots, g_n\}$ respectively. Terms f_i and g_j denote the ith and jth events of F and G respectively. Note that sets F and G may include any kind of events, including events that are occurrences of different actions.

4) forall: The expression

```
forall $i$ in $\{1 \ldots n\}$
    $P(f_i, g_i)$
```

is equivalent to the expression

$$P(f_1, g_1) \owedge P(f_2, g_2) \owedge \cdots \owedge P(f_n, g_n)$$

Here $P(f_i, g_i)$ is an event ordering constraint expression between the events denoted by f_i and g_i.

5) exists: The expression

$$\begin{array}{l}\texttt{exists } i \texttt{ in } \{1 \ldots n\} \\ \qquad P(f_i, g_i)\end{array}$$

is equivalent to the expression

$$P(f_1, g_1) \ovee P(f_2, g_2) \ovee \cdots \ovee P(f_n, g_n)$$

Operators forall and exists are extensions of operators $\owedge$ and $\ovee$ respectively.

Example 3.3 *(Event ordering constraint expression)* For expression E,

$$E = ((a \olessthan b) \ovee (c \olessthan d)) \owedge ((a \olessthan c) \ovee (b \olessthan d))$$

the ordering constraint set is

$$\mathcal{O}(E) = \{\{(a,b),(a,c)\},\{(a,b),(b,d)\},\{(c,d),(a,c)\},\{(c,d),(b,d)\}\}$$

$\triangleleft$

3.3 Representation of Programs

There are two unique aspects in our model of concurrent programming: the first is the separation of interaction definitions from programs specifications, and the second is the choice of “events” as the basis for interaction. We now examine how the two aspects affect representations of programs.

Concurrent programs aim to model applications where entities exist and perform independently. During an execution, an entity performs certain operations. Occasionally, it interacts with its environment at certain points. We call such execution points *interaction points*. Programming languages such as CSP[Hoa78] model such entities as processes and model their interactions with other entities by sending and receiving messages on channels of communications. Sends and receives, therefore, form the interaction points of processes.

We observe that there are two elements of an interaction point: the first is its identification. This underlines the fact that a program may interact at the interaction point. In CSP, for instance, names of communication channels along with the send and receive operations identify the interaction points of a process. The second is its *role* in an interaction. The role of an interaction point determines the manner in which a program participates in an interaction at the interaction point. For instance, the role associated with a “receive” interaction point determines a process’s behavior at the interaction point, that is, the process is delayed until a message arrives.

In our model of programming, the two elements of an interaction point — its identification and its role — are separated. Interaction points of a program are identified when the program is specified. However, the roles of the interaction points are determined (by event ordering constraint expressions) when the program is composed with other programs by the composition operator. Since "events" form the basis for defining event ordering constraint expressions, we choose events to represent interaction points. We now present a representation for programs, which contains, in addition to its composition definition, definitions of interaction points.

In sequential imperative programming languages, programs are represented by blocks [CK89]. A block, say f_s, has two components: i) interface of the form $f_s(p_1,\ p_2,\ \ldots,\ p_n)$, where variables $p_1, p_2, \ldots, p_n$ parameterize the execution behavior of f_s; and ii) composition of operations, specifying the computations of f_s. A block identifies — implicitly — two interaction points: i) *entry* point, where control and values of the parameters are passed and ii) *exit* point, where the block terminates and returns any values. All other execution points are encapsulated by the block abstraction.

We extend the notion of sequential blocks. We call such entities *interacting blocks*. The following are the elements of an interacting block:

1. Interface: An interacting block f_c has an interface of the form
$$f_c(p_1, \ldots, p_n\ ;\ i_1, \ldots, i_m)$$
Here p_1, p_2, …, and p_n are parameter variables. Interaction parameters i_1, i_2, …, and i_m denote interaction points of f_c.
2. Implementation: The implementation of an interacting block is partitioned into two. The first specifies the sequential and concurrent composition of a program. It determines the execution behaviors of the program. The second contains a mapping between the interaction parameters i_1, i_2, …, and i_m and the events of the program.

The composition of two interacting blocks $f_c(v_1, \ldots, v_n\ ;\ i_1, \ldots i_m)$ and $g_c(w_1, \ldots, w_l\ ;\ j_1, \ldots, i_k)$ is, thus, specified by the expression:

```
$f_c(v_1, \ldots, v_n) \parallel g_c(w_1, \ldots, w_l)$
   where
      $\mathcal{E}(e_1, \ldots, e_p)$
```

Here, the event ordering expression $\mathcal{E}$ is over a set of events $\{e_1, \ldots, e_p\}$ such that every event in this set is an interaction point of either f_c or g_c.

3.4 Examples

This section contains two examples, each highlighting aspects of the C-YES model. The emphasis in the first example is on the composition and interaction specification mechanisms. In the second example, it is on the representation of concurrent programs by interacting blocks. The language that we use here contains features from C [KR78] and a few extensions in order to present our ideas. We explain these extensions where they are used.

3.4.1 Example 1: Producer/Consumer.

3.4.1 Example 1: Producer/Consumer. Let `Producer` and `Consumer` be two programs, whose abstract representations are:

```
Producer(Prod(int i)) = {            Consumer(Cons(int j)) = {
  while (1) {                            while (1) {
          :                                      :
      Produce(Buf);                          Consume(Buf);
          :                                      :
  }                                      }
  Prod(int i) = (Produce, i)             Cons(int j) = (Consume, j)
}                                    }
```

Interaction parameters `Prod(i)` and `Cons(j)` are interaction points of `Producer` and `Consumer` programs respectively. Parameter `Prod(i)` names all `Produce` events. It denotes a set of events such that `Prod(k)` denotes `(Produce, k)` (kth occurrence of `Produce`) in the `Producer` program. Parameter `Cons(j)` is similarly defined. We will write `(Prod, i)` and `(Cons, j)` for `Prod(i)` and `Cons(j)` respectively.

We now derive a composition of the `Producer` and `Consumer` programs. The actions `Produce` and `Consume` interact with each other. All other actions are non-interacting and hence execute in parallel during the execution of the composite program. The following is such a composition of the two programs:

```
Comp = (Producer ∥ Consumer) where ConsExp
```

The above expression specifies that during an execution of `Comp`, all events in `Producer` and `Consumer` execute in parallel except for those that must satisfy the ordering constraints imposed by the event ordering constraint expression `ConsExp`.

The simplest interaction arises from the mutual exclusion constraint between the actions: no occurrences of `Produce` and `Consume` should execute in parallel. The following event ordering constraint expression captures the mutual exclusion constraint between the ith and jth occurrences of `Produce` and `Consume` actions:

```
((Prod, i) ⧀ (Cons, j)) Ⓥ ((Cons, j) ⧀ (Prod, i))
```

The expression for mutual exclusion among all occurrences of `Produce` and `Consume`, therefore, is:

```
ConsExp = forall i in {1...}
            forall j in {1...}
              ((Prod, i) ⧀ (Cons, j)) Ⓥ ((Cons, j) ⧀ (Prod, i))
```

There are many possible executions of `Comp` that satisfy the event ordering constraint expression `ConsExp`. One such execution is shown in figure 2. Only the interacting events are shown in the figure. Here all occurrences of `Produce` dominate the first occurrence of `Consume`, thereby causing the starvation of `Consumer`. The interaction in the figure is captured by the following event ordering constraint expression:

```
ConsExp1 =  forall i in {1...}
                   (Prod, i) ⧀ (Cons, 1)
```

Note

$$\langle \forall v : v \in \xi_s(\texttt{Comp}) : Sat(v, \texttt{ConsExp1}) \Rightarrow Sat(v, \texttt{ConsExp}) \rangle$$

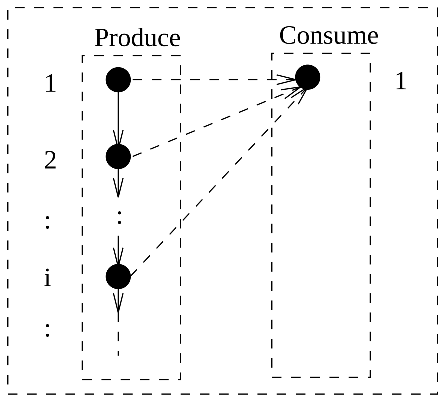

Fig. 2. Domination of `Produce` events over `Consume` events

A different interaction semantics for the `Producer` and `Consumer` programs can be specified by constructing a different event ordering constraint expression. For instance, assume that `Producer` and `Consumer` access a buffer of size one. The semantic constraints on accessing and modifying the buffer are: i) No data is consumed until it is produced, and ii) No data is produced until the previously produced data has been consumed. The following event ordering constraint expression captures these constraints:

```
ConsExp2 = forall i in {1...}
             ((Prod, i) ⧀ (Cons, i)) ⩓ ((Cons, i) ⧀ (Prod, i+1))
```

Figure 3 shows the event dependency graph of the computation of the composition. Note that there are no starvations or deadlocks.

3.4.2 Example 2: Gaussian Elimination. We develop below a concurrent program for the Gaussian elimination algorithm. The purpose here is to highlight the important ideas inherent in the notion of interacting blocks, and to discuss their usefulness in the representation of concurrent programs.

For a $n \times n$ matrix A, the Gaussian elimination algorithm [Ste73] solves the linear equation

$$Ax = b$$

There are two distinct steps in the algorithm. The first step, called *forward elimination*, transforms A into an upper triangular matrix, whereas the second

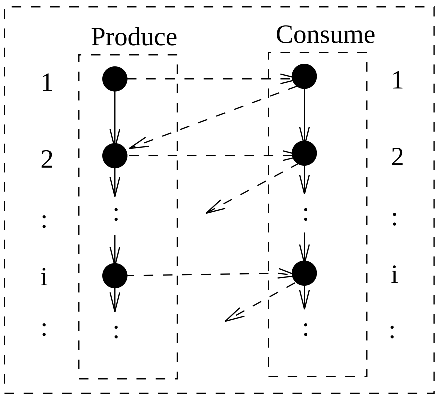

Fig. 3. Single buffer interaction between `Producer` and `Consumer`

step uses the transformed matrix to derive the solutions of the unknowns. We will focus our attention here on the forward elimination step.

There are $(n-1)$ steps, each called *pivot,* in the forward elimination step. In the ith pivot step, elements $A[i+1, i]$ through $A[n, i]$ are reduced to zero, while modifying certain other elements of A. We represent each pivot by an interacting block. The forward elimination program is a constrained concurrent composition of the $(n-1)$ pivot programs. The event ordering constraint expression associated with the composition represents the interaction among the pivot steps.

A motivation for structuring the forward elimination program in this manner is to show that i) there is concurrency among the different pivot steps, and ii) this concurrency can be easily expressed by our composition and interacting block mechanisms. The interacting block associated with the ith pivot step is shown in figure 4. The details of the definition in figure 4 are :

```
P(int i; Access(int aIndJ, int aIndK), Change(int cIndJ, int cIndK)) = {
   int j, k;
   { /*implements functionality of P(i) */
      for (j = i+1; j < n; j = j + 1)
         for (k = i+1; k < n; k = k+1)
            A[j, k] = A[i, k] - (A[j, k]*(A[i, i]/A[j, i]));
   }
   /* Map interaction points to internal execution points*/
   Access(int aIndJ, int aIndK) = (read.A[j, k], 1);
   Change(int cIndJ, int cIndK) = (write.A[j, k], 1);
}
```

Fig. 4. Representation of the ith Pivot step

1. Interface: The interface of `P(i)` defines three parameters. The first, i, identifies the ith pivot. The interaction parameters, `Access(int aIndJ, int`

`aIndK)` and `Change(int cIndK, int cIndK)`, denote the interaction points of `P(i)`.

2. Implementation: The first part of the implementation specifies the computations of the ith pivot. The implementation shown here is sequential. Most implementations of the forward elimination step focus on exploiting parallelism inherent within each pivot step.

   The second part specifies how interaction parameters are mapped to the actual events of `P(i)`. The parameter `Access(int aIndJ, int aIndK)` denotes a set of `P(i)`'s *first* read events of elements of A. For instance, term `P(i).Access(2, 3)` denotes the *first* read of element `A[2, 3]` that `P(i)` executes. Since all such *first* reads in the ith pivot step interact with the writes to the corresponding elements of the $(i-1)$th pivot steps, we include these read events in the set denoted by `Access` by the following definition:

   ```
       Access(int aIndJ, int aIndK) = (read.A[j, k], 1);
   ```

   The variables `j` and `k` in the above definition refer to the variables `j` and `k` of the implementation of `P(i)`. The parameters `aIndJ` and `aIndK` are used to select unique events from the set that `Access` denotes. The other interaction parameter `Change(cIndJ, cIndK)` is defined similarly.

The forward elimination program can now be specified as a constrained concurrent composition of the $(n-1)$ pivot steps. Let us look at the interaction between the ith and (i+1)th pivot steps: Pivot `P(i+1)` should not read or write to any `A[j, k]` until `P(i)` has modified this `A[j, k]`. There is an explicit ordering between the reads and writes of the two steps and the event ordering constraint expression must capture this data dependency. The following expression represents such a dependency:

```
(forall aj, ak in P(i+1).Access
      P(i).Change(aj, ak) ⧀ P(i+1).Access(aj, ak))
                ∧
(forall cj, ck in P(i+1).Change
      P(i).Change(cj, ck) ⧀ P(i+1).Change(cj, ck))
```

In the above expression, `aj, ak, cj` and `ck` are variables that range over the sets that `Access` and `Change` denote. The expression captures the orderings among the interacting reads and writes of `P(i)` and `P(i+1)`. The forward elimination program is shown in figure 5.

3.5 Discussion

We now examine the characteristics of our program composition and program representation mechanisms.

```
ForwardElimination = {
   $\|_{i=1}^{n-1}$ P(i) where
      forall i in {1, ..., (n-2)}
         (forall aj, ak in P(i).Access
               P(i).Change(aj, ak) $\textcircled{<}$ P(i+1).Access(aj, ak) )
                     $\textcircled{\wedge}$
         (forall cj, ck in P(i).Change
               P(i).Change(cj, ck) $\textcircled{<}$ P(i+1).Change(cj, ck))
}
```

Fig. 5. Forward elimination program

3.5.1 Event Based Program Composition. The constrained concurrent composition of programs specifies two kinds of relationships among the events of component programs: concurrency and interaction. The notion of concurrency is assumed to be universal, that is, events are concurrent by default. Interaction among events is modeled by interaction expressions, which determine execution orderings among the events. The model contains several unique ideas such as separation of program and interaction specifications, "events" as the basis for interaction, abstraction of interaction, and algebraic interaction expressions. We discuss each in detail below.

The composition mechanism separates interaction specifications from program specifications. A program in our model does not include any synchronization information; it merely specifies what its computations are. Since programs interact with other programs only when they are composed, interaction specifications are elements of concurrent compositions. They are defined when such compositions are defined.

One of the implications of the above separation is that a program specification can be used in many concurrent compositions, each defining different event ordering constraint expressions. The interaction behavior of the program may be different in each of the compositions. For instance, the two event ordering constraint expressions `ConsExp` and `ConsExp2` specify different interaction semantics for the same `Producer` and `Consumer` programs.

Interaction is specified at the "event" level. The composition mechanism is based on the observation that fundamental interaction occurs between two events. All other interactions can be represented by suitable combinations of primitive interaction expressions. The name space in our model, therefore, is fine-grained, since it contains names for actions as well as their individual occurrences.

One of the implications of such a fine-grained name space is that interaction relationships can be defined among all events directly. This is unlike other approaches, where the name space contains a fixed set of named actions. Also, the interaction among the occurrences of these actions is predetermined, and cannot be changed.

Our concurrent constrained composition mechanism, in conjunction with the choice of "events" as the basis for interaction, allows one to represent programs that are highly concurrent. We show this by extending the producer-consumer example. Assume that `Producer` and `Consumer` programs access a buffer such that

```
Produce = P(buffer[0]); P(buffer[1])
Consume = C(buffer[0]); C(buffer[1])
```

Here, every `Produce` event creates information in `buffer[0]` and `buffer[1]` and every `Consume` event retrieves information from `buffer[0]` and `buffer[1]`. Note that `Produce` and `Consume` here are not atomic actions anymore; they are composed (with the sequential operator ';') from actions `P` and `C` respectively.

Expression `ConsExp2` can be used here to specify an interaction between `Produce` and `Consume`. However, it overconstrains the executions of `P` and `C` events. It introduces unnecessary ordering among events when there should be none. In order to derive an event ordering constraint expression that does not impose any ordering constraints among the events that occur at different buffers, we need to identify `P` and `C` as the basis of interaction — not `Produce` and `Consume`. The redefined interaction points `Prod` and `Cons`, therefore, are:

```
Prod(i) = (P, i)
Cons(j) = (C, j)
```

The derivation first specifies single buffer interactions at `buffer[0]` and `buffer[1]`, and then combines the two expressions to construct an expression for both the buffers.

For all $i \in \{\,1\ldots\,\}$, let

$$\texttt{(P1, i)} = \texttt{(Prod, 2i-1)} \tag{3.5.1}$$

$$\texttt{(P2, i)} = \texttt{(Prod, 2i)} \tag{3.5.2}$$

$$\texttt{(C1, i)} = \texttt{(Cons, 2i-1)} \tag{3.5.3}$$

$$\texttt{(C2, i)} = \texttt{(Cons, 2i)} \tag{3.5.4}$$

$\mathcal{E}_1$
```
= {Single buffer interaction at Buf[0]}
  forall $i$ in {1 ...}
     (P1, $i$) $\textcircled{<}$ (C1, $i$) $\textcircled{\wedge}$
     (C1, $i$) $\textcircled{<}$ (P1, $i+1$)

= { eqn 3.5.1, eqn 3.5.3 }
  forall $i$ in {1 ...}
     (Prod, $2i-1$) $\textcircled{<}$ (Cons, $2i-1$) $\textcircled{\wedge}$
     (Cons, $2i-1$) $\textcircled{<}$ (Prod, $2i+1$)

= { $k = 2i-1$ }
  forall $k$ in {1, 3, ...}
     (Prod, $k$) $\textcircled{<}$ (Cons, $k$) $\textcircled{\wedge}$
```

$\mathcal{E}_2$
```
= {Single buffer interaction at Buf[1]}
    forall $j$ in {1 ...}
        (P2, $j$) $\textcircled{<}$ (C2, $j$) $\textcircled{\wedge}$
        (C2, $j$) $\textcircled{<}$ (P2, $j+1$)

= { eqn 3.5.2, eqn 3.5.4 }
    forall $j$ in {1 ...}
        (Prod, $2j$) $\textcircled{<}$ (Cons, $2j$) $\textcircled{\wedge}$
        (Cons, $2j$) $\textcircled{<}$ (Prod, $2j+2$)

= { $l = 2j$ }
    forall $l$ in {2, 4, ...}
        (Prod, $l$) $\textcircled{<}$ (Cons, $l$) $\textcircled{\wedge}$
```

```
(Cons, $k$) $\circledless$ (Prod, $k+2$)                    (Cons, $l$) $\circledless$ (Prod, $l+2$)

          { Combine $\mathcal{E}_1$ and $\mathcal{E}_2$ }
          ConsExp3 = $\mathcal{E}_1$ $\circledwedge$ $\mathcal{E}_2$ =
                    forall $i$ in $\{1 \ldots\}$
                          (Prod, $i$) $\circledless$ (Cons, $i$) $\circledwedge$
                          (Cons, $i$) $\circledless$ (Prod, $i+2$)
```

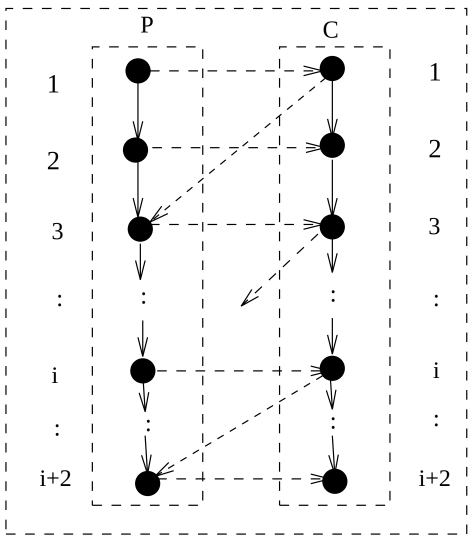

Fig. 6. Two buffer interaction between `Producer` and `Consumer`

Figure 6 shows the event dependency graph associated with an execution of `Comp`. Observe that

$$\langle\, \forall\, i:\ i \in \{\, 1\ \ldots \,\}:\ (\mathtt{P}, i+1) \parallel_e (\mathtt{C}, i)\, \rangle$$

Intuitively, it means that `Consumer` can access `buf[0]` while `Producer` is modifying `buf[1]`. Also, the above interaction expression can be extended to define interaction over a buffer of size `N`:

```
$\mathcal{E}$ = forall $i$ in $\{1 \ldots\}$
          (Prod, $i$) $\circledless$ (Cons, $i$) $\circledwedge$ (Cons, $i$) $\circledless$ (Prod, $i+N$)
```

The interaction specification mechanism captures the fundamental abstractions of interaction. It specifies interaction by suitable ordering relations among the interacting events of programs. It does not depend on the semantics of these events. Hence, the model can be used for both shared and distributed memory programs. For instance, the event ordering constraint expressions `ConsExp1` and `ConsExp2` in the producer-consumer example specify interaction in accessing a buffer. The buffer can be a shared memory structure or a distributed resource. Implementations of the `Produce` and `Consume` actions determine the shared or distributed nature of the buffer.

The interaction expressions are algebraic in nature. They are composed from a set of primitive ordering constraint expressions and the interaction operators. The approach is constructive in that one first identifies the events that interact, and represents their interaction through a set of primitive ordering constraint expressions. An event ordering constraint expression is then constructed from the primitive expressions and the interaction operators. This facilitates *modular* [Blo79] construction of event ordering constraint expressions. For instance, in the two-buffer producer-consumer example, we derive separate event ordering constraint expressions for the two buffers and then use the $\owedge$ operator to combine the two expressions.

Since the properties of the basic primitives and the operators are well defined, it is possible to reason with the interaction specifications in a rigorous manner. For instance, it is possible to prove that

$$\langle \forall v :\ v \in \xi_s(\texttt{Comp}) :\ Sat(v, \texttt{ConsExp2}) \Rightarrow Sat(v, \texttt{ConsExp}) \rangle$$

which states that the interaction expression `ConsExp2` guarantees mutual exclusion between the `Produce` and `Consume` actions.

3.5.2 Concurrent Program Representation. An interacting block extends the notion of sequential block by incorporating mechanisms to identify a set of events as interaction points. The interface of an interacting block includes interaction parameters denoting interaction points of the block. The implementation defines the mappings between the interaction parameters and the events of the block.

The separation between the interaction parameters (defined at the interface) and their mappings (defined in implementation) provides a powerful mechanism for isolating the definition of the interaction behavior of a program from the implementation of the program. It is, therefore, possible to change the implementation, the interaction parameters, or the interaction parameter mappings without changing other aspects of the program definition. We explore how these changes can be made and what they mean in terms of design and implementation of concurrent programs.

It is possible to change a program's implementation without changing any composition expressions that include this program. This is possible because of the separation between how interaction points are defined (in implementations) and how they are used (in event ordering constraint expressions). Note, however, that changes in the implementation should not affect the semantic nature of the interaction points and their participation in the composition definition. We clarify this by showing that it is possible to modify `P(i)`'s implementation without changing either the definitions of the interaction points or the `ForwardElimination` program.

The idea is to change those aspects of the program that do not affect either the interaction points or the orderings between the interaction points of the different pivots. In P(i), it is easy to do so, since we can parallelize one or some aspect of the pivot without changing either the interaction points (reads and writes) or the ordering constraints between them. The implementation in figure 7 is one such realization.

```
P(int i; Access(int aIndJ, int aIndK), Change(int cIndJ, int cIndK)) = {
   int j, k;
   { /*implements functionality of P(i) */
      $\|_{j=i+1}^{j=n-1}$
         $\|_{k=i+1}^{k=n-1}$
               A[j, k] = A[i, k] - (A[j, k]*(A[i, i]/A[j, i]));
   }
   /* define interaction points*/
}
```

Fig. 7. Parallel implementation of P(i)

The body of P(i) now utilizes the parallelism inside each pivot step. The definitions of Access, Change, and ForwardElimination remain unchanged, since there are no changes either in the interaction points or in the orderings among interacting events. The above algorithm is optimally concurrent in that no event is delayed unless it is semantically dependent on some other event.

Another implication of the separation between the interaction parameter and an interacting block's implementation is that the interaction behavior of a program can be changed by modifying either the mappings between the interaction parameters and the internal events, or the event ordering constraint expressions, or both. This is useful in determining those interaction points and interaction expressions that provide optimal performance for a given algorithm by balancing the costs involved in computation and interaction.

In the Gaussian elimination example, the interacting block P(i) waits for P(i+1) to modify an element A[j, k] before it can read or write this element. The different pivot steps, therefore, interact at the level of single elements of the matrix. In many systems (e.g., message passing systems), such closely coupled interactions can be very inefficient. A modified version of ForwardElimination is shown in figure 8. In this implementation, the interaction between the pivots occurs at the level of rows: P(i+1) reads or writes a given row j only after P(i) has modified this row.

The above defines an ordering between the writes to the last element of a row by P(i) and reads and writes to the first element of a row by P(i+1). The composition relies on the fact that a row j is modified sequentially within P(i). It exploits the ordering relationships between the reads and writes in a single row. The above composition expression, therefore, cannot be used if the rows are not modified sequentially in P(i) (as is the case in figure 7).

```
ForwardElimination = {
       {
          $\|_{i=1}^{n-2}$ P(i) where
             forall i in {1, ..., (n-2)}
                ( forall aj in P(i+1).Access
                     P(i).Change(aj, n) $\circledless$ P(i+1).Access(aj, i+1))
                         $\circledwedge$
                ( forall cj in P(i+1).Change
                     P(i).Change(cj, n) $\circledless$ P(i+1).Change(cj, i+1) )
       }
}
```

Fig. 8. Row level interaction among pivots of forward elimination program

4 Related work

We now look at the interaction mechanisms that different concurrent programming languages use. We classify two broad classes of interaction mechanisms on the basis of how common information is manipulated by concurrent programs in these languages.

4.1 Unstructured

In unstructured approaches, programming models do not impose any constraints on how shared information can be manipulated or exchanged inside programs. The models identify a set of synchronization primitives, which are used to synchronize accesses to shared information or to access distributed information. The interaction between the synchronization primitives is defined at the "action" level. All interactions are constructed from suitable combinations of these primitives. Examples of unstructured synchronization primitives are semaphores [Dij65, Dij68], message-passing primitives [Hoa78, Per87], and write-once shared variables [Sha89, FT89] of logic programming languages.

Interactions among programs in these languages are realized indirectly through the executions of suitable synchronization primitives. The synchronization primitives are at a low level of abstraction, which makes it difficult to define concurrent programs. Also, since the synchronization primitives are embedded within the specifications of the programs, one must reason, operationally, with the logic of these programs.

4.1.1 Structured Approaches In structured approaches, a module $M = (I_1, I_2, \cdots, I_n)$ provides a set of services through interfaces I_1, I_2, $\ldots$, I_n. Any accesses or modifications to the data structures of M must be made through the interfaces. Examples of structured approaches are Monitor [Hoa74], abstract data types with Path Expressions [CH74, And79] or abstract expressions [AHV85], Mediators [GC86], rendezvous–based models [Geh84, GR86], remote–procedure call models [BN84], and concurrent object–oriented models [Ame87, YBS87, TS89, CK92, WKH92].

There are two possible sources of concurrency and interactions among programs in such models: i) External interaction and concurrency: these arise when a program C requests for a service through an interface I_k. The requested service and C can both execute in parallel and interact. ii) Internal interaction and concurrency: these arise because of the concurrent invocations of services within M. The concurrent requests interact because they share the data structures that M encapsulates.

Models based on structured approaches support hierarchical and modular software development methodologies and hence should be a basis for software development in the concurrent domain as well. However, support for concurrency and interaction in these models is weak. The reasons are the following:

1. The external interaction in most languages such as ADA [Geh84] and Monitors [Hoa74] is based on synchronous call-return semantics. This limits the possible concurrency between the calling and called computations.
2. Models that do allow for external concurrency among the calling and called computations employ ad-hoc mechanisms such as `future` [YBS87] for specifying external interaction between the two computations. There is no general mechanism for specifying arbitrary external interactions.
3. Many programming models such as Monitors [Hoa74] and ADA [Geh84] do not support internal concurrency.
4. Models such as Mediator [GC86] do allow concurrent executions within a module. However, they either support only non-interacting concurrent executions or use unstructured primitives to specify interactions. Such models inherit the problems associated with the unstructured primitives.

5 Conclusions and Future Work

There are two elements in a program definition in our model: the first specifies its computations (what the program does when it is executed), and the second identifies its interaction points (places where the program may interact with its environment). Interaction points are abstractions of a subset of execution points of the program. A constrained concurrent composition combines a set of programs by specifying both ordering relationships among their interaction points as well as concurrency among all other execution points.

In our model, "events" form the basis for identifying interaction points as well as for specifying interaction among programs. The latter is specified by constructing an algebraic expression from a set of primitive interactions and interaction operators.

Our programming model isolates a program's specification from its interaction behaviors. The separation between the interaction and program specifications allows one to reuse the program definition in many concurrent compositions. Also, the notion of interacting block separates how interaction points are defined (within a program's implementation) and how they are used (in interaction expressions). Hence, it is possible to change a program's implementation without changing any composition expressions as long as certain semantic properties of the interaction points are preserved. In addition, the separation allows one to experiment with different interaction expressions in order to examine different interaction behaviors of the program. This provides a flexible mechanism for finding those interaction points that try to balance the cost involved in computation and interaction.

The choice of "events" as a basis for interaction and the algebraic mechanism for interaction specifications allow direct and precise formulation of interaction definitions among programs. The interaction expressions are modular and algebraic. Also, it is possible to rigorously analyze interaction expressions in order to determine certain classes of properties of programs.

The programming model, therefore, supports the specification of concurrent programs, which are i) highly concurrent and asynchronous, ii) portable across both shared and distributed memory architectures, and iii) represented as expressions over component programs. The composition also supports precise and direct specifications of interactions.

We are investigating a number of issues that deal with the theoretical, language design, and implementation aspects of our model. Active work is proceeding on the integration of our concurrent programming model within the object–oriented framework. The issues that we consider here deal with the representation of concurrent and interacting objects, method compositions, inheritance of methods and interactions expressions, and association between events and method activations. Further, we aim to extend an existing object–oriented programming language to define a concurrent programming language. We also plan to define execution semantics for this language. In addition, we are looking at extending the set of interaction operators and their associated semantics. One of the interesting operators is conditional ordering operator. We are examining its semantics and its relationship with the $\vee$ and `exists` operators.

References

[AHV85] F. Andre, D. Herman, and J. P. Verjus. *Synchronization of Parallel Programs.* The MIT Press, Cambridge, MA, 1985.

[Ame87] Pierre America. POOL-T: A Parallel Object–Oriented Language. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 199–220. The MIT Press, 1987.

[And79] S. Andler. Predicate Path Expression. In *Proc. Sixth ACM Symposium on Principles of Programming Languages*, pages 226–236, 1979.

[And91] Gregory R. Andrews. *Concurrent Programming.* The Benjamin/Cummings Publishing Company, Redwood City, CA, 1991.

[Blo79] Toby Bloom. Evaluating Synchronization Schemes. In *Proc 7th Symposium on Operating Systems Principles*, pages 24–32. ACM, 1979.

[BN84] Andrew Birrell and B. J. Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1):39–59, 1984.

[CH74] R. H. Campbell and A. N. Habermann. The Specification of Process Synchronization by Path Expressions. In *Lecture Notes on Computer Sciences*, volume 16, pages 89–102. Springer Verlag, 1974.

[CK89] Gianna Cioni and Antoni Kreczmar. Modules in High Level Programming Languages. In *Advanced Programming Methodologies*, pages 247–340. Academic Press, Ltd., 1989.

[CK92] K. Mani Chandy and Carl Kesselman. Compositional C++: Compositional Parallel Programming. Technical Report Caltech-CS-TR-92-13, Cal Tech, 1992.

[Dij65] E. W. Dijkstra. Solution of a Problem in Concurrent Programming Control. *Communication of the ACM*, 8(9):569, 1965.

[Dij68] E. W. Dijkstra. The Structure of the THE Multiprogramming System. *Communication of the ACM*, 11(5):341–346, 1968.

[FT89] Ian Foster and Stephen Taylor. *Strand: New Concepts in Parallel Programming.* Prentice Hall, Englewood Cliffs, N. J., 1989.

[GC86] J. E. Grass and R. H. Campbell. Mediators: A Synchronization Mechanism. In *Sixth International Conference on Distributed Computing Systems*, pages 468–477, 1986.

[Geh84] Narain H Gehani. *Ada: Concurrent Programming.* Prentice Hall, Englewood Cliffs, N.J., 1984.

[GR86] Narain H Gehani and W. D Roome. Concurrent C. *Software – Practice and Experience*, 16(9):821–844, 1986.

[Hoa74] C. A. R. Hoare. Monitor: An Operating System Structuring Concept. *Communication of the ACM*, 17(10):549–557, 1974.

[Hoa78] C. A. R. Hoare. Communicating Sequential Processes. *CACM*, 21(8):666–677, 1978.

[KR78] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice–Hall, Englewood Cliffs, New Jersey, 1978.

[Per87] Brinch Hansen Per. Joyce - A Programming Language for Distributed Systems. *Software – Practice and Experience*, 17(1):29–50, 1987.

[Pra86] Vaughan Pratt. Modeling Concurrency with Partial Order. *International Journal of Parallel Programming*, pages 33–71, 1986.

[Sha89] Ehud Shapiro. The Family of Concurrent Logic Programming Languages. *ACM Computing Surveys*, 21(3):413–510, 1989.

[Ste73] G. W. Stewart. *Introduction to Matrix Computations.* Academic Press, New York, 1973.

[TS89] Chris Tomlinson and Mark Scheevek. Concurrent Object Oriented Programming Languages. In Won Kim and F. H. Lochovsky, editors, *Object Oriented Concepts, Databases, and Applications*, pages 79–124. ACM Press, 1989.

[WKH92] Barbara B. Wyatt, K. Kavi, and Steve Hufnagel. Parallelism in Object–Oriented Languages: a Survey. *IEEE Software*, 9(6), 1992.

[YBS87] A. Yonezawa, J. Briot, and E. Shibayama. Modeling and Programming in Object–Oriented Concurrent Language ABCL/1. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 55–89. The MIT Press, 1987.

This article was processed using the LaTeX macro package with LLNCS style