COMPUTER SYSTEMS LABORATORY

DEPARTMENTS OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
STANFORD UNIVERSITY · STANFORD, CA 94305

# Gem: A Tool for Concurrency Specification and Verification

Amy L. Lansky
Susan S. Owicki

Technical Report No. 83-251

November 1983

The author is a fellow of the Fannie and John Hertz Foundation. This work was also supported in part by NSF Grant MCS-80-05336.

# Gem: A Tool for Concu rrendy Specification and Verification

Amy L. Lansky
Susan S. Owicki

Technical Report No. 83-251

November 1983

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305

## Abstract

The GEM model of concurrent computation is presented. Each GEM computation consists of a set of partially ordered events, and represents a particular concurrent execution. Language primitives for concurrency, code segments, as well as concurrency problems may be dcscribed as logic formulae (restrictions) on the domain of possible GEM computations. An event-oriented method of program verification is also presented. GEM is unique in its ability to easily describe and reason about synchronization properties. .

Key Words and Phrases: Concurrency, specification, verification, events, partial ordering.

# GEM: A Tool for Concurrency Specification and Verification

Amy L. Lansky[1], Susan S. Owicki
Stanford University

## Abstract

The GEM model of concurrent computation is presented. Each GEM computation consists of a set of partially ordered events, and represents a particular concurrent execution. Language primitives for concurrency, code segments, as well as concurrency problems may be described as logic formulae (restridtions) on the domain of possible GEM computations. An event-oriented method of program verification is also presented. GEM is unique in its ability to easily describe and reason about synchronization properties.

## 1. Introduction

Our understanding of distributed systems and computational concurrency is far from complete. Languages that include concurrency are harder to describe formally than sequential ones. Verification of concurrent programs is complicated by the unpredictable order in which the actions of independent processes occur.

This paper introduces the Group Element Model (GEM), a *behavioral* or *event-oriented* model of concurrent computation. GEM events are related by partial orderings. Because of this, concurrent action can be modeled in a realistic fashion. A given GEM computation can represent an execution which is truly distributed as well as one in which concurrent action is simulated on a single processor.

We have found GEM's event-oriented approach to be broad enough for general purpose language and problem specification, and powerful enough to serve as a basis for an event-oriented method of concurrent program verification. GEM is especially well-suited for describing and reasoning about *synchronization properties* -- properties that deal with the ordering of events. Synchronization properties such *as priority* are not easily expressed by techniques which rely upon assertions about program variable values.

Many formal models of concurrency have not been applied to language description, or have only been used to describe a single language. In contrast, GEM has already been used to describe three very different language primitives for concurrency: the *Monitor* [9, 11], *Communicating Sequential Processes (CSP)* [12], and *ADA tasks* [1]. These descriptions are fairly concise, were easy to formulate, and capture language structure and semantics in an intuitive fashion.

GEM has been used to provide descriptions of traditional concurrency problems such as the One Slot Buffer, Bounded Buffer, and Readers/Writers problems, as well as two distributed applications: an algorithm for performing updates to a distributed database [5], and an asynchronous, distributed version of the Game of Life [29]. The model has also been used to verify Monitor, CSP, and ADA programs. Properties such as lack of deadlock and functional correctness have been proven of the two distributed applications.

---

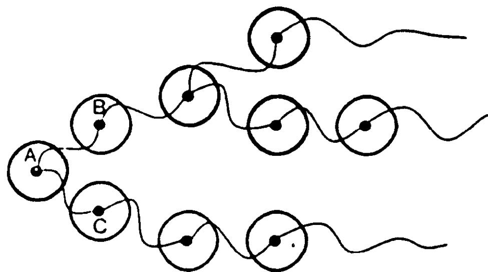
[1] The author is a fellow of the Fannie and John Hertz Foundation. This work was also supported in part by NSF Grant MCS-80-15336.

Sections 2-8 dcscribc GEM's method of representing concurrcnt execution and show how problem, language, and program specifications are formed. The GEM verification method is describcd in scction 9. Finally, an overview of related work is prcscntcd in section 10.

## 2. Concurrency: An Intuitive View

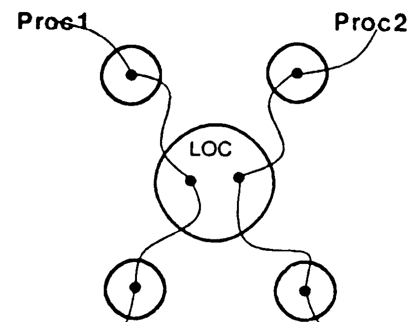
Before WC embark on a detailcd description of how GEM models concurrcnt execution, let us examine concurrency on a more intuitive level. Whcn we think of execution, WC think of actions occurring in timc as well as in space. In a sequential execution, there is only one locus of activity. Each action enables or causes the next action to occur, resulting in activity which is totally ordered in time.

In a concurrent execution, there are many loci of activity. Suppose, for the prcscnt, that each action occurs at its own personal locus of activity. Some actions enable others to occur, but actions may occur independently of one another as well. For example, an action A may fork, enabling two actions B and C. Actions B and C may then enable their own subtrecs of activity, each of which is unrclated to the other.



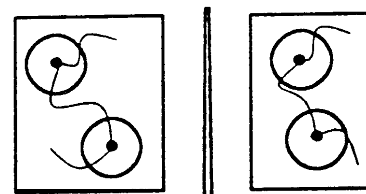

Although the actions which occur within the subtrees belonging to B and C may be ordered in time, such a tcmporal ordering is a result of chance. In fact, it is possiblc for any two actions from different subtrees to occur simultaneously. When it is logically possible for two events to occur simultancously, we will say thcy are *potentially concurrent*, or that therc is no *observable* order bctwecn them. From this cxamplc, it is clear that both thc enabling and observable temporal relationships bctwecn actions in a concurrent computation is partial.

We havc now considered thc role played by the cnabling relationship between actions. What role docs *locus* of activity play? WC have bccn assuming that cach action has its own locus of activity. There arc, howcvcr, scts of actions which logically occur at the same location. Consider, for' cxamplc, a single computcr storage ccll. In most models of computer cxccution, all actions (rcads and writes) which take place at thc samc mcmory location arc forccd to be totally ordcrcd in timc. Although thcsc actions may be causally unconncctcd (thcrc is no cnabling rclationship bctwecn thcm), thcy must occur in *some* scqucntial ordcr. For instance, such a situation ariscs whcn a mcmory location is shared between several logical processes of activity.




Thus there are activities which arc observably ordered in time without being causally related. In our model, all events at a single locus of activity must be totally ordered.

Some loci of activity are naturally grouped together. For example, it would be natural to group all loci of activity belonging to a single procedure activation, or to a single server process. Actions within such a grouping are often rclatcd semantically as well as by-the -enabling relationship. Semantic groupings arc also often tied to the traditional language concept of scope. For instance, the actions of two different processes might bc clustered into two separate groups. Because cach process is not allowed to access the . variables belonging to the other other process, it becomes natural to associatc thc two groups with a "firewall" which limits the enabling relationships between them.



Given this intuitive discussion of concurrent cxccution, we shall now proceed to explain GEM's modcl of cxccution in a more formal way. GEM rcprcsents the actions within a concurrcnt cxecution by objccts called *events*. Thc cnabling rclationship bctwccn actions is modclcd by the *enable relation* $\leadsto$. Loci of forced scqucntial activity are modeled

by *elements*, and this forccd sequcntial order is called the *element order* $=\!\blacktriangleright$. The *tctnporal order* $\Rightarrow$ betwccn actions (thc observablc order in time) is simply thc transitivc closure of $\leadsto$ and $=\!\blacktriangleright$ (minus idcntity). Semantically related loci of activity are modclcd by GEM *groups.*

## 3. Computations and Specifications

GEM is made up of two components: a model of execution and a spccification language. A GEM ***computation*** is an abstract rcprcsentation of concurrent execution. Each computation consists of a possibly infinitc set of objects called ***events,*** a partial relation $\leadsto$ (the ***enable relation),*** and two strict partial orders: $=\!\blacktriangleright$ (the ***element order),*** and $\Rightarrow$ *(the **temporal** order).* Events may bc also clustered into sets called ***elements*** and ***groups.***

A language or concurrency problem may be described by characterizing it as a GEM ***specification*** $\sigma$. Each specification is composed of a sct of logic formulac (restrictions) over thc domain of all possible GEM computations. A computation C is *legal* with respect to a specification $\sigma$ (*legal*(C,$\sigma$)) if C satisfies each restriction in $\sigma$. Thus, each GEM specification admits a set of computations with a particular desired behavior or meaning. For example, the restrictions describing the ADA language primitives admit GEM computations representing legal ADA program executions. A solution to thc database updatc problem is one that obeys all restrictions specifying that problem.

As onc might cxpcct, thcrc are certain properties that must be truc of *all* legal computations. These propertics are described by a set of GEM ***legality restrictions*** which arc automatically part of any GEM specification. For example, the following must be true of all legal GEM computations: the temporal order betwccn cvents is a transitive and irrcflexive order, and must be equivalent to the transitive closure of the enable rclation and the elemcnt order (minus idcntity). The way in which a computation's cvcnts arc clustcrcd into elcmcnts and groups imposes legality rcstrictions on thc element order and enable relation.

Each GEM legality restriction has a formal charactcrization. In this paper, however, we will prcscnt thesc properties informally, as we describe thc basic conceptual components of GEM.[2]

[2] This papcr is intcnded as a survcy of GEM rather than a complctc dcscriplion. For furthcr dctails refer to [17].

## 4. Events, Elements, and Groups

A GEM ***event*** reprcscnts a logical action that is regarded as atomic rclativc to other events in its computation. For example, an assignmcnt statemcnt, the forking of a proccss, and completion of an input requcst may bc modeled as evcnts. An cvent may be viewed as a structured object composed of the following kinds of information: data parameters, thread idcntificrs (sec section 8.3). and the names of thc elcmcnt and group scts to which it belongs. Each evcnt is a unique occurrence in a computation, with its own identity and associatcd data. A set of similar events may be charactcrizcd by an ***event class*** description.

As already stated, a GEM specification is used to restrict the sct of all possiblc computations to those that rcprcscnt mcaningful cxecutions for a particular languagc or problem. The first way in which this is done is to idcntify which events may occur., Specifically, the events which may legally occur within a computation arc those belonging to a specified list of elements: if *legal*(C,$\sigma$) and event e is in C, then e must belong to some element specified in $\sigma$.

***Elements*** model the elementary components of a languagc or problem whose associated actions musf for some reason, occur sequcntially. In this scnse, they arc similar to ***actors*** [6,10].

Each elcment is a unique entity and is charactcrized by a name and the events which belong to it. Moreover, every GEM event ***must*** belong to exactly one element. If an event belongs to an element, we say it "occurs" at that clement. All of the events occurring at the samc element must bc totally ordcred by the elcmcnt order, $=\!\blacktriangleright$.[3] Each element may bc associated with an explicit list of restrictions upon its events.

Let us look at an example. Suppose we wished to model an integer variable Var. Two types of events can bc associated with a variable, Assign and Getval. In addition, wc wish to assert that cvents of typc Var.Assign and Var.Getval occur sequcntially, whether they arc causally related or not. In essence, wc are asserting a lock on access to variable Var. Wc could writc the following GEM specification of Var:

[3] If wc do not wish to imposc an elcmcnt order on events. we can simply associatc each cvent with its own clcmcnt. In this limiting casc, cvcnts arc only rclatcd by the cnablc rclation and temporal ordcr.

```
Var = ELEMENT
EVENTS
    Assign(newval: INTEGER)
    Getval(oldval,: INTEGER)
RESTRICTIONS
    .
END Var
```

A specification which includes Var admits computations containing Var.Assign and Var.Getval events which obey the restriction

```
(∀ e1,e2:{Var.Assign, Var.Getval})
[ e1=▶e2 ∨ e2=▶e1 ∨ e1=e2 ]
```

as well as any additional restrictions explicitly associated with Var.

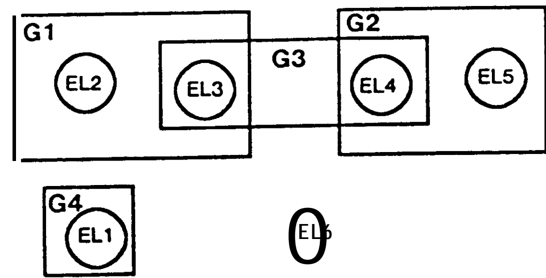
Note that because each event must belong to exactly one element, and all events occurring at the same element are totally ordered, a given event may be uniquely identified by naming the element at which it occurs and its occurrence number. For example, the assignment event denoted by $Var.assign_i$ (or simply $Var^i$) is the $i^{th}$ event at Var.

Events within a computation may also be clustered into *groups.* Groups are sets of elements and/or other groups, and are used to describe the compound structure of more complex language and problem components. For instance, a process might be modeled as a group containing local variable elements, local procedure groups, and a group representing the code of the process.

The group is a natural vehicle for modeling the static scoping rules found in most programming languages. Conventional scope rules limit access to certain variables or procedures. A GEM specification limits access to certain events by placing those events within specific groups. Each group structure is associated with a set of legality restrictions which limit the enable relation ↝ between events. For example, suppose we were given a specification of the following form.

```
ELEMENTS EL1,EL2,EL3,EL4,EL5,EL6

G1 = GROUP(EL2,EL3)
G2 = GROUP(EL4,EL5)
G3 = GROUP(EL3,EL4)
G4 = GROUP(EL1)
```




The allowed communications are as follows:[4]

| *An event in:* | *May enable any event in:* |
|---|---|
| EL1 | EL1,EL6 |
| EL2 | EL2,EL3,EL6 |
| EL3 | EL2,EL3,EL4,EL6 |
| EL4 | EL3,EL4,EL5,EL6 |
| EL5 | EL4,EL5,EL6 |
| EL6 | EL6 |

Note how G1, G2, and G4 could be representations of processes, with G3 modeling a message channel between G1 and G2, and EL6 representing a resource shared by all three processes.

Certain events may be designated as *port events,* and serve as "access holes" to their groups. Using ports, we could model the concept of *data abstraction* as follows:

```
Abstraction = GROUP(Datum1,...,DatumN,
                    Oper1,...,OperM)
PORTS(Oper1.Start,...,OperM Start).
```

Events occurring at elements outside of Abstraction may directly access only events of the form Operi.Start, not those at some Datumj.

Groups are a very flexible structuring device. They may be disjoint, hierarchical, or overlapping,

---

[4] Let e@EL denote the fact that event e belongs to element EL. Given that e1@EL1 and e2@EL2, e1 can enable e2 if access(EL1,EL2) ∨ [e2 is a port of G ∧ access(EL1,G)]. We define access(x,y) to be true if either 1) x and y belong to the same group or 2) y is global to x. If we assume that all elements and groups within a specification are enclosed within a single surrounding group, we can define access(x,y) as follows:

```
access(x,y) ≡ (∃ G)[Y∈G ∧ contained(X,G)]
```

where Y∈G denotes direct membership and

```
contained(X,G) ≡ X∈G ∨
   (∃ G')[X∈G' ∧ contained(G',G)]
```

and may be created, deleted, or changed dynamically.[5] As a result, a variety process intercommunication structures can be modeled. Like element specifications, each group specification may be associated with a set of restrictions upon its events.

## 5. Event Relations

We have already seen how elements and groups can be used to restrict the relationships between events. In this section, we examine the enable relation, element order, and temporal order in greater detail.

The *enable relation* $\rightsquigarrow$ models the passing of control between actions during an execution. For instance, in a computation involving message passing, a message send can *enable* a message receipt. A process fork can *enable the* first event in each forked process. Consecutive events within a process *enable* one another.

The enable relation may also be used as a vehicle for describing data transfer. For example, a GEM restriction could specify "message passing" between two events **send** and **receive** as follows:

***If* send** *enables* **receive**, *then their parameters must be equal.*

```
send ⤳ receive ⊃
send.par1 = receive.par2
```

The enable relation is partial, irreflexive, and *not transitive*.[6]

As stated before, each event must occur at exactly one element. If we use the notation **occurred(e)** to assert the occurrence of e and **e@EL** to represent the fact that event e occurred at element **EL**, we have:

```
occurred(e) ⊃ (∃ ! EL)[ e@EL ]
```

All events belonging to the same element must be totally ordered by the element order $=\!\blacktriangleright$.

```
(e1@EL∧e2@EL) ⊃ (e1=▶e2)∨(e2=▶e1)
```

In addition, two events belonging to different elements *cannot* be related by $=\!\blacktriangleright$.

```
(e1=▶e2) ∨ (e2=▶e1) ⊃
(∃ EL)[(e1@EL ∧ e2@EL)]
```

The element order is partial, irreflexive and transitive.

To clarify the difference between the enable relation and the element order, consider the following. Suppose two processes **Proc1** and **Proc2** communicate solely through a variable **Var.** Let **assign1** and **assign2** be two events representing assignments to **Var.** Event **assign1** represents an assignment to **Var** within **Proc1**, and **assign2**, an assignment within **Proc2.** Although **assign2** may follow assign1 in the element order (**assign1=▶assign2**), **assign1** and **assign2** have nothing to do with each other as far as execution control flow is concerned: (**¬assign1⤳assign2**). In fact, we would have **assign1⤳assign2** only if **assign1** and **assign2** were successive assignments to **Var** belonging to the same process.

Both **e1⤳e2** and **e1=▶e2** imply that **e1** temporally precedes **e2.** Actually, the temporal relationships implied by $\rightsquigarrow$ and $=\!\blacktriangleright$ are the only observable temporal relationships within a distributed computation. Hence we define the third order, the ***temporal*** *order* $\Rightarrow$, to be exactly the transitive closure of $\rightsquigarrow$ and $=\!\blacktriangleright$ (minus identity). Because a GEM computation models concurrent action, the temporal order is necessarily partial. For instance, suppose el and e2 are events belonging to distinct, non-interacting processes. In this case, **e1** and **e2** are potentially concurrent:

```
¬[(e1⇒e2) ∨ (e2⇒e1)]
```

---

[5]Computations grow monotonically, even in the presence of dynamic group structures. This is because changes to group structure are represented as events. A complete discussion **of GEM** dynamic group structures may be found in [17]. The **reader** should also be aware that the scoping rules established by groups can be subverted by passing the identities of events (continuations) as data parameters.

[6]This lack of transitivity results from a need to describe direct event causality. This is required, for instance, in order to express restrictions such as "each message send can enable only one message receipt".

## 6. GEM Type Descriptions

Suppose we wished to specify a computation with many variables,. processes, procedures, etc. As one might cxpcct, the descriptions of similar objects are likely to bc similar, if not identical. Instead of including a separate complete specification of, say, each variable, it would bc convenient to describe just once those qualities which all variables hold in common. Given this generic variable description, each variable may be described by a list of differences from the specified norm.

To achieve this goal, GEM includes a flexible type description facility. Group and clement types may be declared. Types may bc parameterized as well as defined as refinements of other types. Each instance of a given type is an element or group with a structure identical to that of its type description, except for any explicitly mentioned differences. Semantically, the GEM type system may be viewed as a simple text substitution facility.

As a simple illustration of GEM type descriptions, let us return to our description of an integer variable Var. We might form an IntegerVariable element type description:

```
IntegerVariable = ELEMENT TYPE
EVENTS
  Assign(newval:INTEGER)
  Getval(oldval:INTEGER)
RESTRICTIONS
   .
END IntegerVariable
```

**Var**, as well as any other integer variable, may then be described as an instance of type IntegerVariable:

```
Var = IntegerVariable ELEMENT
```

Alternatively, we might describe a generic *untyped* variable element type. A typed variable element type could be described as a refinement of this more basic variable description. The construction is as follows:

```
Variable = ELEMENT TYPE
EVENTS
  Assign(newval:VALUE)
  Getval(oldval:VALUE)
RESTRICTIONS

END Variable
```

```
TypedVariable(t:TYPE) =
  Variable ELEMENT TYPE
/ADD RESTRICTION
(∀ assign:Assign)[TYPE(assign.newval)=t]

Var = TypedVariable(INTEGER) ELEMENT
```

## 7. Histories

A GEM computation represents a complete execution. Sometimes, however, we need to be able to talk about particular points in the progress of a computation. We call such points ***histories.*** A history contains information describing "what has happened so far." If a history a belongs to a computation C, it is simply a *prefix* of that computation. Namely, $\alpha$ consists of a subset of the events belonging to C and the partial orders and relations between those events. All predecessors of an event within a history must belong to that history: $\alpha \models [\text{occurred}(e2) \wedge e1 \Rightarrow e2)] \supset \alpha \models \text{occurred}(e1)$. We denote the prefix relationship between two histories, or between a history and a computation by $\alpha0 \subset \alpha1$ or $\alpha \subset C$, respectively.

Equipped with the notion of histories, we define a ***valid history sequence (vhs)*** as a set of histories that has the following properties:

$S = \alpha0, \alpha1, \ldots$ is a valid history sequence iff

1. *The sequence is monotonically increasing:*
   $\alpha0 \subset \alpha1 \subset \ldots$

2. *Two events occur for the first time in the same history only if they are potentially concurrent:*
   $(\forall\ \alpha i \in S,\ i \geq 1)\ (\forall\ ei, ej \in \{\alpha i - \alpha i\text{-}1\})\ [\neg\ ei \Rightarrow ej]$

For example, suppose we had a computation of the following form:

```
   ⇒ e2 ⇒
e1          e4
   ⇒ e3 ⇒
```

The possible histories are:

```
α0: e1
a1: e1,e2
a2: e1,e3
a3: e1,e2,e3
a4: e1,e2,e3,e4
```

The possible valid history sequences for this computation include:

$\alpha0, \alpha1, \alpha3, \alpha4$
$\alpha0, \alpha2, \alpha3, \alpha4$
$\alpha0, \alpha3, \alpha4$

Notice that the last history sequence listed above describes an execution in which e2 and e3 occur "at the same time."

History sequences have the *tail closure properly:* ifs is a vhs, then any tail of S is a vhs.

$$S=\alpha0,\alpha1,\ldots\alpha i-1,\alpha i,\alpha i+1,\ldots \text{ is a vhs } \supset S[i] = \alpha i,\alpha i+1,\ldots \text{ is a vhs}$$

One way of viewing a GEM computation is as the set of all of its valid history sequences.

Given these definitions, we may apply GEM restrictions not only to specific histories or to the entire computation (*immediate assertions*), but also to sequences of histories *(temporal assertions)*. Specifically, we follow the formulation of temporal logic as defined in [22]. For immediate assertions P, we may assert $\alpha \models P$ (P is true of history *a).* If we say an immediate assertion P is true of a history sequence S, we mean that it is true of the first history in that sequence: $S\models P \equiv \alpha0 \models P$.

We may now define the temporal operators *henceforth (Cl)* and *eventually (0)* as follows:

P is *henceforth* true for sequence S if it is true of every tail sequence of S. If P is an immediate assertion, this means that P is true for every history in S.

$$S \models \Box P \quad \equiv (\forall i \geq 0)\ S[i] \models P$$

P is *eventually* true for sequence S if it is true of some taii sequence of S. If P is an immediate assertion, this means that P is true for some history in S.

$$S \models \Diamond P \quad \equiv (\exists i \geq 0)\ S[i] \models P$$

We shall demonstrate how the Cl operator is used to describe safety properties such as priority. The 0 operator is used to describe liveness properties, such as progress.

## 8. GEM Restrictions

As we have seen, a specification $\sigma$ consists of a set of clement and group descriptions, each of which is associated with a set of implicit legality restrictions about the enable relation and the element order. Each element or group may also be associated with a set of explicit restrictions upon its events. In this section, we examine ways in which these restrictions are formulated.

### 8.1. GEM Predicates

Among the predicates which may be applied to GEM computations and histories are the following:[7]

| | |
|---|---|
| **occurred(e)** | *Event* e *occurred.* |
| **e @ EL** | *Event* e *occurred at element* **EL** |
| **e1 ↝ e2** | *e1 precedes e2 in the enable relation* |
| **e1 =▸ e2** | *e1 precedes e2 in the element order* |
| **e1 ⇒ e2** | *e1 precedes e2 in the temporal order* |

### 8.2. Restriction Abbreviations

Restrictions are first-order logic formulae composed of GEM predicates, the two temporal operators Cl and' 0 , and equality ( = ) between events, groups, and event data. A typical example of a restriction utilizing =▸ is given below. It is part of the Variable ELEMENT description and states that a value retrieval event, Getval, must yield the value last assigned to Variable.

```
Variable = ELEMENT TYPE
EVENTS
   Assign(newval:VALUE)
   Getval(oldval:VALUE)
RESTRICTIONS
1) (∀ assign:Assign, getval:Getval)
   [(assign=▸getval) ∧ ¬(∃ assign':Assign)
    [assign=▸assign'=▸getval]] ⊃
       assign.newval = getval.oldval
END Variable
```

[7] We shall use the notation e : E to state that e is an event belonging to the event class described by E. In general, we use lower case event names to denote specific events. and capitalized names to denote event classes.

In writing specifications, 'many restrictions arise repeatedly. When these restrictions are complicated, it is useful to abbreviate them with some operator or predicate. Each such abbreviation represents a common computational pattern within concurrent systems. Among the abbreviations commonly used in GEM are the following:

*1. E1 is a prerequisite to E2*

```
E1 → E2 ≡
(∀ e2:E2)[occurred(e2) ⊃
          (∃ ! e1:E1)[e1↝e2]]
∧(∀ e1:E1)(∃ at most one e2:E2)[e1↝e2]
```

An event class **E1** is a prerequisite to an event class **E2** if every event e2 must be enabled by one event e1, and each event e1 can enable only one event e2. For example, if a sequential piece of code consists of actions E1, E2, E3, and E4, we would have restriction E1 → E2 → E3 → E4. In the Monitor primitive, **Release** of a wait upon a condition must be enabled by exactly one **Signal**, and every **Signal** can enable only one **Release**: **Signal → Release.**

*2. Nondeteministic prerequisite of E*

```
{Event Class Set} →+ E ≡
 (∀ e:E)[occurred(e) ⊃
 (∃ ! e':{Event Class Set})[e'↝e]]
∧
 (∀ e':{Event Class Set})
 (∃ at most one e:E) [e'↝e]
```

**{Event Class Set}** is a nondeterministic prerequisite to event class **E** if every event e must be enabled by a single event belonging to one of the event classes in the set, and each event of a class belonging to the event class set can enable only one event e. For example, suppose we modeled CSP input and output statements as follows:[8]

```
! = ELEMENT TYPE
EVENTS
   Req({par}:SET OF VALUE)
   End()
RESTRICTIONS
1) Req → End
END !

? = ELEMENT TYPE
EVENTS
   Req()
   End({par}:SET OF VALUE)
RESTRICTIONS
1) Req → End
END ?

outP = ! ELEMENT
inpP = ? ELEMENT
```

Suppose inpP and outP were input and output elements belonging to process **P**. Given a specific CSP program containing **P**, we could formally characterize two sets of events, **inputset(inpP)** and **outputset(outP)**:

**inputset(inpP):**
Set of **!.Req** events belonging to elements which could legally send data to **inpP**

**outputset(outP):**
Set of **?.Req** events belonging to elements which could legally receive data from **outP**

We would then add the restrictions

```
inputset(inpP) →+ inpP.End
outputset(outP) →+ outP.End
```

Simultaneity of I/O exchange in CSP is represented by the restriction

```
(∀ inp:?, out:!)
[inp.req↝out.end ⇄ out.req↝inp.end]
```

*3. Event FORK*

```
E → {Event Class Set} ≡
  (∀ Ei∈{Event Class Set})[E → Ei]
```

*Event JOIN*

```
{Event Class Set} → E ≡
  (∀ Ei∈{Event Class Set})[Ei → E]
```

[8] Due to the limited scope of this paper, these descriptions are quite simplistic. Actual input and output element specifications would incorporate the identity of the desired partner for communication.

*4. Intermediate control points*

$$e1 \text{ at } E2 \equiv \text{occurred}(e1) \wedge \neg(\exists\, e2{:}E2)[e1 \rightsquigarrow e2]$$

e1 at E2 is true if e1 has not enabled an event of class E2.

$$new(e) \equiv \text{occurred}(e) \wedge \neg(\exists\, e')[e \Rightarrow e']$$

**new** ( e ) is true if no event has observably followed e.

### 8.3. GEM Thread Notation

Equipped with the descriptive tools we have presented so far, we may begin to build a specification of the Reader's Priority version of the Readers and Writers problem. We begin with a specification of the following form:

```
User = ELEMENT TYPE
EVENTS
   Read(loc:1..N)
   FinishRead(info: VALUE)
   Write(loc:1..N, info: VALUE)
   FinishWrite
RESTRICTIONS

END User

DataBase = GROUP TYPE(control:RWControl,
        {data[loc:1..N]}:SET OF Variable)

RWControl = ELEMENT TYPE
EVENTS
   ReqRead(loc:1..N)
   StartRead(loc:1..N)
   EndRead(info: VALUE)
   ReqWrite(loc:1..N, info: VALUE)
   StartWrite(loc:1..N, info:VALUE)
   EndWrite
RESTRICTIONS

END RWControl

RWProblem = GROUP (db:DataBase,
                  {u}:SET OF User)
RESTRICTIONS
1) u.Read(loc) →
   db.control.ReqRead(loc) →
   db.control.StartRead(loc) →
   db.data[loc].Getval(info) →
   db.control.EndRead(info) →
   u.FinishRead(info)
2) u.Write(loc, info) →
   db.control.ReqWrite(loc, info) →
   db.control.StartWrite(loc,info) →
   db.data[loc].Assign(info) →
   db.control.EndWrite →
   u.FinishWrite
END RWProblem
```

Each **user** call **to Read** or **Write** results in a request, start of operation, fulfillment of operation, end of operation, and return to the user.

In order to define various forms of priority as well as the mutual exclusion property, we need to be able to label those events which occur as a result of each user request. These labels can then be used to distinguish between events occurring due to different requests. GEM's thread mechanism was devised precisely for this task. As stated earlier, the enable relation models how control is passed between events. **A** ***thread*** is an identifier associated with a *chain* of enabled events of a particular specified form. Each thread may be thought of as defining a sequential process. For the Readers/Writers problem, we define a thread to correspond to the actions occurring within a specific Readers/Writers database transaction. Using a *path expessiorz*-like notation [4], we define thread type $\pi$RW as follows:

```
πRW = (u.Read::db.control.ReqRead::
         db.control.StartRead::
         db.data[loc].Getval::
         db.control.EndRead::
         u.FinishRead
      |
       u.Write::db.control.ReqWrite::
         db.control.StartWrite::
         db.data[loc].Assign::
         db.control.EndWrite::
         u.FinishWrite)
```

By including this thread description within **RWSolution,** we assert the following restrictions upon any legal Readers/Writers computation:

- A unique $\pi$RW thread identifier is created and associated with each **Read** or Write event that occurs.
- This identifier is "passed" along its related control path, as long as events enable one another in the order prescribed, until **a** FinishRead or FinishWrite occurs.

The notation **e** $\pi$RW-i indicates that event e is part of a thread instance $\pi$RW-i of type $\pi$RW.

To complete the Readers and Writers specification, we proceed as follows. First, we augment element `RWControl` with the following restriction describing the "writers exclude others" property. The first clause states that writers exclude readers, and the second, that writers exclude other writers.

**Mutual Exclusion Restriction:**

```
¬(∃ startread πRW-i:StartRead,
    startwrite πRW-j:StartWrite)
  [occurred(startread πRW-i) ∧
  ¬occurred(endread πRW-i) ∧
   occurred(startwrite πRW-j) ∧
  ¬occurred(endwrite πRW-j)]
∧
¬(∃ startwrite πRW-i:StartWrite,
    startwrite πRW-j:StartWrite)
  [occurred(startwrite πRW-i) ∧
  ¬occurred(endwrite πRW-i) ∧
   occurred(startwrite πRW-j) ∧
  ¬occurred(endwrite πRW-j)]
```

where `πRW-i` and `πRW-j` are two distinct `πRW` thread identifiers.

GEM is also able to express priority properties easily. Reader's priority for the Readers/Writers problem may be described as follows:

*If a request for a read and a write are pending at the satne time, the read must be serviced before the write.*

```
[reqread πRW-i at StartRead ∧
 reqwrite πRW-j at StartWrite] ⊃
□ [occurred(startwrite πRW-j) ⊃
   occurred(startread πRW-i)]
```

### 8.4. History vs. Current State

Because a GEM computation contains information about all events that have occurred, restrictions may refer to the *entire* past of a computation or history, not just the current value of variables *(current slate).* Description and verification methods which rely upon variable values to describe state are limited in the sense that variables *abbreviate* state -- *they* cannot easily provide information about all that has occurred in the past. Even if auxiliary variables are used, the specification of many propertics becomes unwieldy. For example, a state-oriented description of reader's priority would have to encode the ordering of event occurrences in terms of auxiliary program variables.

The GEM priority restriction is rather simply and intuitively stated in terms of the event occurrence information available within GEM histories, augmented with the use of temporal operators.

The possibility of utilizing complete computational information in GEM specifications often allows a more intuitive and general description of computational propertics than state-based methods. A GEM problem description is a representation of a problem that may be applied to the analyses of programs written in any language. In contrast, assertion or state-oriented methods of specification usually describe a problem in terms of the variable values of a specific program implementing that problem.

## 9. GEM Verification Methodology

GEM computations contain information about the order of event occurrence, as well as information about their associated data values. For example, associated with each **Assign** event belonging to a **Variable** element is the value assigned. It is not hard to see how proof techniques which make use of assertions about variable values could be applied to GEM computations. Moreover, GEM's temporal operators allow one to describe liveness properties such as progress.[9] Thus, temporal logic methods can be applied to GEM computations as well. Despite this, we have found it useful to develop an alternate proof methodology that is especially well-suited to GEM. Because GEM problem specifications are often expressed in terms of event orders and relations, it makes sense to utilize a proof method geared towards reasoning about these relationships between events. We briefly outline this method below.

Suppose we wish to prove that a given program solves a specific problem. Our first step would be to describe that problem by a GEM problem specification ***P***. For example, we have already described a specification for the Readers and Writers problem in the preceding section. Next, we would describe the program by formulating a GEM

[9] An example of a weak progress requirement is the following one: if all the prerequisites of an event e are fulfilled in history $\alpha$, and e could legally extend history a. and this remains true until e actually occurs, then e must eventually occur. A stronger progress requirement would guarantee e to occur if there is an infinite subsequence of histories in which it could legally enter the computation.

program specification *PROG.* Program specifications are generated by instantiating the groups and elements which make up GEM's description of the underlying implementation language.

For example, suppose we were given the following monitor as part of the implementation of the Reader's Priority Readers/Writers problem (the data itself must be located outside of the monitor):

```
ReadersWriters: MONITOR
BEGIN
   readqueue, writequeue:CONDITION;
   readernum INTEGER;
  /*readernum is positive if reading,
    negative if writing*/

ENTRY PROCEDURE StartRead;
BEGIN
IF readernum < 0 THEN
   WAIT(readqueue);
readernum := readernum + 1;
SIGNAL(readqueue):
END;

ENTRY PROCEDURE EndRead;
BEGIN
readernum := readernum - 1;
IF readernum = 0 THEN
   SIGNAL(writequeue);
END;

ENTRY PROCEDURE StartWrite;
BEGIN
IF readernum ≠ 0 THEN
   WAIT(writequeue):
readernum := -1:
END;

ENTRY PROCEDURE EndWrite;
BEGIN
readernum := 0:
IF queue(readqueue) THEN
   SIGNAL(readqueue)
ELSE
   SIGNAL(writequeue);
END;

/*initialization*/
readernum := 0;
END ReadersWriters
```

In addition, we are given the following GEM specification of the Monitor primitive itself:

```
Monitor = GROUP TYPE(lock:MonitorLock,
           {entry}:SET OF MonitorEntry,
           {cond}:SET OF Condition,
           init:Initialization,
           {var}:SET OF Variable)
          PORTS(lock.Req)
RESTRICTIONS
```

*Restrictions describing how a monitor functions. This would include rules for wailing and signalling, initialization, etc.*

```
END Monitor
```

where `MonitorLock`, `MonitorEntry`, `Condition`, Initialization, and `Variable` are previously defined element or group types.

The monitor program given above could then be described by the following GEM specification:

```
ReadersWriters = Monitor GROUP
   (RWLock, {StartRead,EndRead,
    StartWrite,EndWrite},
   {readqueue,writequeue},RWInit,
   {readernum})
```

Every restriction within `Monitor` applies to `ReadersWriters`. In addition, each component of `ReadersWriters` must also be fully specified. For instance, the `StartRead` `MonitorEntry` specification includes a description of the statements which belong to entry `StartRead`. Given an underlying language specification, translation of a program into a GEM program specification is quite simple and mechanical enough to lend itself to automation.

Finally, a proof is completed as follows:

1. For each group, element, event type, event parameter, and thread in *P*, choose a corresponding object in *PROG.* We call these *the significant* objects of *PROG.*
2. Prove that each restriction *Ri* in *P* is satisfied by the corresponding significant objects in *PROG.*

   $$(\forall Ri \in P)\ [PROG\ sat\ Ri]$$

   GEM includes various rules which define what is meant by sat. One intuitive explanation is the following: *If we examine a computation which is legal with respect to PROG, and only take note of significant objects, those significant objects exhibit the same behavior as a computation that is legal with respect to P.*

To illustrate this proof method, we present an

informal argument showing that reader's actually do get priority in the monitor solution presented above. First, we set up the following event correspondences:

| *PROBLEM* | *PROGRAM* |
|---|---|
| ReqRead | Entry StartRead:BEGIN |
| StartRead | EntryStartRead: readernum:=readernum+1 |
| EndRead | Entry EndRead: readernum:=readernum-1 |
| ReqWrite | EntryStartWrite:BEGIN |
| StartWrite | EntryStartWrite:readernum:=-1 |
| EndWrite | EntryEndWrite:readernum:=0 |

Assume Chat we have already proved Chat potential(startwrite) ⊃ **readernum** = 0[10] and **new(startread)** ⊃ **readernum** > **0.** We have also proved that all events occurring in monitor entries or initialization code arc totally ordered by the temporal order ⇒. An informal proof of the Reader's priority property follows.

ReadersPriority RESTRICTION:
**[reqread** $\pi$RW-i **at** StartRead ∧
**reqwrite** $\pi$RW-j **at** StartWrite] ⊃
□ [occurred(startwrite $\pi$RW-j) ⊃
occurred(startread $\pi$RW-i)]

Proof. By contradiction.

Lct S be a valid history sequence for which the ReadersPriority restriction does not hold. Then S must have histories $\alpha 0$ and $\alpha 3$ such that $\alpha 0 \subset \alpha 3$ and

$\alpha 0$ ⊨ **reqread** $\pi$RW-i **at** StartRead ∧
**reqwrite** $\pi$RW-j **at** StartWrite

$\alpha 3$ ⊨ occurred(startwrite $\pi$RW-j) ∧
¬ occurred(startread $\pi$RW-i)

[10]An event is potential if it has not occurred, but all of its prerequisites have been fulfilled. The expression "Var = N" is an abbreviation for

(∃assign1(N):Var.Assign)[occurred(assign1(N)) ∧
¬(∃assign2:Var.Assign)[assign1(N)⇒assign2]]

If wc choose $\alpha 3$ to bc the smallest history having this property, then wc additionally know that $\alpha 3$ ⊨ **new(startwrite** $\pi$RW-j).

By examining the code of entry StartRead (keeping in mind that execution of monitor events is sequential), we see that the process executing thread $\pi$RW-i must have waited on condition **readqueue,** giving us

**reqread** $\pi$RW-i⇒**wait** $\pi$RW-i(readqueue)⇒
**startwrite** $\pi$RW-j

If wc choose al as the smallest history containing this **wait** event, then we have al ⊨ **new(wait** $\pi$RW-i(readqueue)), and thus al ⊨ readernum<0, and $\alpha 0 \subset \alpha 1 \subset \alpha 3$. Now let a2 be a history such that $\alpha 1 \subset \alpha 2 \subset \alpha 3$ and $\alpha 2$ ⊨ potential(startwrite $\pi$RW-j). (Such an a2 exists because **startwrite** $\pi$RW-j must be **potential** before it occurs). Then a2 ⊨ readernum=0.

Since readernum<0 in al and readernum=0 in $\alpha 2$, there must be some assignment event **assignzero** at variable **readernum** which causes its value to change from less than 0 to 0. Moreover, since all assignments to **readernum** occur sequentially, we can choose **assignzero** to be the first such assignment. We also know that **assignzero** musteitherbe a StartRead event or **End Write** event (these are the only two events which could cause **readernum** to increase). Let $\beta 1$ and $\beta 0$ be histories such that

$\beta 1$ ⊨ new(assignzero)
$\beta 0$ ⊨ potential(assignzero)

Case 1.

Suppose that **assignzero** is of class StartRead. Because **new(startread)** ⊃ readernum>0, we must have readernum≥0 before execution of ass i **gnzero.** This means that $\beta 0$ ⊨ readernum≥0. But $\alpha 1 \subset \beta 0 \subset \alpha 2$, which contradicts the choice of ass ignzero as the first event to increase the value of **readernum to 0.** Contradiction

***Case 2.***

Suppose Chat **assignzero is** an EndWrite event. Then we have

```
wait πRW-i(readqueue)⇒endwrite⇒
startwrite πRW-j
```

Moreover, when this endwrite occurs, we know that at least one process is waiting on condition readqueue, namely the process which executed wait πRW-i(readqueue). Therefore, after endwrite, readqueue will get signalled. The code of entry StartRead guarantees that all waiting readers will be signalled before any other process executes in the monitor. Sequential execution within the monitor then implies:

```
reqread πRW-i⇒endwrite⇒
startread πRW-i⇒startwrite πRW-j
```

Contradiction ∎

## 10. Related Work

The model of computation most similar to GEM is the message-based *actor* model [6, 10]. Greif [6] has utilized actors to model the monitor primitive, and to analyze various aspects of concurrent execution. An actor system is composed of actors, the events occurring at actors, and various partial orders on these events. However, an actor event is a message receipt, rather than an arbitrary logical action. Moreover, each event may be activated by only a single predecessor. Dynamic process creation and flexible forms of intercommunication can be described by actor systems. However, scope rules are not modeled specifically, nor is there any means of naming *particular* chains of control. Nor do actor systems incorporate the use of temporal logic. In addition, the actor model has not been used as a tool for general purpose language description or program verification.

Another event-oriented model related to GEM is Lamport's [15, 16]. In this model, as in ours, an event is a logical action at any desired level of detail. However, non-atomic events are also considered. Lamport has used his model as a framework for solving such problems as the synchronization of distributed clocks.

Several other models have been used to describe concurrency, although they are fairly different from GEM. A detailed review of theoretical models of concurrency may be found in [18]. *Petri Nets* [23] are a good representative of the class of *transition models.* They have been widely used and studied extensively, but are static in structure and not expressive enough to easily illustrate data-dependent properties. Another class of models may be described *as algebraic* or *functional.* Such models usually focus on the input-output sequences of a set of processes or modules, which may be combined algebraically. Milner and Milne's *Communicating Behaviors* [19, 20], Kahn and MacQueen's *Stream Processing Networks* [13, 14], and Pratt's *Process Model* [26] are in this category. Models in this class are usually used for describing problems rather than languages.

Some descriptive methods have been designed specifically for application to languages. *Denotational semantics* [28] has been used to provide functional descriptions of many languages and problems, but has some difficulty with concurrency. Adaptations of denotational methods for concurrent applications have been explored [24, 27] but are not easy to use.

Methods for comparing or analyzing concurrent language structures are not widespread. Guarino-Reid's model [7] is formulated as a special-purpose language made up of modules interconnected by ports, and is used to describe various "abstract communication constructs." A method for evaluating language concurrency primitives is described by Bloom [3].

Much research has been devoted to the verification of concurrent programs. Assertion-oriented methods have been extended to deal with concurrency [21]. Temporal logic has been used successfully, especially in the verification of communications protocols [8, 22, 25]. Most verification techniques certify program correctness by proving assertions about program state cast in terms of variable values. A. specialized, event-oriented method of verification for Serializers is utilized by Atkinson's automatic verifier [2].

## 11. Conclusion

We have described GEM, a model of concurrent computation which is simple and general enough to describe a variety of language concurrency primitives and problems. GEM specifications include information about the structure of a language or problem as well as its intended semantics. Structure 'is described by stating how certain events occur at specific elements, and how those elements are clustered into groups. Language or problem behavior is described by restricting event occurrences to certain partial orders and relations. Specifieations are often stated as synchronization

properties, rather than invariant statements about variable values.

GEM has already been used to describe three very different language primitives: the Monitor, a primitive based on communication via shared data; CSP, which consists of processes communicating via messages; and ADA's tasking mechanism, which uses the rendezvous for interprocess communication. GEM has also been used to specify the One-Slot and Bounded Buffer problems, five versions of the Readers/Writers problem, and two distributed applications: an algorithm for updating a distributed database and an asynchronous version of the Game of Life.

GEM can also be used as a verification tool. Various properties of the Monitor have been proved such as sequential execution of monitor entries. Monitor, CSP, and ADA solutions to the One-Slot Buffer, Bounded Buffer, and Reader's Priority Readers/Writers problems have been verified. Properties such as progress and functional correctness have been proved of the two distributed problems mentioned above.

## Acknowledgments

Russ Atkinson had a great influence upon the formulation of GEM. Others who have helped by reading drafts of this paper are Pierre Wolper, Keith Marzullo, Steven Rubin, and Polle Zellweger.

## References

1. *Reference Manual for the AD A Programming Language.* United States Department of Defense, 1980.

2. Atkinson,R. Automatic Verification of Serializers. Tech. Rept. TR-229, MIT Laboratory for Computer Science, March, 1980.

3. Bloom,T. Synchronization Mechanisms for Modular Programming Languages. Tech. Rept. TR-211, MIT Laboratory for Computer Science, Jan., 1979.

4. Campbell,R.H. and Habermann,A.N. The Specification of Process Synchronization by Path Expressions. In *Lecture Notes in Computer Science 16*, Springer-Verlag, 1974.

5. Ellis,C.A. Consistency and Correctness of Duplicate Database Systems. Proceedings of the 6th Annual Symposium on Operating System Principles, ACM, Nov., 1977, pp. 67-84.

6. Greif,I. Semantics of Communicating Parallel Processes. Tech. Rept. TR-154, MIT Project MAC, sep t., 1975.

7. Guarino-Reid,L. Control and Communication in Programmed Systems. Tech. Rept. CMU-CS-80-142, Carnegie-Mellon University, Dept. of Computer Science, Sept., 1980.

8. Hailpem,B.T. Verifying Concurrent Processes Using Temporal Logic. Tech. Rept. 195, Computer Systems Laboratory, Stanford University, Aug., 1980.

9. Hansen,P.B.. *Operating System Principles.* Prentice Hall, Englewood Cliffs, New Jersey, 1973.

10. Hewitt,C. and Baker H. Jr. Laws for Communicating Parallel Processes. In *IFIP 77*, Gilchrist,B., Ed.,North-Holland, Amsterdam, 1977, pp. 987-992.

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

12. Hoare,C.A.R. "Communicating Sequential Processes." *Comm. ACM 21*, 8 (Aug. 1978), 666-677.

13. Kahn,G. The Semantics of a Simple Language for Parallel Programming. In *IFIP 74*, North-Holland, Amsterdam, 1974.

14. Kahn,G. and MacQueen,D.B. Coroutines and Networks of Parallel Processes. In *IFIP 77*, Gilchrist,B., Ed.,North-Holland, Amsterdam, 1977, pp. 993-998.

15. Lamport,L. "Times, Clocks, and the Ordering of Events in a Distributed System." *Comm. ACM 21*, 7 (July 1978).

16. Lamport,L. "A New Approach to Proving the Correctness of Multiprocess Programs." *ACM-TOPLAS 1*, 1 (July 1979).

17. Lansky, A. Specification and Analysis of Concurrency. Department of Computer Science, Stanford University. Ph.D. thesis (to appear).

18. MacQueen, D.B. Models for Distributed Computing. Tech. Rept. 351, INRIA - Paris, France, April, 1979.

19. Milne, G. and Milner, R. Concurrent Processes and Their Syntax. Tech. Rept. CSR-2-77, University of Edinburgh, Department of Computer Science, May, 1977.

20. Milner, R. Synthesis of Communicating Behavior. 7th Symposium on Mathematical Foundations of Computer Science, Lecture Notes in Computer Science, Springer Verlag, Zakopane, Poland, Sept., 1978.

21. Owicki, S. and Gries, D. "Verifying Properties of Parallel Programs: An Axiomatic Approach." *Comm. ACM 19*, 5 (May 1976), 279-285.

22. Owicki, S., and Lamport, L. '*Proving Liveness Properties of Concurrent Programs." ACM *Transactions on Programming Languages and Systems 4*, 3 (July 1982), 455-495.

23. Peterson, J.L. "Petri *Nets." ACM Compufing Surveys 9*, 3 (Sept. 1977), 221-252.

24. Plotkin, G.D. "A Power Domain Construction." *SIAM Journal on Computing 5* (1976), 452-487.

25. Pnueli, A. The Temporal Logic of Programs. 18th Annual Symposium on Foundations of Computer Science, IEEE, Oct., 1977, pp. 46-57.

26. Pratt, V.R. On the Composition of Processes. Ninth Annual ACM Symposium on the Principles of Programming Languages, ACM, Jan., 1982, pp. 213-223.

27. Smyth, M.B. "Power Domains." *Journal of Computer and System Sciences 16*, 1 (1978), 23-36.

28. Stoy, J.E.. *Denotational Semantics= The Scott-Strachey Approach to Programming Language Theory.* MIT Press, 1377.

29. Winkowski, J. Game of Life as a Synchronous Process That Can Be Realized Asynchronously. Tech. Rept. 286, Institute of Computer Science, Polish Academy of Sciences, Warsaw, Poland, 1977.