Gem: A Tool for Concurrency Specification and Verification

Amy L. Lansky
Susan S. Owicki


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 Concurrent Specification and Verification

Amy L. Lansky
Susan S. Owicki


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 described 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. Lanksy, 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 (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.

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.
Sections 2-8 describe GEM’s method of representing concurrent execution and show how problem, language, and program specifications are formed. The GEM verification method is described in section 9. Finally, an overview of related work is presented in section 10.

2. Concurrency: An Intuitive View

Before we embark on a detailed description of how GEM models concurrent execution, let us examine concurrency on a more intuitive level. When we think of execution, we think of actions occurring in time 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 present, 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 subtrees of activity, each of which is unrelated to the other.

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

We have now considered the role played by the enabling relationship between actions. What role does locus of activity play? We have been assuming that each action has its own locus of activity. There are, however, sets of actions which logically occur at the same location. Consider, for example, a single computer storage cell. In most models of computer execution, all actions (reads and writes) which take place at the same memory location are forced to be totally ordered in time. Although these actions may be causally unconnected (there is no enabling relationship between them), they must occur in some sequential order. For instance, such a situation arises when a memory location is shared between several logical processes of activity.

Thus there are activities which are 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 related semantically as well as by-the-enabling relationship. Semantic groupings are also often tied to the traditional language concept of scope. For instance, the actions of two different processes might be clustered into two separate groups. Because each process is not allowed to access the variables belonging to the other other process, it becomes natural to associate the two groups with a “firewall” which limits the enabling relationships between them.

Given this intuitive discussion of concurrent execution, we shall now proceed to explain GEM’s model of execution in a more formal way. GEM represents the actions within a concurrent execution by objects called events. The enabling relationship between actions is modeled by the enable relation \( \triangleright \). Loci of forced sequential activity are modeled...
by elements, and this forced sequential order is called the *element order* := . . . The *temporal order* \( \Rightarrow \) between actions (the observable order in time) is simply the transitive closure of \( \Rightarrow \) and \( \Rightarrow \) (minus identity). Semantically related loci of activity are modeled by GEM groups.

3. Computations and Specifications

GEM is made up of two components: a model of execution and a specification language. A GEM computation is an abstract representation of concurrent execution. Each computation consists of a possibly infinite set of objects called events, a partial relation \( \triangleright \) (the enable relation), and two strict partial orders: \( \Rightarrow \) (the *element order*), and \( \Rightarrow \) (the *temporal order*). Events may be 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 set of logic formulae (restrictions) over the domain of all possible GEM computations. A computation \( C \) is *legal* with respect to a specification \( \sigma(\text{legal}(C,\sigma)) \) 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 the database update problem is one that obeys all restrictions specifying that problem.

As one might expect, there are certain properties that must be true of all legal computations. These properties are described by a set of GEM legality restrictions which are automatically part of any GEM specification. For example, the following must be true of all legal GEM computations: the temporal order between events is a transitive and irreflexive order, and must be equivalent to the transitive closure of the enable relation and the element order (minus identity). The way in which a computation's events are clustered into elements and groups imposes legality restrictions on the element order and enable relation.

Each GEM legality restriction has a formal characterization. In this paper, however, we will present these properties informally, as we describe the basic conceptual components of GEM.\(^2\)

4. Events, Elements, and Groups

A GEM event represents a logical action that is regarded as atomic relative to other events in its computation. For example, an assignment statement, the forking of a process, and completion of an input request may be modeled as events. An event may be viewed as a structured object composed of the following kinds of information: data parameters, thread identifiers (see section 8.3), and the names of the element and group sets to which it belongs. Each event is a unique occurrence in a computation, with its own identity and associated data. A set of similar events may be characterized by an *eventclass* description.

As already stated, a GEM specification is used to restrict the set of all possible computations to those that represent meaningful executions for a particular language or problem. The first way in which this is done is to identify which events may occur. Specifically, the events which may legally occur within a computation are those belonging to a specified list of elements: if \( \text{legal}(C,\sigma) \) and event \( e \) is in \( C \), then \( e \) must belong to some element specified in \( \sigma \).

Elements model the elementary components of a language or problem whose associated actions must for some reason, occur sequentially. In this sense, they are similar to actors[6,10].

Each element is a unique entity and is characterized 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 element. All of the events occurring at the same element must be totally ordered by the element order, \( \Rightarrow \). Each element may be associated with an explicit list of restrictions upon its events.

Let us look at an example. Suppose we wished to model an integer variable \( \text{Var} \). Two types of events can be associated with a variable, \( \text{Ass} \) and \( \text{Getval} \). In addition, \( \text{WC} \) wish to assert that events of type \( \text{Var}\_\text{Assign} \) and \( \text{Var}\_\text{Getval} \) occur sequentially, whether they are causally related or not. In essence, \( \text{WC} \) are asserting a lock on access to variable \( \text{Var} \). \( \text{WC} \) could write the following GEM specification of \( \text{Var} \):

\[^3\]If \( \text{WC} \) do not wish to impose an element order on events, we can simply associate each event with its own element. In this limiting case, events are only related by the enable relation and temporal order.

---

2This paper is intended as a survey of GEM rather than a complete description. For further details refer to [17].
A specification which includes \texttt{Var} admits computations containing \texttt{Var.Assign} and \texttt{Var.Getval} events which obey the restriction:

\[
(V \ e_1, e_2 : \{\texttt{Var.Assign, Var.Getval}\}) \quad [ e_1 \Rightarrow e_2 \lor e_2 \Rightarrow e_1 \lor e_1 = e_2 ]
\]
as well as any additional restrictions explicitly associated with \texttt{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 \texttt{Var.assigni} (or simply \texttt{Var}^i) is the \texttt{i}th event at \texttt{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:

\textbf{ELEMENTS EL1,EL2,EL3,EL4,EL5,EL6}

\textbf{G1 = GROUP(EL2,EL3)}
\textbf{G2 = GROUP(EL4,EL5)}
\textbf{G3 = GROUP(EL3,EL4)}
\textbf{G4 = GROUP(EL1)}

The allowed communications are as follows:

An event in:\qquad May enable any event in:

\begin{align*}
\text{EL1} & \quad \text{EL1, EL6} \\
\text{EL2} & \quad \text{EL2, EL3, EL6} \\
\text{EL3} & \quad \text{EL2, EL3, EL4, EL6} \\
\text{EL4} & \quad \text{EL3, EL4, EL5, EL6} \\
\text{EL5} & \quad \text{EL4, EL5, EL6} \\
\text{EL6} & \quad \text{EL6}
\end{align*}

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

Certain events may be designated as \textit{port events}, and serve as “access holes” to their groups. Using ports, we could model the concept of data abstraction as follows:

\textbf{Abstraction = GROUP(Datum1,...,DatumN, Oper1,...,OperM)}
\textbf{PORTS(O per1.Start,...,OperM.Start)}.

Events occurring at elements outside of \textbf{Abstraction} may directly access only events of the form \texttt{Oper1.Start}, not those at some Datum.j.

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

---

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

\[
\text{access}(x,y) \equiv (3 \ G)[Y \in G \ A \ \text{contained}(X,G)]
\]

where \texttt{Y \in G} denotes direct membership and

\[
\text{contained}(X,G) \equiv X \in G \lor (3 \ G')[X \in G' \ A \ \text{contained}(G',G)]
\]
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 \( \Rightarrow \) 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.

\[
\text{send} \Rightarrow \text{receive} \quad \text{send.par1} = \text{receive.par2}
\]

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 \( \text{occurred}(e) \) to assert the occurrence of \( e \) and \( e \in \text{EL} \) to represent the fact that event \( e \) occurred at element \( \text{EL} \), we have:

\[
\text{occurred}(e) \subseteq (\exists ! \text{EL})(e \in \text{EL})
\]

All events belonging to the same element must be totally ordered by the element order = . . .

\[
(e1\Rightarrow e2) \vee (e2\Rightarrow e1)
\]

In addition, two events belonging to different elements cannot be related by = . . .

\[
(e1\Rightarrow e2) \vee (e2\Rightarrow e1) \subseteq (\exists \text{EL}[(e1\in \text{EL} \land e2\in \text{EL}])
\]

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 \( \text{Proc1} \) and \( \text{Proc2} \) communicate solely through a variable \( \text{Var} \). Let \( \text{assign1} \) and \( \text{assign2} \) be two events representing assignments to \( \text{Var} \). Event \( \text{assign1} \) represents an assignment to \( \text{Var} \) within \( \text{Proc1} \), and \( \text{assign2} \), an assignment within \( \text{Proc2} \). Although \( \text{assign1} \) and \( \text{assign2} \) have nothing to do with each other as far as execution control flow is concerned:

\[
(\neg \text{assign1}\Rightarrow \text{assign2}).
\]

In fact, we would have \( \text{assign1}\Rightarrow \text{assign2} \) only if \( \text{assign1} \) and \( \text{assign2} \) were successive assignments to \( \text{Var} \) belonging to the same process.

Both \( e1\Rightarrow e2 \) and \( e1\Rightarrow e2 \) imply that \( e1 \) temporally precedes \( e2 \). Actually, the temporal relationships implied by \( \Rightarrow \) and \( \Rightarrow \) 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 \( \Rightarrow \) and \( \Rightarrow \) (minus identity). Because a GEM computation models models concurrent action, the temporal order is necessarily partial. For instance, suppose \( e1 \) and \( e2 \) are events belonging to distinct, non-interacting processes. In this case, \( e1 \) and \( e2 \) are potentially concurrent:

\[
\neg[(e1\Rightarrow e2) \vee (e2\Rightarrow e1)]
\]

5 Computed grow monotonically, even in the presence of dynamic group structures. This is because changes to group structure arc 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 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”.

and may be created, deleted, or changed dynamically. 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.
6. GEM Type Descriptions

Suppose we wished to specify a computation with many variables, processes, procedures, etc. As one might expect, the descriptions of similar objects are likely to be similar, if not identical. Instead of including a separate complete specification, say, each variable, it would be 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 element types may be declared. Types may be 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 
(...)
```

```
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, a 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:

```
a \subseteq [\{e2 \mid e1 \rightarrow e2\}] \cap \
\{e \mid \text{occurred}(e)\}
```

We denote the prefix relationship between two histories, or between a history and a computation by \(a \subseteq c\) or \(a \subseteq C\), 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 = a0, a1, \ldots 
```

is a valid history sequence iff

1. The sequence is monotonically increasing:

```
a0 \subseteq a1 \subseteq \ldots
```

2. Two events occur for the first time in the same history only if they are potentially concurrent:

```
(\forall a_i \in S, i \geq 1) (\forall e_i, e_j \in \{a_i - a_i-1\}) 
\left(\neg e_i \Rightarrow e_j\right)
```

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

```
e1 \Rightarrow e2 \Rightarrow e4
```

The possible histories are:

```
a0: 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:

\[ \alpha_0, \alpha_1, \alpha_3, \alpha_4 \]
\[ \alpha_0, \alpha_2, \alpha_3, \alpha_4 \]
\[ \alpha_0, \alpha_3, \alpha_4 \]

Notice that the last history sequence listed above describes an execution in which \( e_2 \) and \( e_3 \) occur “at the same time.”

History sequences have the tail closure properly: if \( S \) is a vhs, then any tail of \( S \) is a vhs.

\[ S = \alpha_0, \ldots, \alpha_i-1, \alpha_i, \alpha_i+1, \ldots \] is a vhs \( \supset S[i] = \alpha_i, \alpha_i+1, \ldots \) is a vhs

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

Given these definitions, WC 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 \( \rho \), we may assert \( \alpha \models \rho \) if \( \rho \) is true of history \( \alpha \). If we say an immediate assertion \( \rho \) is true of a history sequence \( S \), we mean that it is true of the first history in that sequence: \( S \models \rho \iff \alpha_0 \models \rho \).

We may now define the temporal operators henceforth (\( \lozenge \)) and eventually (\( \Box \)) as follows:

\( \Box \rho \) is henceforth true for sequence \( S \) if it is true of every tail sequence of \( S \). If \( \rho \) is an immediate assertion, this means that \( \rho \) is true for every history in \( S \).

\[ S \models \Box \rho \iff (\forall i \geq 0) \ S[i] \models \rho \]

\( \rho \) is eventually true for sequence \( S \) if it is true of some tail sequence of \( S \). If \( \rho \) is an immediate assertion, this means that \( \rho \) is true for some history in \( S \).

\[ S \models \lozenge \rho \iff (\exists i \geq 0) \ S[i] \models \rho \]

WC shall demonstrate how the \( \lozenge \) operator is used to describe safety properties such as priority. The \( \Box \) 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 element 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:

\[ \text{occurred}(e) \] \( e \) occurred.
\[ e \in EL \] \( e \) occurred at \( \text{element} \in EL \)
\[ e \preceq e' \] \( e \) precedes \( e' \) in the enable relation
\[ e \preceq e' \] \( e \) precedes \( e' \) in the element order
\[ e \Rightarrow e' \] \( e \) precedes \( e' \) in the temporal order

8.2. Restriction Abbreviations

Restrictions are first-order logic formulae composed of GEM predicates, the two temporal operators \( \lozenge \) and \( \Box \), and equality (\( = \)) between events, groups, and event data. A typical example of a restriction utilizing \( \models \) is given below. It is part of the \text{Variable} description and \text{States} that \( a \) value retrieval event, \text{Getval}, must yield the value last assigned to \( a \).

\text{Variable} \( \equiv \text{ELEMENT TYPE} \)

\text{EVENTS}
\text{Assign(newval:VALUE)}
\text{Getval(oldval:VALUE)}

\text{RESTRICTIONS}

1. \( (\forall \text{assign:Assign, getval:Getval}) \neg (3 \text{assign':Assign}) \) \[ \neg (\text{assign'Getval}) \]
\text{assign}.newval = getval . oldval

\text{END Variable}

7We 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. **EI is a prerequisite to E2**

\[ EI \rightarrow E2 \equiv (\forall e2:E2)[ \text{occurred}(e2) \supset (\exists ! e1:EI)[e1 \rightarrow e2]] \wedge (\forall e1:EI)[(\exists \text{at most one } e2:E2)[e1 \rightarrow e2]] \]

An event class EI 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 EI, E2, E3, and E4, we would have restriction \( EI \rightarrow E2 \rightarrow E3 \rightarrow E4 \). 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**.

2. **Nondeterministic prerequisite of E**

\[ \{\text{Event Class Set}\} \rightarrow E \equiv (\forall e:E)[ \text{occurred}(e) \supset (\exists ! e':\{\text{Event Class Set}\})[e' \rightarrow e]] \wedge (\forall e':\{\text{Event Class Set}\})[(\exists \text{at most one } e:e)[e' \rightarrow e]] \]

\{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:

\[ \text{inputset}(\text{inpP}) \]

\[ \text{outputset}(\text{outP}) \]

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

\[ (\forall \text{inp}:? , \text{out}:!) [\text{inp}.\text{req} \rightarrow \text{out}.\text{end} \iff \text{out}.\text{req} \rightarrow \text{inp}.\text{end}] \]

3. **Event FORK**

\[ E \rightarrow \{\text{Event Class Set}\} \equiv (\forall Ei:\{\text{Event Class Set}\})[E \rightarrow Ei] \]

**Event JOIN**

\[ \{\text{Event Class Set}\} \rightarrow E \equiv (\forall Ei:\{\text{Event Class Set}\})[Ei \rightarrow E] \]

---

8Due 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

\[ e_1 \text{ at } E_2 \equiv \text{occurred}(e_1) \land \neg (\exists e_2 : E_2)[e_1 \Rightarrow e_2] \]

\( e_1 \text{ at } E_2 \) is true if \( e_1 \) has not enabled an event of class \( E_2 \).

\[ \text{new}(e) \equiv \text{occurred}(e) \land \neg (\exists e' : E_2)[e \Rightarrow e'] \]

\( \text{new}(e) \) is true if no event has observably followed \( e \).

8.3. GEM Thread Notation

Equipped with the descriptive tools WC 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

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 expression-like notation [4], we define thread type \( \pi_{RW} \) as follows:

\[ \pi_{RW} = (u.Read::db.control.ReqRead::
   db.control.StartRead::
   db.data[loc].Getval::
   db.control.EndRead::
   u.FinishRead)
\]

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 \( e \in \pi_{RW}^{-i} \) 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:

\[
\neg(\text{startread}\ m_{RW-i}:\text{StartRead, startwrite}\ m_{RW-j}:\text{StartWrite}) \\
\neg\text{occurred(startread} m_{RW-i}) \lor \neg\text{occurred(endread} m_{RW-i}) \lor \text{occurred(startwrite} m_{RW-j}) \lor \neg\text{occurred(endwrite} m_{RW-j})
\]

\[
\neg(\text{startwrite}\ m_{RW-i}:\text{StartWrite, startwrite}\ m_{RW-j}:\text{StartWrite}) \\
\neg\text{occurred(startwrite} m_{RW-i}) \lor \neg\text{occurred(endwrite} m_{RW-i}) \lor \text{occurred(startwrite} m_{RW-j}) \lor \neg\text{occurred(endwrite} m_{RW-j})
\]

where \( m_{RW-i} \) and \( m_{RW-j} \) are two distinct \( m_{RW} \) 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 same time, the read must be serviced before the write.

\[
\text{[reqread} m_{RW-i} \text{at StartRead} A \text{reqwrite} m_{RW-j} \text{at StartWrite}] \lor \text{[occurred(startwrite} m_{RW-j}) \lor \text{occurred(startread} m_{RW-i})]
\]

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 state). 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 properties 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’s 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 properties 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 \( \text{ASSIGN event belonging to a \( \text{VAR} \) 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. GEM operators allow one to describe liveness properties such as progress. 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 \( \text{event} e \) are fulfilled in history \( a \), 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 \( WC \) 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):

\[
\text{ReadersWriters: MONITOR}
\begin{align*}
\text{BEGIN} & \\
\text{readqueue, writequeue: CONDITION;} & \\
\text{readernum: INTEGER;} & \\
\text{/* readernum is positive if reading,} & \\
\text{negative if writing */} & \\
\text{ENTRY PROCEDURE StartRead;} & \\
\text{BEGIN} & \\
\text{IF readernum < 0 THEN} & \\
\text{WAIT(readqueue);} & \\
\text{readernum := readernum + 1;} & \\
\text{SIGNAL(readqueue);} & \\
\text{END;} & \\
\text{ENTRY PROCEDURE EndRead;} & \\
\text{BEGIN} & \\
\text{readernum := readernum - 1;} & \\
\text{IF readernum = 0 THEN} & \\
\text{SIGNAL(writequeue);} & \\
\text{END;} & \\
\text{ENTRY PROCEDURE StartWrite;} & \\
\text{BEGIN} & \\
\text{IF readernum \neq 0 THEN} & \\
\text{WAIT(writequeue);} & \\
\text{readernum := -1;} & \\
\text{END;} & \\
\text{ENTRY PROCEDURE EndWrite;} & \\
\text{BEGIN} & \\
\text{readernum := 0;} & \\
\text{IF queue(readqueue) THEN} & \\
\text{SIGNAL(readqueue) ELSE} & \\
\text{SIGNAL(writequeue);} & \\
\text{END;} & \\
\text{/* initialization */} & \\
\text{readernum := 0;} & \\
\text{END ReadersWriters}
\end{align*}
\]

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

\[
\text{Monitor = GROUP TYPE(lock: MonitorLock,} & \\
\text{\{entry\}: SET OF MonitorEntry,} & \\
\text{\{cond\}: SET OF Condition,} & \\
\text{\{init\}: Initialization,} & \\
\text{\{var\}: SET OF Variable} & \\
\text{PORTS(lock.Req) RESTRICTIONS} & \\
\text{Restrictions describing how a monitor functions.} & \\
\text{This would include rules for waiting and signalling,} & \\
\text{initialization, etc.} & \\
\text{END Monitor} & \\
\text{where MonitorLock, MonitorEntry,} & \\
\text{Condition, Initialization, and Variable are} & \\
\text{previously defined element or group types.} & \\
\text{In addition, each component of ReadersWriters} & \\
\text{must also be fully specified. For instance, the StartRead MonitorEntry} & \\
\text{specification includes a description of the statements} & \\
\text{which belong to entry StartRead. Given an} & \\
\text{underlying language specification, translation of a} & \\
\text{program into a GEM program specification is quite} & \\
\text{simple and mechanical enough to lend itself to} & \\
\text{automation.} & \\
\text{Finally, a proof is completed as follows:} & \\
1. \text{For each group, element, event type, event} & \\
\text{parameter, and thread in} \ P, \text{choose a} & \\
\text{corresponding object in} \ PROG. \text{We call these} & \\
\text{the significant objects of} \ PROG. & \\
2. \text{Prove that each restriction} \ Ri \text{in} \ P \text{is satisfied} & \\
\text{by the corresponding significant objects in} & \\
\text{PROG.} & \\
(\forall Ri \in P) [PROG \text{ sat } Ri] & \\
\text{GEM includes various rules which define what} & \\
\text{is meant by sat. One intuitive explanation is the} & \\
\text{following: If we examine a computation} & \\
\text{which is legal with respect to} \ PROG, \text{and only} & \\
\text{take note of significant objects, those significant} & \\
\text{objects exhibit the same behavior as a} & \\
\text{computation that is legal with respect to} \ P. & \\
\text{To illustrate this proof method,} \ WC \text{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:

<table>
<thead>
<tr>
<th>PROBLEM</th>
<th>PROGRAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReqRead</td>
<td>EntryStartRead:BEGIN</td>
</tr>
<tr>
<td>StartRead</td>
<td>EntryStartRead:readernum:=readernum+1</td>
</tr>
<tr>
<td>EndRead</td>
<td>EntryEndRead:readernum:=readernum-1</td>
</tr>
<tr>
<td>ReqWrite</td>
<td>EntryStartWrite:BEGIN</td>
</tr>
<tr>
<td>StartWrite</td>
<td>EntryStartWrite:readernum:=0</td>
</tr>
<tr>
<td>EndWrite</td>
<td>EntryEndWrite:readernum:=0</td>
</tr>
</tbody>
</table>

Assume that we have already proved Chat potential (startwrite) ⊆ readernum = 0 and new (startread) ⊆ readernum > 0. We have also proved that all events occurring in monitor entries or initialization code are totally ordered by the temporal order ⊑. An informal proof of the Reader’s priority property follows.

ReadersPriorityRESTRICTION:
[reqread πRW-i at StartRead ∨ reqwrite πRW-j at StartWrite] ⊑
[occurred(startwrite πRW-j) ⊑ occurred(startread πRW-i)]

Proof. By contradiction.

Let S be a valid history sequence for which the ReadersPriority restriction does not hold. Then S must have histories α0 and α3 such that α0 ⊑ α3 and

α0 ⊑ reqread πRW-i at StartRead ∨ reqwrite πRW-j at StartWrite

α3 ⊑ occurred(startwrite πRW-j) ∨ occurred(startread πRW-i)

If we choose α3 to be the smallest history having this property, then we additionally know that α3 ⊑ new (startwrite πRW-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 πRW-i must have waited on condition readqueue, giving us

reqread πRW-i → wait πRW-i(readqueue) → startwrite πRW-j

If we choose α as the smallest history containing this wait event, then we have α ⊑ new (startwrite πRW-j), and thus α ⊑ readernum < 0, and α ⊑ readernum < 0, and α ⊑ readernum < 0, and α ⊑ readernum < 0, and α ⊑ readernum < 0. Now let α2 be a history such that α1 ⊑ startwrite πRW-j. (Such an α2 exists because startwrite πRW-j must be potential before it occurs). Then α2 ⊑ readernum = 0.

Since readernum < 0 in α and readernum = 0 in α2, 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 must either be a StartRead event or EndWrite event (these are the only two events which could cause readernum to increase). Let β1 and β0 be histories such that

β1 ⊑ new(assignzero)
β0 ⊑ potential(assignzero)

Case 1.

Suppose that assignzero is of class StartRead. Because new (startread) ⊑ readernum < 0, we must have readernum < 0 before execution of assignzero. This means that β0 ⊑ readernum < 0. But α1 ⊑ β0 ⊑ α2, which contradicts the choice of assignzero 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

β1 ⊑ new(assignzero)
β0 ⊑ potential(assignzero)

Case 1.

Suppose that assignzero is of class StartRead. Because new (startread) ⊑ readernum < 0, we must have readernum < 0 before execution of assignzero. This means that β0 ⊑ readernum < 0. But α1 ⊑ β0 ⊑ α2, which contradicts the choice of assignzero as the first event to increase the value of readernum to 0. Contradiction

Case 2.
wait \( \piRW_i(\text{readqueue}) \Rightarrow \text{endwrite} \Rightarrow \text{startwrite} \piRW_j \)

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

\[
\text{reqread} \piRW_i \Rightarrow \text{endwrite} \Rightarrow \text{startread} \piRW_i \Rightarrow \text{startwrite} \piRW_j
\]

Contradiction

10. Related Work

The model of computation most similar to GEM is the message-based \textit{actor} model [6,10]. Greif’s [6] has utilized \textit{actors} to model the monitor primitive, and to analyze various aspects of concurrent execution. An actor system is composed of actors, \textit{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 \textit{rules} are not \textit{modeled specifically}, nor is there any means of naming \textit{particular} chains of control. Nor do actor systems \textit{incorporate} the use of temporal logic. In addition, the actor \textit{model} has \textit{not been} used as a tool for \textit{general} purpose language \textit{description} or program \textit{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 \textit{desired level} of detail. However, non-atomic events are also considered. Lamport has \textit{used} his model as a framework for solving such \textit{problems} as the synchronization of \textit{distributed} clocks.

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

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

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

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

11. Conclusion

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


