A restrictive definition of concurrency for discrete-event modeling

Bo Sandén
George Mason University
bsanden@gmu.edu
ISSE-TR-94-114
Revised

February 27, 1995

Abstract

Concurrency is an increasingly popular modeling concept in object-oriented analysis, software design, process modeling and other areas. Many approaches view their problem domains in terms of independent, concurrent processes that occasionally need to be synchronized. This is potentially an extremely concurrent model, where "essential" processes that represent independent progress may be lost in the wealth of more trivial processes. This is particularly important in a software implementation, where various overhead costs are incurred by each process.

The objective of this work is to find a formal definition of concurrency that is restrictive in that only essential processes (in some sense) are considered concurrent. The definition is based on simultaneity: threads (processes) are concurrent iff their occurrences are arbitrarily close in time. The formal apparatus is outlined in the following:

An *occurrence* is a unique happening at a specific time. A *problem* is a set of timed *traces* of occurrences. Such a set of traces is given as the starting point of the analysis. Occurrences are usually discussed in terms of *events*, where an event is any rule that defines a set of occurrences.

A *thread* is an event whose occurrences are separated in time. Given a problem, the goal is to find a *thread model* such that each occurrence in the problem belongs to exactly one thread. The rationale is that when the system runs, finite time is required to process each occurrence. Within each thread, the interval between occurrences allows for this. Simultaneous occurrences belong to different threads, which may execute on different processors or interleaved on a single one.

A thread model is *minimal* if all the threads have simultaneous occurrences. The rationale here is economy: this eliminates redundant threads that occur in lockstep or sequentially, one following the other, and consequently do not contribute to the essential concurrency. All minimal thread models of a given problem have the same number of threads, which is the *concurrency level* of the problem.

1 Introduction

Continuous activities are intuitively concurrent if they run in parallel, that is, there is an interval during which all the activities are continuously going on. In discrete-event environments, where reality is described in terms of distinct occurrences without duration, concurrency has no similarly obvious meaning. As a basis for the design of concurrent software, this paper proposes a definition based on simultaneity: threads (processes) are concurrent iff their occurrences are arbitrarily close in time. The objective is to define concurrency for the purposes of modeling and design, where it is now often used ad hoc. The definition proposed here allows the modeler or designer to base the decision of where to use concurrency on a determination of whether individual events occur at the same time. This determination can be made objectively based on explicit assumptions.

Concurrency is an increasingly popular modeling concept. Identifying essential concurrency is a useful way to separate concerns in a complex system, which can be reduced to a set of interacting sequential processes. Essential concurrency exists for instance (a) when a process may be held up waiting for resources held by another, (b) when single processes must wait for external events while other processes can proceed, and (c) when each process operates according to an individual schedule. In these cases, each process can intuitively be regarded as a unit of independent progress.

Many object-oriented analysis approaches include dynamic models, where an object is given life in the form of a process. There is sometimes a tendency in object-oriented modeling to regard concurrency as a normal state of affairs, which is disrupted by the occasional need for synchronization. For example, Cox views objects as "little independent automata, each busily working away in parallel ..." [2]. One problem with such very concurrent models is that essential concurrency inherent in a problem is not identified and may be lost among a wealth of processes, some of which may operate in lockstep or sequentially, one after the other. The processes described under (a), (b) and (c) above may not be identified among the "independent automata" but instead span several such automata.

An additional problem with a very concurrent model has to do with implementation. A software designer or a builder of a simulation model needs to map some of the processes found in a description of the problem onto tasks[1] executing on a computer. In that situation, a cost is incurred for the start-up of every new task, for each time unit of the task's existence and possibly for its dismantling. Costs are also associated with task synchronization and communication. These costs are not only in terms of tangible resources, such as CPU cycles, but also in terms of complexity, since process start-up, rescheduling, synchronization, dismantling, etc., require considerable manipulation. With this background, a reasonable research endeavor may be formulated as follows: "To formally define concurrency in environments where a cost is associated with the start of each concurrent process and/or for synchronization and/or for communication between processes."

[1] "Task" is used for a process managed during the execution of software. This is the term used in Ada, for example.

This paper proposes a simple concurrency concept that is applicable in practice yet amenable to formal treatment: The concurrency exists between sequential *threads*. In a *thread model* of a certain problem, each occurrence is attributed to exactly one thread. A $\delta > 0$ exists, such that the occurrences in each thread are separated by δ units of time. In a *minimal* thread model, all the threads are concurrent in the sense that each thread has at least one occurrence that is simultaneous with an occurrence in each other thread.

No notation for describing the possible sequences of occurrences in a thread is proposed, nor is any one existing notation adopted. The modeler may choose to represent individual threads as state machines (using state transition diagrams or Statecharts [5]), regular expressions (using either the traditional notation from automata theory or Jackson diagrams [7, 8]) or recursive expressions as in CSP [6]. The purpose of the proposed concurrency definition is *not* to provide another notation for communicating concurrent processes. In fact, the definition abstracts away from the structure of each process and sequence of event occurrences it produces, and is instead only concerned with whether a concurrent process is justified.

1.1 Practical justification

The concurrency definition proposed here is useful for concurrent, discrete-event modeling in requirements analysis, software design and simulation. Current literature on software requirements shows considerable interest in concurrent models [4]. In software design, tasks as provided by Ada and the threads provided by some operating systems and windowing systems have made concurrency easily available. Furthermore, there is a trend toward using concurrent models for such purposes as process modeling [3]. In all these areas, a well-defined vehicle for reasoning about concurrency in real, practical problems is needed.

Object-oriented analysis is particularly popular, and the Object Modeling Technique, OMT [10][2] is a good example of an object-oriented analysis approach. OMT includes an object model and a dynamic model. Objects are capsules of data (state) and operations, and the object model describes their relationships at each point in time. The dynamic model consists of concurrent processes and describes how the state changes as a result of events. (OMT uses Statecharts [5].) This makes OMT one of several object-oriented analysis techniques where objects are given life and become concurrent processes. OMT takes a liberal view of concurrency. Not only are different objects regarded as concurrent processes, but the modeler is also encouraged to use concurrency to express interleaved sequences of events within individual objects, leading to a potentially very concurrent model.

An important trend in object-oriented development is the desire for seamlessness in the sense that one basic model should support analysis, design and implementation. This means that concurrency identified during analysis tends to remain in the design, where it gives rise to tasks that must be managed and incur the costs that motivate the proposed concurrency definition.

[2] The reference is dated but adequate for the reasoning here.

As mentioned earlier, one concern in the design of concurrent software is that unnecessary tasks incur overhead. Given this background, Entity-life modeling (ELM) relies on threads as defined in this paper and gives heuristics for identifying threads in the problem and mapping them onto tasks [12, 13]. ELM gives special emphasis to systems where multiple entities vie for exclusive access to shared resources, and particularly where each entity needs simultaneous, exclusive access to multiple resources. The entities are modeled as threads. In control systems, it is often possible to design away deadlocks by imposing a certain protocol on these entities as to the order in which they acquire exclusive access to shared resources [14].

Section 2 contains the basic definitions. Section 3 contains a number of examples. Section 4 compares the definitions made here with other concurrency concepts. Section 5 includes conclusions and outlines further work.

2 Definitions

The basic, proposed concurrency definitions follow here. The first part is concerned with describing problems, the second with reasoning about them. Problem description is based on *occurrences*. An occurrence (of an event) is a concrete, unique happening. An informal example is: "This particular elevator arrives at this particular floor in this particular building at this particular time". A *problem* is defined as a set of timed *traces*, which are ordered sequences of occurrences. A trace describes one possible unfolding of occurrences in a given problem. Each trace is a consistent description of developments in the physical world, but a problem may contain traces that are mutually inconsistent. For example, in an elevator problem one trace may have the elevator arriving at floor 4 at a certain time on a certain day, while in another the elevator arrives at floor 5 at the same time and day. These two occurrences cannot co-exist in one trace. A trace may consist of actual observations of reality or be the result of a simulation or a thought experiment.

An *event* is an abstraction of the notion of occurrence; any rule that defines a set of occurrences is an event. This is intended to capture the common usage in discrete-event modeling. For example, when a system is modeled as a finite automaton, an event causes a state transition. Such an event may have any number of occurrences in the actual system. (The usage may clash with what is common in other contexts, where the word "event" is used for what is here called an "occurrence".) The event concept also subsumes the concept of conditions, which are sometimes used together with events in discrete-event models. Thus an event might be formulated as: "Elevator E arrives at floor 3 during rush hour".

The reasoning about concurrency is based on *threads*. A thread is an event whose occurrences are separated in time. This is the key to the concurrency definition: it distinguishes what is sequential (occurrences of the same thread) from what may be concurrent (occurrences of different threads). Simultaneous occurrences must be in different threads. Since an occurrence has no extension in time, such simultaneity is formalized by the concept of event co-occurrence. A set of events co-occur if for any $\delta > 0$ there is a trace where all the

events in the set occur within an interval of δ time units.

Given a problem, the goal is to find a *thread model* such that each occurrence belongs to exactly one thread. A rationale for this is that when a system is executed, finite time is required to process each occurrence. Within each thread, occurrences are separated by at least δ time units, so a sufficiently fast processor can handle each occurrence as it happens. If o_1 in the thread t_1 is such an occurrence, then any occurrence within δ time units of o_1, which might cut short o_1's processing time, belongs by definition to a different thread, t_2, say. The threads t_1 and t_2 can execute on different processors or be interleaved on a single processor.

A thread model is *minimal* if all threads co-occur, that is, they have occurrences that are arbitrarily close in time. The rationale behind minimal thread models is economy: as stated in the introduction, a cost is associated with each thread. In a minimal thread model, there are no redundant threads.

Traces and threads. The two concepts trace and thread both refer to ordered sets and may perhaps be confused. The following two examples illustrate the difference:

Example 1: In a multi-elevator system, a *trace* may be created by logging every occurrence as it happens, no matter what elevator and what floor it affects. On the other hand, each elevator represents a *thread*, since, regardless of trace, all occurrences affecting that elevator are separated in time.

Example 2: In a multi-processor system, the occurrences may be the executions of single instructions on each processor. Then, each processor represents a *thread*, while each execution of a program generates a *trace*.

2.1 Problem descriptions.

This subsection introduces the concepts necessary to describe problems. They include time, occurrences and traces. The notion of distance in time between occurrences is also introduced.

2.1.1 Time and occurrences.

In this model, **time** is continuous and monotonic. An occurrence is a happening at a point in time and without extension in time.

Example: The arrival of a particular elevator car to a particular floor in a particular building at a particular date and time.

Let **occurrences** be the set of all occurrences

The function

$$time : \mathbf{occurrences} \rightarrow \mathbf{time}$$

is an injection defined for all $o \in \mathbf{occurrences}$ and gives a unique time for each occurrence.

2.1.2 Traces.

A trace is an ordered set of occurrences. Semantically, a trace describes one possible sequence of occurrences in a particular problem. We shall assume that each trace has at most a countable number of occurrences.

Let **traces** be the set of all traces

Then,

$$\mathbf{traces} \in 2^{\mathbf{occurrences}}$$

The ordering of the occurrences in a trace with respect to time is strict; the likelihood of two different occurrences being at the same time is zero. Thus, the times associated with different occurrences are different:

$$\forall S \in \mathbf{traces};\ \forall o_1, o_2 \in S;\ o_1 \neq o_2 \bullet time(o_1) \neq time(o_2)$$

(Restricted to a trace, $S \in \mathbf{traces}$, $time : S \rightarrow \mathbf{time}$ is a total injection.)

The separation in time between two occurrences in the same trace is defined as follows:

$$\forall S \in \mathbf{traces};\ \forall o_1, o_2 \in S \bullet \Delta(o_1, o_2) \stackrel{\text{def}}{=} |\, time(o_1) - time(o_2) \,|$$

Since no two occurrences happen at the same point in time,

$$\forall S \in \mathbf{traces};\ \forall o_1, o_2 \in S \bullet o_1 \neq o_2 \Rightarrow \Delta(o_1, o_2) > 0$$

We shall say that two occurrences o_1 and o_2 are separated (or, more specifically, separated by δ time units) if there is a trace that includes them both and there exists a $\delta > 0$ such that $\Delta(o_1, o_2) > \delta$.

A trace extends in time to infinity. Thus, there is no concept of one trace following another. However, the range of *time* applied to the occurrences in a trace may be limited; that is, a trace may contain a first and/or a last occurrence.

2.1.3 Problems.

The purpose of the analysis discussed here is to find a thread model of a given problem. We define formally a problem, P, as a set of traces. We shall assume that each problem has at most a countable number of traces.

Let **problems** be the set of all problems.

A problem is non-empty if it contains at least one trace, S, such that $S \neq \{\}$. Any other problem is empty. We shall denote the empty problem by $\emptyset$.

Furthermore, we define the function

$$Occset : \mathbf{problems} \rightarrow 2^{\mathbf{occurrences}}$$

as the set of all occurrences in P as follows:

$$\forall\, P \in \mathbf{problems};\ \forall\, o \in \mathbf{occurrences} \bullet o \in Occset(P) \stackrel{\text{def}}{\Longleftrightarrow} \exists\, S \in P;\ o \in S$$

Theorem 2.1.1. For any problem, P, $Occset(P)$ is at most countable.

Proof: The number of traces is at most countable and the number of occurrences in each trace is at most countable. $Occset(P)$ is then also at most countable. Q.E.D.

Note. In a problem concerning real-world occurrences there is only one trace. Multiple traces exist in hypothetical worlds, such as simulations, where different scenarios over the same time span can be explored.

2.1.4 Restrictions of problems.

For a given problem, $P = \{S_j;\ j = 1 \ldots\}$, and a set, $O \subset \mathbf{occurrences}$, the restriction of a P with respect to O is denoted $P \mid O$ and defined as the set $\{S_j \cap O;\ j = 1 \ldots\}$.

Clearly, the following holds;

$$P \mid Occset(P) = P$$

$$P \mid \{\} = \emptyset$$

$$P \mid O = P \mid (O \cap Occset(P))$$

2.2 Events.

An event is a rule that defines a set of occurrences. Examples: Each of a, b, c, d and f is an event:

- a = "*Elevator X arrives at some floor*"
- b = "*Some elevator arrives at floor* 4"
- c = "*Some elevator arrives at some floor*"
- d = "*Something happens*"
- f = "*Elevator X arrives at floor* 4"

With the above notation, the text within quotes following the '=' sign is referred to as the *formulation* of the event.

Let **events** be the set of all events.

The function *occset* maps each event onto its occurrences in a given trace:

$$occset : \mathbf{events} \times \mathbf{traces} \rightarrow 2^{\mathbf{occurrences}}$$

If o is an occurrence and e is an event, we shall say that o belongs to e and that e includes o in a problem P iff the following holds:

$$\forall\, S \in P;\ o \in S \bullet o \in occset(e, S)$$

If e_1 and e_2 are events, we shall say that e_1 includes e_2 in a problem P iff the following holds:

$$\forall\, S \in P \bullet occset(e_2, S) \subseteq occset(e_1, S)$$

By overloading, we define

$$occset : \mathbf{events} \times \mathbf{problems} \rightarrow 2^{\mathbf{occurrences}}$$

as follows:

$$\forall\, e \in \mathbf{events};\ \forall\, P \in \mathbf{problems};\ \forall\, o \in \mathbf{occurrences} \bullet$$

$$o \in occset(e, P) \stackrel{\text{def}}{\Longleftrightarrow} \exists\, S \in P;\ o \in occset(e, S)$$

Note: *Occset* takes a problem as its argument, and *occset* takes an event and either a problem or a trace.

Example: For the above events a through f, and a suitably defined elevator problem, P,

$$occset(a, P) \subseteq occset(c, P)$$

$$occset(b, P) \subseteq occset(c, P)$$

$$occset(c, P) \subseteq occset(d, P)$$

$$occset(a, P) \cap occset(b, P) = occset(f, P)$$

The function *event* maps each occurrence, o onto an event that includes exactly the occurrence o. The following holds:

$$\forall\, o \in \mathbf{occurrences};\ \forall\, P \in \mathbf{problems} \bullet$$

$$o \in Occset(P) \Rightarrow occset(event(o), P) = \{o\}.$$

2.3 Threads.

We are interested in events whose occurrences are separated by a minimum time interval. We shall say that an event is a thread in a problem, P, if this is the case. Formally,

$$\forall\, P \in \mathbf{problems};\ \forall\, e \in \mathbf{events} \bullet$$

$thread_P(e) \stackrel{\text{def}}{\Longleftrightarrow}$

$\exists\, \delta > 0;\ \forall\, S \in P;\ \forall\, o_i, o_j \in occset(e, S), o_i \neq o_j \bullet \Delta(o_i, o_j) > \delta$

Example: $a =$ "*Elevator X arrives at some floor*"

In a suitably defined elevator problem, a is a thread since its occurrences are separated by the minimum time that elapses between arrivals at different floors.

Counter-example: $g =$ "*Some elevator arrives at some floor*"

g is not a thread in a problem, P, if for any $\delta > 0$, we can find a trace in P where two elevators arrive within δ time units of each other. Intuitively, such a trace can be found in a problem where multiple elevators travel independently.

Theorem 2.3.1. An event with no occurrence in a problem, P, is a thread in P.

Theorem 2.3.2. An event with exactly one occurrence in a problem, P, is a thread in P.

To show that an event, e, is a thread in a problem, P, it is sufficient and necessary to show one of the following:

- e has no occurrences in P (Theorem 2.3.1).
- e has exactly one occurrence in P (Theorem 2.3.2).
- In every trace, $S \in P$, any two occurrences in $occset(e, S)$ are separated by some $\delta > 0$.

Note. For any occurrence, o, in a problem, P, $event(o)$ is a thread in P since it has exactly one occurrence.

We shall say that a set, E, of events is a thread set in a problem, P, iff each member of E is a thread in P. Formally,

$\forall\, P \in \textbf{problems};\ \forall\, E \in 2^{\textbf{events}} \bullet$

$threadset_P(E) \stackrel{\text{def}}{\Longleftrightarrow} \forall\, e \in E;\ thread_P(e)$

2.4 Thread models.

A thread model of a given problem, P, is a set of threads, $\{h_1 \ldots\}$ in P, such that each occurrence in each trace in P belongs to exactly one thread.

Equivalently, a set of threads in P, $\{h_1 \ldots\}$, is a thread model of P iff $occset(h_1, P) \ldots$ partition $Occset(P)$.

Formally, define $model_P$ as follows:

$\forall\, P \in \textbf{problems};\ \forall\, E \in 2^{\textbf{events}} \bullet$

$model_P(E) \stackrel{\text{def}}{\Longleftrightarrow}$

$threadset_P(E)\ \wedge\ \bigcup_{h \in E} occset(h, P) = Occset(P)\ \wedge$

$\forall\, j \geq 1;\ k \in 1 \ldots j-1;\ h_j, h_k \in E \bullet occset(h_j, P) \cap occset(h_k, P) = \{\}$

Example: Given the earlier-defined events a, b, and f,

- $a =$ "*Elevator X arrives at some floor*"
- $b =$ "*Some elevator arrives at floor* 4"
- $f =$ "*Elevator X arrives at floor* 4",

a and b cannot be threads in the same model of a problem, P, since both include f.

Theorem 2.4.1. A thread model of a non-empty problem contains at least one thread.

Theorem 2.4.2 The empty set is a thread model of the empty problem.

To show that a set of threads, $\{h_1 \ldots h_n\}$, is a thread model of a problem, P, it is sufficient and necessary to show that :

$\forall\, j \in 1 \ldots n;\ k \in j+1 \ldots n \bullet occset(h_j, P) \cap occset(h_k, P) = \{\}$

and

$occset(h_1, P) \cup \ldots \cup occset(h_n, P) = Occset(P)$

2.5 Co-occurrence.

Co-occurrence captures the intuition that different events occur independently. We shall say that a set of events co-occur in a problem, P, if for every $\delta > 0$, there is a trace where *all* the events occur within δ time units.

$\forall\, P \in \textbf{problems};\ \forall\, E \in 2^{\textbf{events}} \bullet$

$co\_occur_P(E) \stackrel{\text{def}}{\Longleftrightarrow}$

$\forall\, \delta > 0;\ \forall\, e_i, e_j \in E;\ e_i \neq e_j;\ \exists\, S \in P;$

$\exists\, o_i \in occset(e_i, S), o_j \in occset(e_j, S);\ o_i \neq o_j;\ \Delta(o_i, o_j) < \delta$

Theorem 2.5.1. An event without occurrences in a problem, P, co-occurs with no event in P.

Note. The definition of co-occurrence of a set of events does not require that each event co-occur with itself.

To show that two events, e_j and e_k *do not* co-occur in a problem, P, it is necessary and sufficient to show either that one of the events has no occurrence in P, or that there exists a $\delta > 0$ such that in each trace, $S \in P$, each member of $occset(e_j, S)$ is separated from each member of $occset(e_k, S)$ by δ time units.

Theorem 2.5.2. If h_j and $h_k, h_j \neq h_k$, are threads in a problem, P, and do not co-occur, there exists a thread, h, in P, such that $occset(h, P) = occset(h_j, P) \cup occset(h_k, P)$

Proof. If h_j and h_k do not co-occur, the following possibilities exist:

1) $occset(h_j, P) = \{\}$, in which case $h = h_k$

2) $occset(h_k, P) = \{\}$, in which case $h = h_j$

3) $occset(h_j, P) = occset(h_k, P) = \{\}$, in which case $h = \{\}$

4) $occset(h_j, P) \neq \{\}$ and $occset(h_k, P) \neq \{\}$

To prove the theorem in case 4), let h be an event such that $occset(h, P) = occset(h_j, P) \cup occset(h_k, P)$. To show that h is a thread in P, we need to show

$$\exists\, \delta > 0;\ \forall\, S \in P;\ \forall\, o_l, o_m \in occset(h, S), o_l \neq o_m \bullet \Delta(o_l, o_m) > \delta$$

according to the definition of thread. Since h_j and h_k are threads, we know that this holds for any pair $o_l, o_m \in h_j$ and for any pair $o_l, o_m \in h_k$. It remains to show that it holds for any pair $o_j, o_k;\ o_j \in h_j, o_k \in h_k$. But from the the definition of co-occurrence we conclude that

$$\exists\, \delta > 0;\ \forall\, S \in P;\ \forall\, o_j \in occset(h_j, S), o_k \in occset(h_k, S);\ o_j \neq o_k \bullet \Delta(o_j, o_k) > \delta$$

Q.E.D.

2.6 Minimal thread models.

A thread model, M, is minimal iff all the threads in M co-occur. Formally, the property *minmodel* is defined as follows:

$$\forall\, P \in \textbf{problems};\ \forall\, M \in 2^{\textbf{events}} \bullet$$

$$minmodel_P(M) \stackrel{def}{\Longleftrightarrow} model_P(M) \wedge co\_occur_P(M)$$

Theorem 2.6.1. If P is a non-empty problem, and $M_P = \{h_1\}$ is a thread model of P, then M_P is minimal.

Theorem 2.6.2. Each thread in a minimal thread model of a problem, P, includes at least one occurrence in P.

Theorem 2.6.3 The minimal thread model of the empty problem is the empty set of threads.

To show that a thread model, $M_P = \{h_1 \ldots\}$, of a given problem, P, is minimal, it is necessary and sufficient to find for any $\delta > 0$, a trace, S_δ, with one occurrence, o_k, from each thread, $h_k, k = 1 \ldots$, such that $\forall\, i \in 1 \ldots n;\ j \in 1 \ldots i-1;\ \Delta(o_i, o_j) < \delta$.

To show that a thread model, $M_P = \{h_1 \ldots\}$, is not minimal, it is necessary and sufficient to find two threads h_j and h_k in M_P that do not co-occur.

Algorithm 2.6.1 One way to find a minimal thread model for a given problem, P, is defined recursively as follows:

1. Assume that a minimal thread model, $M_{P|O_k}$, of P restricted to some subset, $O_k \subset Occset(P)$ has been found.
2. Find a subset, $O_{k+1} \subseteq Occset(P)$, that contains O_k and where $O_{k+1} \setminus O_k \neq \{\}$.
3. Find a set of threads, T_{k+1}, that include all occurrences in $O_{k+1} \setminus O_k$.
4. For each thread, $h \in T_{k+1}$, if h co-occurs with all the threads in $M_{P|O_k}$, include it in $M_{P|O_{k+1}}$, else reformulate (as necessary) one or more threads in $M_{P|O_{k+1}}$ to include $occset(h, P \mid O_{k+1})$.

Note also, that for $O_0 = \{\}$, $M_{P|O_0} = \{\}$ (by Theorem 2.6.3.), which provides a possible starting point for the recursion.

Theorem 2.6.4. Algorithm 2.6.1 is convergent for any problem with a finite number of occurrences.

Algorithm 2.6.2. Given a thread model, $E = \{h_1, h_2 \ldots\}$ of a problem, P, the following algorithm produces a minimal thread model:

1. Assume that a minimal thread model, M_n has been found of $P \mid (occset(h_1, P) \cup \ldots \cup occset(h_n, P))$
2. If $co\_occur_P(\{h_1 \ldots h_{n+1}\})$ then $M_{n+1} = M_n \cup \{h_{n+1}\}$. Otherwise, there is a thread, $h_i, 1 \leq i \leq n$, such that $\{h_i, h_{n+1}\}$ do not co-occur. But then there exists a thread, h, such that $occset(h, P) = occset(h_i, P) \cup occset(h_{n+1}, P)$, by Theorem 2.5.2. Choose $M_{n+1} = M_n \setminus \{h_i\} \cup h$

Note that $M_1 = \{h_1\}$ is a minimal thread model of $P \mid occset(h_1, P)$, providing a possible starting point for the recursion.

Theorem 2.6.5. Algorithm 2.6.2 is convergent for any thread model with a finite number of threads.

2.7 Properties of minimal thread models.

Theorem 2.7.1. A minimal thread model contains at most a countable number of threads.

Proof: For any problem, P, and any minimal thread model, M_P of P,

$card(M_P) \leq card(Occset(P))$.

By Theorem 2.1.1, $Occset(P)$ is at most countable. Q.E.D.

Theorem 2.7.2. If one minimal thread model of a problem, P, has a finite number, n, of threads, then all minimal thread models of P have n threads.

Proof: Assume that $M_1 = \{h_1 \ldots h_n\}$ and $M_2 = \{g_1 \ldots g_m\}$ are both minimal thread models of a problem, P, with $m < n$. Since M_1 is minimal, $h_1 \ldots h_n$ all co-occur, so for any $\delta > 0$, there is a trace, $S_\delta \in P$ with n different occurrences within δ time units. Since $m < n$, at least two of these occurrences must be in one of $g_1 \ldots g_m$, say g_j. But then g_j is not a thread, leading to contradiction. Q.E.D.

We shall call the number of threads in a minimal thread model of a problem P the *concurrency level* of P and denote it $\mathcal{L}(P)$.

Theorem 2.7.3. $\mathcal{L}(P) = 0$ iff P is empty.

Theorem 2.7.4. There exists a problem with a concurrency level equal to the cardinal number of a countably infinite set ($\aleph_0$).

Proof. Consider a problem, P, with one trace, $S = o_1, o_2, \ldots$ such that $time(o_k) = 1 + 1/2^2 + \ldots + 1/k^2$. The series converges toward a certain finite value, t_∞, say. For any $\delta > 0$, the interval $[t_\infty - \delta \ldots t_\infty]$ contains a countably infinite number of occurrences, making $\mathcal{L}(P) = \aleph_0$. Q.E.D.

3 Examples.

This section illustrates the use of the formalism by establishing thread models of a number of examples. Some assumptions are left out relying on the reader's familiarity with various everyday situations.

3.1 Ball tossing.

Definitions: In a situation where a girl and a boy are tossing a ball to each other, define the following events:

boy = "*Boy catches or tosses*",

$girl$ = "*Girl catches or tosses*",

$ball$ = "*Ball tossed or caught*"

Let P be a set of ball-tossing traces according to the above description.

Theorem: **[1]** $\{ball, boy, girl\}$ is not a thread model of P.
[2] $\{boy, girl\}$ is a thread model of P, but not minimal.
[3] $\{ball\}$ is a minimal thread model of P.

Proof:

[1] $\{ball, boy, girl\}$ is not a thread model of P since each occurrence belongs either to both *boy* and *ball* or to both *girl* and *ball*.

[2] *boy* and *girl* are threads in P since the occurrences are separated by $\delta = 1$ second, say. Furthermore, $\{boy, girl\}$ is a thread model of P since *boy* and *girl* together include all occurrences in P and no occurrence is included in both threads. But *boy* and *girl* do not co-occur, so $\{boy, girl\}$ is not minimal.

[3] Clearly, *ball* is a thread including all occurrences in P. By Theorem 2.6.1 it is minimal. Q.E.D.

Comment: An analyst trying to find a minimal thread model for the ball-tossing problem might begin with the events associated with the girl and find the model $\{h_1\}$, where $h_1 = girl$. When the analyst goes on to consider the occurrences associated with the boy, it turns out that the boy occurrences are separated from those associated with the girl, so they must be included in the same thread. This forces a reformulation of h_1 as the *ball* thread. The association of a thread with an *entity* such as the ball or the girl has no formal significance but is a practically useful way of giving meaning to the thread.

3.2 Elevator system.

This system was mentioned earlier and includes one or more elevators operating in parallel shafts and serving the same set of floors. There are $emax$ elevators and $fmax$ floors, with $emax > 0$ and $fmax > 1$. Any number of the elevators can arrive at a floor at the same time.

Let the following be events:

e_i = "*Elevator i arrives at a floor*" *for* $i = 1 \ldots emax$

f_j = "*An elevator arrives at floor j*" *for* $j = 1 \ldots fmax$

Theorem: **[1]** $\{e_1 \ldots e_{emax}\}$ is a minimal thread model of P_{emax}.
[2] $\{f_1 \ldots f_{fmax}\}$ is a thread model of P_1, but not minimal.
[3] For $emax > 1, f_j$ is not a thread in P_{emax} for any j.

Proof:

[1] e_j is a thread in P_{emax} for each $j \in 1 \ldots emax$ for any $emax > 0$ since one elevator cannot arrive at more than one floor at each point in time in a given trace.

Furthermore, $\{e_1 \ldots e_{emax}\}$ is a thread model for all $emax$ since

$$\forall\, emax \bullet occset(e_1) \cup \ldots \cup occset(e_{emax}) = Occset(P_{emax})$$

$$\forall\, emax;\ \forall\, j, k \in 1 \ldots emax;\ j \neq k \bullet occset(e_j) \cap occset(e_k) = \{\}$$

Finally, $e_1 \ldots e_n$ co-occur, since we can find a trace where all elevators arrive at some floor at the same time.

[2] In P_1, $emax = 1$. For any $j \in 1 \ldots fmax$, f_j is then equivalent to the event

"*The elevator arrives at floor j*",

whose occurrences are separated. Furthermore,

$$\forall\, fmax \bullet occset(f_1) \cup \ldots \cup occset(f_{fmax}) = Occset(P_1)$$

$$\forall\, fmax;\ \forall\, j, k \in 1 \ldots fmax;\ j \neq k \bullet occset(f_j) \cap occset(f_k) = \{\}$$

Thus, $\{f_1 \ldots f_{fmax}\}$ is a thread model of P_1.

By [1], $\{e_1\}$ is a minimal thread model of P_1. Thus $\mathcal{L}(P_1) = 1$ so by Theorem 2.7.2, $\{f_1 \ldots f_{fmax}\}$ cannot be minimal for $fmax > 1$.

[3] f_j is not a thread in P_{emax}, $emax > 1$, for any j since the occurrences that two different elevators arrive at floor j are not separated. Q.E.D.

3.3 Movie star.

Both in the elevator example and the ball-tossing example, threads can be determined on the basis of causality: each elevator must leave a floor before it can arrive at the next, the girl must toss the ball before the boy can catch it, etc., whereas the arrivals of two different elevators at a floor have no direct causal connection.

Now consider instead the life of a movie star described as an alternating series of occurrences of the events *hire* and *fire* on one hand, and a series of alternating *marry* and *divorce* occurrences on the other[3]. Here, causality is limited to each of the two series, but it is reasonable to assume that the star cannot be involved in more than one occurrence of any of these events at a given point in time.

Definition: Let the following be an event:

$s =$ "*Star gets married, divorced, hired or fired*"

Theorem: $\{s\}$ is a minimal thread model of the movie star problem, P.

Proof: s is a thread since with $\delta = 1$ *second* (for example),

$\forall\, o_i, o_j \in occset(s, P);\ o_i \neq o_j \bullet \Delta(o_i, o_j) > \delta$

$\{s\}$ is a thread model since $occset(s, P) = Occset(P)$. By Theorem 2.6.1, it is minimal. Q.E.D.

3.4 File copy.

Consider the problem where a device (such as a computer program) copies a sequential file, IN, onto a different file, OUT, by repeatedly filling a buffer with a block from IN then emptying the buffer into OUT. At any one time, at most one buffer, B_I, can be filled, and at most one buffer, B_O, can be emptied, with $B_I \neq B_O$.

Definitions: Let P_n be a file copy problem with $n > 0$ buffers.

Let the following be events:

$f_i =$ "*Start filling buffer* i", $i = 1 \ldots n$

$e_i =$ "*Start emptying buffer* i", $i = 1 \ldots n$

$t_i =$ "*Start filling or emptying buffer* i", $i = 1 \ldots n$

$f =$ "*Start filling some buffer*"

$e =$ "*Start emptying some buffer*"

[3]This view of stardom was suggested by Ashley McNeile.

Theorem: **[1]** $\{f, e\}$ is a thread model of P_n, for any $n > 0$.
[2] $\{f, e\}$ is a minimal thread model of P_n, $n > 1$.
[3] $\{f, e\}$ is not a minimal thread model of P_1.
[4] $\{t_1 \ldots t_n\}$ is a thread model of P_n, for any $n > 0$.
[5] $\{t_1 \ldots t_n\}$ is a minimal thread model of P_n, $n = 1, 2$.
[6] $\{t_1 \ldots t_n\}$ is not a minimal thread model of P_n, $n \geq 3$.

Proof:

[1] Each of e and f is a thread since only one buffer can be filled at a time, and only one buffer emptied at a time. Furthermore,

$$\forall\, n > 0 \bullet occset(f, P_n) \cup occset(e, P_n) = Occset(P_n)$$

$$\forall\, n > 0 \bullet occset(f, P_n) \cap occset(e, P_n) = \{\}$$

[2] For $n > 1$, e and f co-occur since one buffer can be read and another one written simultaneously. Consequently, the thread model is minimal.

[3] With a single buffer, $e = e_1$ and $f = f_1$ do not co-occur since the buffer cannot be filled and emptied at the same time.

[4] t_i is a thread in P_n for each $i = 1 \ldots n$, for any $n > 0$ since any two occurrences of start filling and start emptying a given buffer are separated. Furthermore,

$$\forall\, n > 0;\ occset(t_1, P_n) \cup \ldots \cup occset(t_n, P_n) = Occset(P_n)$$

$$\forall\, n > 0;\ \forall\, j, k \in 1 \ldots n;\ j \neq k \bullet occset(t_j, P_n) \cap occset(t_k, P_n) = \{\}$$

[5] For $n = 1$, it follows by Theorem 2.6.1 that $\{t_1\}$ is a minimal thread model of P_1.

For $n = 2$, t_1 and t_2 co-occur since, in some trace, one buffer starts filling and the other starts emptying at the same time.

[6] From [2] and Theorem 2.7.2 it follows that a minimal thread model of $P_n, n > 1$ has exactly 2 threads. Q.E.D.

4 Other concurrency models

4.1 Process algebra.

In Hoare's Communicating Sequential Processes (CSP) [6], a process is a rule that generates traces of event occurrences. An elementary process, P, is described recursively as the solution to an equation such as $P = (x \rightarrow P)$ where x is some event. The right-hand member reads "x then P" and describes a process that generates the event x and then behaves exactly like P. Process interaction occurs when the same event is a possible next event in the independent behavior of each process. Such symmetric interaction can be extended to any number of processes. This allows the modeler to capture each causal constraint as concurrent process, since the production of a trace can only continue with an event that meets all the constraints. In this regard, CSP represents an alternative concurrency definition from the one proposed here[4]

In addition to providing an alternative definition, CSP can also be used as a notation to describe individual threads in the sense of this paper. Several CSP processes may constitute one thread. For example, $T = P;\ Q;\ R$ may be a thread, consisting of the three CSP processes P, Q and R executed in sequence. A thread can also be defined by interleaving CSP processes.

4.2 Actors.

Actors [1] is a general computational model intended particularly for applications in artificial intelligence and targeted at highly parallel computer architectures. Actor systems are modeled in a "structured operational" style. This is contrast to the CSP description style, where a system is broken down into individual processes and the total behavior is deduced from their combined behavior, Agha defines the configuration of an actor system and the transitions between configurations. Figuratively, such a configuration is a kind of snapshot of the whole system, reflecting the "state" of each actor[5]. In contrast with the processes in CSP, which are implicit, an actor is a computational agent with certain capabilities. This is in line with the proposed concurrency definition, which is motivated by costs associated with similar agents.

4.3 Models based on finite automata.

In Statecharts [5], concurrent processes (referred to as *orthogonal states*) are represented as communicating state machines. For notational flexibility beyond what is usual in traditional state transition diagrams, state variables are used in addition to explicit states, and transitions are caused by events and/or conditions on state variables and on the current state

[4]The concurrency concept in CSP gave rise to the task-rich Ada 83 syntax. Ada 9X allows equivalent designs to be expressed with substantially fewer tasks.

[5]Strictly speaking, there is no concept of a global time at which a snapshot might be taken.

of a concurrent process. Each transition may cause an action, which may in turn trigger a transition in another process or assign a value to a state variable, etc. Statecharts provide for viewing processes at different levels of abstraction with *superstates* that decompose into substates. Statecharts are incorporated in various requirements techniques. In object-oriented analysis, they are sometimes used to represent the dynamic behavior of the objects [10].

The Jackson model is intended as an aid for practical software construction. It is represented by JSP [7, 11] and JSD [8]. A process is essentially a regular expression over events and communicates with other processes via messages. Message queues allow each process to proceed at its own rate. While the model is explicitly pragmatic, it is amenable to formalization [15]. A strength of the model is the direct mapping of a process in the reality onto a program control structure in the software. Structure diagrams (or sequence diagrams [13]) are used to represent individual processes. They provide an alternative that is preferable to state transition diagrams (and Statecharts) for certain types of processes.

Both Statecharts and the Jackson model support a concurrency concept based on causality, similar to that supported by CSP: Every constraint on the system that can be so expressed is included as a sequential process, and an event occurs only when all the constraints are met. No particular attempt is made to reduce concurrency. In JSD, this is remedied by implementation in a sequential environment where different processes become subprograms and multiple instances of a process type become data records. That way, the cost associated with the process is much reduced.

4.4 Petri nets.

The Petri net [9] model also expresses concurrency. It is basically a resource model: a *transition, t*, is enabled when the necessary resources are available in the form of tokens at *t*'s input places. As a result, new resources may become available as tokens at *t*'s output places, and may in turn enable other transitions. Each firing of a transition is an occurrence under the proposed definition, and concurrency is defined in terms of transitions that can fire simultaneously. However, there is no explicit notion of a thread, which is instead represented by a series of transitions.

5 Conclusions.

Concurrent threads have been formally defined based on the concept of occurrences that coincide in time. The concept has been shown to capture the intuitive concurrency in a number of small problems. Further work includes the development of larger examples and a comparison with other concurrency definitions.

5.1 Further work on the definitions.

The following are some areas where the concurrency definition must be elaborated:

Realism. A realism criterion is necessary that limits the way events can be defined. For example, the occurrences of "*Pick a lottery ticket*" cannot normally also be described by the two events "*Pick a winning ticket*" and "*Pick a losing ticket*" since it must be possible to know what event occurs as it occurs.

Similar situations arise in communication systems where an incoming message must be accepted before its local addressee is known. This creates a need for *secondary occurrences*: The arrival of a message is an occurrence in the problem domain which may belong to some thread, t_1, but once the the addressee is determined, there may be an occurrence of some thread, t_2, associated with that addressee. Secondary occurrences are another area for theory enhancement.

Universal occurrences. A provision must be made for a class of universal occurrences that do not fit the concept of threads as equivalence classes. These are exceptional occurrences typically emanating from outside the problem domain and with particularly broad effects. Examples are the occurrences of such events as "power cut" and "the system crashes".

Modeling liberty. The realism requirement notwithstanding, the modeler is at liberty to make explicit assumptions that conflict with the physical reality. For example, it may be desirable to abstract away the fact that a certain system is to be executed on one sequential processor, which otherwise limits the system model to a single thread.

Adaptation to physical reality. In a practical implementation with a digital clock, δ would not be less than the distance between clock ticks since there is no way of telling occurrences apart at a higher degree of resolution. But the processing time for an occurrence may be greater than a clock tick. This means that occurrences of the same thread may be too frequent to be handled by one processor, and must be processed by multiple processors working in lockstep.

5.2 Development of further examples.

Along the lines of the examples discussed in the previous section, additional, well-defined, small problems will be developed as tests of the concurrency definition against intuitive concurrency. The intention is that some of the examples can be developed into reference problems that can serve as prototypes for the analyst.

Resource sharing. If the occurrences in a problem are dependent on shared resources, this influences the thread models of the problem. For example, if all occurrences are dependent on a single resource, the concurrency level of the problem is one. Resource sharing and deadlock prevention between threads needs to be studied.

Timing. Events such as "*X seconds have expired since y*" often play an important role in concurrent models either as time-outs or as prompts for action, such as when a quantity needs to be sampled every X seconds.

Interleaving. The earlier-mentioned life of a movie star is an example of interleaving. A more mundane example is a print file which has a physical view with page and line breaks and a logical view with paragraph and sentence breaks. The series of physical breaks and the series of logical breaks are interleaved. Davis discusses the advantage of orthogonal states in Statecharts to express interleaving [4]: A description of the movie star without decomposition into such orthogonal states includes less intuitive, combined states such as married-and-hired and single-and-fired. The two versions of the print file and the two views of the life of a movie star are not concurrent threads with the definition proposed here, but may be with some other definition of concurrency.

Parallel computation. Although the proposed definition is primarily concerned with problem-domain concurrency, it is consistent with execution domain concurrency, i.e., execution on parallel hardware. The execution on any one processor is regarded as a series of occurrences of some execution event. The events on different processors then co-occur, whereas those on each processor are separated.

5.3 Relation between threads and tasks.

The proposed definition is intended to restrict the possible, concurrent descriptions of a given problem. As far as the definition is concerned, these descriptions are equivalent. They potentially lead to different concurrent software designs with threads mapped onto tasks. The software designer must choose a concurrent description that leads to an optimum design. Some heuristics were developed as part of Entity-life modeling, such as the criterion that a thread be delayable or queuable [13].

A *delayable* thread contains events that are triggered by time. The rationale for these threads is that a task has the built-in ability to reschedule itself for execution at a later time (as by means of the delay statement in Ada). The elevator threads are delayable since, for instance, the doors must close automatically after they have been open for a certain period of time.

Queuable threads compete for simultaneous exclusive access to some resources. The rationale for queuable threads is a task's ability to suspend its activity and queue for access to a resource. In a simulation of the dining philosophers' problem, for example, each philosopher has a thread that queues for access first to one fork and then to another.

Additional criteria suggest that each thread should be *varied*. This is meant to reduce the number of threads by including many different events in each one. Furthermore, for conceptual simplicity, multiple instances of a few thread types are more desirable than many different threads.

Additional rules may allow the software designer to reduce the number of tasks below the concurrency level of the problem. Such a reduction may be based on 1) the *probability* that two events co-occur, 2) the *window* in time during which the occurrence can be detected and 3) the consequences of missing an occurrence.

For instance, in the elevator example, buttons on the floors and in the elevators are pressed independently of the elevator movements. Since they may all be pressed simultaneously, there is one thread per button. However, it is unlikely that more than a few buttons are pressed at any one time, a pressed button can be detected at any time while it is being held pressed, and the consequences of a failed detection are very limited (in this particular example). For this reason, there can be considerably fewer button sampling tasks than there are buttons.

5.4 Other research.

One benefit of formally defining concurrency is the potential for comparison of different concurrency concepts. Interesting, potential research projects would be to

- compare the restricted concurrency as defined here to the concurrency concept in Petri nets
- investigate if CSP can be used to define the traces that the present definition is based on.

A more practical effort is to elaborate a (manual) procedure or method for analyzing the concurrency in a given problem. This amounts to devising a "user-friendly" interface on the approach discussed here and might involve steps such as "identify significant occurrences" and perhaps a form or a table for identifying threads, co-occurrences, etc.

An important effort is to investigate how concurrency analysis as described here can be integrated into some of the popular object-oriented analysis approaches, whether as a way to identify "active" objects or separate threads.

Acknowledgements. Jan Hext made a crucial contribution to this work with his critique of earlier versions of the paper. Paul Ammann, Pat Patterson, Ray Schneider and Sean Wang also provided helpful comments and suggestions.

References

[1] G. Agha. *Actors; A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[2] B. J. Cox. *Object Oriented Programming, an Evolutionary Approach.* Addison-Wesley, 1987.

[3] B. Curtis, M. I. Kellner, and J. Over. Process modeling. *Communications of the ACM*, 35(9):75–90, September 1992.

[4] A. M. Davis. *Software Requirements. Objects, Functions and States.* Prentice-Hall, 2 edition, 1993.

[5] D. Harel. Statecharts, a visual approach to complex systems. In *Science of Computer Programming*, volume 8, pages 231–274. 1987.

[6] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall International, 1985.

[7] M. A. Jackson. *Principles of Program Design.* Academic Press, New York, 1975.

[8] M. A. Jackson. *System Development.* Prentice-Hall International, 1983.

[9] J. Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice-Hall, 1982.

[10] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-oriented Modeling and Design.* Prentice-Hall, 1991.

[11] B. I. Sandén. System programming with JSP: Example - a VDU controller. *Communications of the ACM*, 28(10):1059–1067, October 1985.

[12] B. I. Sandén. Entity-life modeling and structured analysis in real-time software design - a comparison. *Communications of the ACM*, 32(12):1458–1466, December 1989.

[13] B. I. Sandén. *Software Systems Construction with Examples in Ada.* Prentice-Hall, 1994.

[14] B. I. Sandén. Designing control systems with entity-life modeling. *Journal of Systems and Software*, 17(4), April 1995.

[15] P. A. Zave. A distributed alternative to finite-state-machine specifications. *ACM TOPLAS*, 7(1):10–36, January 1985.