RC 21924 (98579) 4 January 2001 Computer Science

IBM Research Report

Using Diposets to Characterize the Semantics of Concurrent Systems

John S. Davis II
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Using Diposets to Characterize the Semantics of Concurrent Systems

John S. Davis II
IBM T.J. Watson Research Center

January 2, 2001

Abstract

Modeling and designing concurrent systems is one of the most challenging tasks faced by software engineers as is evinced by many examples of error-laden large and small-scale concurrent systems. The primary difficulty is the inarticulate characterization of the semantics of concurrent systems. The successful development of concurrent systems requires a clear understanding of the types of relationships that exist between the components found in such systems. Two particularly important relationships found in concurrent systems are the *order* relation and the *containment* relation. The order relation represents the relative timing of component actions within a concurrent system and the containment relation facilitates human understanding of a system by abstracting a system's components into layers of visibility.

One major consequence of improper management of the order and containment relationships in a concurrent system is deadlock. *Deadlock* is an undesirable halting of a system's execution and is the most challenging type of concurrent system error to debug. The contents of this publication show that no methodology is currently available that can concisely, accurately and graphically model both the order and containment relations found in complex, concurrent systems. The result of the absence of a method suitable for modeling both order and containment is that analysis of concurrent systems is nearly impossible and the prevention of deadlock is extremely challenging.

I created the diposet as a solution to this problem and introduce the diposet in this publication. A *diposet* is a formal, mathematical structure that is similar in nature to a partially ordered set and is well suited to describing concurrent, computational systems. I define the diposet and show that it uses an *order-centric* approach that offers insight into the relative timing of events in a concurrent system and allows the specification of containment. The diposet offers an approach that is distinct from traditional formal concurrency methods that instead focus on logic which is non-graphical and often inaccessible to the non-mathematician. The formal structure of a diposet is illustrated through the construction of several proofs and theorems and I provide real world examples to show that the diposet can model a wide variety of communication semantics including synchronous and asynchronous message passing.

1 Introduction

Modeling and designing concurrent systems is one of the most challenging tasks faced by software engineers. Anyone who has designed a distributed software system or debugged a multi-threaded program has, no doubt, encountered frustration and uncertainty, the likes of which is rarely matched by the corresponding task in a sequential program. If finding a bug in a sequential program is equivalent to finding a needle in a haystack, then debugging a concurrent program is best described as finding a bug in a set of haystacks that seem to continually exchange positions while the search is in progress. The difficulty of developing concurrent systems is evinced in several high profile, large scale concurrent systems (Gibbs, 1994). For example, the Denver International Airport was designed to accommodate three airlines landing simultaneously but was delayed for months at a rate of $1.1 million a day due to errors in its concurrent baggage handling system. Similarly, in 1994 the Federal Aviation Administration was forced to scale back plans for its distributed air traffic control system due to software errors after a five year delay and a loss of $144 million. These two projects experienced failures in part due to their magnificent scale, but their concurrent nature added significantly (and indeed enabled the great scale), causing much of the problem.

The primary difficulty with designing and modeling concurrent programs is the inarticulate characterization of the semantics of concurrent systems. The fundamental question posed in the study of concurrent system semantics is *what does it mean to be simultaneous?* The challenge of this question lies in the incompatibility between human sequential thinking and concurrent execution. The sequential train of thought that designers engage in does a poor job tracking simultaneous activities in concurrent systems of significant size. The classic paper "No Silver Bullet" (Brooks Jr., 1987) sheds light here when it speaks of four essential properties of software that cause great difficulty for software designers: complexity, conformity, changeability and invisibility. Brooks describes the problem of invisiblity as being due to the fact that software does not live in a three dimensional world and hence is not easily seen by designers. Invisibility is compounded for concurrent software systems; such systems are neither three dimensional or sequential. Designers already burdened with intangible, non-three dimensional concepts must also take on the burden of travelling beyond sequential thought in order to grasp notions of concurrency.

There are many practical problems that result from inarticulate concurrent system semantics. At the *conceptual level*, inarticulate semantics prevents clear communication of a design and leads to noisy information transfer among designers. In some cases, the lack of communication may prevent a single designer from conceiving a design or recalling a design that was conceived in the past. In other cases, the lack of communication may prevent a designer from communicating her conception to a fellow designer in a clean and unambiguous fashion. At the *state-space level*, inarticulate concurrent semantics makes it difficult to maintain two important properties: *safety* and *liveness*. A concurrent system does not satisfy safety if disparate components of a concurrent system arrive at inconsistent interpretations of shared information. A concurrent system does not satisfy liveness if any component of the system halts execution prematurely. At the *execution level*, inarticulate concurrent system semantics can result in perceived non-determinism of a concurrent system. "Simultaneous" instructions in a concurrent program may be arbitrarily interleaved resulting in different interleavings for different execution runs, *even if the program inputs and parameters remain constant.* The result is that a concurrent program appears to be non-deterministic since the resulting trace for a given execution run can involve any one of several possible interleavings.[1] Perceived non-determinism makes it extremely difficult for the beleaguered concurrent system programmer to track down bugs because of the difficulty to reproduce errors.

The difficulties faced by designers of concurrent systems will not only continue but are likely to increase in occurrence due in large part to the growing demand for concurrent systems and the availability of concurrency constructs within the Java$^{\mathrm{TM}}$programming language for designing such systems. While in earlier times concurrent programming was a relatively esoteric art practiced by software few engineers, the need to program multithreaded web servers and support distributed pervasive computing environments has made concurrent programming a requirement for an increasing number of engineers. Furthermore, unlike the specialized concurrency libraries that were necessary in languages such as C++, Java$^{\mathrm{TM}}$has integrated standard concurrency constructs into its core language. The result is that entry level computer scientists are afforded the opportunity to attempt concurrent system design. Unfortunately, this opportunity comes at a high cost. Java$^{\mathrm{TM}}$provides syntax for fine grained concurrency; such syntax makes no guarantees for preventing errors and the result is often that a beginning concurrency programmer in Java$^{\mathrm{TM}}$is given "just enough rope to hang himself." If the industry is to avoid system failures such as those mentioned previously, it is imperative that software engineers be given new tools and methodologies for correctly designing concurrent systems.

I have attempted to establish the difficulty of creating concurrent systems in the previous discussion. The remainder of this document consists of the presentation of a modeling framework that makes concurrent system development easier. An appropriate framework can aid in the specification and analysis of concurrent systems and provide a methodology for describing the varied semantics found in such systems. Furthermore, a qualified framework can give insight into the execution of a concurrent system so that the perception of non-determinism will be reduced and safety and liveness will be maintained. I will judge the modeling framework to be presented based primarily on its applicability to concurrent systems and in particular to its ability to characterize the *order* and *containment* relationships found within typical concurrent systems. As I show in Section 2, order and containment are two very important types of relationships found between the components of a concurrent system.

The ability to characterize order and containment may be sufficient for modeling concurrent systems but that does not mean it will be practical and intuitive for the designer. Secondary to the issues of order and containment, I propose four criteria for comparing alternative concurrent system modeling frameworks.

[1]In actuality, the choice of the realized trace is due to the system's thread scheduler, a process that is often deterministic. Nevertheless, thread schedulers are typically beyond the control or view of a programmer and hence the apparent non-determinism. The absence of "simultaneous" events prevents this phenomenon in sequential programs.

1) **Unambiguous**
A framework that is unambiguous allows one to speak precisely. The absence of ambiguity facilitates understanding and ensures a unique mapping from syntax to semantics.

2) **General**
Generality allows one to speak about many things. As will be addressed later in this paper, there are numerous and varied semantics that are in use within the concurrency community (Andrews, 1991; Girault *et al.*, 1999; Davis *et al.*, 1999). A useful concurrency modeling framework should be able to address a broad subset of these constructs.

3) **Graphical**
A graphical approach allows one to speak visually. Indeed, the proverb "a picture is worth a thousand words" rings true in the design community as much as it does in the world at large and, to this end, contemporary work on graphical programming tools has been very active within the research community. For example, graphical representation is the primary thrust of the Unified Modeling Language (UML) movement (Booch *et al.*, 1999; Booch, 1994; Rumbaugh *et al.*, 1991).

4) **Formal**
A formal methodology allows one to speak with authority. The formal methods community (Alur and Henzinger, 1996; Winskel, 1994; Agha, 1986) studies the composition of systems and the semantics that result. The community places great emphasis on rigor and the ability to prove properties about a system. Such proof techniques facilitate guarantees about a system's execution.

The modeling framework that I created and will present is the diposet. A *diposet* is a formal, mathematical structure that is similar in nature to a partially ordered set and is well suited to describing concurrent, computational systems. Diposets use an *order-centric* approach that offers insight into the relative timing of events in a concurrent system. Diposets also allow the specification of containment. This approach is distinct from traditional formal concurrency methods that instead focus on logic (Andrews, 1991; Magee and Kramer, 1999; Godefroid, 1996). In addition to its suitability for describing order and containment relationships, I will show that the diposet satisfies each of the four characteristics presented above and examples will be given to illustrate the usefulness of the diposet in real world scenarios.

The remaining sections of this paper proceed as follows. In Section 2, I provide an overview of concurrent systems and establish the importance of the order and containment relationships that components of the system form with one another. I then consider several traditional and contemporary methods such as petri nets and directed graphs that are in use for modeling concurrent systems and show that they are insufficient with respect to order and containment. In Section 3, I formally introduce the diposet and in Section 4 I apply the diposet to various semantics and real world problems. This paper concludes in Section 5 with reflection and consideration of future extensions.

2 Order and Containment in Concurrent System Design

A concurrent program specifies a set of two or more processes that are coordinated to perform a task and a set of resources that are shared by the processes (Milner, 1989; Andrews, 1991; Magee and Kramer, 1999; Schneider, 1997). Each *process* consists of a sequential program made up of a sequence of *instructions* and this sequence is often referred to as a *thread of control* or *thread.* The relationships that exist between the processes, threads and instructions found in a concurrent system are numerous and correctly modeling and designing a concurrent system requires a clear understanding of these relationships.

One particularly important relationship in concurrent systems, the *order relation*, deals with the relative time at which instructions are invoked in an executing concurrent program. An example of the order relation is illustrated by instructions within a thread. Because each thread is a sequence, the instructions contained within a thread are totally ordered; i.e., given two distinct instructions, a and b, either a occurs before b or b occurs before a. While all instructions within the same thread are ordered with respect to one another, it is not necessary that instructions in separate threads have such a relationship. To say that two instructions, a and b, are not ordered with respect to each other simply means that although a and b are distinct instructions, a does not occur before b nor does b occur before a. Some interpret such absence of ordering as meaning that a and b are simultaneous instructions (de Bakker and de Vink, 1996).

The *containment relation* is another very important type of relationship found within concurrent systems. An obvious example of containment occurs with threads containing instructions. Another example of containment is instructions containing instructions. Given two distinct components, a and b, in a concurrent system either a contains b, b contains a or neither. Containment facilitates abstraction. It allows a designer to vary the amount of detail that is shown in a design. For example, in some cases a designer needs to know how a thread carries out its duties and in such cases the ability to see the instructions contained in a thread is important. After verifying a thread's activities, the designer may no longer need to consider the thread's details and the view will be refocused to ignore the contents of a thread. The need for abstracting details is seen throughout the object oriented software community (Booch, 1994; Rumbaugh *et al.*, 1991; Booch *et al.*, 1999; Lea, 1997).

The importance of the ordering and containment relationships to concurrent systems is made clear when considering the fundamental problem of interference. The coordination of threads in a concurrent system requires communication between the threads so that when appropriate, the threads may modify their activites based on information from other threads. Communication is accomplished by shared resources and is realized through *communication instructions.* In some cases, a shared resource might be a conduit through which communication messages are transferred. In other cases, a shared resource might be a memory location that multiple threads have read/write access to. While communication is necessary to coordinate threads, undisciplined communication can lead to major problems. If two or more threads access the same shared resource, they can potentially interfere with one another. There are many different types of interference but at its core *interference* occurs when two or more processes attempt to simultaneously change the state of a shared resource.

Interference is one of the fundamental problems faced in concurrent programming and for this reason great emphasis is placed on the ordering of instructions in concurrent programming. If two instructions from different threads modify a common resource, it is essential that one instruction happen before the other so that interference is avoided. We must be clear when we speak of one instruction happening before another. To ensure that instruction a precedes instruction b and thus avoids interference, we often require that instruction a precede instruction b *atomically*; i.e., that the completion of instruction a precede the beginning of instruction b. The atomic specification is where the containment relation becomes important. An instruction is atomic if it does not contain other instructions and it is non-atomic otherwise. To say that instruction a atomically precedes instruction b means that instruction a and all of its contents precede instruction b and all of its contents.

Adding order and containment constraints can be effective in preventing interference; unfortunately, lavish use of ordering and containment constraints can result in incomplete execution of a concurrent program. Consider for example two threads, A and B, such that thread A is instructed to wait on a particular instruction of thread B. If thread B decides to not invoke the particular instruction, perhaps in lieu of a more favorable option, then thread A will end up waiting forever - a potentially undesirable result. For these reasons, concurrent programming can be viewed as the application of techniques and methodologies for enforcing an appropriate level of ordering and containment on a set of multithreaded instructions.

The above discussion of order and containment constraints in concurrent programming highlights the two fundamental properties of concurrent systems: safety and liveness. *Safety* is the property that no *bad* thing happens during the execution of a program (Andrews, 1991; Schneider, 1997). Interference is an example of a bad thing. *Liveness* is the property that something *good* eventually happens (Andrews, 1991; Schneider, 1997). Liveness is violated if a program's execution terminates prematurely. All errors unique to concurrent programs can be stated in terms of safety and liveness. These definitions of safety and liveness have a foundation in mathematical logic. I prefer to cast the definitions into a framework based on ordering and containment. Given such a context, safety is violated in a concurrent program with too few ordering and containment constraints; liveness is violated in a concurrent program with too many ordering and containment constraints.

2.1 A Simple Concurrent Program

Figure 1 can be thought of as a simple concurrent program in that it specifies the ordering of instructions in a concurrent program. Thread A consists of instructions a, b, c, d and e while thread B consists of instructions f, g, h, i and j. Note that the arrows indicate instruction ordering such that the arrowhead indicates the preceding instruction; e.g., in the figure, instruction a occurs before instruction b. The angled arrow in Figure 1 indicates an ordering constraint imposed by

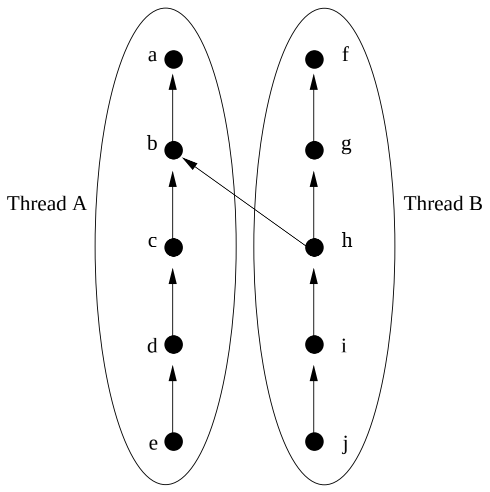

Figure 1: Two Communicating Threads

communication. The arrow does not indicate polarity of the communication but rather serves to illustrate the ordering constraint that the communication imposes. As shown, instruction h must occur after instruction b. Implicitly, instructions i and j must also occur after instruction b. Such constraints between instructions in separate threads would not exist if not for the communication between the threads.

Note that it is not possible to determine the relative ordering of all of the instructions in Figure 1. In particular, we can not determine whether instruction c occurs before or after instruction g. In general, a concurrent program will specify ordering constraints on only a subset of thread instructions. If all instructions between distinct threads were totally ordered, the result would be a single thread. The absence of an ordering specification is usually taken to indicate that relative ordering is inconsequential. In other words, the specification in Figure 1 indicates that instructions c and g can be realized as c followed by g or g followed by c; either realization is allowed and the choice is arbitrary. The notion of arbitrary ordering of unordered instructions can be applied to all of the instructions of a set of threads and results in an interleaving. An *interleaving* is a sequential realization of a set of threads that does not violate any of the ordering constraints of the threads. Figure 2 is an example of an interleaving. Note that threads A and B can be interleaved in either of the five ways shown. What this means is that if the concurrent program specified by Figure 2 were executed, any of the five sequential orderings could represent the actual execution. In fact, each execution can randomly turn out to be any of the five orderings even without changing parameters! Multiple interleavings facilitate both apparent and actual concurrency. In both cases, the goal is to ensure that the sequential realization/model is correct; i.e., equivalent to what the designer wants.

Unfortunately the existence of multiple interleavings for a single concurrent program specification leads to a major difficulty with concurrent programming. The size of the set of interleavings

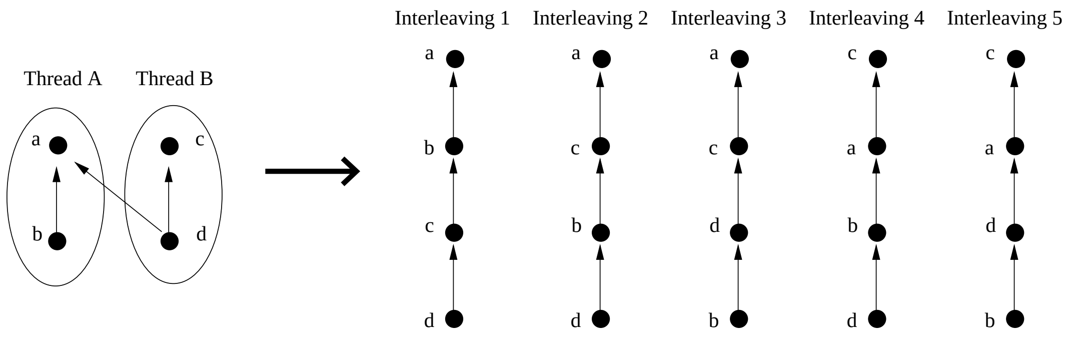

Figure 2: Sequential Interleavings

for a given program is typically unmanageably large. In general, given N threads that each execute M distinct non-communication instructions, there are

$$\frac{(NM)!}{(M!)^N}$$

possible interleavings. Five threads with ten non-communication instructions result in over 4.83×10^{31} possible interleavings.

2.2 Representing Concurrent Systems

A key difficulty in designing and implementing concurrent systems is the absence of effective tools for specifying and representing such systems. Representation tools are extremely important in the design process. Representation tools aid designers in communicating with each other about a given design as well as in finding errors. Graphical representation tools are especially helpful in designing software. For example, graphical representation is the primary thrust of the UML movement (Booch, 1994; Rumbaugh *et al.*, 1991). I will consider graphical representation tools for concurrent programming. In previous sections I have shown several figures (e.g., Figures 1 and 2) in an attempt to graphically represent concurrent programs. Unfortunately, these graphs have significant shortcomings.

In this section, I survey four modeling techniques that are worthy of consideration for modeling concurrent systems and I discuss the pros and cons of each. The four approaches I survey are partially ordered sets, interval orders, graphs and Petri nets. I chose these four modeling techniques because of their widespread use and mathematical rigor (West, 1996; Neggers and Kim, 1998; Peterson, 1981). My metric for measuring these four approaches will be their ability to represent both *containment* and *order* simultaneously. I will show that using this metric, each of these techniques falls short. I will then propose a new formalism for more effectively representing concurrent systems with containment and order; I refer to this formalism as a *diposet*.

2.2.1 Partially Ordered Sets

Definition 2.1. PARTIALLY ORDERED SET
Let X be a set. A **partial order**, R, on X is a binary relation that is reflexive, anti-symmetric and transitive. An ordered pair (X, R) is said to be a **partially ordered set** or a **poset** if R is a partial order on the set X. □

The three conditions on R hold for all $x, y, z \in X$ as follows

- Reflexive: $(x, x) \in R$
- Anti-Symmetric: $(x, y) \in R, (y, x) \in R$ implies $x = y$
- Transitive: $(x, y) \in R, (y, z) \in R$ implies $(x, z) \in R$

I will write $\leq$ for R such that $(x, y) \in R$ if and only if $x \leq y$; similarly $(y, x) \in R$ if and only if $y \leq x$.[2] Other common notations for R include $\sqsubseteq$ and $\preceq$. If $x \leq y$ or $y \leq x$ we say that x and y are *comparable.* If x and y are *incomparable* we write $x \parallel y$. We say that y *covers* x if $x \leq y$ and there is no element $z \in X$ such that $x \leq z \leq y$. The set X of a partially ordered set is called the *ground set.* If all elements of the ground set are comparable, then the set is called a *totally-ordered set* or a *chain.* If none of the elements of the ground set are comparable, then the set is called an *anti-chain.* The *up-set*, $Q \subseteq X$, of element y is defined such that $x \in Q \implies y \leq x$. We write the up-set of element y as y_{up-set}. The *down-set* is defined in a similar fashion.

Partially ordered sets can be graphically represented by Hasse diagrams. A *Hasse diagram* is a graph in which each vertex or point corresponds to one element of the ground set. An arrowed-line is drawn from point x to point y if y covers x.[3] If we interpret the partial order as representing precedence such that $x \leq y$ if y precedes x, then Figures 1 and 2 are examples of Hasse diagrams. For clarification, note that b is covered by a in Figure 1.

It would seem that partially ordered sets are a natural way to express the ordering relationships in concurrent programming systems. If we let each element of a set represent a method or function, then partially ordered sets can represent a program of method calls. Unfortunately, posets are not expressive enough to accurately represent even very simple programs. Consider the code fragment found in Program 2.1 (written in Java$^{\text{TM}}$syntax) where we assume that the methods *do*() and *undo*() do not call any other methods.

Assume a thread that invokes *start*(), *compute*() and then *finish*(). A poset is not able to model the complete relationship between *start*(), *compute*(), *finish*(), *do*() and *undo*(). More specifically, how do we relate *do*() and *undo*() to *compute*(). Both of the Hasse diagrams in Figure

[2]Note that I have chosen to use reflexive notation so that $\leq$ reads "less than or equal." Alternatively I could use irreflexive notation such as $<$, read "less than." Reflexive notation as given in the definition of partially ordered set defines the relation, R, as a *weak inclusion* while irreflexive notation defines the relation, R, as a *strong inclusion.* In some cases the relation associated with strong inclusion is called an *order* as opposed to a *partial order.*

[3]Alternatively, Hasse diagrams can be drawn with arrows from x to y if x covers y. Pay attention to the orientation when viewing a Hasse diagram.

Program 2.1. Example Sequential Code

```
public void start() {
    a = val;
}
public void compute() {
    do();
    undo();
}
public void finish() {
    a = 0;
}
```

3 are less than accurate. The method *compute*() is neither before or after *do*() and *undo*(), yet to say that *compute*() is incomparable to *do*() and *undo*() is not quite right either. The method *compute*() is non-atomic in that it contains *do*() and *undo*(). *The problem illustrated by this example is that partially ordered sets can not represent both the notion of order and the notion of containment.* Order is necessary to relate *start*() and *compute*() while containment is necessary to show that *compute*() is non-atomic.

2.2.2 Interval Orders

An interval order is a special class of partially ordered sets. The name implies that interval orders are amenable to graphical representation, and on the surface an interval order seems suitable for describing elements that are non-atomic. Nevertheless, interval orders can not describe containment and indeed they are less expressive then posets.

Definition 2.2. Interval Order
A poset $(X, \leq)$ is an **interval order** if there is a function $I : X \rightarrow [i(x), t(x)]$ where $i(x), t(x) \in \Re$ so that $x < y$ in X iff $t(x) < i(y)$ in $\Re$. □

An interval order corresponding to Program 2.1 is shown in Figure 4. The primary problem with interval orders is that they can not represent certain posets. In particular, while interval orders can represent incomparable points, they can not represent incomparable chains. Figure 5 illustrates the inability of interval order to represent chains. Given that the intervals of a, b and c are as shown, where do we place the interval for d? Interval d must intersect a and b without intersecting c: an impossible constraint. Hence, an interval order must be free of the poset shown in Figure 5. This precludes a large set of posets and renders interval orders insufficient for our purposes.

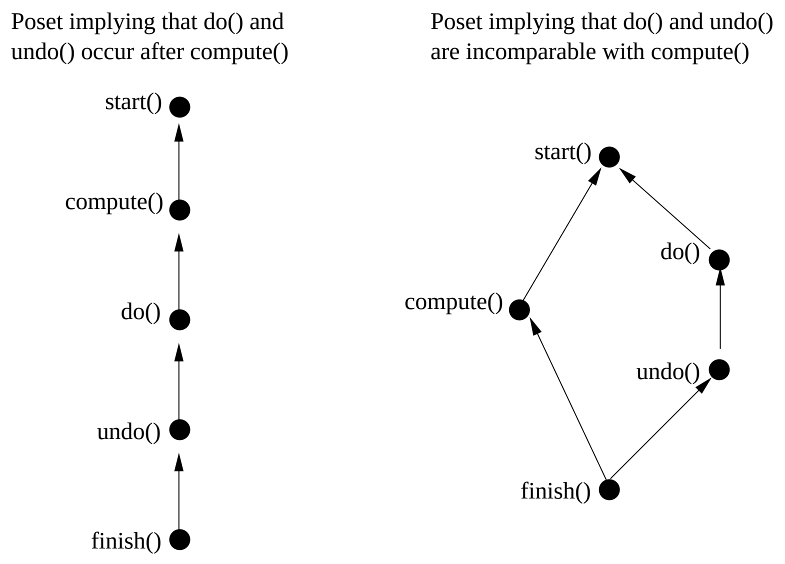
Poset implying that do() and undo() occur after compute()
start()
compute()
do()
undo()
finish()
Poset implying that do() and undo() are incomparable with compute()
start()
do()
compute()
undo()
finish()

Figure 3: Insufficient Poset Representations of Program 2.1

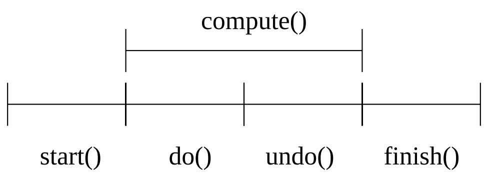

Figure 4: An Interval Order Representation of Program 2.1

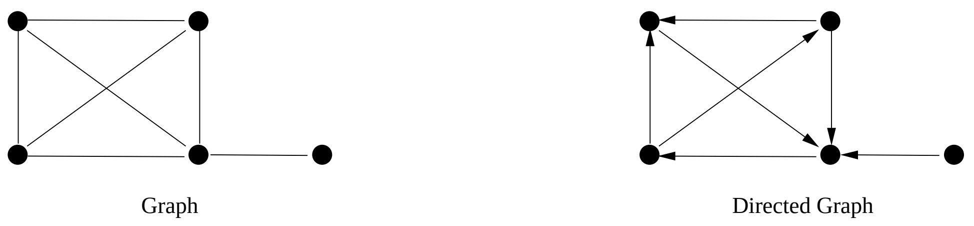

Figure 6: An Example Graph and Directed Graph

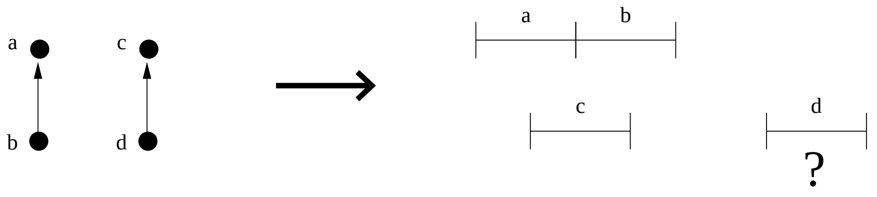

Figure 5: A Parallel Chain Poset With No Corresponding Interval Order

2.2.3 Graphs

A graph, as its name implies, is a mathematical structure that naturally lends itself to visual representation. Graphs are used extensively within the field of computer science. Examples include the representation of language grammars and network connectivity diagrams.

Definition 2.3. Graph
A **graph** G with n **vertices** and m **edges** consists of a **vertex set** $V(G) = \{v_1, ..., v_n\}$ and an **edge set** $E(G) = \{e_1, ..., e_m\}$. Each edge is a set of two (possibly equal) vertices called its **endpoints**. We write uv for an edge $e = \{u, v\}$. If $uv \in E(G)$, then u and v are **adjacent**. $\square$

Graphs are illustrated by diagrams in which a point is assigned to each vertex and a curve is assigned to each edge such that the curve is drawn between the points of the edge's endpoints. An example graph is shown in Figure 6 (on the left). In some cases, it is useful to add directionality to the edges of a graph. A *directed graph* models such directionality and is defined in the following definition. An example directed graph can be found in Figure 6 (on the right) where arrowed curves indicate direction.

Definition 2.4. Directed Graph
A **directed graph** is a graph in which each edge is an ordered pair of vertices. We write uv for the edge (u, v) with u being the **tail** and v being the **head**. $\square$

The definitions above are consistent with that used in many texts on the subject (West, 1996; Chen, 1997). Note that the edge set of a directed graph is simply a relation; e.g., $E(G) \subseteq V(G) \times$

$V(G)$. Focusing on the fact that the edge set of a directed graph is a relation emphasizes the shared traits between directed graphs and many other mathemathical structures. In particular, a relation-oriented definition of directed graph makes it clear that a partially ordered set is a special case of a directed graph.

Graphs and directed graphs both have definitions for several useful characteristics. For our purposes, two particularly useful definitions are path and cycle. Informally, a *path* in a graph is an ordered list of distinct vertices $v_1, ..., v_n$ such that $v_{i-1}v_i$ is an edge for all $2 \leq i \leq n$. A path may consist of a single vertex. A *cycle* is a path $v_1, ..., v_n$ in which $v_n v_1$ is an edge. The *length* of a path (cycle) $v_1, ..., v_n$ is n.

In their basic form, directed graphs and graphs are insufficient for modelling software systems for reasons similar to those cited for partially ordered sets. A directed graph only has a single relation on its set of vertices. A single relation will not sufficiently describe both the order and containment characteristics that are found in the typical object-oriented software program since order and containment are distinct qualities that require individual representation.

2.2.4 Petri Nets

Carl Adam Petri developed Petri theory with a concern for asynchronous communication between components and the causal relationships between events. The basic theory from which Petri nets developed can be found in the dissertation of Carl Petri (Petri, 1962). The definition of a Petri net structure is found below.

Definition 2.5. PETRI NETS
A **Petri net structure**, C, is a four-tuple, $C = (P, T, I, O)$. $P = \{p_1, p_2, ...p_n\}$ is a finite set of **places**, $n \geq 0$. $T = \{t_1, t_2, ..., t_m\}$ is a finite set of **transitions**, $m \geq 0$. The set of places and the set of transitions are disjoint, $P \cap T = \emptyset$. $I : T \rightarrow P^{\infty}$ is the **input** function, a mapping from transitions to bags[4] of places. $O : T \rightarrow P^{\infty}$ is the **output** function, a mapping from transitions to bags of places. □

Tokens can reside in (or are assigned to) the places of a Petri net. A *marking* μ is an assignment of a nonnegative number of tokens to the places of a Petri net. The number of tokens that may be assigned is unbounded. Hence, there are an infinite number of markings for a Petri net.

A Petri net executes by firing its transitions. A transition *fires* by removing tokens from its input places and creating new tokens in its output places. A transition may fire if it is *enabled.* A transition is enabled if each of its input places contains at least as many tokens as connection arcs from the place to the transition. Tokens that cause a transition to be enabled are called *enabling tokens.* When a transition fires, it removes all of its enabling tokens from its input places and then deposits into its output places *one* token for each output arc.

A Petri net is often graphically displayed as shown in Figure 7. In fact, a Petri net is a directed, bipartite multigraph. A *bipartite graph* is a graph that consists of two classes of nodes such that

[4]A bag is like a set except that it allows multiple occurrences of elements.

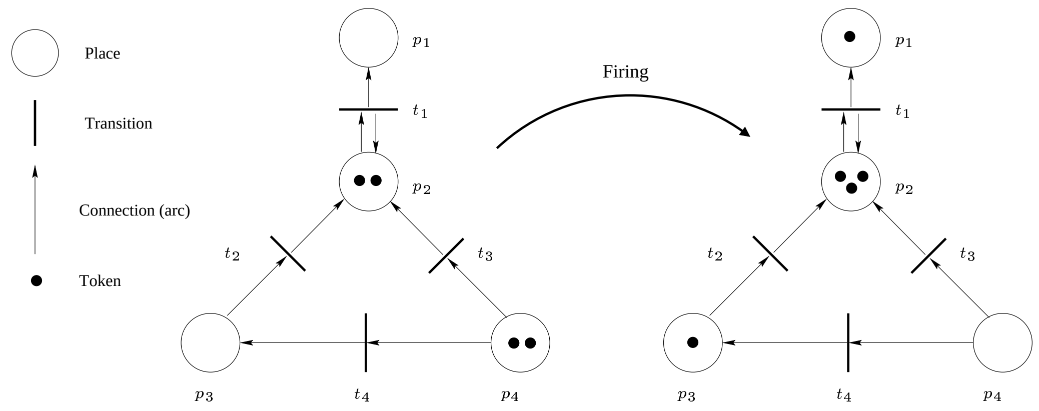

Figure 7: An Example Petri Net With Firing

each edge connects a node from one class to a node in the other class. In a Petri net, every arc (edge) connects a place to a transition. A *multigraph* is a graph that allows multiple edges from one node to another. As shown in Figure 7, several arcs may connect a place/transition pair.

A Petri net is not sufficient for representing order and containment. Even though it consists of two classes of nodes, its bipartite nature would constrain the order and containment relations to occur adjacently. It is not obvious how the containment relation could be graphically displayed using a Petri net, thus making it difficult to represent hierarchy. In addition, Petri nets assume an asynchronous style of communication. While it is true that asynchronous communication can serve as a foundation for synchronous communication (Brookes, 1999), asynchronous primitives can not represent synchronous communication in a succinct manner.

3 Diposets

In the previous sections I have summarized several mathematical formalisms and critiqued their usefulness in the context of describing object-oriented software systems. In each case, I showed that these formalisms were not sufficient for describing the richness of simple software systems. I have developed a new mathematical structure that I refer to as a *diposet*. In the remainder of this section I will define diposet and in subsequent sections I will make a case that diposets are suitable for robustly describing software systems.

The key observation with each of the mathematical structures presented thus far is that a single relation is not satisfactory for describing software systems. One way to deal with this problem is to use a pair of structures for describing software systems. Consider a pair of directed graphs G_1 and G_2 such that $V(G_1) = V(G_2)$. For convenience I will refer to this *paired directed graph* as $\{G_1, G_2\}$. Associated with the pair of directed graphs are two relations, $E(G_1)$ and $E(G_2)$. Each relation spawns various characteristics. For example, $\{G_1, G_2\}$ may have two distinct paths, p_1 and p_2, such that p_1 is associated with $E(G_1)$ and p_2 is associated with $E(G_2)$.

A paired directed graph $\{G_1, G_2\}$ offers the beginnings of a tool equipped for describing a variety of systems that require two types of relations (e.g., order and containment) over a set of elements. In order to make a paired directed graph completely useful, more structure must be added. I created the diposet to fill the need for just such a structure.

Definition 3.1. Diposet
Let X be a set. A **diorder** on X is a pair of binary relations on X referred to, respectively, as the **order relation**, R_O, and the **containment relation**, R_C, such that R_C and R_O are both reflexive, anti-symmetric and transitive. For all $x, y \in X$, if $(x, y) \in R_O$ then $(x, y), (y, x) \notin R_C$. Similarly, for all $x, y \in X$, if $(x, y) \in R_C$ then $(x, y), (y, x) \notin R_O$. A set X that is equipped with a diorder is said to be a **diposet** and is denoted (X, R_O, R_C). □

It is immediately obvious that a diposet is a special case of a paired directed graph. It is also clear that (X, R_O) and (X, R_C) are both partially ordered sets with a common ground set. The ground set X of a diposet is equivalent to the set of vertices V $(= V(G_1) = V(G_2))$ of a paired directed graph. The containment and order relations of a diposet, $\{R_C, R_O\}$, are equivalent to the two sets of edges in a paired directed graph $\{E(G_1), E(G_2)\}$.

We say that the ground set, X, of a diposet consists of *events*. The order relation determines how events are ordered with respect to one another. Consider events $a, b \in X$. If $(a, b) \in R_O$ then we say that $a \leq_O b$. I.e., event a precedes event b. If $(a, b), (b, a) \notin R_O$ then we say that $a \parallel_O b$; e.g., a and b are *incomparable*. The containment relation facilitates non-atomic events and event containment. An event is non-atomic if it contains another event. If $(a, b) \in R_C$ then we say that $a \leq_C b$. I.e., event b is non-atomic and contains event a. If $(a, b), (b, a) \notin R_C$ then we say that $a \parallel_C b$; e.g., a and b are *mutually non-inclusive*. Note the distinction between incomparable and mutually non-inclusive. In the context of diposets, incomparability refers to the order relation; mutual non-inclusiveness refers to the containment relation. Up-set is defined both for order and

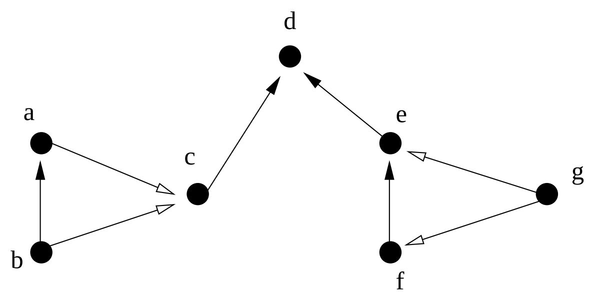

Figure 8: An Example Diposet

containment and is denoted as such; e.g., $up - set_O$ and $up - set_C$ (similar definitions exist for down-set). An *order* (*containment*) *path* in a diposet is a sequence of events $e_1, ..., e_n$ such that $e_1 \leq_O ... \leq_O e_n$ $(e_1 \leq_C ... \leq_C e_n)$.

Note that a direct result of Definition 3.1 is that $R_O \cap R_C = \emptyset$. The fact that R_O and R_C of a diposet do not intersect leads to two results that hold for all diposets:

i) An event can not contain an event that it precedes or that it is preceded by.

ii) An event can not be contained by an event that it precedes or that it is preceded by.

The disjointness of R_O and R_C in a diposet serves as one of the key distinctions between a diposet and a paired directed graph. In a paired directed graph $\{G_1, G_2\}$ it is sufficient for G_1 and G_2 to share a common set of vertices but there is no constraint on the two sets of edges associated with a paired directed graph. For example, it is completely admissable for the edge sets of a paired directed graph to be identical; i.e., $E(G_1) = E(G_2)$. The intuition behind the disjointness of R_O and R_C is that each relation should provide orthogonal information. If the order and containment relations of a diposet provide redundant information, then the usefulness of distinct relations is undermined.

Partially ordered sets are graphically represented via Hasse diagrams. Hasse diagrams serve as a simple way to represent posets with directed graphs where an arrow is drawn from element a to element b if b covers a. Diposets utilize Hasse diagrams as well, with the notion of covering being extended to containment. I.e., b covers a if there does not exist q such that $a \leq_C q \leq_C b$. Given that b covers a according to a containment relation, we say that b is a *cover container* of a. To accomodate both relations in a diposet, diposet Hasse diagrams require two types of arrows. I will use a black arrow head to represent the the order relation and a white arrow head to represent the containment relation. Figure 8 displays an example diposet. From this figure we can see that event a is contained by event c and is incomparable to event d. Event b is preceded by event a and event f is preceded by event d.

In many systems, the kind of containment that can be modelled by a diposet is not sufficiently constrained. Most software systems require that containment be nested. I add this additional

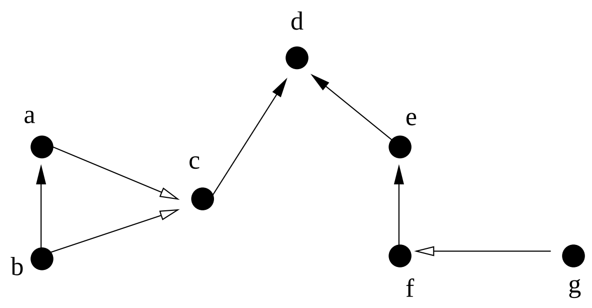

Figure 9: An Example Nested Diposet

constraint with the following principle.

Definition 3.2. The Nested Containment Rule
A diposet, (X, R_O, R_C), satisfies the **nested containment rule** if $\forall x, y, z \in X$, the following conditions are adhered to:
Condition I: If $x \parallel_C y$, then $(z \leq_C x \Longrightarrow z \not\leq_C y)$ and $(z \leq_C y \Longrightarrow z \not\leq_C x)$.
Condition II: If $x \leq_O y$, then $(z \leq_C x \Longrightarrow z \leq_O y)$ and $(z \leq_C y \Longrightarrow x \leq_O z)$.
A diposet that satisfies the nested containment rule is called a **nested diposet**. $\Box$

In plain English, Condition I says that an event can have at most one cover container. Condition II says that each event precedes (is preceded by) each event that its container events precede (are preceded by). An example nested diposet can be found in Figure 9.

A key distinction between the Hasse diagrams of diposets and nested diposets can be seen when comparing Figures 8 and 9. In Figure 9 it is implicit that $a \leq_O d$ by Condition II of Definition 3.2. In a similar fashion, we see that $g \leq_O e$. These assumptions can not be made in a general diposet, and hence in Figure 8 a and d are incomparable while in Figure 9 they are comparable. This distinction between the Hasse diagram for diposets versus nested diposets requires that one clearly state which type of diagram is being displayed, so that confusion can be avoided. Nested diposets are generally more useful than diposets. For example, most computer programs have a nested structure. For this reason, I will focus solely on nested diposets from this point on and I will use the term nested diposet and diposet interchangeably to mean nested diposet. Several interesting results can be derived based on the nested containment rule, as the Weighted Chain Theorem[5] illustrates.

[5]The intuition behind the name "Weighted Chain" is that if ever a subset of a diposet contains a minimum contained element (e.g., an element contained by all other members of the subset), then the minimum forces the elements in the subset to be pulled down like a hanging chain with a weight tied at the bottom.

Theorem 3.1. Weighted Chain Theorem

For nested diposet, (X, R_O, R_C), if there exists $x_0 \in X$ s.t. $\forall x \in X, x_0 \leq_C x$ then all events in X are incomparable.

Proof by Contradiction Suppose that not all events in X are incomparable. Then there must exist two events $y, z \in X$ such that either $y \leq_O z$ or $z \leq_O y$. Consider the former case. We have $y \leq_O z$. Since $x_0 \leq_C y$ by the theorem statement, then we know from Condition II of Definition 3.2 that $x_0 \leq_O z$. Again referring to the theorem statement we have $x_0 \leq_C z$. This contradicts Definition 3.1 since an event can not be contained by an event that it precedes; e.g., the disjointness of R_O and R_C has been violated. Hence our supposition was false. The alternative cases follow in a similar manner. □

In considering the nested containment rule, it is important to be clear on what it does not imply. In particular, note that for a given nested diposet, (X, R_O, R_C), with $x, y, z \in X$

$$x \leq_O y \leq_C z \not\Longrightarrow x \leq_O z$$

The simplest counter example that satisfies the above statement is the following three event nested diposet, $x, y, z \in X$:

$$x \leq_C z$$
$$y \leq_C z$$
$$x \leq_O y$$

Note that if $x \leq_O y \leq_C z \Longrightarrow x \leq_O z$ then Def. 3.1 would be violated; i.e., $R_O \cap R_C \neq \emptyset$.

Definition 3.3. Sequential Nested Diposet (Thread)

A **sequential nested diposet** or **thread** is a nested diposet, $X_{ND} = \{X, R_O, R_C\}$, for which $\exists x_0 \in X$, called the **maximum container** of X, such that $x \leq_C x_0, \forall x \in X$ and such that $\forall x, y \in X$, if x and y have a common cover container, then $x \leq_O y$ or $y \leq_O x$. □

An example thread is shown in Figure 10. It is drawn in an explicit graphical format. Explicit graphical format will be explained in Section 3.1.

Given that each event in a thread has at most one cover container,[6] it is useful to develop a notion of depth. We define depth recursively. The *depth* of the maximum container in a thread is 0. For any event x contained within a thread other than the maximum container, the *depth* of x is the depth of its cover container plus 1.

[6]A characteristic that is true of all nested diposets.

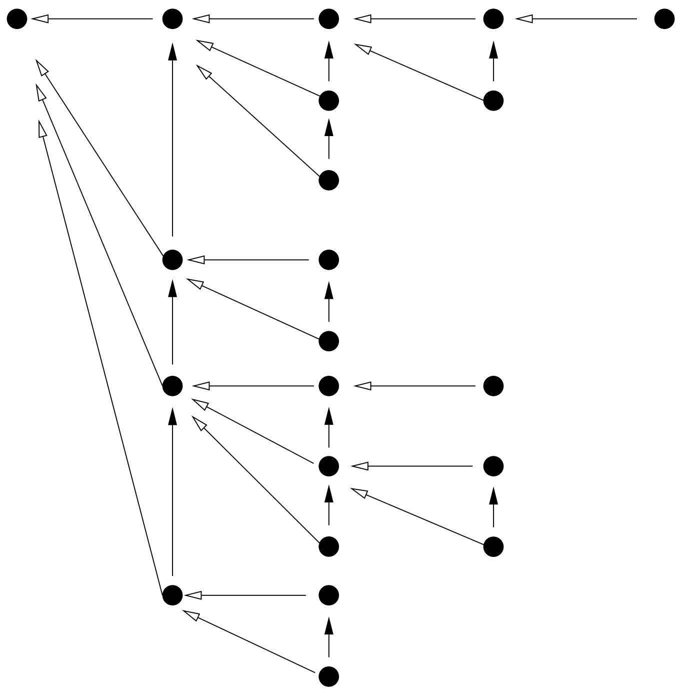
Figure 10: A Sequential Nested Diposet (Explicit Representation)

Theorem 3.2. Connected Thread Theorem
Any two events in a sequential nested diposet (thread) are either related by the order relation or the containment relation but never both.
Direct Proof Consider any two events x, y contained in a sequential nested diposet with ground set X. We know that x and y can not be related by both the order and containment relations by Definition 3.1. In terms of the rest of the proof there are three possible cases as listed below.

i) If x and y are not mutually non-inclusive then x and y must be related by the containment relation and we are done.

ii) If x and y are mutually non-inclusive and have a common cover container then by Definition 3.3 x and y must be related by the order relation and we are done.

iii) If x and y are mutually non-inclusive and do not have a common cover container then apply the following step. Select the event (either x or y) that has the greatest depth.[7] Without loss of generality assume that x has a greater depth then y. If the cover container of x is comparable to y than we are done by virtue of Condition II of Definition 3.2. Otherwise, repeat this step.

□

Theorem 3.3. Acyclic Diposet Theorem
A diposet can not contain order or containment cycles of length 2 or more.
Proof by Contradiction Suppose that a diposet (X, R_O, R_C) contains an order cycle of length 2 or more. Then there must exist a path $e_1, ..., e_n$ with $e_1 \neq ... \neq e_n$ such that $e_1 \leq_O ... \leq_O e_n \leq_O e_1$. By the anti-symmetry property of partially ordered sets, this implies that $e_1 = ... = e_n$. Hence, our supposition must be false and the diposet does not contain a cycle of length 2 or more. Similar reasoning applies to containment cycles of length 2 or more. This completes our proof. □

Note that in general a paired directed graph can contain both order and containment cycles of any length. As will be shown in subsequent sections, the existence of a cycle indicates that a system can not be modelled by a diposet but perhaps can be modelled by a paired directed graph.

In many situations it is useful to label the events of a diposet. For example, multiple events in a diposet's ground set may each share a common label indicating that they represent a common entity or labels may serve as a basis for relating a class of events. A labelling function facilitates this process.

[7] If x and y have the same depth then arbitrarily choose one or the other.

Definition 3.4. Labelled Diposets
A **diposet labelling function**, $f : X \to L$, maps the ground set of a diposet to a **label set**, L. A diposet that is associated with a labelling function and label set is referred to as a **labelled diposet**. $\square$

Many of the example diposets that have been previously shown were labelled. For example, in Figure 9 the label set is $L = \{a, b, c, d, e, f, g\}$ and the labelling function is bijective.

3.1 Types of Order

Thus far I have presented three partially ordered structures: diposet, nested diposet and sequential nested diposet (thread). Nested diposets and sequential nested diposets are especially important for our purposes because of the abundance of nested structures in the field of computer science. When considering a nested diposet, it is always the case that the order relation of a nested diposet can be separated into two subsets: $C_O \cup T_O \subseteq R_O$. C_O is referred to as the set of *communication order relations* and T_O is referred to as the set of *threaded order relations.* For any two events, $x, y \in X$, $C_O(x, y)$ represents a subset of order relations that are associated with x and y; i.e., $C_O(x, y) \subseteq C_O$. In a similar fashion $T_O(x, y) \subseteq T_O$.

The threaded order relation T_O relates events that are in the same thread. For any nested diposet (X, R_O, R_C), we have $T_O(x, y) = \emptyset$ if $x, y \in X$ are not part of the same sequential nested diposet. The communication order relation C_O relates events that are not in the same thread. We have $C_O(x, y) = \emptyset$ if $x, y \in X$ are part of the same sequential nested diposet.

A simple communication order relation is $C_O(x, y) = (x, y)$. As will be shown in Section 4.3, this order relation is equivalent to asynchronous communication between two threads in which the thread containing event y receives from the thread containing event x. A more elaborate communication order relation is

$$C_O(x,y) = \begin{array}{l} \{(x,y')|y' \in y_{up-set_O}\} \cup \{(y',x)|y' \in y_{down-set_O}\} \\ \cup\{(x',y)|x' \in x_{down-set_O}\} \cup \{(y,x')|x' \in x_{up-set_O}\} \end{array} \qquad (1)$$

I show in Section 4.3 that the communication order relation of Equation 1 is a precise characterization of synchronous message passing communication in which x and y represent the sending/receiving events of two communicating threads

Given nested diposet (X, R_O, R_C) we can write

$$R_O = \left[\bigcup_{(x,y)\in X\times X} C_O(x,y) \right] \bigcup \left[\bigcup_{(x,y)\in X\times X} T_O(x,y) \right] \qquad (2)$$

We can leverage the dichotomy found in Equation 2 to simplify our nested diposet Hasse diagrams. Recall that Figure 10 is drawn in an explicit graphical format. By *explicit* I mean that all cover

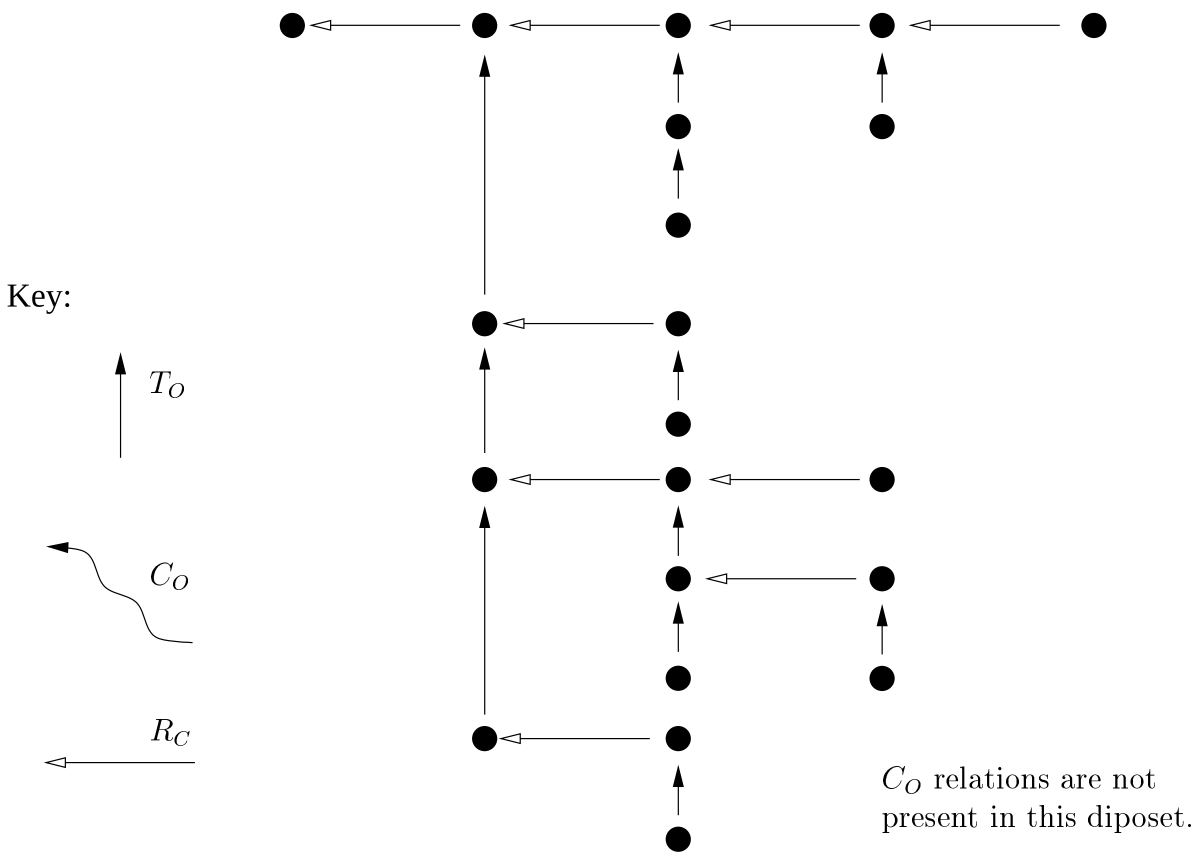

Figure 11: A Sequential Nested Diposet (Implicit Representation)

container relationships are explicitly shown. Alternatively a sequential nested diposet can be represented in an implicit graphical format. The implicit graphical representation of a nested diposet relies on the following three rules.

i) Order relations associated with T_O are drawn with a vertical line.

ii) Order relations associated with C_O are drawn with a non-vertical curve.

iii) Containment relations are drawn with a horizontal line (represented by R_C).

An example sequential nested diposet that is drawn in an implicit format is shown in Figure 11. The thread drawn in Figure 10 is identical to that of Figure 11. The only difference is that the former is represented explicitly while the latter is represented implicitly. In the remainder of this document, I will use implicit representation of sequential nested diposets.

4 Diposets and Concurrent Programming

Diposets are amenable to modeling a wide variety of systems including manufacturing schedules, distributed transactions and hardware systems. Given our interests, we will use nested diposets to model concurrent software systems. In a concurrent system, the ground set of our nested diposet consists of method invocations or code blocks. The label of an invocation is simply the method's name. Hence, multiple invocations of a method share a common label. Note that declaring a nested diposet's ground set as consisting of method invocations can accomodate a rich class of programming constructs including recursion and software objects.

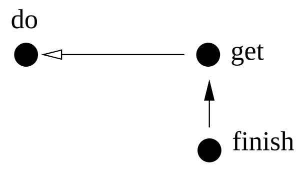

Figure 12: A Nested Diposet Corresponding To Program 4.1

If the body of method a contains an invocation of method b, then we say that $b \leq_C a$. If the body of method a precedes method b (as in method a returns prior to the invocation of method b) then we say that $b \leq_O a$. Consider the code fragment shown in Program 4.1. Here we see both the notion of containment and order. The methods $get()$ and $finish()$ are contained within the method $do()$. E.g., $get \leq_C do$ and $finish \leq_C do$. In addition, the method $finish()$ is ordered to occur after the method $get()$. E.g., $finish \leq_O get$. A single invocation of the method $do()$ would result in the thread displayed in Figure 12.

Program 4.1. Sample Method Calls

```
public void do() {
    get();
    finish();
}
public void get() {
    z1 = x + y;
}
public void finish() {
    z2 = z1++;
}
```

In some cases a nested diposet or a diposet will not be sufficient for describing a software system. In particular, as Theorem 3.3 (the Acyclic Diposet Theorem) declares, a diposet can not contain non-trivial cycles. In cases where inclusion of a cycle is crucial, the structure of a diposet can be relaxed and transformed into a paired directed graph. Paired directed graphs are amenable to describing cycles because they are not beholden to anti-symmetry.

4.1 Safety and Synchronization

Recall that the key problems of concurrent programs fall into two classes: safety and liveness problems. Let us consider how nested diposets can model these constucts. Safety is solved by applying mutual exclusion to the critical section of code that should not be simultaneously accessed by multiple threads. A common way to guarantee safety in a concurrent program is to require lock synchronization to code blocks. Only one process can synchronize with a given lock and thus access to the block of code will necessarily be mutually exclusive.

Safety via lock synchronization can be represented with containment and order relationships. Locks apply to blocks of code, thus we can think of a realized lock as an invoked method. In nested diposet terms, the code that a realized lock synchronizes is contained by the lock. We must make sure that multiple realizations of the same lock are not invoked simultaneously. This is accomplished by ordering the lock invocations. This process is illustrated in Program 4.2 (written in psuedo Java$^{\mathrm{TM}}$code) and Figure 13. Note that the `synchronized` keyword means that the lock for the corresponding method is an instatiation of the `Obj` class. A nested diposet showing a possible interleaving of calls to methods *do* and *undo* that satisfies the synchronization lock constraints is given in Figure 13.

Program 4.2. Synchronized Method Calls

```
public class Obj {
    public synchronized void do() {
        modify();
        change();
    }
    public synchronized void undo() {
        change();
        modify();
    }
    private void change() {
        // Atomic; contains no methods
    }
    private void modify() {
        // Atomic; contains no methods
    }
}
```

4.2 Liveness and Deadlock

The result of liveness problems within concurrent, computational systems are perhaps the most recognizable difficulties that the typical computer user must face. Liveness is closely associated with the inter-dependencies and relative speeds of autonomous threads. Relative thread speeds are tied to the thread scheduling algorithms of operating systems and such algorithms are typically beyond the control of software developers. For this reason, liveness problems have an inherently

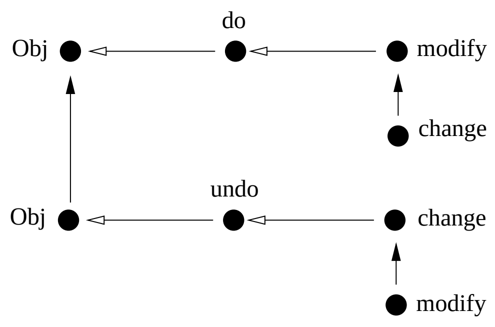

Figure 13: A Possible Interleaving Of Calls To `do()` And `undo()` In Program 4.2

non-deterministic nature from the perspective of the software developer. Although the problems associated with the absence of safety can be just as devastating as those associated with the absence of liveness, safety is much easier to maintain than liveness and thus for most computational systems safety is not a major issue.[8] Doug Lea categorizes liveness into four groups (Lea, 1997).

i) *Contention* occurs when several processes wait on resources but only a subset of the processes gain the resources. Contention is fundamentally related to fairness and is generally a deterministic problem in that it is based on the thread/process scheduling algorithm being used.

ii) *Dormancy* occurs when a waiting thread is not notified that the condition it is waiting on becomes true. This problem is relatively easy to solve with well placed "wake up" mechanisms. For example, in the Java$^{\text{TM}}$programming language a `notify()` or `notifyAll()` method would be used. Dormancy is typically deterministic in that the wake up mechanisms are usually not dependent upon a particular interleaving of threads.

iii) *Deadlock* occurs when a cycle of processes are mutually dependent upon each other at the same time. More precisely, N processes each wait on exclusive access to one of N resources while simultaneously holding exclusive access to another one of the N resources such that each process is awaiting access to a distinct resource. Deadlock is typically non-deterministic in that it is dependent upon the relative speeds of the processes acquiring the resources.

iv) *Premature Termination* occurs when a process ceases operation unexpectedly without properly notifying the other processes in the concurrent system. Such termination can result in both safety and liveness problems for the remaining processes. Premature termination is akin to a reversal of dormancy and is relatively easy to solve given appropriate exception handling.

[8]While corrupt data (the result of safety problems) are farely rare, who among us has not witnessed the *blue screen of death*?

Each of the types of liveness problems can cause a concurrent program to halt in an undesirable manner. While they are all challenging to deal with, in my experience deadlock stands out in a class of its own. In the best case scenario, deadlock is tied to the interleaving of the threads involved. This means that deadlock will non-deterministically occur based on the relative speeds of the threads and how the relative speeds impact thread interleaving. In the worst case scenario deadlock is not dependent upon relative thread speeds. In this case deadlock is intrinsic in the semantics of the communicating threads and there is no hope of evasion. Hence, in the worst case scenario there is nothing one can do while in the best case scenario one's view of the situation is blurred by randomness. Given the heightened difficulty of deadlock, I will focus on its representation.

Definition 4.1. DEADLOCK
A paired directed graph exhibits **deadlock** if and only if it contains a cycle. □

Nested diposets can not exhibit deadlock as per the Acyclic Diposet Theorem (Theorem 3.3). What Definition 4.1 tells us is that a software system that can be modelled by a nested diposet can not exhibit deadlock. To determine if a system exhibits deadlock, apply the relavent order and containment relationships and attempt to construct a diposet model of the system. *If it is possible to apply order and containment relationships without violating the nested containment rule and arrive at a cycle, then deadlock can occur.* In such instances the model is not a diposet but rather a paired directed graph that is not anti-symmetric. Otherwise, the model is a diposet and by definition it is deadlock free.

Deadlock often comes about through the use of multiple synchronization locks. As stated in the previous section on safety, synchronization locks that have a common label typically have an ordering constraint that requires that they be comparable. In conjunction with the order constraint on synchronization locks, deadlock-prone code often implements such locks so that they are contained by one another. This containment constraint can often contradict the ordering constraint and lead to deadlock. To illustrate this phenomenon see Figure 14. The first section (part a) of the figure shows a diposet consisting of two distinct threads each involving two events with the displayed labels. If we treat these events as the holding of synchronization locks, then we know that an ordering relation must be applied between the separate threads so that the locks are not concurrent. The next three sections of Figure 14 show distinct application of order constraints to the two threads. In each case, the applied order constraints do not violate the nested containment rules. In part d of Figure 14 the thick lines indicate that a cycle exists - *deadlock!* Given the order constraints imposed by the synchronization locks, it is possible for this system to experience deadlock and in fact the model in part d of Figure 14 is not a diposet.

Figure 15 consists of an alternative configuration such that the containment constraint of the left thread is reversed. Again, order constraints are applied to the nested diposet, however, because of the reversed containment constraint, order constraints must be applied in a manner different from Figure 14. In no case can order constraints be applied without violating the nested containment rule and lead to cycles. Thus, the configuration of this software system is not deadlock-prone. Note

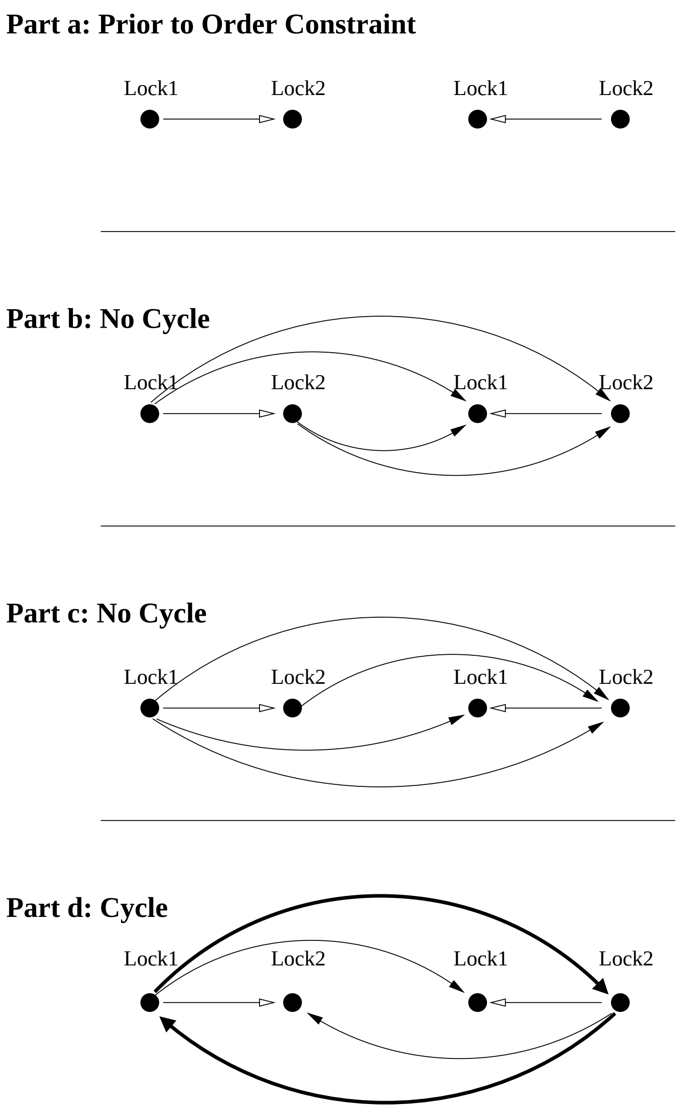

Figure 14: Order/Containment Constraints Leading To Deadlock

that there are only two ways to apply order constraints without violating the nested containment rules in Figure 15.

In considering Figures 14 and 15 note how the order and containment constraints come about in concurrent programs. Containment constraints are typically determined at compile time. How the source code of a program is written determines what the containment constraints will be. Order constraints between threads are typically determined at run-time and are a function of

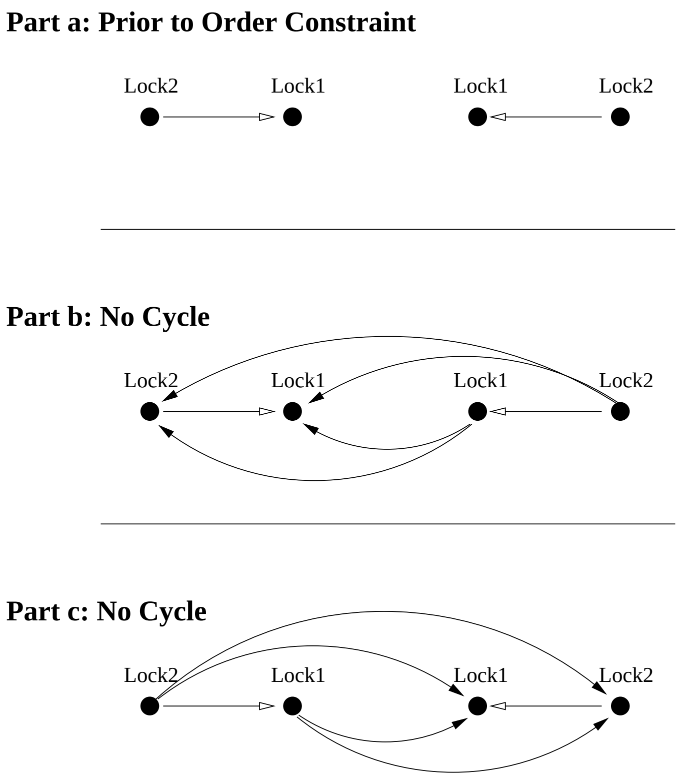

Figure 15: Order/Containment Constraints That Do Not Lead To Deadlock

relative thread speeds. This is why we show the containment constraints first followed by the order constraints.

4.3 Communication Semantics

Communication between threads imposes an order constraint on their composite diposet. These order constraints are precisely the communication order relations discussed in Section 3.1. In this section I will discuss two important communication styles and describe their corresponding communication order relationsh. Message passing is one of the fundamental ways to communicate within a concurrent system. Message passing communication assumes that components are connected via channels through which messages are transmitted. There are two types of message passing communication: synchronous and asynchronous. *Synchronous message passing* requires both the sender and receiver connected by a channel to be synchronized when a communication occurs. *Asynchronous message passing* does not require the sender and receiver to be simultaneously engaged and involves a storage facility in which messages can be placed by the sender until the receiver is ready.

The communication order relations for asynchronous message passing is very simple. Given that the sending event is denoted x and the receiving event is denoted y, an asynchronous message passing communication order relation for x and y is written

$$C_O(x,y) = \{(x,y)\}.$$

In other words, x must precede y. A graphical example of such a relation is shown in Figure 16. Here the left thread communicates to the right thread. Event x is the sending event and event y is the receiving event.

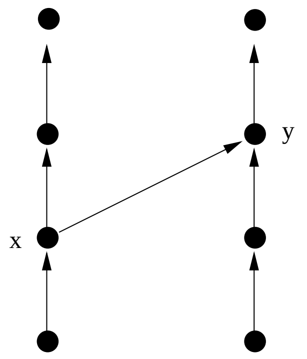

Figure 16: A Diposet Representing Asynchronous Communication

The communication order relation for synchronous message passing is significantly more complex than asynchronous message passing. Given that the sending event is denoted x and the receiving event is denoted y, a synchronous message passing communication order relation for x and y is equivalent to that given in Equation 1. For convenience I have rewritten Equation 1 below. Note that synchronous message passing is symmetric; i.e., $C_O(x,y) = C_O(y,x)$. Thus, there is no need to differentiate a sender and receiver. The graphical representation of synchronous message passing is shown in Figure 17 in which event x and y are synchronous.

$$C_O(x,y) = \begin{array}{l} \{(x,y')|y' \in y_{up-set_O}\} \cup \{(y',x)|y' \in y_{down-set_O}\} \\ \cup\{(x',y)|x' \in x_{down-set_O}\} \cup \{(y,x')|x' \in x_{up-set_O}\} \end{array}$$

4.4 An Example: PtPlot And The Java$^{\text{TM}}$Swing Package

I conclude this paper with an informative and real world example by demonstrating how diposets can model the threading mechanism that is part of the Swing package of the Java$^{\text{TM}}$programming language. The Java$^{\text{TM}}$Swing package consists of a set of graphical user interface (GUI) components that have a pluggable look and feel. The *pluggable look and feel* lets one design a single set of GUI

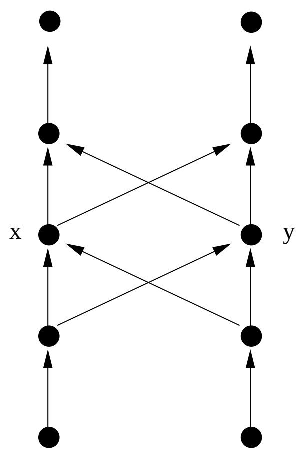

Figure 17: A Diposet Representing Synchronous Communication

components that can automatically have the look and feel of any OS platform (e.g., Microsoft Windows^{TM}, Sun Solaris^{TM}, Apple Macintosh^{TM}). As with all GUIs, the Swing graphical user interface must respond both to human input such as mouseclicks and text entry as well as computer input such as new images or windows to display. Responding to both computer and human input is an inherently concurrent process. Swing addresses this concurrency with a single event dispatch thread for all GUI operations.

The Swing *event dispatch thread* takes events (e.g., the pressing of a button or clicking of a mouse) and schedules them to occur in a sequential order. The `invokeAndWait()` and `invokeLater()` methods are available so that other threads in a program can access the event dispatch thread (these methods are part of the `javax.swing.SwingUtilities` class). The `invokeAndWait()` method communicates synchronously with the event dispatching thread. The `invokeLater()` method communicates asynchronously with the event dispatching thread. Improper use of the `invokeAndWait()` or `invokeLater()` methods is a greater source of confusion among Swing users and can result in deadlock.[9]

PtPlot, created by Edward A. Lee and Christopher Hylands of UC Berkeley, is an example Java^{TM}program that uses the Swing package (Davis *et al.*, 1999, chapter 10).[10] PtPlot consists of Java^{TM}classes (many of which are Swing classes) that plot data on a graphical display. The main thread in the program is part of the `Plot` class `run()` method. This thread (I'll refer to it as the *PtPlot thread*) repeatedly calls the `Plot.addPoint()` method. `addPoint()` synchronizes on the `Plot` object lock and then attempts to draw points on the display. This latter task (drawing points on the display) requires the PtPlot thread to communicate with the Swing event dispatch thread. Separate

[9]For a glimpse at the headaches faced by users of the two invoke methods, view `http://forum.java.sun.com/` and search on `invokeLater`.

[10]PtPlot is available at `http://ptolemy.eecs.berkeley.edu/java/ptplot`.

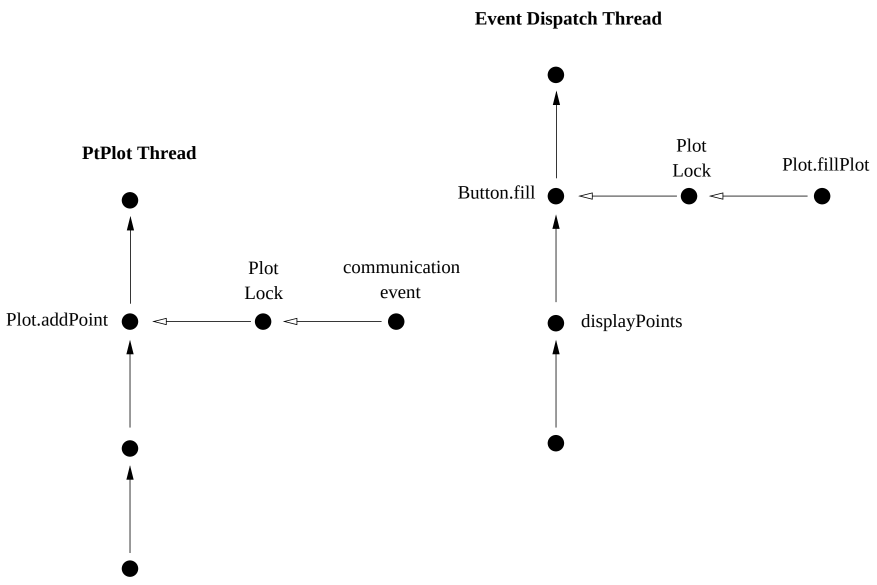

Figure 18: The Separate Threads In PtPlot

from the Plot thread are several buttons for modifying the view of the PtPlot display. One such button is the *fill button.* If a user clicks on the fill button the `ButtonListener.actionPerformed()` method will be called and this in turn calls the `Plot.fillPlot()` method. The `Plot.fillPlot()` method is synchronized on the `Plot` object lock. Since the fill button is a swing component, `ButtonListener.actionPerformed()` and all of its contents are part of the event dispatch thread.

In order for the PtPlot thread to actually add points to the display, it must communicate with the event dispatch thread either through the `invokeLater()` method or the `invokeAndWait()` method. Diposets illustrate how the former method is deadlock free while the latter is deadlock prone. Figure 18 shows the two separate threads - the PtPlot thread and the event dispatch thread - without communication between them. Two order constraints must be added to this figure. The first constrains the two invocations of the `Plot` lock to not occur concurrently. The second constraint is due to the communication between the PtPlot thread and the event dispatch thread. This second constraint is a function of the `displayPoints` event and the event labelled "`communication event`." Figure 19 shows both constraints added to the two threads. The upper section of Figure 19 consists of the asynchronous constraint that is imposed by `invokeLater()`. The lower section of Figure 19 uses the synchronous constraint of `invokeAndWait()`. In this latter case a cycle exists.

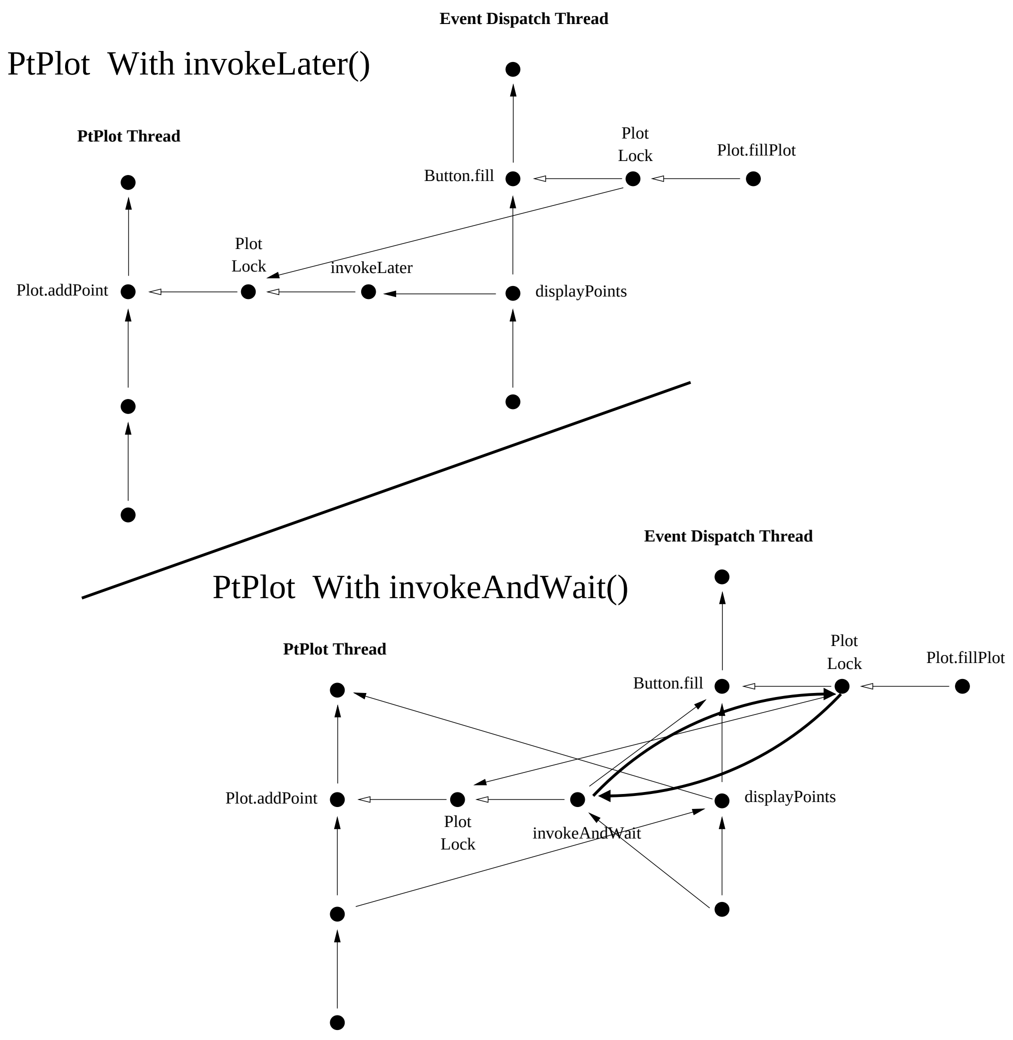

Figure 19: PtPlot and the Java^{TM}Swing Event Dispatch Thread

5 Conclusion

The goal of this paper was to introduce the diposet and illustrate its usefulness as a tool for modeling and designing concurrent systems. To achieve this goal, I first established the difficulty of concurrent system design in Section 1 and I argued the importance of a design methodology that is unambiguous, general, graphical and formal. In Section 2, I considered the properties of concurrent systems and recognized the importance of the order and containment relationships between the components found in concurrent systems. I then considered several techniques that satisfied my criteria of Section 1 including petri nets and directed graphs but showed that these methods were insufficient because of their inability to characterize order and containment.

In Section 3, I introduced the diposet. I presented the definition of the diposet as well as several key variants, proved related theorems and illustrated the graphical representation of the diposet. The contents of this section served to make clear that a diposet is unambiguous, general, graphical and formal. Section 4 consisted of application of the diposet to concurrent programming. In particular, I represented safety and liveness via diposets and used diposets as the basis for a variety of communication semantics. I concluded Section 4 with an example of the usefulness of diposets in modeling concurrency in the Java$^{\text{TM}}$Swing Package.

The concept of using diposets to model and design concurrent systems is a nascent endeavor. I believe that it can serve as the basis for a radically new approach to programming but there is much work to do before such a day arrives. One useful area of study for diposets would endeavor to extend the application of diposets to model a greater set of concurrent system semantics. I applied diposets to message passing semantics but I did not consider shared memory communication. While the set of message passing semantics is large enough to single handedly justify diposets, shared memory systems are widely used and merit consideration for being modeled by diposets as well. Diposets should also be considered outside the space of concurrent software systems. Partially ordered set and lattice theory has been applied to the humanities and social sciences; for example, the mathematics of ordered sets are used to arrive at consensus among selection committees and within the field of market research (Davey and Priestley, 1990). It is likely that diposets will also find fruitful application in these and other disciplines.

Before delving into new application spaces for diposets, perhaps the most important future work on diposets is a study of the practical efficacy of diposets as applied to concurrent programming; i.e., do programmers and software engineers find diposets intuitive and usable? It would be worthwhile to study diposets in the modeling and design of a real world large-scale software system. Only through use can we truly understand the effectiveness of diposets. Such a study would reveal the aspects of the diposet that should be highlighted and the disadvantages of the diposet that should be removed.

References

Agha, G. A. (1986). *ACTORS: A Model of Concurrent Computation in Distributed Systems*. The MIT Press Series in Artificial Intelligence. MIT Press, Cambridge.

Alexander, C., Ishikawa, S., and Silverstein, M. (1977). *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press.

Allen, R. and Garlan, D. (1997). A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*.

Alur, R. and Henzinger, T. A. (1994). Real-time system = discrete system + clock variables. In T. Rus and C. Rattray, editors, *Theories and Experiences for Real-time System Development*, AMAST Series in Computing 2, pages 1–29. World Scientific.

Alur, R. and Henzinger, T. A. (1996). Reactive modules. In *Proceedings of the 11th IEEE Symposium on Logic in Computer Science*, pages 207–218.

Andrews, G. R. (1991). *Concurrent Programming: Principles and Practice*. Benjamin/Cummings, Redwood City, California.

Backus, J. (1978). Can programming be liberated from the von neumann style? a functional style and its algebra of programs. *Communications of the ACM*, **21**(8), 613–641.

Basten, T., Kunz, T., Black, J. P., Coffin, M. H., and Taylor, D. J. (1997). Vector time and causality among abstract events in distributed computations. *Distributed Computing*, **11**(1), 21–39.

Benveniste, A. (1998). Compositional and uniform modeling of hybrid systems. *IEEE Transactions on Automatic Control*, **43**(4), 579–584.

Bhattacharyya, S. and Lee, E. (1994). Looped schedules for dataflow descriptions of multirate signal processing algorithms. *IEEE Transactions on Signal Processing*, **42**(5).

Boehm, B. W. (1976). Software engineering. *IEEE Transactions on Computers*, **C-25**(12), 1226–1241.

Booch, G. (1994). *Object-Oriented Analysis and Design*. The Benjamin/Cummings Pulishing Company, Redwood City, CA, 2nd edition.

Booch, G., Rumbaugh, J., and Jacobson, I. (1999). *The Unified Modeling Language User Guide*. Addison-Wesley, Reading, MA.

Breal, M. (1991). *The Beginnings of Semantics: Essays, Lectures and Reviews*. Stanford University Press, Stanford, California.

Brookes, S. D. (1999). Communicating parallel processes. In *Symposium in Celebration of the Work of C.A.R. Hoare*.

Brookes, S. D. and Dancanet, D. (1995). Sequential algorithms, deterministic parallelism, and intensional expressiveness. In *22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Francisco, CA. ACM Press.

Brooks Jr., F. P. (1975). *The Mythical Man-Month*. Addison-Wesley, Reading, MA.

Brooks Jr., F. P. (1987). No silver bullet: Essence and accidents of software engineering. *Computer*, **20**(4), 10–19.

Brown, R. (1988). Calendar queues: A fast o(1) priority queue implementation for the simulation event set problem. *Communications of the ACM*, **31**(10), 1220 – 1227.

Buck, J. (1994). Static scheduling and code generation from dynamic dataflow graphs with integer-valued control streams. In *Proceedings of the 28th Annual Asilomar Conference on Signals, Systems, and Computers*, Pacific Grove, CA.

Buck, J., Ha, S., Lee, E. A., and Messerschmitt, D. G. (1994). Ptolemy: A framework for simulating and prototyping heterogeneous systems. *International Journal of Computer Simulation*, **4**, 155–182.

Chandy, K. M. and Misra, J. (1981). Asynchronous distributed simulation via a sequence of parallel computations. *Communications of the ACM*, **24**(11), 198–206.

Chang, K.-T. and Krishnakumar, A. S. (1993). Automatic functional test generation using the extended finite state machine model. In *Proceedings of the 30th Design Automation Conference*, pages 86–91. The Association for Computing Machinery, Inc.

Chase, C. M. and Garg, V. K. (1998). Detection of global predicates: Techniques and their limitations. *Distributed Computing*, **11**, 191–201.

Chen, W. (1997). *Graph Theory And Its Engineering Applications*. World Scientific, Singapore.

Chu, P.-Y. M. and Liu, M. T. (1989). Global state graph reduction techniques for protocol validation in the efsm model. In *Eighth Annual International Phoenix Conference on Computers and Communications*, pages 371–377.

Conway, M. E. (1963). Design of a separable transition-diagram compiler. *Communications of the ACM*, **6**(7).

Dalpasso, M., Bogliolo, A., and Benini, L. (1999). Virtual simulation of distriputed ip-based designs. In *Proceedings of the 36th Design Automation Conference*, pages 50–55. The Association for Computing Machinery, Inc.

Daniel, R. (1998). Embedding Ethernet connectivity. *Embedded Systems Programming*, **11**(4), 34 – 40.

Davey, B. A. and Priestley, H. A. (1990). *Introduction to Lattices and Order*. Cambridge University Press.

Davis, J. S., Galicia, R., Goel, M., Hylands, C., Lee, E. A., Liu, J., Liu, X., Muliadi, L., Neuendorffer, S., Reekie, J., Smyth, N., Tsay, J., and Xiong, Y. (1999). Ptolemy II: Heterogeneous concurrent modeling and design in java. Memorandum No. UCB/ERL M99/40, University of California, Berkeley, Department of EECS, University of California, Berkeley, CA, 94720.

Davis, R. E. (1989). *Truth, Deduction, and Computation: Logic and Semantics for Computer Science*. Computer Science Press.

Davis II, J. S., Goel, M., Hylands, C., Kienhuis, B., Lee, E. A., Liu, J., Liu, X., Muliadi, L., Neuendorffer, S., Reekie, J., Smyth, N., Tsay, J., and Xiong, Y. (1999). Overview of the ptolemy project. Memorandum No. UCB/ERL M99/37, University of California, Berkeley, Department of EECS, University of California, Berkeley, CA, 94720.

de Bakker, J. and de Vink, E. (1996). *Control Flow Semantics*, chapter True Concurrency, pages 473–490. Foundations of Computing. MIT Press, Cambridge, Massachusetts.

Dijkstra, E. (1965). Solution of a problem in concurrent programming control. *Communications of the ACM*, **8**(9), 569.

Dijkstra, E. (1968a). The structure of the "the" multiprogramming system. *Communications of the ACM*, **11**(5), 341–346.

Dijkstra, E. W. (1968b). *Programming Languages*, chapter Co-operating Sequential Processes, pages 43–112. NATO Advanced Study Institute. Academic Press, London.

Fidge, C. J. (1991). Logical time in distributed systems. *Computer*, **24**(8), 28–33.

Foster, I. and Kesselman, C., editors (1999). *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, Inc., San Francisco, California, 1st edition.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Garey, M. and Johnson, D. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, New York.

Gibbs, W. W. (1994). Software's chronic crisis. *Scientific American*, **271**(3), 86–95.

Girault, A., Lee, B., and Lee, E. (1999). Hierarchical finite state machines with multiple concurrency models. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, **18**(6).

Godefroid, P. (1996). *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State Space Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin.

Gordon, M. J. C. (1979). *The Denotational Description of Programming Languages*. Springer-Verlag.

Gunter, C. A. (1992). *Semantics of Programming Languages: Structures and Techniques*. Foundations of Computing Series. The MIT Press.

Halbwachs, N. (1993). *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, Dordrecht.

Hoare, C. A. R. (1985). *Communicating Sequential Processes*. Prentice-Hall International, New Jersey.

Hopcroft, J. E. and Ullman, J. D. (1979). *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, Massachusetts.

Hudak, P. (1989). Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, **21**(3), 359–411.

Jefferson, D. R. (1985). Virtual time. *ACM Transactions on Programming Languages and Systems*, **7**(3), 404–425.

Jones, C. B. (1999). Compositionality, interference and concurrency. In *Symposium in Celebration of the Work of C.A.R. Hoare*.

Kahn, G. (1974). The semantics of a simple language for parallel programming. In *Proceedings of the IFIP Congress 74*, pages 471–475, Paris, France. International Federation for Information Processing, North-Holland Publishing Company.

Kahn, G. and MacQueen, D. B. (1977). Coroutines and networks of parallel processes. In *Proceedings of the IFIP Congress 77*, pages 993–998, Paris, France. International Federation for Information Processing, North-Holland Publishing Company.

Kienhuis, A. (1999). *Design Space Exploration of Stream-based Dataflow Architectures: Methods and Tools*. Delft University of Technology, Amsterdam, The Neterlands.

Kundu, J. and Cuny, J. E. (1995). The integration of event- and state-based debugging in ariadne. In *International Converence on Parallel Processing*, volume II, pages 130–134. CRC Press.

Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, **21**(7), 558–565.

Lea, D. (1997). *Concurrent Programming in Java*. Addison-Wesley, Reading, MA.

Lee, E. (1999a). Embedded software - an agenda for research. Memorandum No. UCB/ERL M99/63, University of California, Berkeley, Department of EECS, University of California, Berkeley, CA, 94720.

Lee, E. (1999b). Modeling concurrent real-time processes using discrete events. *Annals of Software Engineering*, **7**, 25–45.

Lee, E. and Messerschmitt, D. (1987). Synchronous data flow. *Proceedings of the IEEE*.

Lee, E. and Xiong, Y. (2000). System-level types for component-based design. Memorandum No. UCB/ERL M00/8, University of California, Berkeley, Department of EECS, University of California, Berkeley, CA, 94720.

Lee, E. A. (1997). A denotational semantics for dataflow with firing. Memorandum No. UCB/ERL M97/3, University of California, Berkeley, Department of EECS, University of California, Berkeley, CA, 94720.

Lee, E. A. and Parks, T. M. (1995). Dataflow process networks. *Proceedings of the IEEE*, **83**(5), 773–801.

Lee, E. A. and Sangiovanni-Vincentelli, A. (1997). A denotational framework for comparing models of computation. Memorandum No. UCB/ERL M97/11, University of California, Berkeley, Department of EECS, University of California, Berkeley, CA, 94720.

Luckham, D. C., Vera, J., and Meldal, S. (1995). Three concepts of system architecture. Technical Report CSL-TR-95-674, Stanford Computer Science Lab.

Lynch, N. A. (1996). *Distributed Algorithms*. Morgan Kaufmann Publishers, San Francisco.

Lynch, N. A. and Fischer, M. J. (1981). On describing the behavior and implementation of distributed systems. *Theoretical Computer Science*, **13**(1), 17–43.

Magee, J. and Kramer, J. (1999). *Concurrency: State Models & Java Programs*. John Wiley & Sons.

Mattern, F. (1989). Virtual time and global states of distributed systems. In *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226.

Matthews, S. G. (1993). An extensional treatment of lazy data flow networks. *Theoretical Computer Science*, **151**(1), 195 – 205.

Meyer, B. (1999). Every little bit counts: Toward more reliable software. *Computer*, **32**(11), 131–133.

Milner, R. (1989). *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, London.

Milner, R. (1993). Elements of interaction: Turing award lecture. *Communications of the ACM*, **36**(1), 78–89.

Milner, R. (1999). Computing and communication - what's the difference? In *Symposium in Celebration of the Work of C.A.R. Hoare*.

Minas, M. (1995). Detecting quantified global predicates in parallel programs. In *First International EURO-PAR Conference*, volume 966 of *Lecture Notes in Computer Science*, pages 403–414, Stockholm, Sweden. Springer-Verlag.

Morgan, C. (1985). Global and logical time in distributed algorithms. *Information Processing Letters*, **20**(4), 189–194.

Moschovakis, Y. N. (1994). *Notes on Set Theory*. Springer-Verlag.

Mowbray, T. J. and Malveau, R. C. (1997). *CORBA: Design Patterns*. Wiley Computer Publishing.

Murphy, D. (2000). Still flying high. *San Francisco Examiner*, page J1.

Murthy, P. K. (1996). *Scheduling Techniques for Synchronous and Multidimensional Sychronous Dataflow*. Ph.D. thesis, University of California, Berkeley.

Neggers, J. and Kim, H. (1998). *Basic Posets*. World Scientific, Singapore.

Nicollin, X. and Sifakis, J. (1994). The algebra of timed process, ATP: Theory and application. *Information and Computation*, **114**, 131–178.

Nygaard, K. and Dahl, O. (1981). *The Development of the Simula Languages*. Academic Press, New York, NY.

Passerone, R., Rowson, J., and Sangiovanni-Vincentelli, A. (1998). Automatic synthesis of interfaces between incompatible protocols. In *Proceedings of the 35th Design Automation Conference*. The Association for Computing Machinery, Inc.

Peterson, J. (1981). *Petri Net Theory and the Modeling of Systems*. Prentice Hall, Englewood Cliffs, NJ.

Petri, C. (1962). *Kommunikation mit Automaten*. German language, University of Bonn, Bonn, Germany.

Pino, J. L., Bhattacharyya, S. S., and Lee, E. A. (1995). A hierarchical multiprocessor scheduling system for dsp applications. In *Proceedings of the 29th Annual Asilomar Conference on Signals, Systems, and Computers*, Pacific Grove, CA.

Press, I. C. S., editor (1997). *Proceedings of the Sixth IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems*, Tunis, Tunisia.

Righter, R. and Walrand, J. (1989). Distributed simulation of discrete event systems. *Proceedings of the IEEE*, **77**(1), 99–113.

Rowson, J. A. and Sangiovanni-Vincentelli, A. (1997). Interface-based design. In *Proceedings of the 34th Design Automation Conference*, pages 178–183. The Association for Computing Machinery, Inc.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W. (1991). *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, NJ.

Schmidt, D. A. (1986). *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, Newton, Massachusetts.

Schneider, F. (1997). *On Concurrent Programming*. Graduate Texts in Computer Science. Springer Verlag, New York.

Sebesta, R. W. (1996). *Concepts of Programming Languages*. Addison-Wesley, 3rd edition edition.

Shiple, T. R. (1993). Survey of equivalences for transition systems. Unpublished, Department of EECS, University of California, Berkeley.

Smyth, N. (1998). *Communicating Sequential Processes Domain in Ptolemy II*. Memorandum No. UCB/ERL M98/70, University of California, Berkeley, Department of EECS, University of California, Berkeley, CA, 94720.

Stoltenberg-Hansen, V., Lindstrom, I., and Griffor, E. R. (1994). *Mathematical Theory of Domains*. Cambridge University Press.

Sztipanovits, J., Karsai, G., and Bapty, T. (1998). Self-adaptive software for signal processing. *Communications of the ACM*, **41**(5), 66–73.

Tennent, R. D. (1991). *Semantics of Programming Languages*. Prentice Hall, London.

Tomlin, C., Pappas, G. J., and Sastry, S. (1998). Conflict resolution for air traffic management: A study in multiagent hybrid systems. *IEEE Transactions on Automatic Control*, **43**(4), 509–521.

van Glabbeek, R. J. and Weijland, W. P. (1996). Branching time and abstraction in bisimulation semantics. *Journal of the ACM*, **43**(3), 555–600.

Vuillemin, J. (1974). Correct and optimal implementations of recursion in a simple programming language. *Journal of Computer and System Sciences*, **9**(3), 332–354.

Wegner, P. (1997). Why interaction is more powerful than algorithms. *Communications of the ACM*, **40**(5).

West, D. (1996). *Introduction to Graph Theory*. Prentice Hall, Upper Saddle River, New Jersey.

Wexler, J. (1989). *Concurrent Programming in Occam 2*. Ellis Horwood Series in Computers and Their Applications, England.

Wilson, L. B. and Clark, R. G. (1993). *Comparative Programming Languages*. Addison-Wesley, second edition edition.

Winskel, G. (1994). *The Formal Semantics of Programming Languages: An Introduction*. Foundations of Computing. MIT Press, Cambridge.

Yates, R. K. (1993). Networks of real-time processes. In *International Conference on Concurrency Theory*, Lecture notes in computer science, pages 384–397.

Young, J. S., MacDonald, J., Shilman, M., Tabbara, A., Hilfinger, P., and Newton, A. R. (1998). Design and specification of embedded systems in java using successive, formal refinement. In *Proceedings of the 35th Design Automation Conference*. The Association for Computing Machinery, Inc.

Yourdon, E., editor (1979). *Classics in Software Engineering*. Yourdon Press, New York, NY.

Yourdon, E. (1993). *Decline & Fall of the American Programmer*. Yourdon Press, Englewood Cliffs, New Jersey.

Index