A Theory of Distributed Time

Sean W. Smith

December 1993

CMU-CS-93-231

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Natural intuition organizes experience into a linear sequence of discrete events, but this approach is inappropriate for asynchronous distributed systems, where information is distributed and perception is delayed. Distributed environments require a distributed notion of time, to abstract away not only irrelevant physical detail but also irrelevant temporal and computational detail. By expressing distributed systems concepts that are difficult to talk about in terms of real time and by distinguishing what really "happens" from what physically occurred, a theory of *distributed time* would provide a natural framework for solving problems in distributed environments. This paper lays the groundwork for that claim by formally building such a theory. This research improves on previous work on time in distributed systems by supporting temporal relations more general than partial orders, by supporting abstraction through multiple levels of temporal relations, by separating the family of temporal relations an application consults from the particular clock implementations that track them, and by providing a single arena to consider these issues for a wide range of applications.

This research was sponsored by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under grant number F33615-93-1-1330. Additional support was provided by NSF Grant CCR-8858087, by matching funds from Motorola and TRW, by the U.S. Postal Service, and by an ONR Graduate Fellowship. The author is grateful to IBM for equipment to support this research.

Carnegie Mellon Computer Science Technical Report CMU-CS-93-231.

Keywords: Distributed systems, concurrency, security and protection, checkpoint/restart, fault tolerance

Contents

List of Figures

Acknowledgments

The author is extremely grateful to Doug Tygar for his advice and continual encouragement, and to Dave Johnson for his painstaking reading and discussion of many early drafts of this document.

Chapter 1

Introduction

Traditionally, we think of computation as some set of things that happen. Since things happen in real time, we can use real time to organize these events into a linear sequence. By imposing a discrete structure on events, this traditional view already performs abstraction: full physical detail does not express what really "happens." The advent of asynchronous distributed computation extends this abstraction to time: if two events occur without knowledge of each other, then their real time sequence does not matter [La78,Pr86]. Expressing what really "happens" in a *distributed* computation requires a theory of *distributed time* that abstracts away both irrelevant physical detail and irrelevant temporal detail.

A theory of *distributed time* has practical motivations and uses. Many application problems in asynchronous distributed systems reduce to asking questions about temporal relations other than the natural real time sequence. Thinking in terms of these alternative temporal relations would clarify these problems; providing clocks for these relations would simplify protocol design. Indeed, building protocols for these problems requires confronting these clock issues in one form or another. However, doing wonderful things with alternative temporal relations requires understanding the underlying framework. This paper considers the question of the appropriate notion of time for distributed systems, and develops formal mechanisms for a theory of *distributed time*. Later papers will use these mechanisms to build a framework for secure applications.

Previous research developed the notion of time as a partial order. Lamport [La78] used partial orders to track causal dependency in distributed systems; Pratt [Pr86] argued for the universality of partial order time. Fidge [Fi88] and Mattern [Ma89] explored partial order time and built vector clocks; the author explored security issues in tracking partial order time.[1] Other research includes calls for departing from the order of real time ([Je85] uses total orders; [Gr75] uses partial orders), and explorations of the role of partial orders and asynchrony in application problems such as communication [BiJo87, PBS89], distributed debugging [Fi89, Sp89], deadlock detection [Ma87, TaLo91], distributed snapshots [ChLa85, Ma93], and rollback recovery [StYe85, Jo89, JoZw90, PeKe93].

This paper improves on earlier work by providing a single, general theory of distributed time suitable for a wide range of applications. By supporting temporal relations more general than

[1]The author's Ph.D. proposal [Sm91] discusses these issues, and presents a secure protocol for partial order time. [ReGo93] also explores security for partial order clocks; more recent work by the author [SmTy93] improves on these earlier protocols. [AmJa93] considers some related security issues.

partial orders[2] and by supporting hierarchies of temporal abstraction, this theory can express the computational abstraction appropriate for families of application problems. By providing a general approach to distributed time, this theory allows us to unify in a single framework protocols that separately consult and affect time, and to consider once the clock issues central to each separate protocol. By introducing orthogonality between temporal relations and the clocks that track them, this theory allows us to consider (and alter) clock implementations without changing higher-level protocols.

The author's current research [Sm94] involves building a single arena to analyze the temporal aspects of distributed application problems, to design protocols in terms of distributed time primitives, and to independently consider secure implementations of these primitives. This paper provides a theoretical foundation for that work.

1.1. Describing Computation

Loosely speaking, we use time to identify the things that happen and the order in which they happen. What is the best way to describe what actually "happens" in a computation?

Describing Physical Reality On a basic level, computation is a physical activity. Physical devices react to each other and the environment as time progresses. From this perspective, the best description is a straightforward record of the physical activity: the notebook of an omniscient observer who, each time something changes, glances at his watch and jots down what occurred and when. Figure 1.1 gives a toy example.

Abstracting to Discrete Events However, merely recording physical activity is too naive. Even the above toy example reveals a fundamental problem with this approach: *granularity.* Recording a list requires imposing a granularity on actions: one thing happens, then another, then another. This imposition raises two issues.

First, the granularity we desire when describing computation is usually far coarser than the level in an exhaustive physical description. A computational event represents some bundle of physical events. Figure 1.2 illustrates this abstraction.

```
0:01  0:03  0:04  0:07  0:08  0:09  0:11  0:13
LDR1  INR1  STR1  NOP   LDR1  LDR2  ADD   STR1
```

Figure 1.1 An example of an exhaustive physical description is a timestamped list of machine instructions.

[2]For example, non-transitive relations and cyclic relations both have some use.

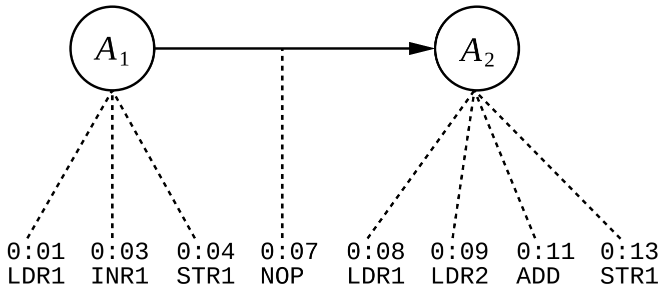

Figure 1.2 We may abstract away from the physical description by bundling basic physical events into computational events. The detailed machine code becomes "event A_1, then event A_2."

Secondly, even constructing an exhaustive physical description begs the granularity question. Why should we record machine instructions, rather than gate firings, transistor activity, or subatomic particles? Event abstraction continues at lower layers. Figure 1.3 sketches one approach.

Abstracting from Real Time If physical computation is taking place in a distributed environment, then the physical description should indicate not only *when* things happen, but also *where.* The computational level should leave concurrent any events that represent simultaneous activity. (See Figure 1.4.)

Suppose the system is *asynchronous* as well. Events A and B were not genuinely simultaneous but only apparently simultaneous: that is, they had no knowledge of each other. Then we may still want to leave them unordered. (See Figure 1.5.)

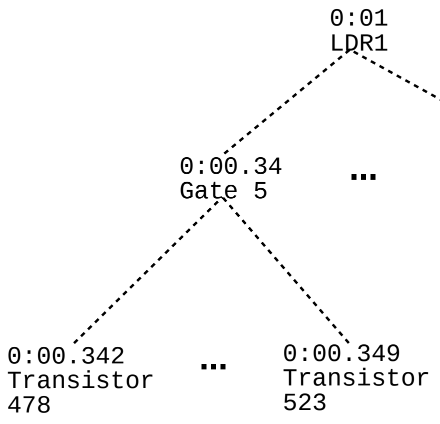

Figure 1.3 The physical description is itself an abstraction: each instruction may represent gate firings or transistor actions. The level of description we choose for our base is essentially arbitrary.

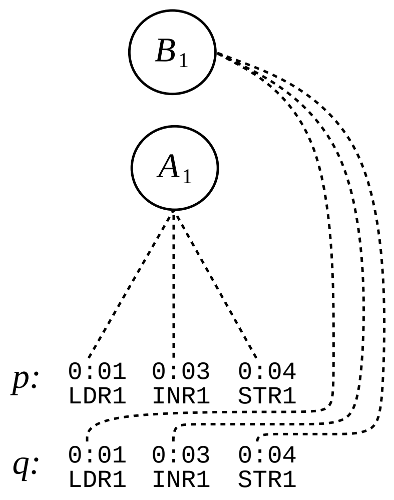

Figure 1.4 Abstract events A_1 and B_1 represent genuinely simultaneous computation; we regard these events as concurrent.

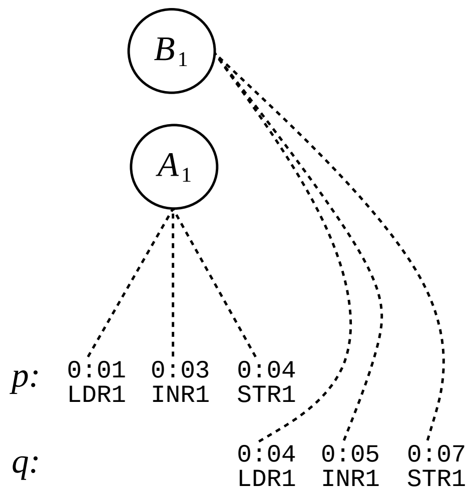

Figure 1.5 Abstract events A_1 and B_1 now represent computation that only "appears" simultaneous; nevertheless, we still regard these events as concurrent.

Lacking any access to real-time clocks and unable to perceive each other except through messages, process p and process q cannot distinguish the actual physical order of A and B in this example. Hence we have not just condensed physical activity to events and removed edges; we have condensed a *set* of physical descriptions to a single computational description. (See Figure 1.6.) The processes should not know which physical description in this class is the "true" description.

Abstracting from Abstractions Situations arise when even a single layer of abstraction does not suffice. For example, consider the problem of *rollback*: modifying the computation so that certain events appear to have never occurred. (Rollback arises arises when considering fault-tolerance and checkpointing [Jo89, JoZw90], and will be considered in subsequent work.) Suppose process p wants to roll back event A_2 and execute $A_2{}'$ instead. Initially we pretended that the computational description, not the physical description, is what "really happens." But now we want to ignore detail in the computational description as well—we want to abstract away the original event A_2, and the rolled back computation that depended on it. Figure 1.7 sketches how rollback induces two levels of abstraction.

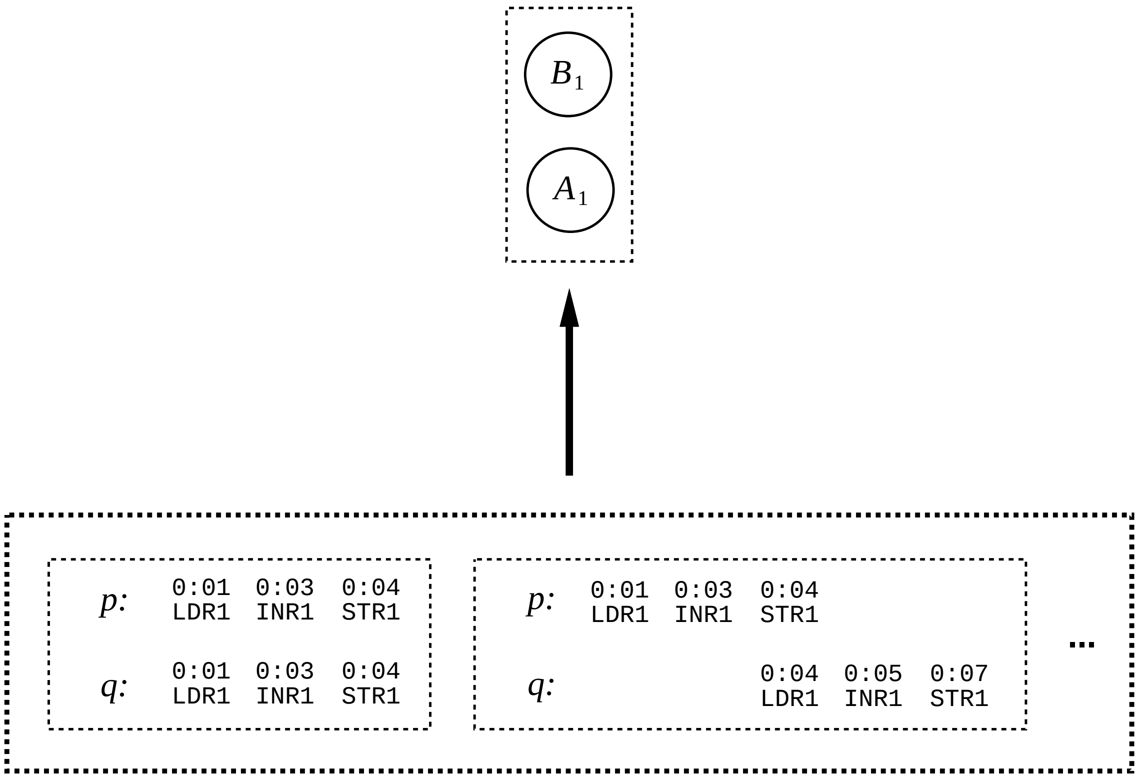

Figure 1.6 An abstract computation graph represents a set of possible physical computations. Once we abstract to the graph, we forget the presumably irrelevant physical detail.

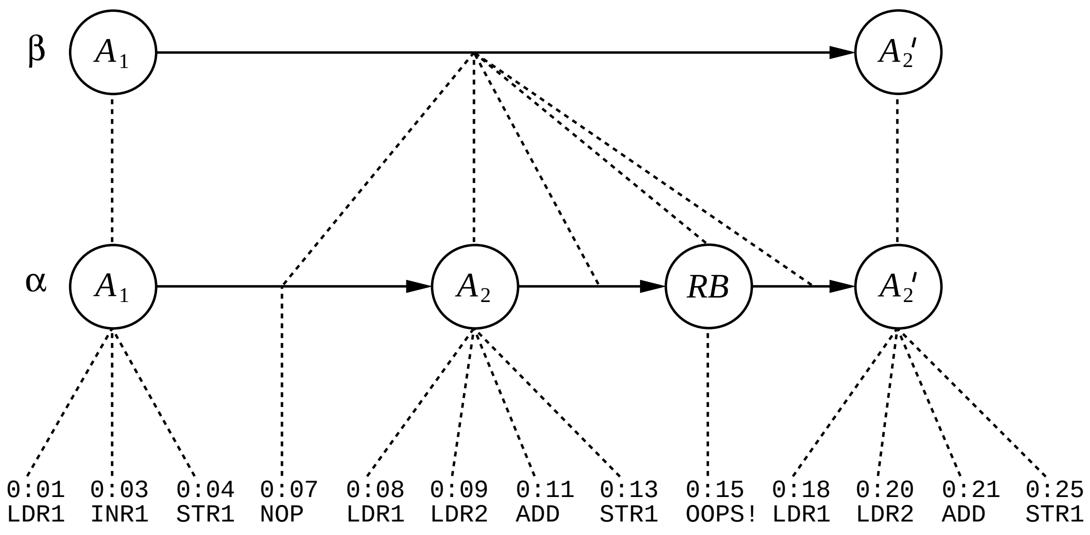

Figure 1.7 Rollback induces two levels of abstraction. We prune away irrelevant machine details to obtain description α; however, we presumably want to prune away irrelevant rollback details to obtain the "real" description β.

1.2. Distributed Time

These informal sketches demonstrate some issues critical to building a theory of time.

- We want to represent a computation as some abstract set of "things that happen," with a relation indicating the temporal order in which these things happened.
- The components in these abstractions themselves represent various parts of the exhaustive physical description.
- These abstractions should permit temporal relations more general than that of linear time.

The rollback example of Section 1.1 motivates two more issues:

- We need to distinguish between the way we obtain the abstract representations, and the representations themselves (since we may have multiple routes to the same representation).
- We will want to apply abstractions to abstractions.

We conclude that a general theory of *distributed time* should contain three components:

- a standard format for these abstract representations (so we can talk about computations)
- a way to specify *time models*: representational transformations on these objects (so we can abstract from one representation to another)

- a way to translate some level of physical description into this format (so our chains of abstraction have some footing in reality)

Once we develop a framework for distributed time, the challenge remains of developing and using models in this framework. Our sketches in Section 1.1 featured two implicit goals:

- to express the ordering that processes in an asynchronous distributed system perceive
- to use some natural level of discrete events

Initially we see two principal motivations for using distributed time models:

Best Approximation of Reality If the complete physical description is unavailable, our time model should express as much as we can know about it.

Convenient Expressiveness If the complete physical description obscures key concepts, then our time model should provide a more appropriate description.

The rollback example of Section 1.1 raises a third motivation:

Virtual Computation If the processes collectively pretend that the "current" computation differs from the one a complete physical description would record, then our time model should express this abstraction.

If an application problem broaches these issues, then distributed time will be relevant to that application. We quickly sketch a few examples:

- The problem of *distributed snapshots* consists of one process trying to take a snapshot of the state of the system at some instant. Distribution and asynchrony impose knowledge limitations that make this task difficult: anything that the process can perceive about the rest of the system is out-of-date.
- The problem of *orphan detection* requires determining if a given event might have perceived (and thus depend on) an aborted event. This perceive/depend relation forms a partial order—real time alone fails to give enough information.
- The problem of *rollback* requires modifying the computation to pretend that a simpler one (or at least a different one) occurred. The processes cooperate to add an additional level of abstraction, and this new level—describing a fault-free computation that never really happened—becomes the "real" computation.

Chapter 15 will return to these topics, and subsequent work will explore them more thoroughly.

1.3. Overview of this Paper

This paper formally develops a theory of distributed time. The initial goal is to build a framework to express the ordering perceivable in asynchronous distributed systems; however, the framework will extend to wider domains.

As we already observed, computation is fundamentally a physical activity; hence talking about abstract representations of computation requires choosing some arbitrary level of physical description. Part I presents our system model, the level of physical description we choose for this work.

Part II builds the machinery for *time models*. This construction follows the schema of Section 1.2: we develop a *computation graph* format for abstract representations, translate the physical description into this format, and build a family of representational transformations

Part III explores the relationship between modeled time and real time—the relationship between logical simultaneity and genuine simultaneity. We extend the time model machinery to apply to parallel computation, and we explore *timeslices*: sets of logically simultaneous events.

Chapter 15 concludes this paper by discussing the second half of the problem: using this theory of time as a framework for a secure applications.

(A guide to the symbols and terminology we use follows the text of this paper.)

Part I

Computation

The immediate focus of our work is building time models for computation in asynchronous distributed systems. However, before we can build models, we need to specify the things we want to model. Part I handles this task. Chapter 2 presents the formalism we use for our distributed system system: a collection of finite automata communicating with each other, and with the outside world (via I/O devices, also automata). Chapter 3 then defines the *system trace* format we use for the ground-level, exhaustive physical description of computation.

(Part I)

Chapter 2

Systems

A *process* is a sequential, localized computational entity. A process may interact with its local environment through a collection of I/O devices. A *system* is a finite collection of processes and I/O devices. Processes and I/O devices have unique *names.* For a given system, let **PROC-NAMES** be the set of process names, **DEV-NAMES** the set of I/O device names, and **NAMES** be their union. (To keep things simple, we assume these sets are static.)

Processes interact with each other and with the I/O devices by asynchronously passing messages that arrive either once (after an unpredictable delay) or not at all. (Thus, the system does not necessarily preserve message order, and may lose messages.) A message is a triple indicating the sender, the destination, and the message content. Formally, define

$$\textbf{MESSAGES} = \textbf{NAMES} \times \textbf{NAMES} \times \Sigma$$

where Σ is the set of finite binary strings.

2.1. Processes

The Automata Model Internally, a process is a deterministic finite automaton operating in real time. Each process has a finite set of states Q (with initial state $q_0 \in Q$) and a send queue S and a receive queue R from $\textbf{MESSAGES}^*$.[1] (These queues are not necessarily FIFO.) Such triples constitute *process configurations*:

$$\textbf{CONFIGS} = Q \times \textbf{MESSAGES}^* \times \textbf{MESSAGES}^*$$

Transition Functions A process also has a transition function δ that specifies transformations of the process configuration.

$$\delta : \textbf{CONFIGS} \rightarrow \textbf{CONFIGS}$$

[1]The notation W^* denotes the set of strings of items from a set W.

However, not just any function will do. All transition functions must respect the operation of the send and receive queues. For example, the send queue lists the messages sent by this process that have not found their way into the network yet. Transition functions must treat the send queues as write-only.

Exactly how a transition function should treat the receive queue—the list of messages that have arrived at that process but have not yet been "received"—is another matter. Should a process be able to execute only "blocking receives," where a *receive* operation causes the process to read a message off its queue (if the queue is nonempty) or wait indefinitely until a message arrives? Should the arrival of a message interrupt the process, so that a *receive* happens spontaneously on the arrival of a message? Or should a process have a poll operation, where it formally determines if a message is waiting?

A Interrupt/Polling δ As an example, we develop a specification that admits transition functions that can do both explicit polling and spontaneous interrupts.

The Informal Version We want such a δ to allow a process to examine its current state and whether or not the receive queue is empty. This information alone then enables one of three types of transitions:

- *send*: the process changes state and adds a message to the send queue.
- *receive*: the process changes state, removes a message from the receive queue, and reads it.
- *compute*: the process only changes state (without modifying the queues).

The *receive* transition can only be enabled if a message is waiting, and only in a *receive* transition may the process actually examine the value of the of the message at the head of the queue.

The Formal Version Let EMPTY be the predicate indicating that a queue is empty, CAR return the first element of a nonempty queue, CDR return the remainder, and APPEND(s, x) return the queue s with the element x appended.

Axiom 2.1 *(Interrupt/Poll)* There exist functions

$$\begin{array}{rccc}
\text{CLASS}: & Q \times \{\textit{true}, \textit{false}\} & \rightarrow & \{\textit{send}, \textit{receive}, \textit{compute}\} \\
\text{STATE}_{si}: & Q \times \{\textit{true}, \textit{false}\} & \rightarrow & Q \\
\text{STATE}_{r}: & Q \times \textbf{MESSAGES} & \rightarrow & Q \\
\text{MESS}: & Q \times \{\textit{true}, \textit{false}\} & \rightarrow & \textbf{MESSAGES}
\end{array}$$

such that

$$\text{CLASS}(q, x) = \textit{receive} \quad \Longrightarrow \quad x = \textit{true}$$

and

$$\delta(q, S, R) = \begin{cases} (\text{STATE}_{si}(q,\ \text{EMPTY}(R)),\ \ \text{APPEND}(S,\ \ \text{MESS}(q,\ \text{EMPTY}(R)),\ \ R) \\ \qquad \text{if CLASS}(q, \text{EMPTY}(R)) = \textit{send} \\ \\ (\text{STATE}_{r}(q,\ \text{CAR}(R)),\ \ S,\ \ \text{CDR}(R)) \\ \qquad \text{if CLASS}(q, \text{EMPTY}(R)) = \textit{receive} \\ \\ (\text{STATE}_{si}(q,\ \text{EMPTY}(R)),\ \ S,\ \ R) \\ \qquad \text{if CLASS}(q, \text{EMPTY}(R)) = \textit{compute} \end{cases}$$

This paper assumes the processes in the example systems have transition functions that satisfy this axiom.

In Real Time A process operates in real time. Each process receives ticks; at each tick, the process transforms its state according to δ. This paper treats transformations as instantaneous (to insure they are atomic), and assumes a past-closed convention (to keep state well-defined). If a tick occurs at time u, the old configuration persists for $t \leq u$, and the new one exists for $t > u$ (until the next tick).

Since the processes are asynchronous, these ticks occur at indeterminate intervals, independently at each process. However, these intervals must be "reasonable." The following axiom presents one characterization of reasonableness.

Axiom 2.2 *(Discrete Behavior)* In any finite period of time, a process receives only a finite number of ticks.

Philosophy Central to the family of time systems we build in this paper is the assumption that the system is indeed asynchronous and distributed. Processes have *no access to real time*: an outside observer can generate timestamps from God's wristwatch, but individual processes never get to look at this device. Further, processes may perceive the rest of the system only through the messages they receive.[2]

2.2. I/O Devices

Throughout its execution, a process may interact with local parts of the outside world—perhaps a hard disk, a user at the console, or a fermentation vat with sensors and valves. From a process's

[2]Local input and output (through I/O devices) provides an avenue for covert communication that violates this distribution requirement. Such pathology lies beyond the scope of this paper (but subsequent research will examine this issue).

point of view, these I/O devices are black boxes. The process can communicate with them and may have some idea of what they might be doing, but the environments essentially have nondeterministic behavior and unobservable state.

Similar to processes, I/O devices appear in our model as automata, with a set of states Q_{DEV} (containing initial state q_0), a send queue, a receive queue, configurations of the form

$$\textbf{DEV-CONFIGS} \quad = \quad Q_{\text{DEV}} \times \textbf{MESSAGES}^* \times \textbf{MESSAGES}^*$$

and a transition function δ.

An I/O-device automaton differs from a process automaton in two important ways. First, the state set Q_{DEV} may be countably infinite (since the real world can be fairly complex). Second, a given configuration may enable transitions to several new configurations (to allow for the randomizing influence of the outside world). The transition function for I/O-device automata maps configurations to *sets* of configurations:[3]

$$\delta \ : \ \textbf{DEV-CONFIGS} \rightarrow \mathcal{P}(\textbf{DEV-CONFIGS})$$

In a transition from configuration c, the automaton takes on one of the new configurations from $\delta(c)$ nondeterministically.

Each process has a (possibly empty) collection of I/O devices. We make the simplifying assumption that these collections are disjoint. The I/O devices that a process uses are private to that process (e.g., only process p communicates with its I/O devices).

As with process automata, asynchronous, independent ticks (satisfying the Discrete Behavior Axiom) trigger transitions in I/O-device automata.

2.3. Message Transmission

The previous sections presented automata models for process behavior and I/O devices. This section completes the picture by formally describing their interaction: message transmission.

A process automaton (or I/O-device automaton) sends a message by appending it to the send queue, and receives a message by examining the receive queue (according to its δ). However, forces external to the actual automata determine how messages get from one queue to the other. In our model, a message added to a send queue remains there an indeterminate amount of time, after which it spontaneously vanishes into the ether. The message might arrive in the appropriate receive queue after some unpredictable positive delay, or it might remain in the ether forever.

As with configuration transitions, these changes are past-closed and instantaneous: the old state exists for time $t \leq u$, and the new state for time $t > u$.

[3]The notation $\mathcal{P}(W)$ denotes the set of all subsets of a set W.

(Conceivably, we may wish to require more predictable message behavior for I/O messages—such as bounded transmission time—because of their connection to a process is presumably more reliable than the network between processes.)

(Part I)

Chapter 3

Traces

Having described what the system is, we now describe what the system does.

3.1. What Influences an Execution

A system consists of a set of process automata, each with its corresponding I/O-device automata. In a given system, each process has an individual program. In a particular execution, the system starts computation at time $t = 0$ with each process and I/O device in its initial configuration $(q_0, \emptyset, \emptyset)$. Naturally the program at each automata influence how the configurations evolve in this execution. But there are woollier influences: the tick sequences, the transitions of I/O-device automata, and the lifetime and fate of messages.

The behavior of these influences—the delays on messages and ticks, the choices of state and fate—is unpredictable from the point of view of a process, or even of an outside observer with perfect knowledge of all the processes (or even of the entire system). But formalizing this nondeterminism is tricky. For example, what mechanism best models the generation of a process's ticks in a particular execution? A simple random choice—e.g., at time t the process obtains a positive real Δ at random, and moves again at $t + \Delta$—does not suffice. Neither does obtaining an increasing sequence from a set of permissible sequences (according to some specified distribution), nor does any mechanism obtaining one process's sequence independently from the other sequences.

In reality, the universe *calculates* this behavior. The delays and transitions that occur in a particular execution depend on the state of the universe when the execution commences. But since the universe is a fairly intractable beast, in our model things just happen unpredictably. We formally acknowledge this lack of determinism.

Axiom 3.1 In an execution, *any* pattern of behavior (obeying the Finiteness Axiom) may occur.

3.2. Observing Computations

An execution begins when an outside observer sets his stopwatch to 0 and simultaneously resets the processes and their I/O-devices to their initial configurations. The automata rules of Chapter 2, and the particular way the ticks, state choices, and message fates unfold, allow the process and I/O-device configurations to be well-defined for all time $t \geq 0$.

A *system trace* is a discrete representation of what an omniscient observer outside the system can realistically perceive of a computation over a finite period of time. In a particular computation, the observer takes a finite series of photos of the system and jots down the time of each photo on the back. We assume the observer is lucky enough to catch all the action by taking at least one photo immediately after every change to a process configuration: after every process tick, and after every message arriving at a process's receive queue or vanishing from a process's send queue. (Since the I/O-device automata are black boxes, we shield their behavior from the observer.)

We can imagine traces to be tables, with one column for each photo. In each column, the first row contains the time of the photo, and the remaining rows (one for each process) contain the process configurations that the photo captures.

Definition 3.2 Suppose a system has n processes, P_1 through P_n. A *system trace* is a finite tuple $T = ((t_0, s_0), ..., (t_k, s_k))$, where each t_i is a nonnegative integer and each s_i is an n-tuple $(c_{i\,1}, ..., c_{i\,n})$ of process configurations, such that

- $t_0 < t_2 < ... < t_k$
- s_0 consists of the initial configurations.
- There exists some system computation such that each $c_{i\,j}$ is the configuration of process P_j at time t_i
- For this computation, let $u_0 < u_1 < ... < u_m$ be the sequence of time values from the closed interval $[t_0, t_k]$ at which a process ticks, a message arrives at a process receive queue, or a message leaves a process send queue. Then:
  - $t_0 < u_0$
  - $u_m < t_k$
  - For each $j < m$, at least one t_i falls between u_j and u_{j+1}.

Part II

Time Models

A system trace provides the maximum amount of physically observable information about a computation. However, this information contains too much detail and too little structure. Consequently, we develop a *time model* framework for transforming the detailed representations to more abstract representations. Presumably, these abstract representations better express the essential aspects of a computation by abstracting away the irrelevant details.

Part II builds this framework. Chapter 4 develops the definition of time models. Chapter 5 explores some basic properties of time models, and presents some basic operators (which themselves take the form of time models—abstracting abstractions). Chapter 6 develops a particular family of time models (to express the hierarchy of abstractions) from real time ordering of all events to partial ordering of interesting events. Collections of models suggest some natural relationships; Chapter 7 explores these relationships.

(Part II)

Chapter 4

Developing a Definition

Loosely speaking, time is a mechanism for ordering things that happen. Talking about a computation requires enumerating the things that happen and placing some type of order on them. Hence, we introduce a *computation graph* format to describe a computation as a particular set of "ordered"[1] objects. Modeling a computation entails taking its description in this format and constructing a new description (also in this format), whose parts may represent various parts of the old description. A *time model* is thus a representational transformation of computation graphs. For these chains of transformed graphs to talk about the physical reality of computing, they require a foundation: a computation graph that explicitly describes computation, rather than one that is just an image of another graph. We provide this foundation by transforming traces into *ground-level computation graphs.*[2]

Section 4.1 develops this *computation graph* representation. Section 4.2 translates system traces to ground-level graphs. Section 4.3 then presents the notion of a *time model* as a particular way of transforming (and presumably abstracting) sets of computation graphs.

4.1. Computation Graphs

4.1.1. A Definition

Abstractly, a computation is some set of discrete events that happen in some particular order. (In this paper, we assume that this set is always finite.)

[1]Strictly speaking, this temporal relation may not always be an order.

[2]As we observed in Section 1.1, the choice of what constitutes ground-level is somewhat arbitrary. In this paper, ground-level graphs come from traces; other uses of this theory might require other foundations.

We express this abstraction as a *computation graph*: a labeled directed graph representing a computation.[3] The graph consists of directed edges and labeled nodes. Each node is an event—a distinct "thing that happens." The label describes the event. We distinguish between events and event labels in order to allow repeated occurrences of the same type of event.

The edges have two roles: to indicate the temporal relation of events, and to indicate the transition from one event to another. The role an edge plays will be clear from the construction of a graph.

4.1.2. Notation

The *atoms* of a graph are its nodes (with labels) its edges. Where convenient, we will regard a graph as the set of its atoms: $x \in \alpha$ refers to an atom x from graph α. Lower-case Greek letters denote computation graphs. Upper-case Roman letters from the beginning of the alphabet denote specific nodes, and lower-case Roman letters starting with x denote specific atoms. Variations on the notation $\mathcal{G}$ will denote special sets of computation graphs—e.g., the graphs obtained in some particular way, with event labels from some specified set.

4.1.3. Subgraphs

We obtain a *subgraph* of a computation graph in the natural way: by pruning away some nodes and edges.

Definition 4.1 A *subgraph* of a computation graph α is the graph obtained by removing from α:

- a subset of the nodes
- a subset of the edges, including any edge incident to a deleted node

When computation graph α' is a subgraph of computation graph α, we write

$$\alpha' \quad \subset \quad \alpha$$

4.1.4. Identity and Isomorphism

We introduce terminology to describe when two computation graphs completely match:

[3]Labeled graphs are essentially identical to ordered multisets, which surface in the literature (such as Pratt's work on partial order time [Pr86]). However, we feel the former representation is more amenable to computer scientists. Further, using graphs rather than pomsets grants us the liberty to use more general temporal relations.

Definition 4.2 Two computation graphs α_1 and α_2 are *identical*

$$\alpha_1 \quad \equiv \quad \alpha_2$$

when they match: when a bijection exists giving an exact matching of nodes and edges.

Since nodes in computation graphs have labels (by definition), for two nodes to match, they must possess the same label. The standard graph-theoretic notion of isomorphism ignores labels and consequently gives a weaker correspondence:

Definition 4.3 Two computation graphs α_1 and α_2 are *isomorphic*

$$\alpha_1 \quad \cong \quad \alpha_2$$

when they are identical, except for the node labeling. That is, we can relabel the nodes in α_1 to obtain a graph α'_1 satisfying $\alpha'_1 \equiv \alpha_2$.

Both identity and isomorphism depend on the existence of a bijection between two graphs. Having explicit access to this bijection will be useful:

Definition 4.4 A *pairing* between two graphs α_1 and α_2 is simply a subset P of $\alpha_1 \times \alpha_2$. If $\alpha_1 \equiv \alpha_2$ and pairing P enumerates the identification, we write

$$\alpha_1 \quad \equiv_P \quad \alpha_2$$

Similarly, if $\alpha_1 \cong \alpha_2$ and pairing P enumerates the isomorphism, we write

$$\alpha_1 \quad \cong_P \quad \alpha_2$$

The correspondence between two identical or isomorphic computation graphs does not necessarily induce unique pairings: consider two copies of an edgeless graph consisting of two nodes with the same label.

Identity and Isomorphism on Subgraphs Suppose two graphs have identical subgraphs:

$$\alpha_1 \supset \alpha'_1 \equiv_P \alpha'_2 \subset \alpha_2$$

Then the pairing P between the subgraphs extends to be a pairing between graphs. The pairing not only enumerates the identification between the subgraphs but also *specifies* the subgraphs. Figure 4.1 illustrates this relationship.

This technique also applies to isomorphic subgraphs.

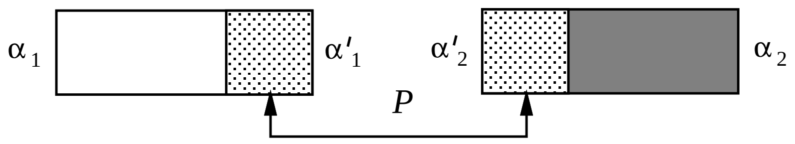

Figure 4.1 The pairing enumerating the identification between two identical subgraphs also specifies the subgraphs. Here, each $\alpha'_i \subset \alpha'_i$, and $\alpha'_1 \equiv_P \alpha_2$. However, P is a pairing not only between the subgraphs α'_i but is also a pairing between the graphs α_i. Given α_1 and α_2 and P, we can figure out that the α_i subgraphs are identical.

4.2. Ground-Level Computation Graphs

Currently we describe real physical computation via traces. Our time models will provide the means for more abstract descriptions. In order to have closure on composition, time models will operate on computation graphs. Thus, in order for time models to apply to real computations, we need to lift traces into this computation graph format.

We want to perform this action with a minimum of abstraction, since abstraction is the duty of models. This lifting is just some sleight of hand so that models can talk about the real world.

4.2.1. Turning Traces into Graphs

Since we will perform abstractions on computation graphs, we need to make sure that the ground-level graph for a trace contains everything of interest in the trace. Consider the trace $T = ((t_0, s_0), ..., (t_k, s_k))$, with $s_i = (c_{i\,1}, c_{i\,2}, ..., c_{i\,n})$. This trace expresses a handful of interesting things about the underlying computation:

- At time t_i, the jth process is in configuration $c_{i\,j}$.
- Either this configuration persists through t_{i+1}, or there exists exactly one time u_i in the open interval (t_i, t_{i+1}) at which this process changes configurations. This change must have one of the following forms:
  - the process undergoes a *send, receive*, or *compute* transition (from Axiom 2.1).
  - a message *depart*s from the send queue of this process
  - a message *arrive*s at the receive queue of this process

  But the exact value of u_i is not known.
- If the trace indicates that two different processes each undergo a configuration change in the interval (t_i, t_{i+1}), the changes occurred at the same instant. (This follows from the definition of trace: otherwise the observer would have taken an intermediate photograph.)

We construct the ground-level computation graph of T by, for each process, creating a node for each of these interesting actions. (Thus, each of these actions becomes an "event.") We then draw edges to represent the basic transitions from action to action at each process. The basic transitions go from photo to photo, if nothing happened, or from photo to configuration change and from configuration change to the next photo.

We choose event labels from the set:

$$\begin{array}{lcll} \textbf{PROC-NAMES} & \times & (& (\{\textit{photo}\} \times \textbf{CONFIGS} \times \textit{non-negative reals}) \\ & & \cup & (\{\textit{send}, \textit{receive}, \textit{depart}, \textit{arrive}\} \times \textbf{MESSAGES}) \\ & & \cup & \{\textit{compute}\} \ ) \end{array}$$

This set just follows the above schema. Each label is a pair containing a process name, and a description of the event: a timestamped photograph or a configuration transition.

4.2.2. Computations and Ground-Level Graphs

Physical computation takes place in space and time. The computation that trace T represents takes place in the space-time region **PROC-NAMES** $\times [t_0, t_k]$ (the cross-product of a discrete set with a closed interval of the reals). The ground-level graph for T has two important properties relating to this region.

- Each atom of the ground-level graph of T naturally represents some part of the underlying region.
  - Event $(p, (\textit{photo}, c, t_i))$ represents the instant (p, t_i) of the photo.
  - A configuration change event $(p, \textit{foo})$ (between the t_i and t_{i+1} photos) represents the instant of transition *foo*: the point (p, u) for the [unknown] instant u in the open interval (t_i, t_{i+1}) when the change occurred.
  - Edges represent the transitions between consecutive events at the same process. We induce the region this edge represents from the regions the events represent: the edge from the (p, t) node to the (p, u) node represents the region $(p, (t, u))$.
- The regions represented by the atoms in the graph of T form a partition of the region represented by T.

Every instant at every process in a computation that trace T describes corresponds to exactly one atom in the ground-level computation graph. Every atom in the ground-level computation graph represents a disjoint set of these instants. Figure 4.2 sketches an example of these properties.

4.2.3. No Abstraction

We reiterate that the ground-level computation graph of T is merely a graph version of the trace T, expanded to include the (inferred) configuration transitions. The graph contains no explicit ordering

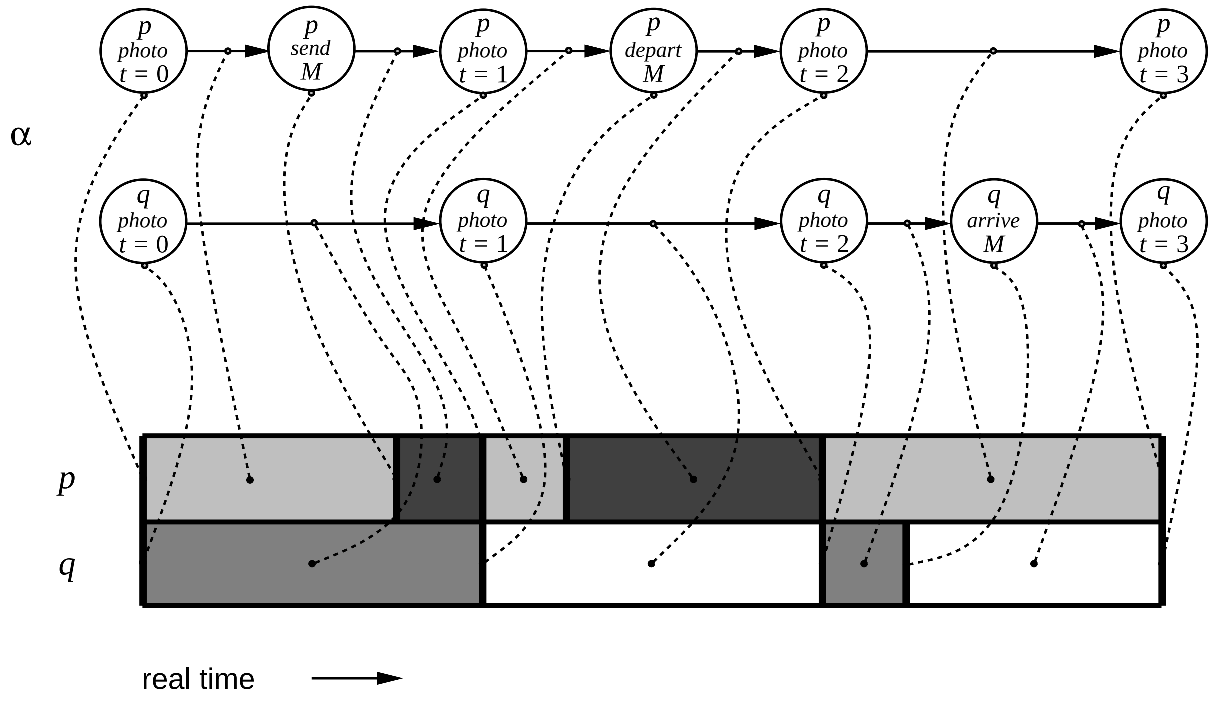

Figure 4.2 A ground-level computation graph represents computational space-time. Each atom in the ground-level graph α represents activity at process p or q at some point or interval of real time.

information that was not already present in the trace. The graph also contains information—such as the times of the photos, and even the existence of the photos—not available to the processes.

The graph version of a trace merely expresses the trace in graph format. Any higher abstraction (such as imposing orders or pruning away uninteresting actions) is the job of a time model.

4.3. Time Models

Formally, a *time model* is a particular way of transforming one set of computation graphs into another set, presumably more abstract. Concomitant with this transformation is a notion of representation: an atom in the transformed graph may represent a set of atoms in the original graph. Time models usually depart from physical reality in order to better express some underlying conceptual structure.

4.3.1. Events and Temporal Relations

Events As we saw in Section 4.1, building computation graphs requires bundling process activity into discrete packages called *events*. We identify events, the basic “things that happen,” with their nodes. Events are atomic in the sense that they provide the fundamental level of granularity in the computation graph: in this graph, one cannot talk about anything finer. The label on an event should describe that event in sufficient detail for the level of abstraction in this graph—for example, if a graph represents what a process perceives about a computation, the labels should make no reference to things that process cannot observe, such as real time.

Temporal Relations In the ground-level computation graphs from Section 4.2, edges represent transitions between events. In more general graphs, the edges will represent a *temporal relation* on events—the “order” in which they happen.

A temporal relation is a binary *precedes* relation on a collection of elements (which, in this paper, will be events). We write $A \longrightarrow B$ to indicate that event A precedes event B in this relation. We also use some variations:

- $A \underline{\longrightarrow} B$ when $A \longrightarrow B$ or $A = B$
- $A \not\longrightarrow B$ when A does not precede B
- $A \longleftrightarrow B$ when $A \longrightarrow B$ and $B \longrightarrow A$
- $A \underline{\longleftrightarrow} B$ when $A \longleftrightarrow B$ or $A = B$
- $A \not\longleftrightarrow B$ when neither $A \longrightarrow B$ nor $B \longrightarrow A$

A relation is *transitive* when (for any events A, B, C) if $A \longrightarrow B$ and $B \longrightarrow C$ then $A \longrightarrow C$. A relation is *antisymmetric* when $A \longrightarrow B$ and $B \longrightarrow A$ cannot both hold for $A \neq B$. A relation is *irreflexive* when no $A \longrightarrow A$. A *partial order* is a relation that transitive, antisymmetric, and irreflexive; a *total order* is a partial order that is *complete*: for any $A \neq B$ either $A \longrightarrow B$ or $B \longrightarrow A$.

We introduce a new term: a *linear time order* is a partial order where concurrency is an equivalence relation whose equivalence classes induce a total order. In a linear time order, we can assign each event A a real number $T(A)$, such that $T(A) < T(B)$ iff $A \longrightarrow B$, for distinct A, B. (A linear time order is just a total order that allows for simultaneous events.)

4.3.2. Representation

From Graph to Graph Events represent discrete units of computation. In the physical system, computation takes place in space and time. Expressing computation as traces imposes a granularity on perception: things happen at processes (the space coordinates), and time values from the trace must delimit the time periods (the time coordinates). The graph version of a trace constructs events and edges by packaging portions of the space-time computation region as single atoms. As Section 4.2.2 observes, this packaging has some convenient properties: each atom represents a disjoint subregion, and together these subregions constitute a partition of the full region.

Constructing a computation graph β to model another computation graph α should proceed in the same fashion. Each atom in β may represent some portion of the computation region that α expresses. (A *ghost* atom is one that represents nothing.) However, this region is no longer space-time, but rather is the graph α. As with traces, the structure of the region forces a granularity on the perceivable subregions: they must be composed of subsets of atoms of α.

We could express this representation in a number of ways (regarding a graph as the set of its atoms):

- as a relation between α and β
- as a partial function from α to β
- as a function from α to $\mathcal{P}(\beta)$
- as a function from β to $\mathcal{P}(\alpha)$

The first approach makes composition of models awkward; the second forces each atom in α to have at most one representative in β (a restriction which we suspect may cause problems eventually); the third makes it difficult to talk about the multiple atoms a single atom might represent.

We conclude that the fourth approach is the cleanest and the most flexible. It most closely follows the principle that each atom in β represents some set of atoms in α. It also allows us to express easily properties such as “each α atom has at most one representative in β” and “β may have no ghost nodes.” Such a function naturally extends to act on sets of atoms: apply the function to the individual elements in the set and take the union of the results.

Terminology Suppose graph β represents graph α. A *representation map* is a function taking each atom of β to a set of atoms of α. (As we will see in the next section, representation maps will accompany model applications.)

Since we’re talking about representation, we’ll adopt a democratic model for terminology. Let x be an atom of β, y an atom of α, and R a representation map from β to α. Then we say:

- x is a *representative* of y (if $y \in R(x)$)
- $R(x)$ is the *constituency* of x
- y is a *constituent* of x (if $y \in R(x)$)

However, we allow for general, Chicago-style democracy:

- Some representatives may have overlapping constituencies.
- Some representatives may have empty constituencies.
- The collection of constituencies might not cover the entire populace.

4.3.3. Models

A Formal Definition We put the elements of Section 4.3.1 and Section 4.3.2 together to produce a formal definition of a time model: a uniform way to build a computation graph whose pieces explicitly represent pieces of another computation graph.

> **Definition 4.5** A *time model* is a partial function $\mathbf{M}$ taking computation graphs to computation graphs, such that (if $\mathbf{M}$ is defined on graph α) the application $\alpha \longmapsto \mathbf{M}(\alpha)$ induces a representation map from $\mathbf{M}(\alpha)$ back to α. We write
>
> $$\langle \mathbf{M},\ \alpha \rangle$$
>
> to indicate this representation map.

Figure 4.3 illustrates the action of a time model and its representation map.

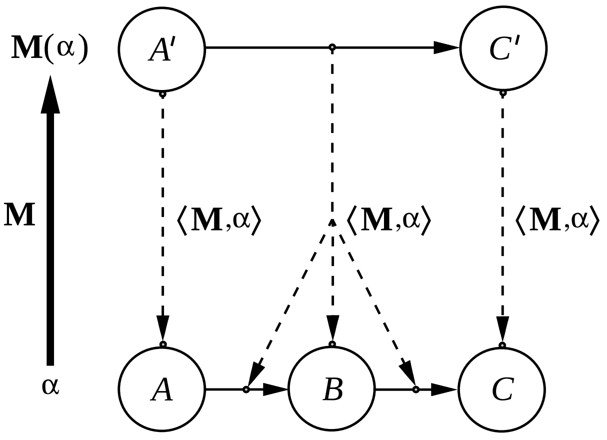

Figure 4.3 Model M transforms computation graph α to computation graph $\mathbf{M}(\alpha)$. The representation map $\langle\, \mathbf{M},\ \alpha\, \rangle$ takes each atom of $\mathbf{M}(\alpha)$ back to the set of atoms in α it represents. The bold arrow indicates the action of $\mathbf{M}$; the dashed arrows indicate the action of $\langle\, \mathbf{M},\ \alpha\, \rangle$.

Conventions Models are partial functions, so the *domain* of a model is the set $\mathcal{D}$ of graphs for which it is defined.

When β is understood to be a particular computation graph that model $\mathbf{M}$ generates from graph α, we write

$$A \longrightarrow B \text{ in } \mathbf{M}$$

to indicate that event A precedes event B in β (and, implicitly, that events A and B appear in β). (In some situations, we will want to emphasize the *model*, not the particular graph names. This shorthand makes such emphasis possible.)

A time model $\mathbf{M}$ naturally induces transformations of graph sets that its domain contains: $\mathbf{M}(\mathcal{G})$ is the set consisting of the transformed graphs $\{\mathbf{M}(\alpha) : \alpha \in \mathcal{G}\}$.

Composition and Inversion Simple manipulations of functions apply to models too—the only trick is handling the representation maps. For example, composing models yields a model.

Definition 4.6 Suppose model $\mathbf{M}_1$ (with domain $\mathcal{D}_1$) and model $\mathbf{M}_2$ (with domain $\mathcal{D}_2$) satisfy

$$\mathbf{M}_1(\mathcal{D}_1) \quad \subset \quad \mathcal{D}_2$$

Then their composition $\mathbf{M}_2 \circ \mathbf{M}_1$ is the model on domain $\mathcal{D}_1$ taking α to $\mathbf{M}_2(\mathbf{M}_1(\alpha))$, with

$$\langle\, \mathbf{M}_2 \circ \mathbf{M}_1,\ \alpha\, \rangle \quad = \quad \langle\, \mathbf{M}_1,\ \alpha\, \rangle \quad \circ \quad \langle\, \mathbf{M}_2,\ \mathbf{M}_1(\alpha)\, \rangle$$

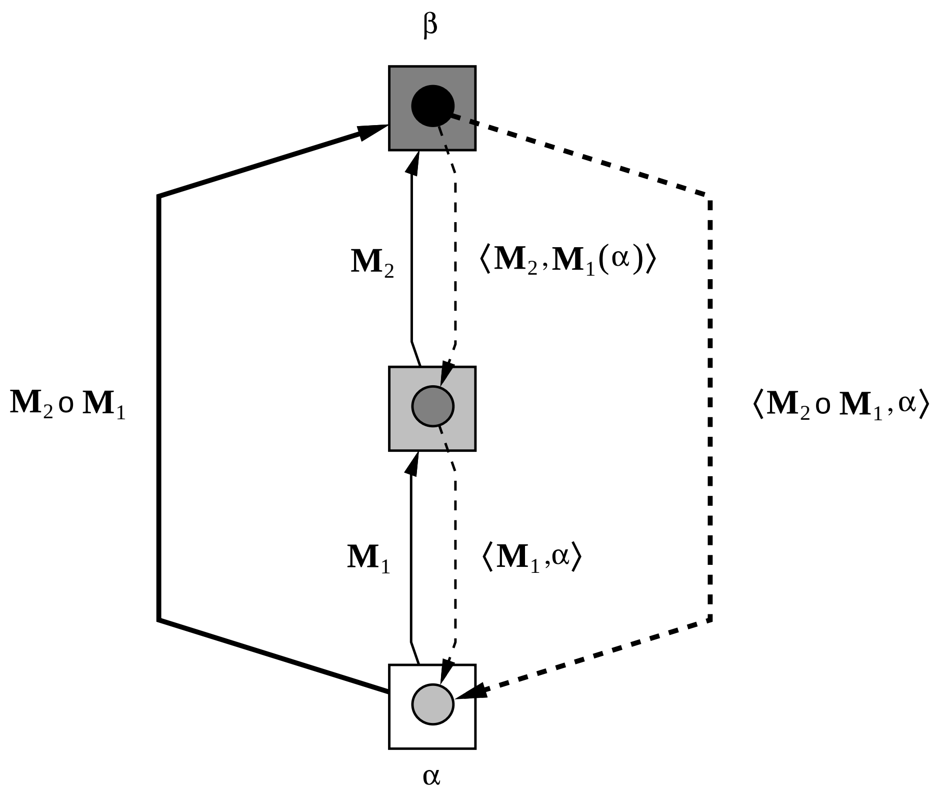
β
$\mathbf{M}_2$
$\langle \mathbf{M}_2, \mathbf{M}_1(\alpha) \rangle$
$\mathbf{M}_2 \circ \mathbf{M}_1$
$\langle \mathbf{M}_2 \circ \mathbf{M}_1, \alpha \rangle$
$\mathbf{M}_1$
$\langle \mathbf{M}_1, \alpha \rangle$
α

Figure 4.4 To obtain $\beta = (\mathbf{M}_2 \circ \mathbf{M}_1)(\alpha)$, we transform α according to $\mathbf{M}_1$, and then transform the result according to $\mathbf{M}_2$. To figure out what an atom of β represents in α, we obtain the set of atoms it represents in $\mathbf{M}_1(\alpha)$, and then figure out what each of these represents in α. Solid arrows indicate the action of $\mathbf{M}_1$ and $\mathbf{M}_2$; the bold solid arrow indicates the action of $\mathbf{M}_2 \circ \mathbf{M}_1$. Dashed arrows indicate the action of the representation maps $\langle \mathbf{M}_1, \alpha \rangle$ and $\langle \mathbf{M}_2, \mathbf{M}_1(\alpha) \rangle$; the bold dashed arrow indicates the action of the representation map $\langle \mathbf{M}_2 \circ \mathbf{M}_1, \alpha \rangle$.

Figure 4.4 illustrates composition of models.

We can also talk about the inverse image of graphs (relative to a given model and class). If α is a graph from $\mathbf{M}(\mathcal{G})$ where $\mathcal{G}$ is understood, then $\mathbf{M}^{-1}(\alpha)$ is the set

$$\{\beta \in \mathcal{G} \ : \ \mathbf{M}(\beta) = \alpha\}$$

A Simple Example The computation that a trace T expresses has a natural synchronized structure. But the ground-level computation graph of T not only fails to express this structure—it also includes items from the trace (the photographs) and items induced from the trace (*arrive* and *depart* nodes) that one ordinarily would not regard as genuine events in the computation. We now introduce a simple time model that abstracts ground-level graphs to graphs that more cleanly represent the computational activity.

Definition 4.7 The model LINEAR takes ground-level graph α to the graph β built as follows. Let α be the graph of trace $T = ((t_0, s_0), ..., (t_k, s_k))$.

Nodes For each process p:

- Create a node $\perp$ in β for the node $(p, \textit{photo}, t_0)$ in α.
- Create a node $\top$ in β for the node $(p, \textit{photo}, t_k)$ in α.
- Node $(p, \textit{photo}, t_i)$ leads to node $(p, \textit{photo}, t_{i+1})$ in α, possibly through an intermediate node A_i. Examine this transition:
  - If the intermediate node A_i exists and is a *send, receive* or *compute*, then create a copy of this node (minus the p name) in β.
  - Otherwise, nothing interesting happened, so create a node *idle* in β.

Edges Thus, there exists a node in β for time t_0 at each process, for time t_k at each process, and for the transition from t_i to t_{i+1} at each process. Draw an edge from node A to node B in β—not necessarily from the same process—whenever any of the following hold:

- A represents the transition from t_i to t_{i+1}, and B represents the transition from t_{i+1} to t_{i+2}
- A represents t_0 and B represents the transition from t_0 to t_1.
- A represents the transition from t_{k-1} to t_k, and B represents t_k
- A represents t_0 and B represents t_1, in the degenerate case when $k = 1$.

The representation map formalizes this natural representation. $\langle$ LINEAR, $\alpha \rangle$ takes each non-*idle* node in β to the corresponding node in α, and the *idle* nodes in β to the atoms lying between the corresponding pair of *photo* nodes. The edges in β between sequential nodes at the same process represent the internal atoms in the paths between the nodes they represent; the cross-process edges are ghosts.

Figure 4.5 shows the application of LINEAR to a simple trace graph.

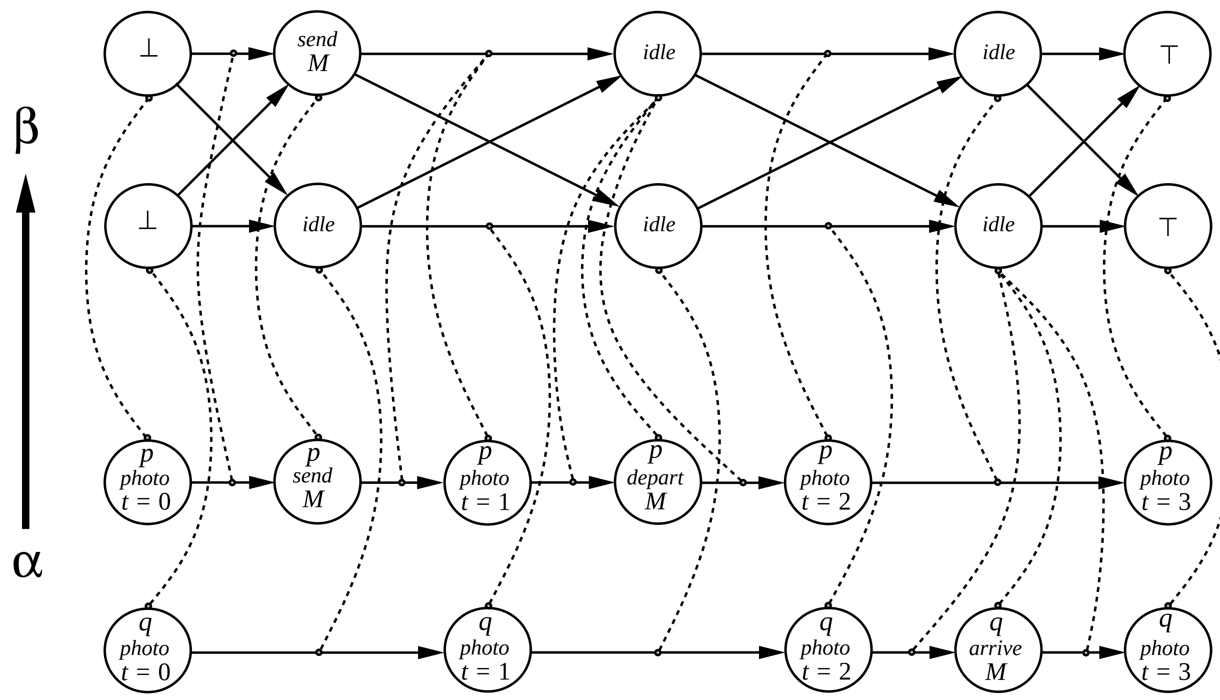

Figure 4.5 We obtain β by applying LINEAR to the simple ground-level computation graph α. Dashed lines connect each atom in β to the atoms it maps to under $\langle \text{LINEAR}, \alpha \rangle$.

The LINEAR model derives its name from the fact that it expresses the basic steps in the natural linear time order on computations. Fully expressing the linear time order requires one more tool (which Section 5.1 will provide).

(Part II)

Chapter 5

Properties and Operators

This chapter presents machinery to talk about some properties of computation graphs and time models. We develop this machinery both by considering the actual properties—of graphs, of sets of graphs, and of models that produce such graphs—and also by considering *operators* on graphs that ensure that some given property holds. (Conveniently, such operators take a familiar form: time models.) Section 5.1, Section 5.2 and Section 5.3 consider some special issues arising from relations. Section 5.4 and Section 5.5 consider the generation and representation issues arising from model applications. Finally, Section 5.6 considers the issues involved in merging computation graphs and merging the models that produce them.

5.1. Transitivity

The computation graphs that we've seen so far (ground-level graphs and their LINEAR images) express events and transitions between events. However, usually we think of temporal relations as being *transitive*: if event A happens before event B, and event B happens before event C, then event A happens before event C.

Defining Transitive Closure Hence, we say that a computation graph α is *transitive* if its relation is transitive: an edge exists from A to B whenever B is reachable from A. We obtain the *transitive closure* $\overline{\alpha}$ of a graph α by adding an edge from A to B whenever a path but no edge exists between them.

A model is *transitive* when it produces only transitive computation graphs. Taking the transitive closure of a model seems a natural operation, but the representational aspect of models makes this operation somewhat non-trivial. Suppose model $\mathbf{M}$ acts on graph α. Clearly we want the transitive closure of $\mathbf{M}$ to produce a transitive version of $\mathbf{M}(\alpha)$. Unless $\mathbf{M}(\alpha)$ already is transitive, we will need to add edges. Which edges should we add? What should these edges represent?

In this paper, we choose the simplest approach:[1] simply take the transitive closure of each graph that $\mathbf{M}$ produces, and let any new edges be ghosts. Here we begin to see some of the expressiveness of time models: the transitive closure operator is itself a time model that copies a graph and adds edges. We call this model TRANS, and use the shorthand:

$$\overline{\mathbf{M}} \quad = \quad \text{TRANS} \circ \mathbf{M}$$

Using Transitive Closure As we have mentioned, usually we think of temporal relations as transitive. However, these relations usually arise by first considering some "basic" transitions on events. Having an explicit transitive closure operator allows us to follow this technique when building models.

For example, the transitive closure $\overline{\text{LINEAR}}$ expresses the full linear time ordering of process actions induced by real time.

Asking about precedence in graph $\overline{\mathbf{M}}(\alpha)$ is equivalent to asking about paths in $\mathbf{M}(\alpha)$. (Having the flexibility to talk about both the "full" version and the "single-step" version of a time model will be useful in subsequent papers when we consider knowability issues.)

5.2. Bounds

Is there a well-defined "earliest" or "latest" event in a computation? In this section, we define what this means and present an operator to force models that produce extremal events to produce *unique* extremal events.

The Property An event is *minimal* in a graph if no event precedes it. Similarly, an event is *maximal* if no event succeeds it.

A computation graph α is *bounded* when it contains a *unique* minimum that precedes all other events in $\overline{\alpha}$, and a *unique* maximum that follows all other events in $\overline{\alpha}$. When a graph is bounded, the unique extrema are its *bounding* nodes.

[1]Another approach would be to add an edge for each nontrivial path from event A to event B. The new edge would represent the internal atoms in this path. This alternative approach allows us to reach through some precedence assertion to the individual steps that cause it to hold. This ability might be useful: for example, it makes it easier to state one of our preliminary security results [Sm91]: an honest process (using a certain clock implementation) will always detect the presence of an edge, if the events that edge represents occur only at honest processes.

Should we eventually be interested in the more complicated form of closure, we could insert that between $\overline{\mathbf{M}}$ and $\mathbf{M}$ by first defining TRANS_1 (which adds representative edges for each nontrivial path) and then TRANS_2 (which replaces all edges from A to B by a single representative ghost edge). Then we would re-define TRANS as the composition $\text{TRANS}_2 \circ \text{TRANS}_1$.

A model $\mathbf{M}$ is bounded when it produces only bounded graphs.

A graph α is *transitively bounded* when $\overline{\alpha}$ is bounded; a model $\mathbf{M}$ is *transitively bounded* when $\overline{\mathbf{M}}$ is bounded.

An Operator Suppose model $\mathbf{M}$ produces graphs whose transitive closure contains minima and maxima. One way to insure that $\mathbf{M}$ is transitively bounded is to collapse the extrema into single events.

Definition 5.1 The model EXTREMA takes a graph α to the graph β as follows:

Nodes Partition the nodes of the transitive closure $\overline{\alpha}$ into three sets: $S_\perp$ containing the minima, $S_\top$ containing the maxima, and S_{other} the remaining nodes. The nodes of β consist of one copy of each node in S_{other}, plus a new node labeled $\perp$ if $S_\perp$ is nonempty, plus a new node labeled $\top$ if $S_\top$ is nonempty.

Edges The node construction induces a natural surjection F from nodes in α to nodes in β. Use this surjection to draw edges: if an edge exists from A to B in α, then draw one from $F(A)$ to $F(B)$ in β. (Thus F extends to a surjection F' from atoms to atoms.)

The representation map $\langle\, \text{EXTREMA},\ \alpha \,\rangle$ is the inverse of the surjection F'.

Applying EXTREMA to a model does not necessarily yield a transitively bounded model. For an easy counterexample, suppose model $\mathbf{M}$ produces graphs that are simple cycles. Since $\mathbf{M}$ produces no minima or maxima, we have

$$\text{EXTREMA} \circ \mathbf{M} \quad = \quad \mathbf{M}$$

and thus $\overline{\mathbf{M}}$ is not bounded.

5.3. Cycles

Given a directed graph, a natural question is to ask whether it contains any cycles. This question applies to our work, since time models produce computational graphs.

Hence, we say that a node A in graph α is *acyclic* when no cycle in α contains it. We say that graph α is *acyclic* when it contains no cycles. Finally, we say that a model $\mathbf{M}$ is *acyclic* when it produces only acyclic graphs.

Conversely, a node A is *cyclic* if it is contained in a cycle; a graph α is *cyclic* if it contains a cycle.

5.4. Generators

On a basic level, we might regard possible system behavior—what the system does in a given set of circumstances—as a set of possible system traces. Our time theory allow us to regard possible system behavior instead as a set of possible computation graphs. However, this set of graphs cannot stand alone as a descriptive entity; we need to specify how this particular set originates in the ground-level graphs.

This specification consists of two things: a model $\mathbf{M}$ and a graph set $\mathcal{G}_2$ such that $\mathcal{G}_1 = \mathbf{M}(\mathcal{G}_2)$. We say that such a model is a *generator* of set $\mathcal{G}_1$. If $\mathcal{G}_2$ consists of ground-level computation graphs, then $\mathbf{M}$ is a *grounding* generator of $\mathcal{G}_1$: each graph in $\mathcal{G}_1$ is grounded in physical reality. If $\mathbf{M}$ produces no ghost events in $\mathcal{G}_1$, then it is a *concrete generator* of $\mathcal{G}_1$: each event in a $\mathcal{G}_1$ graph has concrete meaning in its $\mathcal{G}_2$ pre-image.

Some interesting scenarios will develop when a set of models induces multiple grounding generators for a single graph set. For example, the graph β in Figure 1.7 might arise from failure-free execution or from a faulty execution simulating (through rollback) a failure-free execution. Subsequent papers will present a more thorough exploration of this topic.

5.5. Disjoint and Complete Models

Suppose computation graph α lies in the domain of model $\mathbf{M}$. The new graph $\mathbf{M}(\alpha)$ represents the original graph α. How expressive is this representation? Two issues arise immediately.

- Do the atoms in $\mathbf{M}(\alpha)$ have unique meanings?
- In the $\mathbf{M}(\alpha)$ graph, can we still talk about every atom in α?

We introduce two terms to handle these issues. Model $\mathbf{M}$ is *disjoint* on α if the constituencies of the atoms in $\mathbf{M}(\alpha)$ are disjoint (that is, no atom of α has multiple representatives in $\mathbf{M}(\alpha)$). Model $\mathbf{M}$ is *complete* on α if the constituencies of $\mathbf{M}(\alpha)$ completely cover α (that is, each atom of α has at least one representative in $\mathbf{M}(\alpha)$). If $\mathbf{M}$ is complete and disjoint on α and $\mathbf{M}(\alpha)$ is free of ghosts, then $\langle\, \mathbf{M},\ \alpha \,\rangle$ partitions the atoms of α.

The model $\mathbf{M}$ is itself disjoint when it is disjoint on every graph in its domain; similarly, the model $\mathbf{M}$ is complete when it is complete on every graph in its domain.

Suppose graph α lies in the domain of model $\mathbf{M}$. If $\mathbf{M}$ is complete, every atom in α is represented in $\mathbf{M}(\alpha)$. We can use $\mathbf{M}(\alpha)$ to talk about every atom of α (although we may not be able to distinguish some atoms). If $\mathbf{M}$ is disjoint, every atom from α that is represented in $\mathbf{M}(\alpha)$ is represented uniquely. We may not be able to talk about every part of the original graph α, but we can distinguish everything we can talk about.

Every model we consider in this paper will be disjoint.

5.6. Merging Graphs and Models

Suppose computation graph α lies in the domain of two models $\mathbf{M}_1$ and $\mathbf{M}_2$. We have two different abstractions of α: the graphs $\beta_1 = \mathbf{M}_1(\alpha)$ and $\beta_2 = \mathbf{M}_2(\alpha)$. (Perhaps each β_i isolates and abstracts some particular aspect of α).

How can we merge β_1 and β_2 to obtain a single, more complete abstraction of α? How can we merge $\mathbf{M}_1$ and $\mathbf{M}_2$ into a model that always produces this more complete abstraction?

5.6.1. Merging Graphs

Suppose we are given two computation graphs α_1 and α_2, and we we want to construct a graph that retains all the information in both. The semantics of computation graphs make this task tricky: a graph may have multiple nodes with the same label. Suppose α_1 and α_2 each have a node labeled A. Should we merge these nodes or keep them separate? What if α_1 and α_2 instead have identical subgraphs α'_i a bit more complicated than the singleton A? If α_1 and α_2 have multiple pairs of identical subgraphs, which pair should we merge?

To rectify this confusion, we need to explicitly the pairs of atoms we will identify (that is, the pairs of atoms that will take on the same identity in the merged graph). Section 4.1.4 gives us the necessary tools.

Definition 5.2 Suppose computation graph α_1 has subgraph α'_1, computation graph α_2 has α'_2, and $\alpha'_1 \equiv_P \alpha'_2$ We obtain the *union with respect to* P

$$\alpha_1 \quad \cup_P \quad \alpha_2$$

by joining the two graphs α_i and merging the two atoms in each pair in P.

Of course a quick and dirty solution to the problem of merging graphs is to take the disjoint union: deliberately keep all nodes and edges separate, and obtain a disconnected graph with two components α_1 and α_2. This is just taking the union with respect to the empty pairing.

Figure 5.1 illustrates the two forms of graph union.

5.6.2. Merging Models

Suppose $\mathbf{M}_1$ and $\mathbf{M}_2$ share domain $\mathcal{D}$. Merging these models by merging the graphs they produce requires specifying which atoms in these graphs will be identified. Hence, for each $\alpha \in \mathcal{D}$, we need to exhibit a pairing P between the $\mathbf{M}_i(\alpha)$. However, not just any pairing will do, since the atoms in a transformed graph represent atoms in the original graph. The pairing must respect this representation.

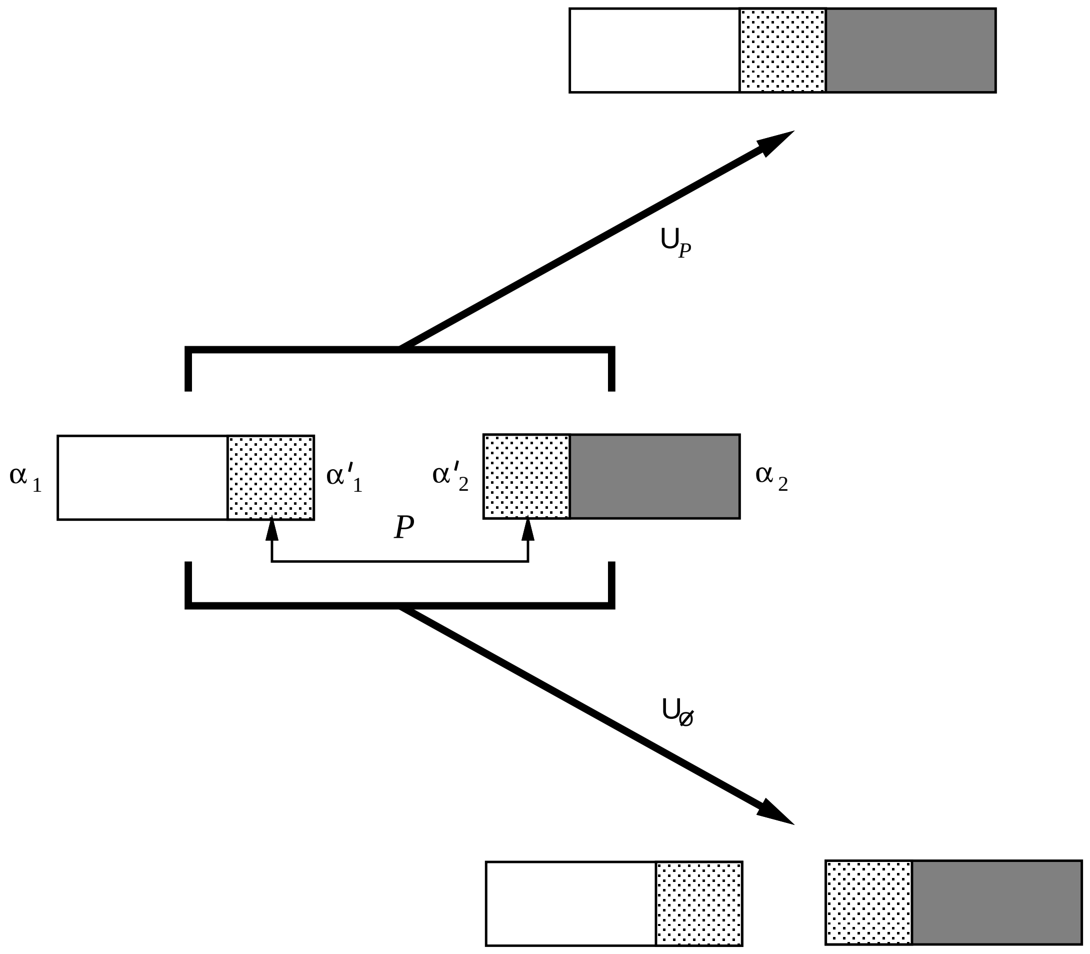

Figure 5.1 Suppose two computation graphs α_1 and α_2 have identical subgraphs (α'_1 and α'_2, respectively), matched by pairing P (left). We obtain the union $\alpha_1 \cup_P \alpha_2$ by merging the α'_i according to the pairing P (top right); we obtain the disjoint union $\alpha_1 \cup_\emptyset \alpha_2$ by keeping both graphs separate and disconnected (bottom right).

The possible presence of ghosts makes constructing this list slightly nontrivial. Should two ghost nodes with the same label be considered the same? What about two ghost edges?

In this paper, we take the most straightforward approach to this dilemma—we keep ghost nodes distinct but we merge ghost edges that obviously coincide.

Let $\mathbf{M}_1$ and $\mathbf{M}_2$ share domain $\mathcal{D}$, and let α be a graph from $\mathcal{D}$. Let α_i be the image $\mathbf{M}_i(\alpha)$. Suppose node A_1 in α_1 and node A_2 in α_2 have the same label and represent the same (nonempty) part of α:

$$\langle\, \mathbf{M}_1,\ \alpha \,\rangle(A_1) \;=\; \langle\, \mathbf{M}_2,\ \alpha \,\rangle(A_2) \;\neq\; \emptyset$$

Then clearly we should regard A_1 and A_2 as the same node in the merged graph.

For preserving edges, we drop the prohibition against ghosts, but add another rule: the endpoints must be common. If edge E_i connects node A_i to node B_i in graph α_i, node A_1 is identified with node A_2, node B_1 is identified with node B_2, and $\langle\, \mathbf{M}_1,\ \alpha \,\rangle(E_1) = \langle\, \mathbf{M}_2,\ \alpha \,\rangle(E_2)$ then we identify these edges.

Definition 5.3 For models $\mathbf{M}_1$ and $\mathbf{M}_2$ and graph α in the domain of both, let COMM$(\mathbf{M}_1, \mathbf{M}_2, \alpha)$ denote the pairing between $\mathbf{M}_1(\alpha)$ and $\mathbf{M}_2(\alpha)$ constructed as above.

That is, COMM$(\mathbf{M}_1, \mathbf{M}_2, \alpha)$ is a list of pairs of nodes and pairs of edges. A pair of nodes (A_1, A_2) is in the list iff the A_i have the same label and the same non-empty constituency; a pair of edges (E_1, E_2) is in the list iff (A_1, A_2) and (B_1, B_2) are in the list (where E_i connects A_i to B_i), and the E_i constituencies are equal.

The COMM pairing behaves as desired:

Proposition 5.4 Let α lie in the domain of models $\mathbf{M}_1$ and $\mathbf{M}_2$. Let P be the pairing COMM$(\mathbf{M}_1, \mathbf{M}_2, \alpha)$ and let $\beta_i = \mathbf{M}_i(\alpha)$. Then

1. The atoms of β_i occurring in the pairing P form a subgraph, β'_i
2. $\beta'_1 \equiv_P \beta'_2$

Proof These results follow directly from Definition 5.3, Definition 4.4 and Definition 4.1. □

We can now extend union to models:

Definition 5.5 The *union* of model $\mathbf{M}_1$ and $\mathbf{M}_2$ is the model $\mathbf{M}_1 \cup \mathbf{M}_2$ on the intersection of their domains, with

$$(\mathbf{M}_1 \cup \mathbf{M}_2)(\alpha) \;=\; \mathbf{M}_1(\alpha) \cup_{\text{COMM}(\mathbf{M}_1, \mathbf{M}_2, \alpha)} \mathbf{M}_2(\alpha)$$

Since the representation maps $\langle\, \mathbf{M}_1,\ \alpha \,\rangle$ and $\langle\, \mathbf{M}_2,\ \alpha \,\rangle$ agree on pairs of atoms from $\mathbf{M}_1(\alpha)$ and $\mathbf{M}_2(\alpha)$ that get identified, define the representation map $\mathbf{M}_1 \cup \mathbf{M}_2$ as follows:

$$\langle\, \mathbf{M}_1 \cup \mathbf{M}_2,\ \alpha \,\rangle \quad = \quad \begin{cases} \langle\, \mathbf{M}_1,\ \alpha \,\rangle & \text{on atoms from } \mathbf{M}_1(\alpha) \\ \langle\, \mathbf{M}_2,\ \alpha \,\rangle & \text{on atoms from } \mathbf{M}_2(\alpha) \end{cases}$$

Of course, the quick and dirty approach to merging models works as well:

Definition 5.6 The *disjoint union* of model $\mathbf{M}_1$ and $\mathbf{M}_2$ is the model $\mathbf{M}_1 \cup_{\emptyset} \mathbf{M}_2$ on the intersection of their domains. $\mathbf{M}_1 \cup_{\emptyset} \mathbf{M}_2$ takes takes α to $\mathbf{M}_1(\alpha) \cup_{\emptyset} \mathbf{M}_2(\alpha)$ with the representation map

$$\langle\, \mathbf{M}_1 \cup_{\emptyset} \mathbf{M}_2,\ \alpha \,\rangle \quad = \quad \begin{cases} \langle\, \mathbf{M}_1,\ \alpha \,\rangle & \text{on atoms from } \mathbf{M}_1(\alpha) \\ \langle\, \mathbf{M}_2,\ \alpha \,\rangle & \text{on atoms from } \mathbf{M}_2(\alpha) \end{cases}$$

We extend these operations to act on finite sets of models in the natural way.

$$\cup\{\mathbf{M}_1, ..., \mathbf{M}_k\} \quad = \quad \mathbf{M}_1 \cup \mathbf{M}_2 \cup ... \cup \mathbf{M}_k$$

This operation is well-defined:

Proposition 5.7 The above two unions on models are associative. For models $\mathbf{M}_1, \mathbf{M}_2, \mathbf{M}_3$:

1. $(\mathbf{M}_1 \cup \mathbf{M}_2) \cup \mathbf{M}_3 = \mathbf{M}_1 \cup (\mathbf{M}_2 \cup \mathbf{M}_3)$
2. $(\mathbf{M}_1 \cup_{\emptyset} \mathbf{M}_2) \cup_{\emptyset} \mathbf{M}_3 = \mathbf{M}_1 \cup_{\emptyset} (\mathbf{M}_2 \cup_{\emptyset} \mathbf{M}_3)$

Proof The disjoint union case is trivial. For the other case, observe that nodes don't go away. Let A_i be from $\mathbf{M}_i$; if you merge A_1 and A_2 in $(\mathbf{M}_1 \cup \mathbf{M}_2)$, then A_2 will still be around in $(\mathbf{M}_2 \cup \mathbf{M}_3)$ to merge with A_1. A similar argument works for edges. $\square$

(Part II)

Chapter 6

Developing a Family of Models

Section 5.1 developed the LINEAR model to express the linear time ordering that real time induces. Yet the crux of the discussion in Chapter 1 is that real time sequences are not sufficient—so this chapter uses the model tools from Chapter 5 to formally develop an alternative model: partial order time.

Section 6.1 develops a collection of timelines: models imposing a linear structure on events at a single process. Section 6.2 presents two models relating events at different processes. Section 6.3 uses these components and the tools from Chapter 5 to assemble the *partial order time* model POT.

Figure 6.1 shows the compositional development of this family of models.

6.1. Within Processes

A ground-level computation graph gives a linear sequence of events for each process: a start point, a sequence of process actions, and a stop point.

We've already seen LINEAR perform this abstraction:

> **Definition 6.1** For each process $p \in$ **PROC-NAMES**, define LINLINE$_p$ to be the model that takes a ground-level graph α and returns the LINEAR(α) subgraph corresponding to process p.

However, these timelines still contain elements that we would not normally consider part of the computation: the *idle* events. We introduce a model to abstract them away:

> **Definition 6.2** Define the model NONIDLE to remove the *idle* events from graphs.
>
> - NONIDLE applies only to graphs α whose *idle* nodes have in-degree one and out-degree one. (For example, LINLINE graphs meet this criteria.)
> - Such α have well-defined maximal *idle* chains. NONIDLE copies the entire graph, but replaces each maximal chain with a single edge.

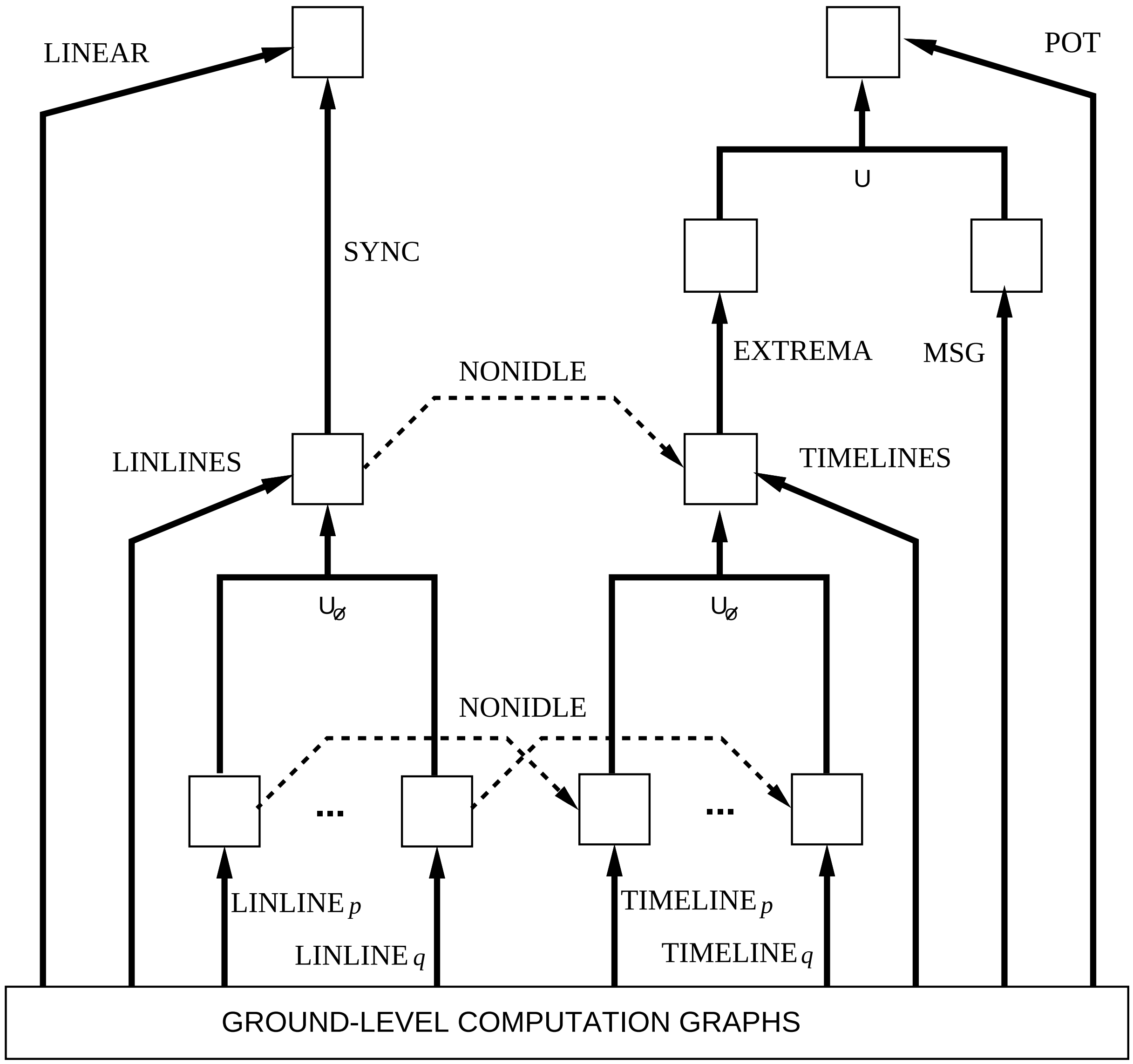

Figure 6.1 The models we discuss here fit into a composition hierarchy. The boxes indicate sets of computation graphs; an arrow **M** between two boxes indicates that model **M** is a surjection from the one set onto the other. What's more, the functional identities we illustrate here are also model identities—e.g., the model LINEAR equals the composition of models SYNC ∘ LINLINES.

- The graph NONIDLE(α) consists thus of atoms from α and new edges. The atoms from α represent themselves; the new edges represent the chain they replaced.

Figure 6.2 illustrates the action of the NONIDLE model.

We can now define the linear timeline of interesting events.:

Definition 6.3 For process $p \in$ **PROC-NAMES**, let

$$\text{TIMELINE}_p \quad = \quad \text{NONIDLE} \circ \text{LINLINE}_p$$

We frequently want to consider the set of timelines as a whole, so we set up some shorthand:

Definition 6.4 Define the models LINLINES and TIMELINES:

$$\begin{aligned} \text{LINLINES} &= \cup_{\emptyset}\{\text{LINLINE}_p \ : \ p \in \textbf{PROC-NAMES}\} \\ \text{TIMELINES} &= \cup_{\emptyset}\{\text{TIMELINE}_p \ : \ p \in \textbf{PROC-NAMES}\} \end{aligned}$$

6.2. Across Processes

Messages We define a model that captures a cross-process order induced by message passing:

Definition 6.5 The model MSG on ground-level computation graph α retains only *send* and *receive* nodes, and draws a ghost edge from A to B only when B is the receipt of the message sent at A.

Edges are ghosts in MSG because all we want to know is if a message got through or not. If we were interested in exploring fault tolerance in message transmission, then perhaps we would want to expand what an edge represents.

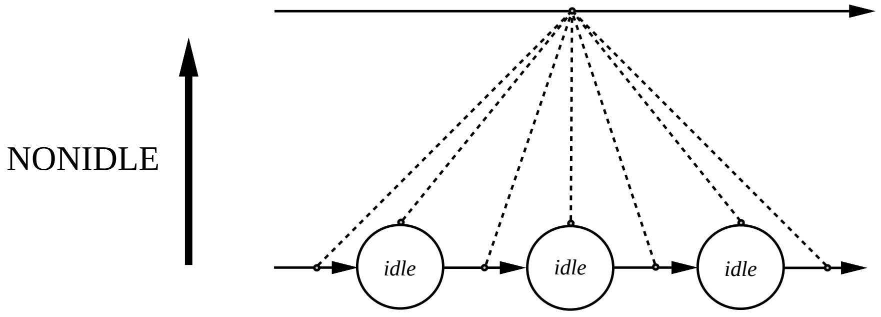

Figure 6.2 The model NONIDLE replaces maximal chains of *idle* nodes by a representative edge.

Linear Synchronization As an aside, we can define a model SYNC that links up equal length straight-line graphs by grouping each "column" of events into an equivalence class.

> **Definition 6.6** Let the model SYNC act on a collection of equal length timelines, one per process, by drawing a ghost edge from the m node at process P_i to the $m+1$ node at each process P_j ($j \neq i$).

Whether SYNC actually performs meaningful synchronization depends on the graphs it acts on—whether the equivalence classes can be meaningfully regarded as synchronized units.

For example, SYNC allows us to give a bottom-up definition of LINEAR:

$$\text{LINEAR} \quad = \text{SYNC} \circ \text{LINLINES}$$

6.3. Partial Order Time

The model LINEAR induces the linear time order $\overline{\text{LINEAR}}$. This only makes sense, as the trace ordering follows real time. However, our building blocks allows us to define an alternative:

> **Definition 6.7** Define the partial order time model POT:
>
> $$\text{POT} \quad = \quad \text{MSG} \cup (\text{EXTREMA} \circ \text{TIMELINES})$$

Essentially capturing Lamport's causal dependency partial order, the POT model is the primary focus of the remainder of this paper.

(Part II)

Chapter 7

Relationships Between Models

The handful of models presented so far suggest some natural ways we can consider one model to be "part" of another. For example:

- POT(α) is always a subgraph of $(\text{EXTREMA} \circ \text{LINEAR})(\alpha)$.
- If two graphs α_1 and α_2 give the same POT image, then they give the same TIMELINE$_p$ image.
- Indeed, given any graph generated by POT, we can uniquely identify the component that TIMELINE$_p$ generates.
- In a rough sense, POT almost appears to be a model on its TIMELINES components.

This chapter presents formal machinery to describe these relationships. Section 7.1 describes forms of *containment* (the first bullet); Section 7.2 presents *refinement* (the second bullet); and Section 7.3 presents *components* (the third bullet). Finally, Section 7.4 describes how a set of components may comprise a *decomposition* of a model, and how we can factor this decomposition out of the model (the fourth bullet).

7.1. Containment

We want to describe the relationship when the action of one model always includes the action of another. Section 7.1.1 develops the *containment* relation; Section 7.1.2 introduces a related tool, the *containment map*, and Section 7.1.3 sketches some uses of containment.

7.1.1. The Containment Relation

Suppose two models $\mathbf{M}_1$ and $\mathbf{M}_2$ share[1] the same domain $\mathcal{D}$. A minimum requirement for $\mathbf{M}_1$ to be *contained* in $\mathbf{M}_2$ is that for any $\alpha \in \mathcal{D}$, $\mathbf{M}_1(\alpha)$ is isomorphic.[2] However, once again the representational aspect of models complicates things. The atoms in $\mathbf{M}_1(\alpha)$ and $\mathbf{M}_2(\alpha)$ carry additional meaning: their constituencies in α. For $\mathbf{M}_1$ to be contained in $\mathbf{M}_2$, we also require that corresponding constituencies also satisfy a containment relation.

To avoid some pathological situations, we will require uniqueness of pairing.

Definition 7.1 Suppose models $\mathbf{M}_1$ and $\mathbf{M}_2$ act on the same domain $\mathcal{D}$. Model $\mathbf{M}_1$ is *contained* in $\mathbf{M}_2$, written

$$\mathbf{M}_1 \quad \overset{\sim}{\subset} \quad \mathbf{M}_2$$

when for each $\alpha \in \mathcal{D}$, there exists a *unique* pairing P between $\mathbf{M}_1(\alpha)$ and $\mathbf{M}_2(\alpha)$ satisfying these two conditions:

1. **Isomorphism** There exists $\beta \subset \mathbf{M}_2(\alpha)$ such that

$$\mathbf{M}_1(\alpha) \quad \cong_P \quad \beta$$

2. **Constituency Containment** If $(x_1, x_2) \in P$ then

$$\langle\, \mathbf{M}_1,\ \alpha \,\rangle(x_1) \quad \subset \quad \langle\, \mathbf{M}_2,\ \alpha \,\rangle(x_2)$$

The symbol for containment ($\overset{\sim}{\subset}$) contains two elements, suggesting *isomorphism* ($\cong$) and *subgraph* ($\subset$). These concepts characterize containment: $\mathbf{M}_1 \overset{\sim}{\subset} \mathbf{M}_2$ when each $\mathbf{M}_1$ graph is *isomorphic* to a *subgraph* of the corresponding $\mathbf{M}_2$ graph (with representation behaving nicely).

Special Cases Suppose $\mathbf{M}_1 \overset{\sim}{\subset} \mathbf{M}_2$, with domain $\mathcal{D}$. Then for each $\alpha \in \mathcal{D}$, Definition 7.1 tells us that two graphs—$\mathbf{M}_1(\alpha)$ and a a subgraph of $\mathbf{M}_2(\alpha)$—will satisfy Condition 1 and Condition 2. However, each of these conditions has a natural alternative that is more restrictive:

$1'$. **Identity** The graphs are identical.

$2'$. **Constituency Equality** The constituency of each atom in the $\mathbf{M}_2$ subgraph equals the constituency of the corresponding atom in the $\mathbf{M}_1$ graph.

[1]Naturally, we can make any pair of models share the a domain by replacing the individual domains with their intersection.

[2]We could require *identity* instead of isomorphism, but that would lead to some label awkwardness in Chapter 8 when we want to merge individual process graphs into a global graph. The relabeling that isomorphism permits will be convenient.

We obtain special cases of containment by replacing the original conditions with their stronger versions:

Definition 7.2 Suppose $\mathbf{M}_1 \overset{\sim}{\subset} \mathbf{M}_2$.

- If $\mathbf{M}_1$ and $\mathbf{M}_2$ also satisfy Definition 7.1 with Condition 1 replaced by Condition $1'$, we say that $\mathbf{M}_1$ *directly contains* $\mathbf{M}_2$ and write

$$\mathbf{M}_1 \quad \overset{-}{\subset} \quad \mathbf{M}_2$$

- If $\mathbf{M}_1$ and $\mathbf{M}_2$ also satisfy Definition 7.1 with Condition 2 replaced by Condition $2'$, we say that $\mathbf{M}_1$ *strongly contains* $\mathbf{M}_2$ and write

$$\mathbf{M}_1 \quad \overset{\sim}{\subseteq} \quad \mathbf{M}_2$$

Since the two conditions of Definition 7.1 are independent, we can strengthen both simultaneously, giving a third version:

- If $\mathbf{M}_1$ and $\mathbf{M}_2$ also satisfy Definition 7.1 with Condition 1 replaced by Condition $1'$ and Condition 2 replaced by Condition $2'$, we say that $\mathbf{M}_1$ *strongly directly contains* $\mathbf{M}_2$ and write

$$\mathbf{M}_1 \quad \overset{-}{\subseteq} \quad \mathbf{M}_2$$

Each of these relations is clearly transitive.

Figure 7.1 distinguishes containment from direct containment; Figure 7.2 distinguishes containment from strong containment.

Proposition 7.3 For any $\mathbf{M}_1, \mathbf{M}_2$,

1. $\mathbf{M}_1 \overset{-}{\subseteq} \mathbf{M}_2 \implies \mathbf{M}_1 \overset{\sim}{\subseteq} \mathbf{M}_2$
2. $\mathbf{M}_1 \overset{-}{\subseteq} \mathbf{M}_2 \implies \mathbf{M}_1 \overset{-}{\subset} \mathbf{M}_2$

Proof Condition $1'$ implies Condition 1, and Condition $2'$ implies Condition 2. $\square$

7.1.2. The Containment Map

It will be useful to transform the unique pairing from Definition 7.1 into a function:

Definition 7.4 Suppose $\mathbf{M}_1 \overset{\sim}{\subset} \mathbf{M}_2$, with shared domain $\mathcal{D}$. For $\alpha \in \mathcal{D}$, let P be the unique pairing satisfying Definition 7.1. Define the *containment map* $\langle\langle\, \mathbf{M}_2,\ \mathbf{M}_1,\ \alpha \,\rangle\rangle$

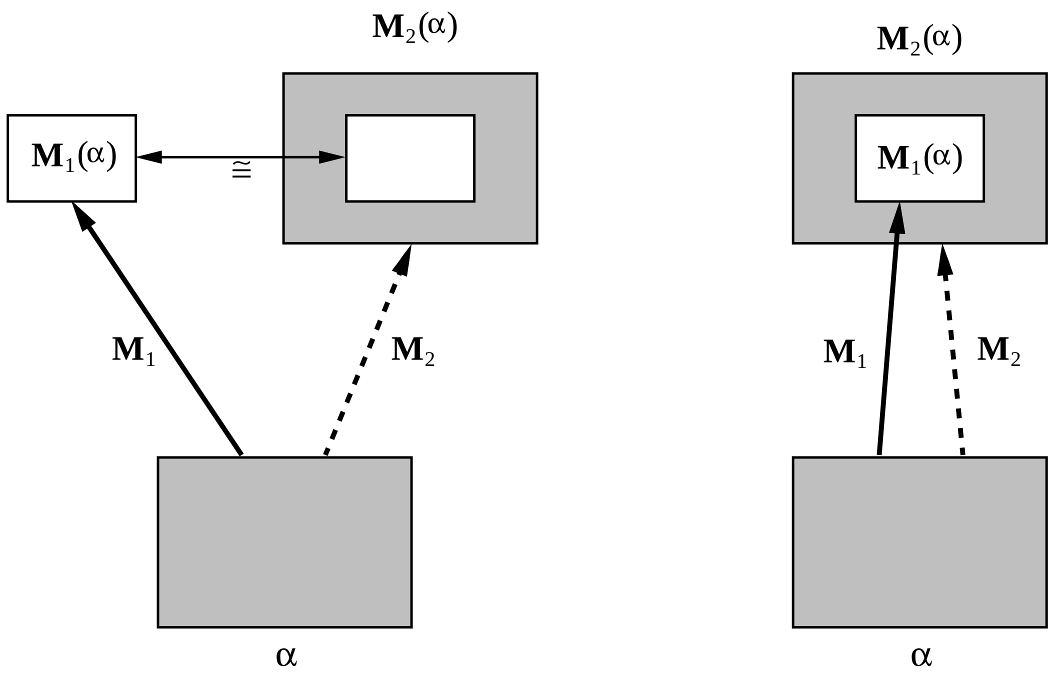

Figure 7.1 Containment of any form requires that one model always produces a graph isomorphic to a subgraph of what another model produces. However for direct containment, this isomorphism is in fact the identity. The left diagram shows ordinary containment: $\mathbf{M}_1 \mathrel{\tilde{\subset}} \mathbf{M}_2$; the right diagram shows direct containment: $\mathbf{M}_1 \mathrel{\bar{\subset}} \mathbf{M}_2$.

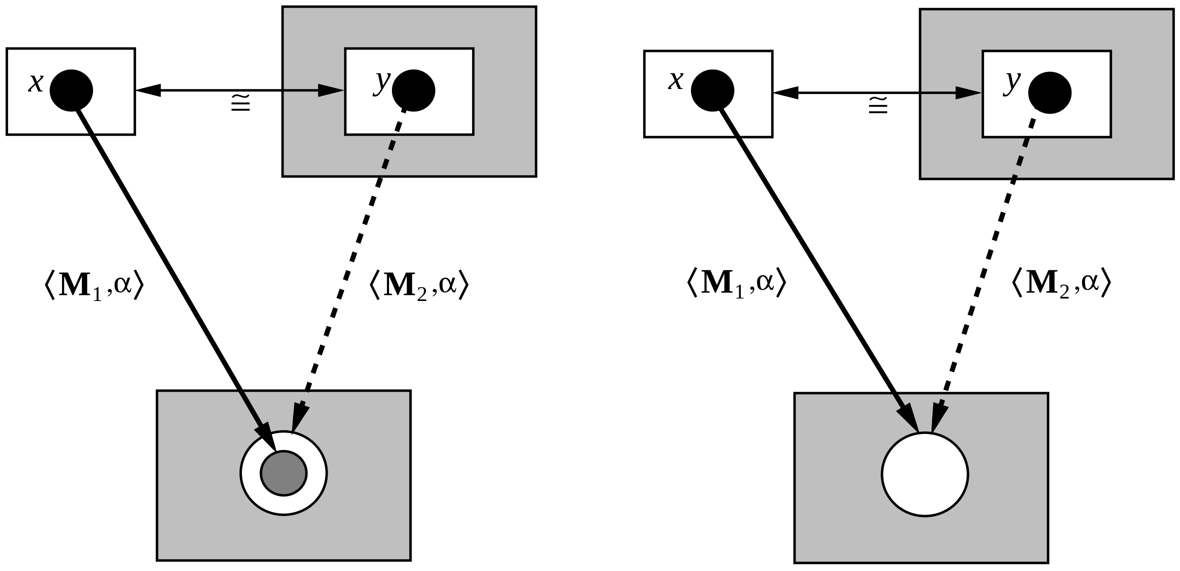
x
y
$\cong$
$\langle \mathbf{M}_1,\alpha \rangle$
$\langle \mathbf{M}_2,\alpha \rangle$
x
y
$\cong$
$\langle \mathbf{M}_1,\alpha \rangle$
$\langle \mathbf{M}_2,\alpha \rangle$

Figure 7.2 If $\mathbf{M}_1 \mathrel{\tilde{\subset}} \mathbf{M}_2$, then the M_1 representation on any atom x yields a subset of what the M_2 representation yields on the matching atom y (left). For strong containment $\mathbf{M}_1 \mathrel{\tilde{\subseteq}} \mathbf{M}_2$, the representations are equal (right).

to be the bijection that P determines from the $\mathbf{M}_2(\alpha)$ subgraph back to $\mathbf{M}_1(\alpha)$. That is,

$$\langle\langle\, \mathbf{M}_2,\ \mathbf{M}_1,\ \alpha\,\rangle\rangle(x_2) \;=\; x_1 \quad\Longleftrightarrow\quad (x_1, x_2) \in P$$

We can regard $\langle\langle\, \mathbf{M}_2,\ \mathbf{M}_1,\ \alpha\,\rangle\rangle$ as a partial function on all of $\mathbf{M}_2(\alpha)$.

Figure 7.3 illustrates the action of the containment map.

Hiding Awkward Notation Strictly speaking, the $\langle\langle\, \mathbf{M}_2,\ \mathbf{M}_1,\ \alpha\,\rangle\rangle$ function is partial. As such, it is not defined for some elements of its domain: namely, the atoms from $\mathbf{M}_2(\alpha)$ that are not part of the subgraph corresponding to $\mathbf{M}_1(\alpha)$. In order to prevent statements like "take the union of $\langle\langle\, \mathbf{M}_2,\ \mathbf{M}_1,\ \alpha\,\rangle\rangle$ over the set W" from becoming too awkward—because we would have to explicitly specify the subset of W for which the containment map is defined—we will adopt the convention that identification maps are "defined" on the remaining elements in the domain, but they just produce the empty set.

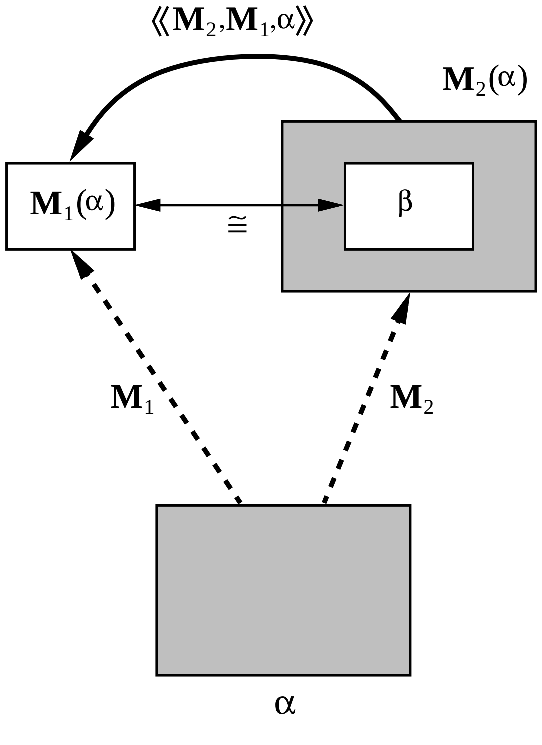

Figure 7.3 When one model contains another, a unique pairing connects the graphs they produce. Here we see that $\mathbf{M}_1 \overset{\sim}{\subset} \mathbf{M}_2$, so $\mathbf{M}_1(\alpha)$ is isomorphic to a subgraph β of $\mathbf{M}_2(\alpha)$. The containment map $\langle\langle\, \mathbf{M}_2,\ \mathbf{M}_1,\ \alpha\,\rangle\rangle$ acts on all of $\mathbf{M}_2(\alpha)$ to take this subgraph β back to $\mathbf{M}_1(\alpha)$.

This convention allows statements like the following:

$$\mathbf{M}_1(\alpha) \quad = \quad \left(\bigcup_{x \in \mathbf{M}_2(\alpha)} \langle\langle\, \mathbf{M}_2,\ \mathbf{M}_1,\ \alpha \,\rangle\rangle(x) \right)$$

7.1.3. Using Containment

Temporal Relations Suppose $\mathbf{M}_1 \overset{\sim}{\subset} \mathbf{M}_2$ act on graph α. Then we obtain $\mathbf{M}_2(\alpha)$ by copying $\mathbf{M}_1(\alpha)$, changing the labels, and adding more edges and nodes. This observation yields the following facts:

Proposition 7.5 Suppose $\mathbf{M}_1 \overset{\sim}{\subset} \mathbf{M}_2$ act on graph α. Let A_2 and B_2 be nodes in $\mathbf{M}_1(\alpha)$. Suppose $\langle\langle\, \mathbf{M}_2,\ \mathbf{M}_1,\ \alpha \,\rangle\rangle$ is defined on these nodes; define:

$$\begin{aligned} A_1 &= \langle\langle\, \mathbf{M}_2,\ \mathbf{M}_1,\ \alpha \,\rangle\rangle\ (A_2) \\ B_1 &= \langle\langle\, \mathbf{M}_2,\ \mathbf{M}_1,\ \alpha \,\rangle\rangle\ (B_2) \end{aligned}$$

Then:

$$\begin{aligned} A_1 \rightrightarrows B_1 \quad &\Longrightarrow \quad A_2 \rightrightarrows B_2 \\ A_2 \not\leftrightarrow B_2 \quad &\Longrightarrow \quad A_1 \not\leftrightarrow B_1 \end{aligned}$$

Proof Edges in the $\mathbf{M}_1$ graph show up in its isomorphic image in the $\mathbf{M}_2$ graph. □

Ignoring Edges Situations arise when we would rather ignore the edge constituencies when worrying about containment. To handle these cases, we introduce a new operator:

Definition 7.6 The model GHOSTIFY transforms a computation graph by forcing all edges to be ghosts.

Transitive Closure Taking the transitive closure will not cause containment to stop holding:

Proposition 7.7 For models $\mathbf{M}_1, \mathbf{M}_2$:

$$\begin{aligned} \mathbf{M}_1 \overset{\sim}{\subset} \mathbf{M}_2 \quad &\Longrightarrow \quad \overline{\mathbf{M}_1} \overset{\sim}{\subset} \overline{\mathbf{M}_2} \\ \mathbf{M}_1 \overset{-}{\subset} \mathbf{M}_2 \quad &\Longrightarrow \quad \overline{\mathbf{M}_1} \subset \overline{\mathbf{M}_2} \end{aligned}$$

Proof TRANS adds only ghost edges; and if TRANS adds an edge to the $\mathbf{M}_1$ graph that didn't already exist in the $\mathbf{M}_2$ graph, then TRANS will also add that edge to the $\mathbf{M}_2$ graph. □

Proposition 7.7 does not hold for strong containment: suppose $\mathbf{M}_2$ already has transitive edges in its $\mathbf{M}_1$ image, except these edges are not ghosts.

Examples of Containment The family of models from Chapter 6 provides a number of natural examples of containment:

Proposition 7.8 For $p \in$ **PROC-NAMES**:

$$\begin{array}{rcl}
\text{TIMELINE}_p & \bar{\subset} & \text{EXTREMA} \circ \text{TIMELINES} \\
\text{EXTREMA} \circ \text{TIMELINES} & \bar{\subseteq} & \text{POT} \\
\text{TIMELINE}_p & \bar{\subset} & \text{POT} \\
\text{GHOSTIFY} \circ \text{POT} & \bar{\subset} & \text{EXTREMA} \circ \overline{\text{LINEAR}} \\
\text{LINLINES} & \bar{\subseteq} & \text{LINEAR}
\end{array}$$

Proof

1. The p timeline shows up in the complete set; the representations coincide exactly except for the transitive extrema.
2. Only the message edges (and the transitive edges they imply) are missing.
3. Containment is transitive.
4. The POT graph is clearly a subgraph. The nodes have identical representations. But the edges of POT that do not appear in LINEAR will correspond to ghost edges in $\overline{\text{LINEAR}}$. The POT versions of these edges may actually represent something; hence the GHOSTIFY.
5. Only the SYNC edges are missing.

□

7.2. Refinement

Suppose we have two time models act on the same domain of computation graphs. Section 7.1 provides the terminology to talk about the situation when one model's graphs always contain images of the other model's graphs. However, our research has demonstrated the need to talk about a more subtle correlation: if model $\mathbf{M}_1$ collapses a set of input graphs by taking each of them to the same output graph, then model $\mathbf{M}_2$ also collapses this set. This property allows us to compare $\mathbf{M}_1$ and $\mathbf{M}_2$ graphs without having to go all the way back to the input graphs.

Formally, a model $\mathbf{M}$ with domain $\mathcal{D}$ induces a natural partition on a set $\mathcal{G} \subset \mathcal{D}$: just take the collection of sets $\mathbf{M}^{-1}(\mathbf{M}(\mathcal{G}))$. If two models $\mathbf{M}_1$ and $\mathbf{M}_2$ on the same domain have the property that, for any set, the $\mathbf{M}_2$ partition is strictly coarser, then specifying the computation graph $\mathbf{M}_1(\alpha)$ also determines the specific computation graph $\mathbf{M}_2(\alpha)$. In some sense, the actual value of α is irrelevant.

Definition 7.9 Suppose models $\mathbf{M}_1$ and $\mathbf{M}_2$ on the domain $\mathcal{D}$ have the property that, for all $\alpha, \alpha' \in \mathcal{D}$:

$$\mathbf{M}_1(\alpha) = \mathbf{M}_1(\alpha') \quad \Longrightarrow \quad \mathbf{M}_2(\alpha) = \mathbf{M}_2(\alpha')$$

Then we say that $\mathbf{M}_1$ *refines* to $\mathbf{M}_2$, and we write $\mathbf{M}_1 \rhd \mathbf{M}_2$.

Clearly, refinement is transitive.

The relation $\mathbf{M}_1 \rhd \mathbf{M}_2$ induces a function from $\mathbf{M}_1(\mathcal{D})$ to $\mathbf{M}_2(\mathcal{D}))$: we write $\beta_1 \rhd \beta_2$ when $\beta_i \in \mathbf{M}_i(\mathcal{D})$ and $\mathbf{M}_1^{-1}(\beta_1) \subset \mathbf{M}_2^{-1}(\beta_2)$. (This function does not extend to be a model itself because of the lack of any kind of representation. We have no well-defined correspondence between the atoms of a particular graph that $\mathbf{M}_2$ produces and the atoms of the graph that $\mathbf{M}_1$ produces on the same input.)

Figure 7.4 illustrates these relationships.

Transitive Closure Taking the transitive closure will not cause refinement to stop holding:

Proposition 7.10 For models $\mathbf{M}_1, \mathbf{M}_2$:

$$\mathbf{M}_1 \rhd \mathbf{M}_2 \quad \Longrightarrow \quad \overline{\mathbf{M}_1} \rhd \overline{\mathbf{M}_2}$$

Proof This follows directly from Definition 7.9. □

Abstraction Hierarchies Refinement allows us to put models into “abstraction hierarchies”: chains of models on a given domain that monotonically lose information—or gain abstraction.

Proposition 7.11 For any $p \in$ **PROC-NAMES**:

$$\text{LINEAR} \rhd \text{POT} \rhd \text{TIMELINES} \rhd \text{TIMELINE}_p$$

$$\text{LINEAR} \rhd \text{LINLINES} \rhd \text{LINLINE}_p$$

Proof These assertions follow directly from the definitions of the models. □

7.3. Components

Suppose $\mathbf{M}_1 \stackrel{\sim}{\subset} \mathbf{M}_2$, with shared domain $\mathcal{D}$. Then for any $\alpha \in \mathcal{D}$, $\mathbf{M}_1(\alpha)$ shows up in $\mathbf{M}_2(\alpha)$. However, this is still not sufficient to talk about the $\mathbf{M}_1$ component of a graph $\beta \in \mathbf{M}_2(\mathcal{D})$: suppose

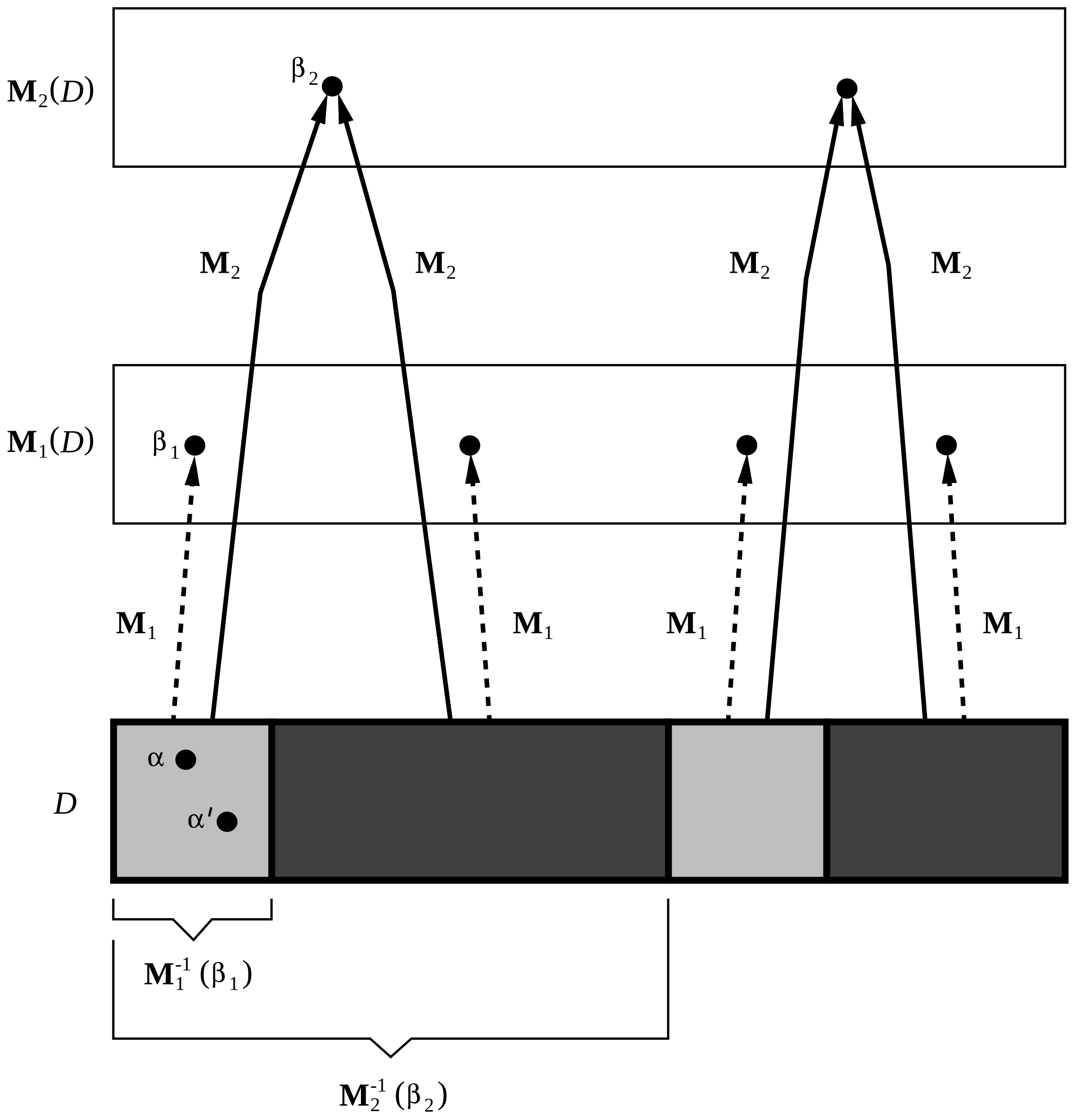

Figure 7.4 This diagram illustrates refinement: $\mathbf{M}_1 \trianglerighteq \mathbf{M}_2$. The dashed arrows indicate the action of $\mathbf{M}_1$; the solid arrows indicate the action of $\mathbf{M}_2$. We see that when $\mathbf{M}_1$ identifies two graphs (for example, α and α') by taking them to the same image, then $\mathbf{M}_2$ also identifies those two graphs. We see that the $\mathbf{M}_1$ value determines the $\mathbf{M}_2$ value: for example, knowing that $\mathbf{M}_1$ takes a graph to β_1 is sufficient to conclude that $\mathbf{M}_2$ takes that graph to β_2. We write $\beta_1 \trianglerighteq \beta_2$ to describe this relationship.

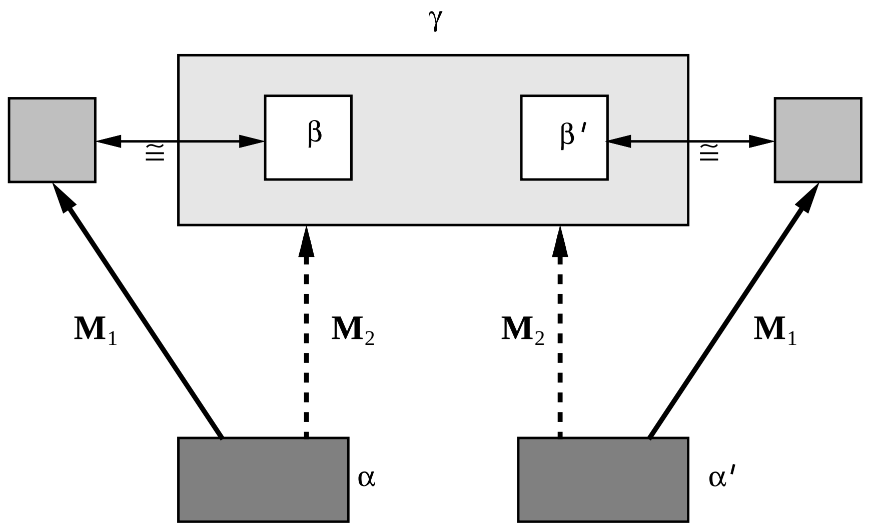

Figure 7.5 Containment does not guarantee well-defined components. Although $\mathbf{M}_1 \overset{\sim}{\subset} \mathbf{M}_2$ model $\mathbf{M}_1$ may be isomorphic to different subgraphs depending on the original graph. Here, $\gamma = \mathbf{M}_2(\alpha) = \mathbf{M}_2(\alpha')$ but β, the subgraph isomorphic to $\mathbf{M}_1(\alpha)$ differs from β', the subgraph isomorphic to $\mathbf{M}_1(\alpha')$.

γ is the M_2 image of both α and α' in $\mathcal{D}$ (that is, $\mathrm{M}_2(\alpha) = \mathrm{M}_2(\alpha') = \beta$), but $\mathrm{M}_1(\alpha) \neq \mathrm{M}_1(\alpha')$. Figure 7.5 illustrates this counterexample.

Talking unambiguously about the M_1 component of a graph generated by M_2 requires both containment and refinement:

Definition 7.12 Suppose M_1 and M_2 act on the same domain. M_1 is a *component* of M_2

$$\mathrm{M}_1 \quad \overset{\sim}{\sqsubset} \quad \mathrm{M}_2$$

when $\mathrm{M}_1 \overset{\sim}{\subset} \mathrm{M}_2$ and $\mathrm{M}_2 \;\triangleright\; \mathrm{M}_1$.

Each special case of containment (from Definition 7.2) gives rise to a corresponding special case of components:

Definition 7.13 Suppose M_1 and M_2 act on the same domain.

- If $\mathrm{M}_1 \overset{-}{\subset} \mathrm{M}_2$ and $\mathrm{M}_2 \;\triangleright\; \mathrm{M}_1$, then M_1 is a *direct component* of M_2:

$$\mathrm{M}_1 \quad \overset{-}{\sqsubset} \quad \mathrm{M}_2$$

- If $\mathrm{M}_1 \overset{\sim}{\subseteq} \mathrm{M}_2$ and $\mathrm{M}_2 \;\triangleright\; \mathrm{M}_1$, then M_1 is a *strong component* of M_2:

$$\mathrm{M}_1 \quad \overset{\sim}{\sqsubseteq} \quad \mathrm{M}_2$$

- If $\mathbf{M}_1 \overline{\sqsubseteq} \mathbf{M}_2$ and $\mathbf{M}_2 \; \triangleright \mathbf{M}_1$, then $\mathbf{M}_1$ is a *strong direct component* of $\mathbf{M}_2$:

$$\mathbf{M}_1 \quad \overline{\sqsubseteq} \quad \mathbf{M}_2$$

Each of these relations is transitive.

Informally, $\mathbf{M}_1 \tilde{\sqsubset} \mathbf{M}_2$ when the containment isomorphism takes the $\mathbf{M}_1$ graph to a well-defined subgraph of $\mathbf{M}_2$. One can take any graph produced by $\mathbf{M}_2$ and unambiguously select the $\mathbf{M}_1$ component.

Transitive Closure As with containment, taking the transitive closure will not cause non-strong containment to stop holding.

Proposition 7.14 For models $\mathbf{M}_1, \mathbf{M}_2$:

$$\begin{array}{rcl} \mathbf{M}_1 \tilde{\sqsubset} \mathbf{M}_2 & \Longrightarrow & \overline{\mathbf{M}_1} \tilde{\sqsubset} \overline{\mathbf{M}_2} \\ \mathbf{M}_1 \overline{\sqsubset} \mathbf{M}_2 & \Longrightarrow & \overline{\mathbf{M}_1} \overline{\sqsubset} \overline{\mathbf{M}_2} \end{array}$$

Proof This follows directly from Proposition 7.7 and Proposition 7.10. □

Examples Our family of models provides some examples of components.

Proposition 7.15 For each $p \in$ **PROC-NAMES**:

$$\begin{array}{rcl} \text{TIMELINE}_p & \overline{\sqsubset} & \text{POT} \\ \text{LINLINE}_p & \overline{\sqsubseteq} & \text{LINEAR} \end{array}$$

Proof Proposition 7.8 gives containment; Proposition 7.11 gives refinement. □

7.4. Decomposition

We have seen in Chapter 6 that our two more complex time models, LINEAR and POT, each have a fairly significant straight-line substructure. The LINEAR model has LINLINES; the POT model has TIMELINES.

Informally, we want to be able to talk about temporal orderings both in such higher-level models and in their substructures. The LINEAR model easily grants this ability. Not only is LINLINES a component of LINEAR; the "factorization"

$$\text{LINEAR} \;=\; \text{SYNC} \;\circ\; \text{LINLINES}$$

gives us a straightforward way to talk about the LINLINES graph of a computation as an intermediate step on the way to the LINEAR graph.

Performing the same task with the POT model is challenging. It cannot be the case that TIMELINES $\overset{\approx}{\sqsubset}$ POT because TIMELINES $\overset{\approx}{\subset}$ POT cannot hold: since the global extrema of POT graphs bind together the local extrema of TIMELINES graphs, a bijection cannot exist. However, the POT model does contain the individual TIMELINE$_p$ models. Further, the collection of these individual component models refines to the POT model.

Suppose we defined a model MSG$'$ that takes graphs with *send* and *receive* events and adds the MSG edges:

$$\text{MSG}'(\alpha) \quad = \quad \alpha \cup \text{MSG}(\alpha)$$

Then we could factor POT as well:

$$\text{POT} \quad = \quad (\text{MSG}' \circ \text{EXTREMA}) \quad \circ \quad \text{TIMELINES}$$

In this paper, we lay the foundations for work with models more general than POT and LINEAR. Hence, we want to isolate the general rule at work in this factorization. This section carries out this task. In Section 7.4.1 we explore the relationship between a model $\mathbf{M}$ and a single $\mathbf{M}_1 \overset{\approx}{\sqsubset} \mathbf{M}$. In Section 7.4.2 we demonstrate that a sufficiently rich set of components $\{\mathbf{M}_1, ..., \mathbf{M}_k\}$ will form a *decomposition* of a model $\mathbf{M}$: a substructure that we can factor out.

7.4.1. Model and Component

Suppose $\mathbf{M}_1$ is a submodel of $\mathbf{M}$. For any α in the shared domain, the containment map $\langle\langle\, \mathbf{M},\ \mathbf{M}_1,\ \alpha \,\rangle\rangle$ takes the atoms of the $\mathbf{M}$ graph back to the atoms of the $\mathbf{M}_1$ graph. Suppose, for all α, some further properties hold:

- $\mathbf{M}_1 \ \triangleright \mathbf{M}$
- The containment map $\langle\langle\, \mathbf{M},\ \mathbf{M}_1,\ \alpha \,\rangle\rangle$ is defined on all non-ghosts in the $\mathbf{M}$ graph.
- Whenever an atom of the $\mathbf{M}$ graph represents anything, it represents the same thing its $\langle\langle\, \mathbf{M},\ \mathbf{M}_1,\ \alpha \,\rangle\rangle$ image in the $\mathbf{M}_1$ graph.

The first property implies that each $\mathbf{M}_1$ graph determines an $\mathbf{M}$ graph, and the second and third imply that the atoms of the $\mathbf{M}_1$ graph determine (through the containment map) the constituencies of the atoms of the $\mathbf{M}$ graph.

Hence we can obtain a model $\mathbf{M}_2$ satisfying $\mathbf{M} = \mathbf{M}_2 \circ \mathbf{M}_1$.

Such an induced model would be practically the identity—we're just taking the $\mathbf{M}_1$ graph, relabeling the nodes, and possibly adding ghosts. However, if we had a set of components $\{\mathbf{M}_i\}$ satisfying a few convenient properties, rather than just the single submodel $\mathbf{M}_1$, then we can induce a model that is not so trivial. This insight yields the technique of decomposing models into components.

7.4.2. Decomposing Models into Components

When a collection of models $\mathbf{M}_1$, $\mathbf{M}_2$, ..., $\mathbf{M}_k$ are each components of a model $\mathbf{M}$, then each atom in $\mathbf{M}(\alpha)$ maps to a set (possibly empty) of atoms in the collection of graphs $\mathbf{M}_i(\alpha)$. If this set determines the α representation of the $\mathbf{M}$ atom, and the disjoint union of the collection refines to $\mathbf{M}$, then we can do some fun things.

Definition 7.16 Suppose $\mathbf{M}$ is a model on domain $\mathcal{D}$, $\{\mathbf{M}_1, ..., \mathbf{M}_k\}$ is a finite set of models on the same domain, and $\mathbf{M}' = \cup_{\emptyset}\{\mathbf{M}_1, ..., \mathbf{M}_k\}$. Then $\mathbf{M}'$ is a *decomposition* of $\mathbf{M}$, with *decomposition set* $\{\mathbf{M}_1, ..., \mathbf{M}_k\}$, when:

- $\mathbf{M}_i \overset{\simeq}{\sqsubset} \mathbf{M}$ for each i
- $\mathbf{M}' \quad \rhd \quad \mathbf{M}$
- For all graphs $\alpha \in \mathcal{D}$ and for all atoms $x \in \mathbf{M}(\alpha)$

$$\langle\, \mathbf{M},\ \alpha\, \rangle(x) \quad = \quad \bigcup_i \ (\ \langle\, \mathbf{M}_i,\ \alpha\, \rangle \circ \langle\langle\, \mathbf{M},\ \mathbf{M}_i,\ \alpha\, \rangle\rangle\ \ (x)\ )$$

Figure 7.6 illustrates the representation condition (the third bullet).

Proposition 7.17 If $\mathbf{M}'$ is a decomposition of $\mathbf{M}$, then $\overline{\mathbf{M}'}$ is a decomposition of $\overline{\mathbf{M}}$.

Proof This assertion follows from Proposition 7.10, Proposition 7.14 and Definition 7.16. $\square$

Proposition 7.18 The following hold:

1. TIMELINES is a decomposition of POT.
2. LINLINES is a decomposition of LINEAR.

Proof Both statements assert that a model decomposes to a disjoint union of a set of models. Proposition 7.15 gives that each element in the set is a component of the model. Proposition 7.11 gives refinement; and the definitions of the models gives the representation condition. $\square$

Model Your Own Decomposition When a collection of components $\{\mathbf{M}_i\}$ is a decomposition set for model $\mathbf{M}$, then we can determine whatever any $\mathbf{M}$ atom represents from what its $\{\mathbf{M}_i\}$ atoms represent. Hence we can perform the desired factorization and insert $\{\mathbf{M}_i\}$ between the input graphs and $\mathbf{M}$.

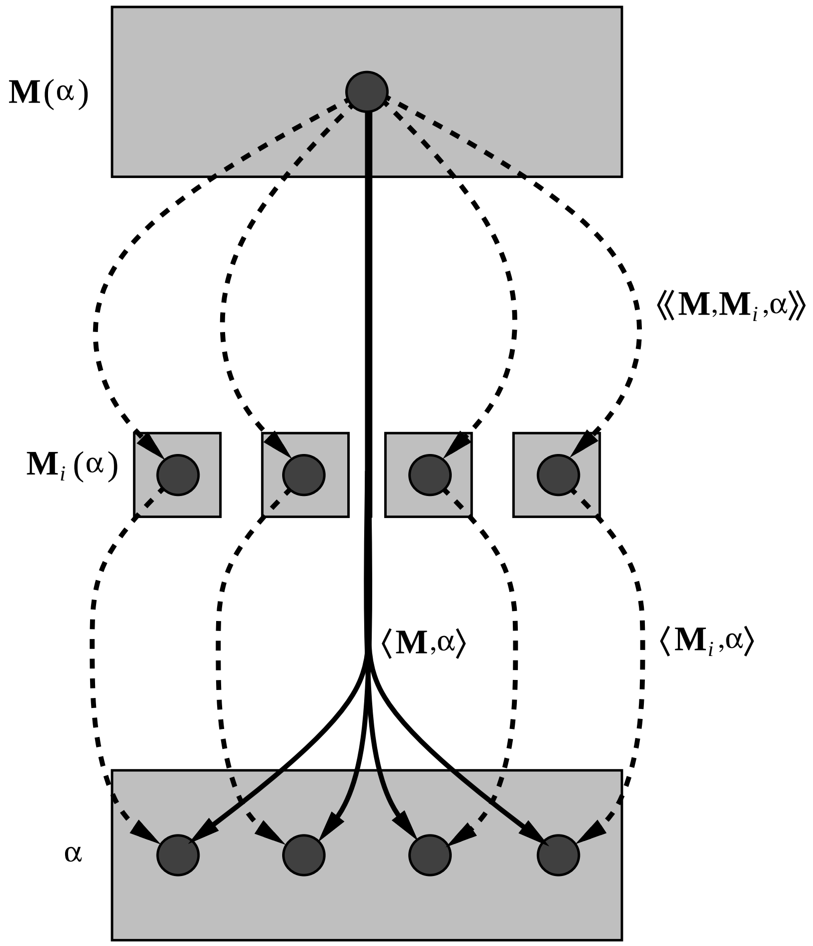
$\mathbf{M}(\alpha)$
$\langle\langle \mathbf{M},\mathbf{M}_i,\alpha\rangle\rangle$
$\mathbf{M}_i(\alpha)$
$\langle \mathbf{M},\alpha\rangle$
$\langle \mathbf{M}_i,\alpha\rangle$
α

Figure 7.6 For a set of components to form a decomposition of a model, the constituencies of the set should determine the constituencies of the model. In this example, we consider the model $\mathbf{M}$ and the set of components $\{\mathbf{M}_i\}$. We have two routes from an atom in $\mathbf{M}(\alpha)$ back to α. We can go directly through the representation map $\langle\, \mathbf{M},\ \alpha\,\rangle$ (solid arrow); or we can go to each $\mathbf{M}_i(\alpha)$ through the containment maps $\langle\langle\, \mathbf{M},\ \mathbf{M}_i,\ \alpha\,\rangle\rangle$, and from there to α through the representation maps $\langle\, \mathbf{M}_i,\ \alpha\,\rangle$ (dashed arrows). For the model $\mathbf{M}' = \cup_{\emptyset}\{\mathbf{M}_i\}$ to be a decomposition, these routes must always yield the same set of atoms in α.

Definition 7.19 Suppose $\mathbf{M}$ on domain $\mathcal{D}$ has decomposition $\mathbf{M}'$, with decomposition set $\{\mathbf{M}_1, ..., \mathbf{M}_k\}$. Define the *factoring model* $\mathbf{M}/\mathbf{M}'$ on the domain $\mathbf{M}'(\mathcal{D})$ as follows.

Let $\beta = \mathbf{M}'(\alpha)$, for $\alpha \in \mathcal{D}$. Then $\mathbf{M}/\mathbf{M}'$ takes β to the γ from $\mathbf{M}(\mathcal{D})$ satisfying $\beta \mathrel{|\!\!\triangleright} \gamma$. The representation map applies each containment map to an atom and collects the results:

$$\langle\, \mathbf{M}/\mathbf{M}'\,,\ \beta\,\rangle(x) \;\;=\;\; \{\langle\langle\, \mathbf{M},\ \mathbf{M}_i,\ \alpha\,\rangle\rangle(x) \;\;:\;\; 1 \le i \le k\}$$

Figure 7.7 sketches the structures from Definition 7.19.

Theorem 7.20 *(Factorization)* Suppose model $\mathbf{M}$ has decomposition $\mathbf{M}'$. Then

$$\mathbf{M} \;\;=\;\; \mathbf{M}/\mathbf{M}' \;\circ\; \mathbf{M}'$$

Proof This assertion follows directly from Definition 7.16 and Definition 7.19: the composition on the right gives the same transformation action and representation map as the model $\mathbf{M}$. $\square$

The observations from the beginning of this section are special cases of the Factorization Theorem:

$$\begin{aligned} \text{LINEAR}/\text{LINLINES} &= \text{SYNC} \\ \text{POT}/\text{TIMELINES} &= \text{MSG}' \circ \text{EXTREMA} \end{aligned}$$

Proposition 7.21 Suppose model $\mathbf{M}$ has decomposition $\mathbf{M}'$, with decomposition set $\{\mathbf{M}_1, ..., \mathbf{M}_k\}$. Under **M/M'** , for any i, an atom x from $\mathbf{M}$ represents either nothing at $\mathbf{M}_i$ or an image of itself at $\mathbf{M}_i$.

Proof By Definition 7.16, $\mathbf{M}_i \overset{\simeq}{\sqsubset} \mathbf{M}$, hence $\mathbf{M}_i \overset{\sim}{\subset} \mathbf{M}$. Thus $\mathbf{M}_i(\alpha)$ is isomorphic to a subgraph of $\mathbf{M}(\alpha)$, and this relationship determines the representations in $\mathbf{M}/\mathbf{M}'$. $\square$

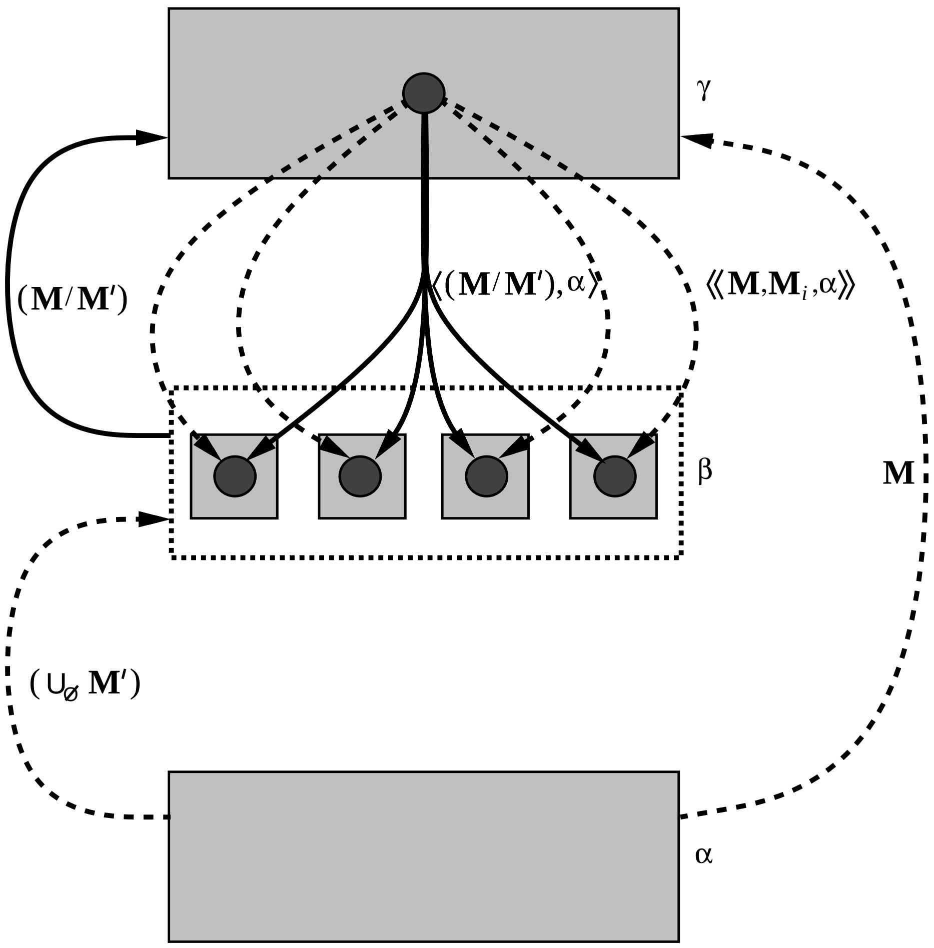

Figure 7.7 Model $\mathbf{M}$ with decomposition $\mathbf{M}'$ induces a *factoring model* $\mathbf{M}/\mathbf{M}'$. The decomposition $\mathbf{M}' = \cup_\varnothing\{\mathbf{M}_i\}$ takes α (bottom) to β (middle) ; the model $\mathbf{M}$ takes α to γ (top). The factoring model $\mathbf{M}/\mathbf{M}'$ takes β to γ; in the new model, an atom of γ represents the union of its images in each $\mathbf{M}_i$ component. (Solid arrows indicate both the action and the representation of the new model. Dashed arrows indicate the action of $\mathbf{M}$, $\mathbf{M}'$, and the containment maps.)

Part III

Simultaneity

The desire to distinguish between genuine real time and the temporal relations that processes themselves perceive motivated the development of the model family in Chapter 6. The LINEAR model expresses the former; the POT model expresses the latter.

A natural concept from traditional linear time is *simultaneity*: at any given moment, a single photograph describes the state of the system. However, simultaneity is one of the first casualties of asynchrony in a distributed system. We can still talk about “consistent” global states, but these states may never have physically occurred. Time ceases to be a nicely behaved sequence of individual moments.

Part III explores these issues, and extends the previous work of Mattern [Ma89] and Johnson [Jo89, JoZw90].

Section 7.4 observed that the goal here is to lay a foundation for more general work with more general models than POT and LINEAR; hence Part III begins in Chapter 8 by characterizing these models.

Chapter 9 explores logical simultaneity—global states, in terms of the semantics of time models—and shows how these can form a lattice structure. Chapter 10 discusses two convenient vector structures that arise from logical simultaneity. We then relate logical simultaneity to the simultaneity of real time: Chapter 11 explores the basic structures; Chapter 12 examines why models such as POT fail to give the desired simultaneity properties; and Chapter 13 proposes some solutions.

Finally, Chapter 14 makes some deeper observations about the structure of logical global states.

(Part III)

Chapter 8

Parallel Models

This chapter characterizes the models to which the machinery of of Part III applies: computation that takes place at different processes in *parallel*. Section 8.1 introduces the structure of the *multiprocess pair*: a model describing the local process computation, along with one describing the global system computation. Section 8.2 presents some tools for models from this structure, and Section 8.3 isolates some interesting subsets of multiprocess pairs—including the subset *parallel pairs*, describing parallel computation.

8.1. Multiple Processes

We want a two-level perspective on system behavior: things happen locally at processes, but these things also happen globally in the system. We define a mechanism to provide this dual perspective:

> **Definition 8.1** Suppose models $\mathbf{M}$ and $\mathbf{M}'$ on ground-level computation graphs satisfy the following:
>
> 1. $\mathbf{M}'$ is a decomposition of $\mathbf{M}$, with decomposition set $\{\mathbf{M}_i\}$
> 2. Each $\mathbf{M}_i$ describes events at a unique process.[1]
> 3. The factoring model $\mathbf{M}/\mathbf{M}'$ has no ghost events.
>
> Then we say that $(\mathbf{M}, \mathbf{M}')$ is a *multiprocess pair*. Model $\mathbf{M}$ is a *multiprocess* model, with *multicomponent* $\mathbf{M}'$.

The family of models from Chapter 6 provides some natural (and intentional) examples: both $(\text{LINEAR}, \text{LINLINES})$ and $(\text{POT}, \text{TIMELINES})$ are multiprocess pairs.

[1]Actually, there's no reason why the "process" for this decomposition should be the same as the "process" for the basic system model of Chapter 2. This work should easily extend to handle such wrinkles as process migration and virtual processes.

Suppose multiprocess pair $(\mathbf{M}, \mathbf{M}')$ acts on graph α. The multicomponent $\mathbf{M}'$ transforms graph α into a set of *local* process descriptions; the model $\mathbf{M}$ transforms graph α to the *global* system description. The factoring model $\mathbf{M}/\mathbf{M}'$ takes the *local* process descriptions to the global system descriptions; events from different processes may merge, but no new events are added.

Multiple Perspectives The two models in a multiprocess pair provide two views of a computation: as independent local threads, and as a unified global whole. Frequently we want to make another kind of distinction: between basic transitions and full transitive precedence. For example, in the $(\text{POT}, \text{TIMELINES})$ we may want to distinguish between *immediate* precedence

$$A \longrightarrow B \quad \text{in} \quad \text{TIMELINES}$$

and *transitive* precedence

$$A \longrightarrow B \quad \text{in} \quad \overline{\text{TIMELINES}}$$

Proposition 7.17 tells us that if $(\mathbf{M}, \mathbf{M}')$ is a multiprocess pair, then so is $(\overline{\mathbf{M}}, \overline{\mathbf{M}'})$. So suppose we want to build transitive temporal relations arising from some notion of "basic transition steps." If we build $(\mathbf{M}, \mathbf{M}')$ so that edges express basic steps, then $(\overline{\mathbf{M}}, \overline{\mathbf{M}'})$ is a parallel pair giving the full transitive steps. Thus the multiprocess pair $(\mathbf{M}, \mathbf{M}')$ provides four views of an underlying computation α. Figure 8.1 illustrates the multiple perspectives.

8.2. Tools

We now introduce some tools to facilitate using the multiprocess pair machinery.

8.2.1. Projection

Working with multiprocess pairs will frequently require constructing objects with the structure "one thing per each process component." We use standard notation to move between each object and the individual entries:

> **Definition 8.2** When a set has the property that each $p \in$ **PROC-NAMES** is unambiguously associated with a unique element of the set, we use projection to select these elements. For example, $\overline{\mathbf{M}'}$ is the disjoint union of process models; $\pi_p \overline{\mathbf{M}'}$ refers to the model for process p.

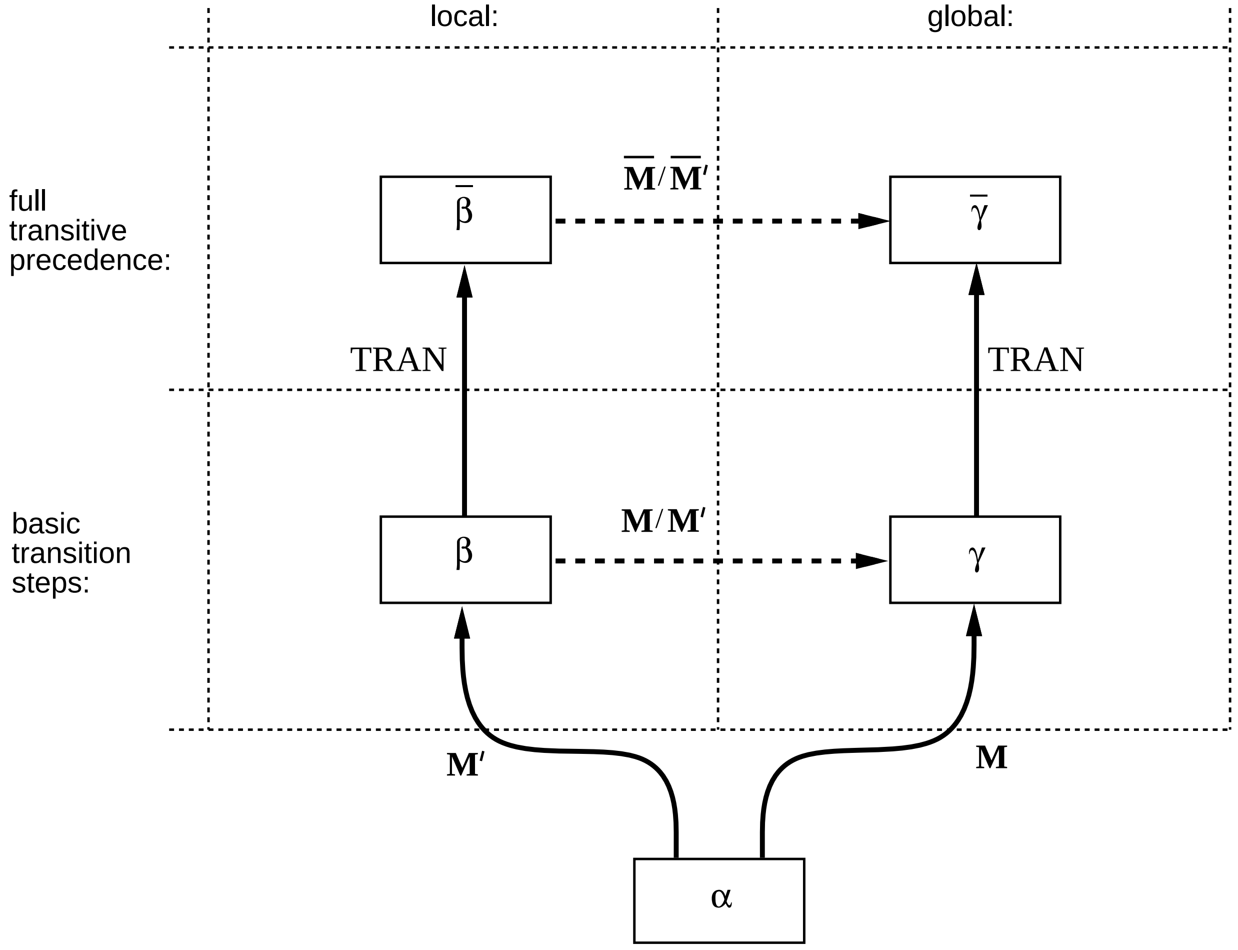

Figure 8.1 A multiprocess pair provides four views of a computation, according to two independent choices: whether we use the model or the multicomponent, and whether or not we take the transitive closure. Here, pair $(\mathbf{M}, \mathbf{M}')$ acts on graph α, with $\beta = \mathbf{M}'(\alpha)$ and $\gamma = \mathbf{M}(\alpha)$. Graph β provides the basic transition step version of the local computation; graph $\overline{\beta}$ provides the full transitive closure. Graphs γ and $\overline{\gamma}$ provide the global system descriptions.

8.2.2. Events in Models and Multicomponents

Multiple Roles Suppose graph α lies in the shared domain of multiprocess pair $(\mathbf{M}, \mathbf{M}')$. Each atom that $\mathbf{M}$ produces has its origins in the process components from $\mathbf{M}'$. In this sense, an atom x from $\mathbf{M}(\alpha)$ has multiple roles:

- as itself: $x \in \mathbf{M}(\alpha)$
- as the atom (if any) it represents at a particular process component of $\mathbf{M}'$: $\langle\langle\, \mathbf{M},\ \pi_p\, \mathbf{M}',\ \alpha\, \rangle\rangle(x)$

An atom x from $\overline{\mathbf{M}}(\alpha)$ has two more roles:

- as its $\overline{\mathbf{M}'}$ constituency: $\langle\ \overline{\mathbf{M}}/\overline{\mathbf{M}'}\,,\ \overline{\mathbf{M}'}(\alpha)\,\rangle(x)$
- as the atom (if any) it represents at a particular process component of $\overline{\mathbf{M}'}$: $\langle\langle\, \overline{\mathbf{M}},\ \pi_p\, \overline{\mathbf{M}'},\ \alpha\, \rangle\rangle(x)$

To simplify discussing these multiple roles, we introduce some notational shortcuts:

Definition 8.3 When multiprocess pair $(\mathbf{M}, \mathbf{M}')$ acting on graph α are understood, define these operators on atoms x from $\mathbf{M}(\alpha)$:

- $x|_{\mathbf{M}'} \quad = \quad \langle\ \mathbf{M}/\mathbf{M}'\,,\ \mathbf{M}'(\alpha)\,\rangle(x)$
- $x|_p \quad = \quad \langle\langle\, \mathbf{M},\ \pi_p\, \mathbf{M}',\ \alpha\, \rangle\rangle(x)$, for $p \in$ **PROC-NAMES**

For atoms x in $\overline{\mathbf{M}}(\alpha)$ define two more operators:

- $x|_{\overline{\mathbf{M}'}} \quad = \quad \langle\ \overline{\mathbf{M}}/\overline{\mathbf{M}'}\,,\ \overline{\mathbf{M}'}(\alpha)\,\rangle(x)$
- $x|_{\overline{p}} \quad = \quad \langle\langle\, \overline{\mathbf{M}},\ \overline{\mathbf{M}'},\ \alpha\, \rangle\rangle(x)$, for $p \in$ **PROC-NAMES**

Extend each operator to act on sets of atoms by applying it to each element of the set and collecting the results.

The operators from Definition 8.3 possess simple mnemonics: $x|_{\mathit{foo}}$ takes atom x to whatever it represents in *foo*. These operations also satisfy some easy identities:

Proposition 8.4 For multiprocess pair $(\mathbf{M}, \mathbf{M}')$, let x be an atom in a graph generated by $\mathbf{M}$. Then:

$$x|_{\mathbf{M}'} \quad = \quad \{\, x|_p \quad : \quad p \in \textbf{PROC-NAMES}\}$$

If x is an atom in a graph generated by $\overline{\mathbf{M}}$, then

$$x|_{\overline{\mathbf{M}'}} \quad = \quad \{\, x|_{\overline{p}} \quad : \quad p \in \textbf{PROC-NAMES}\}$$

If x is a node, then for any $p \in$ **PROC-NAMES**:

$$x|_{\overline{p}} \quad = \quad x|_p$$

Proof These assertions follows directly from Definition 8.3 and Definition 7.16, and the fact that transitive closure only adds edges. □

From "Nodes" to "Events" Time models produce computation graphs. However, in practice these graphs describe computations; nodes in a graph represent correspond to *events* in the computation.

Until now, we've talked about computation graphs as graphs; hence we've referred to nodes as nodes. But now we want to begin using graphs as descriptions of computations; **hence we shift terminology from "node" (the object in the graph) to "event" (the reality presumably behind this object).**

8.2.3. Temporal Relations in Models and Multicomponents.

Localizations Since the individual process models are in fact *components* of the global model, a path between two events in the multicomponent induces an edge between those events in a multiprocess model. It will be useful to talk about these edges without having to move down to the multicomponent and back up. We can perform this by trimming down the $\overline{\mathbf{M}}$ graph to include only those edges arising out of $\overline{\mathbf{M}'}$. These will be the non-ghosts in the factoring model.

Definition 8.5 The *localization* of multiprocess pair $(\mathbf{M}, \mathbf{M}')$ is the model $\mathbf{L}$ that takes the event set from $\mathbf{M}$ and draws the edges induced by $\overline{\mathbf{M}'}$.

For example, the model EXTREMA $\circ$ $\overline{\text{TIMELINES}}$ is the localization of multiprocess pair $(\text{POT}, \text{TIMELINES})$.

Proposition 8.6 Suppose multiprocess pair $(\mathbf{M}, \mathbf{M}')$ with localization $\mathbf{L}$ acts on graph α. Let $\gamma = \mathbf{M}(\alpha)$, $\gamma_L = \mathbf{L}(\alpha)$ and $\beta = \mathbf{M}(\alpha)$.

Events $A, B \in \gamma$ satisfy $A \longrightarrow B$ in γ_L iff some $A' \in A|_{\mathbf{M}'}$ and $B' \in B|_{\mathbf{M}'}$ satisfy $A' \longrightarrow B'$ in $\overline{\beta}$.

Proof This follows from the definitions. If $A' \longrightarrow B'$ in some transitive process component, then $A \longrightarrow B$ in $\overline{\gamma}$ by an edge representing an edge in $\overline{\beta}$. □

Corollary 8.7 Let $(\mathbf{M}, \mathbf{M}')$ be a multiprocess pair. If $\overline{\mathbf{M}'}$ is acyclic, then the localization has no self-loops: no edge connects a node to itself.

While the localization is not transitive, it does possess a "local transitivity." Suppose when we construct the localization, we label each edge with the process model from whence it came. Then the localization has the property that if a path exists whose edges are all labelled with process p, then an equivalent edge exists, also labeled with p.

Localization adds an intermediate perspective between the local $\mathbf{M}'$ and the global $\mathbf{M}$: we use the events from $\mathbf{M}$ but retain only the edges from $\overline{\mathbf{M}'}$. Since taking the transitive closure of the localization would ruin Proposition 8.6, the localization perspective is intermediate on that axis too. Figure 8.2 illustrates the revised view.

Local Closures Another useful operation is collecting the events that locally precede some set of system events.

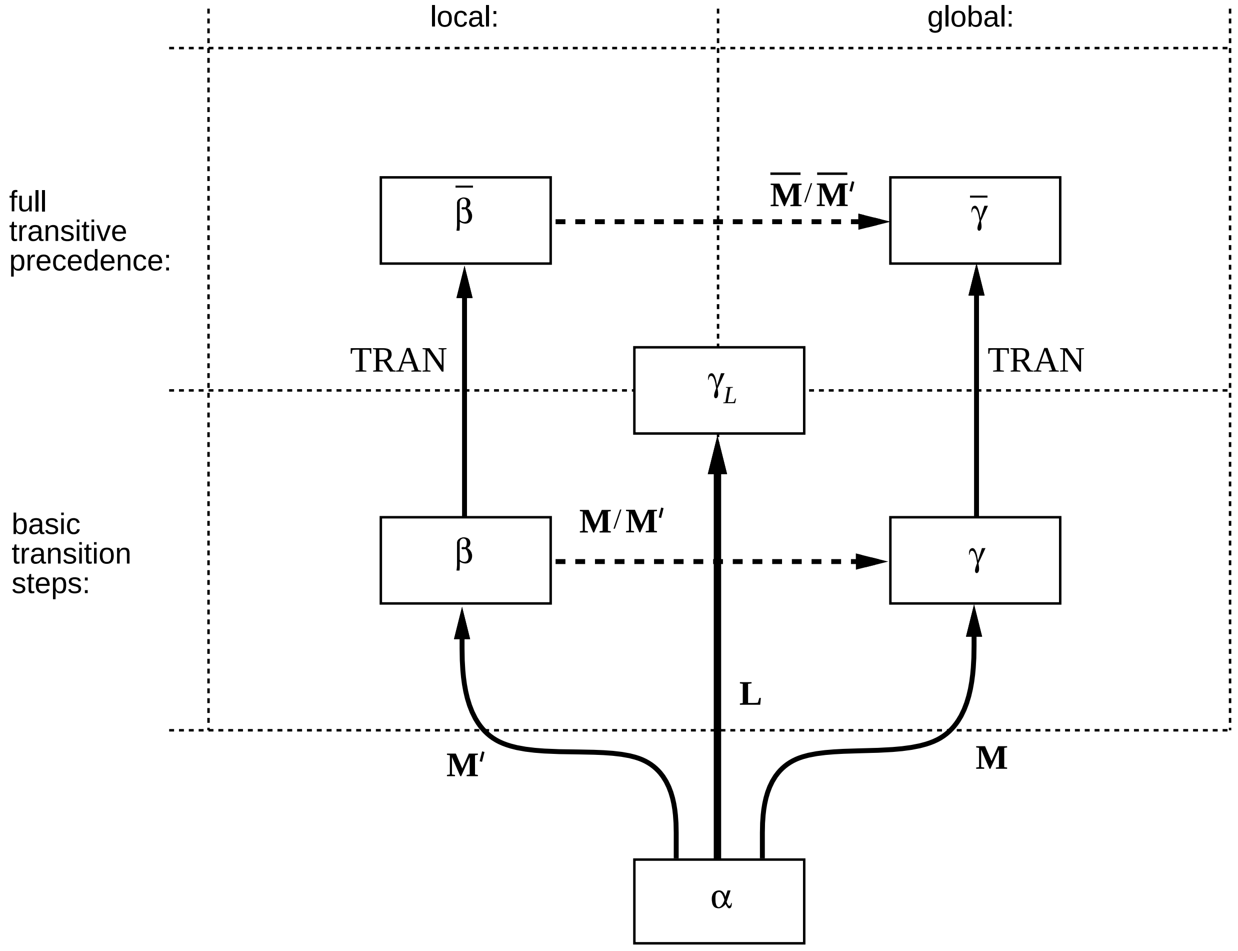

Figure 8.2 The localization of a a multiprocess pair provides a intermediate perspective between the local and the global, and between the transitive and the non-transitive. With localization, we now have five views: the localization is central. Here, pair $(\mathbf{M}, \mathbf{M}')$ with localization $\mathbf{L}$ acts on graph α, with $\beta = \mathbf{M}'(\alpha)$, $\gamma_L = \mathbf{L}(\alpha)$ and $\gamma = \mathbf{M}(\alpha)$.

Definition 8.8 Suppose multiprocess pair $(\mathbf{M}, \mathbf{M}')$ with localization $\mathbf{L}$ acts on graph α. Let $\gamma = \mathbf{M}(\alpha)$, $\beta = \mathbf{M}'(\alpha)$, and $\gamma_L = \mathbf{L}(\alpha)$.

For a set of events X from γ, define its *local past-closure* $\lceil X \rceil$ to be the set of event that precede or equal X in γ_L.

$$\lceil X \rceil \quad = \quad \{A \ : \ \text{for some } B \in X,\, A \Longrightarrow B \text{ in } \gamma_L\}$$

Define the *local future-closure* $\lfloor X \rfloor$ similarly.

Local closures select events from an $\mathbf{M}$ graph on the basis of the relations of their pre-images in $\overline{\mathbf{M}'}$. Hence, if a set X consists of POT events at different processes, $\lceil X \rceil$ contains only copy of $\perp$, rather than a copy for each process.

8.3. Variations

8.3.1. Concurrent Pairs

The definition of multiprocess pair says very little about how the global model glues together the individual process components. For example, we could take multicomponent LINLINES and merge one process's maximum with another's minimum.

To describe the situation when the activity at different processes takes place concurrently, we introduce a special term:

Definition 8.9 Suppose multiprocess pair $(\mathbf{M}, \mathbf{M}')$ has domain $\mathcal{D}$. We say that $(\mathbf{M}, \mathbf{M}')$ is *concurrent* when for any $\alpha \in \mathcal{D}$,

1. If A and B are maxima from different process components in $\overline{\mathbf{M}'}$, then
$$(\overline{\mathbf{M}}/\overline{\mathbf{M}'})\,(A) \quad \not\longleftrightarrow \quad (\overline{\mathbf{M}}/\overline{\mathbf{M}'})\,(B)$$
2. If A and B are minima from different process components in $\overline{\mathbf{M}'}$, then
$$(\overline{\mathbf{M}}/\overline{\mathbf{M}'})\,(A) \quad \not\longleftrightarrow \quad (\overline{\mathbf{M}}/\overline{\mathbf{M}'})\,(B)$$

Both $(\text{LINEAR}, \text{LINLINES})$ and $(\text{POT}, \text{TIMELINES})$ are concurrent.

If the model is transitively bounded as well, then the extrema have a simple structure:

Proposition 8.10 If a concurrent model is transitively bounded, then the unique global maximum represents the individual process maxima (and similarly for the minima).

Proof Let $\mathbf{M}$ be the concurrent model. Fix an input graph α, and let A be the global maximum in $\overline{\mathbf{M}}(\alpha)$. By Definition 8.1, event A has to represent some event A' at some process p. Event A' must be a p maximum, for otherwise A could not be maximal. Let $B' \neq A'$ be a maximum at process q (not necessarily distinct from p) and let B be its image in $\mathbf{M}$. If $B \neq A$, then $B \longrightarrow A$ (because A is maximal) and hence the model could not be concurrent. $\square$

8.3.2. Multilinear Pairs

The definition of multiprocess pair also says very little about the individual process components. We introduce a special term to describe when these components look like timelines:

> **Definition 8.11** A multiprocess pair $(\mathbf{M}, \mathbf{M}')$ is *multilinear* when the individual process components produce only straight-line graphs.

8.3.3. Parallel Models

Definition 8.9 ensures that the local process components happen in "parallel." Definition 8.11 ensures that process components are timelines. Together, these conditions describe what we usually regard as "parallel computation." Figure 8.3 illustrates this taxonomy.

> **Definition 8.12** Suppose multiprocess pair $(\mathbf{M}, \mathbf{M}')$ is concurrent, and each $\mathbf{M}'$ component always produces straight-line graphs. We say that $\mathbf{M}$ is *parallel* and that $(\mathbf{M}, \mathbf{M}')$ is a *parallel pair.*

Both $(\text{LINEAR}, \text{LINLINES})$ and $(\text{POT}, \text{TIMELINES})$ are parallel pairs.

Other Directions Part III explores properties of parallel pairs. However, more advanced work may require dealing with more general varieties; for example, rollback may require allowing process components to be trees rather than straight-line graphs. Hence, future research may involve slightly generalizing our parallel pair machinery.

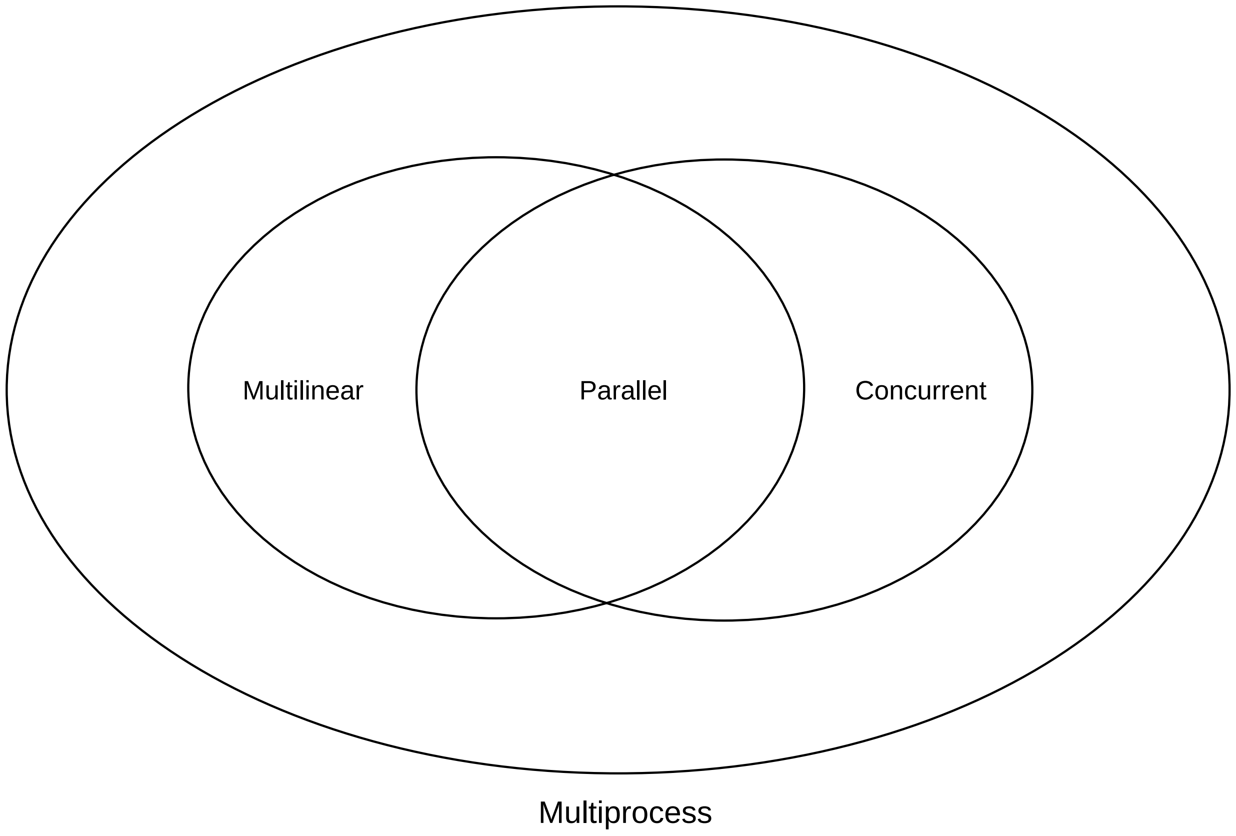

Figure 8.3 A *parallel pair* is a multiprocess pair that is concurrent, and where each process component is a straight-line graph.

(Part III)

Chapter 9

Logical Simultaneity

A time model M on ground-level computation graphs imposes a web of ordering on the events in an unfolding computation. A maximal set of mutually concurrent events represents a logical slice of time across this computation—"logical" in the sense that in the semantics of the time model, this set describes a possible moment of simultaneity.

Section 9.1 considers a number of approaches to describing logical global states in parallel pairs, and shows how they all arise from *timeslices*: sets of events forming logical slices of time.[1] Section 9.2 presents some natural operations on event sets; Section 9.3 uses these operations to establish the set of timeslices forms a lattice.

The literature diverges on the exact definitions of many of the terms that arise here (e.g., "consistent cut"); to avoid any ambiguity, we take pains to indicate clearly the definitions we use.

9.1. Timeslices

9.1.1. Vectors and Cuts

We want an object to express a system-wide "system state." Informally, this should be a tuple of events, one per process. However, the fact that events can occur at multiple processes complicates matters.

Thus, in general this "one-per-process" rule has two possible formal characterizations. We introduce terms for both:

Definition 9.1 Suppose $(\mathrm{M}, \mathrm{M}')$ is an multiprocess pair, and γ is a graph from M.

1. An event *vector* is an *array* of events from γ with the constraint that the process p entry occurs at process p.

[1][Sp89] uses the term "timeslice" (and [Ma89] uses "time slice"); the timeslices there are special cases of the timeslice here.

2. An event *cut* is a *set* of events from γ such that for each process p, *exactly* one event occurs at p.

Vectors are *arrays*, rather than sets or multisets, because events may occur at multiple processes. Indexing allows the entries in a vector to carry a banner indicating their origin. Suppose distinct events A and B both occur at processes p and q, and a vector V contains both A and B. Without indexing, we could not tell which was the process p entry of vector V.

Every cut is the event set from a unique vector: the cut provides exactly one event for each vector entry. However, not every vector has an event set that is a cut: suppose events $A \neq B$ both occur at both p and q; vector V may contain both.

In graph theory, a cut is a set of nodes whose removal leaves the graph disconnected. In our usage, a cut is a set of events that cuts each timeline in a parallel pair.

9.1.2. Timeslices

So far, we've just used the fact that each process's component describes a concurrent part of the computation. A computation graph specifies temporal ordering on events, and hence on the events in a set.

Definition 9.2 A set of events X in a computation graph is *mutually concurrent* when no events A, B in X (not necessarily distinct) satisfy $A \longrightarrow B$.

When a set of events is mutually concurrent, then—in the semantics of the model—no event in this set happened before another event. If the set is maximal, then any other event in the computation must have happened either before or after some event in this set. Thus in terms of the model, this set describes a possible simultaneous moment.

Definition 9.3 A *timeslice* from a computation graph α is a maximal set of mutually concurrent events. An α-timeslice is a timeslice in graph α. An $\mathbf{M}$-timeslice is a timeslice in a graph that model $\mathbf{M}$ generates.

Suppose $\mathbf{M}_1 \overset{\sim}{\sqsubseteq} \mathbf{M}_2$. Applying Proposition 7.5 tells us how timeslices from $\mathbf{M}_1$ relate to timeslices from $\mathbf{M}_2$:

- Timeslices from $\mathbf{M}_1$ map into sets of events in $\mathbf{M}_2$ (since we might gain edges in $\mathbf{M}_2$).
- Conversely, timeslices from $\mathbf{M}_2$ map to subsets of timeslices in $\mathbf{M}_1$ (since we may lose edges and even events going back to $\mathbf{M}_1$).

9.1.3. Timeslices in Parallel Pairs

If we're using a parallel pair to describe temporal precedence on global events, then we have three perspectives: the transitive local model, the localization, and the transitive global model. We've already seen timeslices from two of these structures.

Timeslices in the Local Model Timeslices from the transitive local model are isomorphic to *vectors.*

Proposition 9.4 Let $(\mathbf{M}, \mathbf{M}')$ be a parallel pair. Then V is a vector in $\mathbf{M}$ iff $\{ (\pi_p V)|_{\overline{p}} \ : \ p \in$ **PROC-NAMES**$\}$ is a timeslice in $\overline{\mathbf{M}'}$.

Proof $\overline{\mathbf{M}'}$ produces a collection of total orders, one from each process. □

Timeslices in the Localization Timeslices in the localization are *cuts*.

Proposition 9.5 Let $(\mathbf{M}, \mathbf{M}')$ be a parallel pair, with localization $\mathbf{L}$. A set X is a cut in $\mathbf{M}$ iff X is a timeslice in $\mathbf{L}$.

Proof Suppose X is a $\mathbf{L}$-timeslice. If process p is not represented, then any A touching p is mutually concurrent with any element of X. If process p is represented twice, then X cannot be mutually concurrent. Conversely, no distinct A, B in a cut X can precede each other (by Proposition 8.6), and no A can be a self-loop (by Corollary 8.7). But any other event in the graph must touch some process p and hence be ordered with $\pi_p X$, so X is maximal. □

Timeslices in the Global Model This third case is tricky: timeslices in the transitive global model are at least partial cuts.

Proposition 9.6 Let $(\mathbf{M}, \mathbf{M}')$ be a parallel pair. If X is a timeslice in $\overline{\mathbf{M}}$ then X is a partial cut.

Proof Since precedence in the localization implies precedence in the transitive model, a timeslice in the latter is at least a partial timeslice in the former. Apply Proposition 9.5. □

In general, timeslices from the global model may not be full cuts. We follow the literature in introducing a term to describe when they are.

Definition 9.7 Let $(\mathbf{M}, \mathbf{M}')$ be a parallel pair. A *consistent cut* is a cut that is also an $\overline{\mathbf{M}}$-timeslice.

Chapter 12 will consider the properties of a parallel pair necessary to ensure that all timeslices are consistent cuts.

Definition 9.7 differs from the order-theoretic meaning of the term “consistent cut.” In order theory, a consistent cut is a graph-theoretic cut whose members share a common upper bound (i.e., a common descendant).

9.2. Set Precedence and Operations

Precedence The edges in a computation graph specify precedence on events. We can use these edges to induce a precedence relation on sets of events.

Definition 9.8 Suppose X and Y are sets of events in a graph γ. We say that X precedes Y in γ

$$X \quad \prec_\gamma \quad Y$$

when an X event precedes a Y event in γ, and all $A, B \in X \cup Y$ satisfy

$$A \longrightarrow B \text{ in } \gamma \quad \Longrightarrow \quad A \in X \quad \wedge \quad B \in Y$$

To determine the γ-precedence of two sets X and Y, we build a subgraph of γ by taking the events from these sets and drawing any relevant γ edges. Set X precedes set Y when all edges go from an X event to a Y event, and at least one edge exists.

Relative Minima and Relative Maxima We can also use edges to transform sets of events.

Definition 9.9 Suppose X is a set of events in some graph γ. Define $\min_\gamma(X)$ to be the set of relative minima:

$$\min_\gamma(X) \quad = \quad \{A \in X \quad : \quad B \in X \implies B \not\longrightarrow A \text{ in } \gamma\}$$

Define $\max_\gamma(X)$ to be the set of relative maxima.

We omit the subscript when the graph is understood.

Precedence and Relative Min/Max Clearly, the relative minima of $X \cup Y$ can follow neither X nor Y; similarly the relative maxima can precede neither X nor Y. With some stronger conditions on X and Y, we can establish that the relative minima and relative maxima are actually the tightest bounds on X and Y. A lattice structure emerges.

9.3. Lattices

In this section, we show that the set of timeslices in a transitive graph forms a lattice. The precedence relation and minima and maxima operations from Section 9.2 give the appropriate structure.

Section 9.3.1 gives some basic definitions. Section 9.3.2 proves the main result. Section 9.3.3 considers the implications for vectors, cuts, and consistent cuts.

9.3.1. Definitions

First, we recall some standard definitions.

Definition 9.10 Suppose W is a nonempty ordered set, and y, z are two elements of W. Element $x \in W$ is an *upper bound* of y and z if, in the order, x follows both y and z. Element x is a *least upper bound* of y and z if x precedes any other upper bound x' of y and z. Define *lower bound* and *greatest lower bound* symmetrically.

A *lattice* is a nonempty ordered set such that any two elements in the set have both a least upper bound and a greatest lower bound in the set.

The standard term for "least upper bound" in a lattice is *join*; the standard term for "greatest lower bound" is *meet.*

9.3.2. Timeslices

Proving that timeslices from a transitive graph form a lattice is tricky. Intuition suggests that Section 9.2 should provide the tools: $\prec$ precedence provides the order, min$(X \cup Y)$ should be $X \sqcap Y$ and max$(X \cup Y)$ should be $X \sqcup Y$.

Intuition fails, because not all timeslices are consistent cuts. While $\prec_{\overline{\gamma}}$ establishes a partial order on $\overline{\gamma}$-timeslices, the relative minima and relative maxima operations may only produce *proper subsets* of timeslices. These mutually concurrent sets extend to timeslices—but showing that there exists unique extrema in the set of these extensions is not trivial.

We prepare for the main result with a series of lemmas.

Comparing Timeslices If two timeslices are different, then some pair of entries must be ordered:

Lemma 9.11 Let X and Y be timeslices in a graph. If $X \neq Y$ then some $A \in X$ and $B \in Y$ satisfy $A \longrightarrow B$ or $B \longrightarrow A$.

Proof Since $X \neq Y$ and timeslices are maximal, we can choose $A \in (X \setminus Y)$. If A were concurrent with everyone in Y, then A would be in Y—hence such a B must exist. □

Since timeslices are mutually concurrent sets, we can strengthen the $\prec_{\overline{\gamma}}$ property:

Lemma 9.12 If timeslices X and Y in a graph satisfy $X \prec Y$, then for any A and B from $X \cup Y$,

$$A \longrightarrow B \quad \Longrightarrow \quad A \in (X \setminus Y) \;\wedge\; B \in (Y \setminus X)$$

Proof Timeslices cannot contain events that precede each other. □

Partial Order In a *transitive* graph, the $\prec$ relation forms a partial order on timeslices:

Lemma 9.13 The $\prec$ relation is a partial order on the set of timeslices in a transitive graph.

Proof We establish the three properties.

1. *The $\prec$ relation is antisymmetric.* Let X and Y be timeslices. If $X \prec Y$ then there exists $A \in X$ and $B \in Y$ with $A \longrightarrow B$. If $Y \prec X$ as well, then $A, B \in X \cap Y$. Hence neither could be timeslices.
2. *The $\prec$ relation is irreflexive.* If $X \prec X$ then some $A, B \in X$ satisfy $A \longrightarrow B$.
3. *The $\prec$ relation is transitive.* Let timeslices X, Y, Z satisfy $X \prec Y \prec Z$. Suppose $A, B \in (X \cup Z)$ satisfy

   $$A \quad \longrightarrow \quad B$$

   but $A \notin X$ and $B \notin Z$. Since X and Z are timeslices, the events cannot lie together, so $A \in Z$ and $B \in X$. If $C \underline{\longrightarrow} A$ for some $C \in Y$, then $C \underline{\longrightarrow} B$, contradicting $X \prec Y$. But $A \underline{\longrightarrow} C$ for some $C \in Y$ violates $Y \prec Z$. If $A \longleftrightarrow A$ then A could not be part of timeslice Z. Hence $\{A\} \cup Y$ is mutually concurrent, so Y could not be a timeslice. Thus all $Z \cup X$ edges go from X to Z.

   Suppose no $A \in X$ and $B \in Z$ satisfy $A \longrightarrow B$. Then every $A \in X$ either appears in Z or is mutually concurrent with everyone in Z—in which case it appears in Z. Hence $X = Z$. Apply the antisymmetry case.

□

The Maxima and Minima Operations For timeslices in a transitive graph, the relative maxima and relative minima operations produce mutually concurrent sets:

Lemma 9.14 If X and Y are timeslices in a transitive graph, then min$(X \cup Y)$ and max$(X \cup Y)$ are partial timeslices.

Proof Let $Z =$ min$(X \cup Y)$. If Z is not a partial timeslice, then some $A, B \in Z$ satisfy $A \longrightarrow B$. Without loss of generality, assume $A \in X$ and $B \in Y$. However, $B \in Y \cap Z$ implies $B \longrightarrow C$ for some $C \in X$, hence $A \longrightarrow C$, so X could not be a timeslice.

The case for max is similar. □

These sets characterize the set of timeslice bounds:

Lemma 9.15 Suppose X and Y are timeslices in a transitive graph. Timeslice Z is a lower bound of X and Y iff no event in min$(X \cup Y)$ precedes any event in Z; timeslice Z is an upper bound of X and Y iff no event in Z precedes any event in max$(X \cup Y)$.

Proof Let $M =$ min$(X \cup Y)$. Suppose $A \in M$ precedes $B \in Z$. Without loss of generality, assume $A \in X$. Then Z can neither precede nor equal X. Suppose no $A \in M$ precedes anyone in Z. Then no one in $X \cup Y$ can precede anyone in Z. If $Z \neq X$, then Lemma 9.11 implies that someone in Z must precede someone in X. Hence Z is a lower bound of X and Y. The case for max is symmetric. □

Extremal Extensions Timeslices by definition contain only acyclic events. Hence the set of timeslices that a given mutually concurrent set extends to has a unique maximum and a unique minimum—because directed acyclic graphs have maxima and minima.

Lemma 9.16 Suppose X is a partial timeslice in a transitive graph. There exists a unique minimum timeslice and a unique maximum timeslice containing X.

Proof Let W be the set of all acyclic events that are concurrent with every member of X. Since these events are acyclic, our transitive graph induces a transitive acyclic subgraph γ on W. Define $X' =$ min$_\gamma(W)$. Define $\mathcal{W}$ to be the set of timeslices from γ. We make some assertions:

- $X' \in \mathcal{W}$. By definition, the γ minima set is maximal and mutually concurrent.
- $\{X \cup Z \ : \ Z \in W\}$ is the set of timeslices containing X. A set $X \cup Z$ is mutually concurrent and cannot be extended; the non-X elements of a timeslice containing X must be a timeslice in γ.

- For any $Z \in \mathcal{W}$, if $Z \neq X'$ then $(X \cup X') \prec (X \cup Z)$. Otherwise, X' could not have been the minima.

Thus a unique minimum $X \cup X'$ exists, and similarly a unique maximum exists. □

The Timeslice Lattice Hence, timeslices from a transitive graph form a lattice. The $\prec$ relation gives a partial order, and max and min give partial timeslices that extend to the appropriate timeslices.

> **Theorem 9.17** *(Timeslice Lattice)* If nonempty, the set of timeslices in a transitive graph forms a lattice.

Proof By Lemma 9.13, the $\prec$ relation forms a partial order.

Let X and Y be timeslices. By Lemma 9.14, min$(X \cup Y)$ is a partial timeslice. By Lemma 9.16, there exists a unique maximum timeslice Z containing min$(X \cup Y)$. By Lemma 9.15, Z is a lower bound of X and Y. Suppose timeslice Z' is a different lower bound that is not dominated by Z. Then some $A \in Z$ precedes some $B \in Z'$. By Lemma 9.15, B cannot follow anyone in min$(X \cup Y)$. But if B precedes someone in min$(X \cup Y)$, then Z is not a timeslice. Hence B is concurrent with min$(X \cup Y)$, and by Lemma 9.16 must precede or equal someone in Z.

Thus timeslice Z is the greatest lower bound of X and Y; similarly max$(X \cup Y)$ extends to a least upper bound. □

9.3.3. Vectors, Cuts, and Consistent Cuts

Vectors A direct consequence of Section 9.3.2 is that vectors form a lattice.

> **Theorem 9.18** Suppose parallel pair $(\mathbf{M}, \mathbf{M}')$ acts on graph α. Let $\gamma = \mathbf{M}(\alpha)$ and $\beta = \mathbf{M}'(\alpha)$. The set of vectors in γ, if nonempty, forms a lattice.

Proof By the Timeslice Lattice Theorem (Theorem 9.17), the $\overline{\beta}$-timeslices form a lattice; by Proposition 9.4, the γ-vectors are a bijective image of the $\overline{\beta}$-timeslices. □

In fact, this is easily established without Section 9.3.2—in particular, meet and join coincide exactly with the relative minima and maxima.

$$\begin{aligned} X \sqcap Y &= \text{min}_{\overline{\beta}}(X \cup Y) \\ X \sqcup Y &= \text{max}_{\overline{\beta}}(X \cup Y) \end{aligned}$$

We will informally identify vectors with their multicomponent images, and will consequently apply $\prec_{\overline{\beta}}$, $\sqcap$, and $\sqcup$ directly to vectors.

Cuts The set of cuts does not always form a lattice. Since the localization is not transitive, the Timeslice Lattice Theorem (Theorem 9.17) does not apply. Figure 9.1 sketches a counterexample.

If the only multiple-process events were extrema, (such as in POT and LINEAR), then the case of cuts would reduce to that of vectors.

Consistent Cuts However, the set of consistent cuts does form a lattice. When applied to consistent cuts, the cut operations yield consistent cuts. (Mutual concurrency is easy to establish; maximality would be easy if events only occurred at single processes.)

Lemma 9.19 Suppose multiprocess pair $(\mathbf{M}, \mathbf{M}')$ with localization $\mathbf{L}$ acts on graph α. and $\gamma_L = \mathbf{L}(\alpha)$.

If X, Y are consistent cuts, then both $\min_{\gamma_L}(X \cup Y)$ and $\max_{\gamma_L}(X \cup Y)$ are consistent cuts.

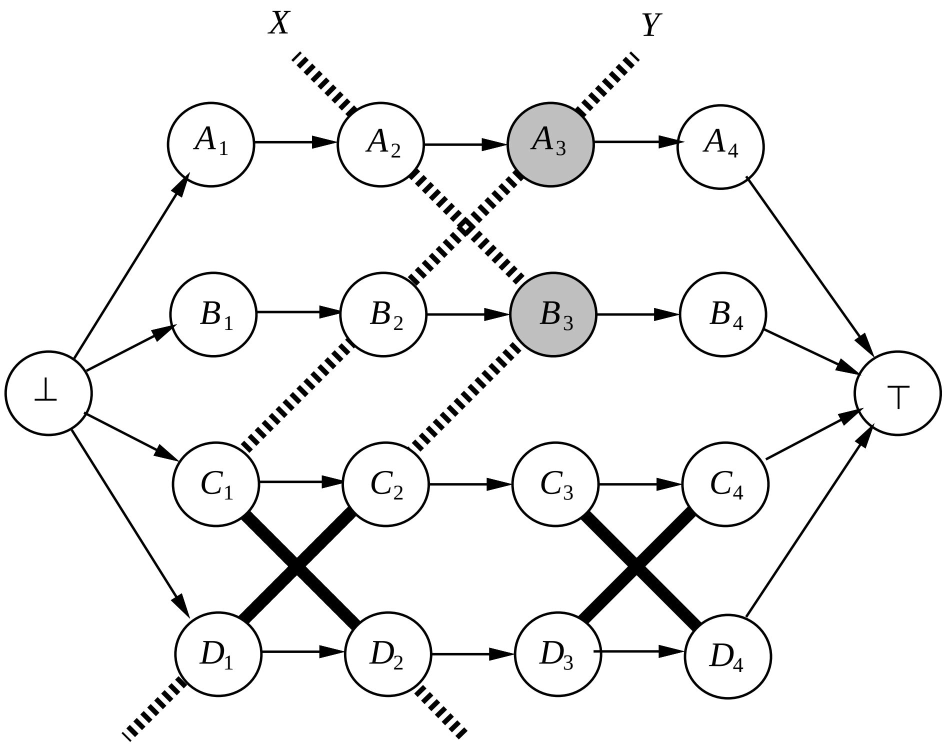

Figure 9.1 Cuts in a parallel pair may not form a lattice. Suppose that the process events linked with bold lines are merged in the global model. Then cuts $X = \{A_2, B_3, (C_2D_1)\}$ and $Y = \{A_3, B_2, (C_1D_2)\}$ have only a partial cut $\{A_3, B_3\}$ as their relative maxima. This partial cut extends to two different dominating cuts—add (C_4D_3) or (C_3D_4). Each of these extensions is concurrent in the set-precedence order. Hence cuts X and Y have no *cut* as a least upper bound.

Proof Let $\gamma = \mathbf{M}(\alpha)$.

Suppose $\pi_p X \longrightarrow \pi_p Y$ in γ_L and $\pi_q Y \longrightarrow \pi_q X$ in γ_L. Then $\pi_p X \not\longrightarrow \pi_q Y$ in $\overline{\gamma}$, for otherwise X is not consistent. Similarly $\pi_q Y \not\longrightarrow \pi_p X$ in $\overline{\gamma}$. Hence $\min_{\gamma_L}(X \cup Y)$ is mutually concurrent.

Since all events in X and Y are acyclic in γ, they must be acyclic in γ_L, so any event in $X \cup Y$ follows or equals someone in the relative minima. Suppose some process p were not represented in $\min_{\gamma_L}(X \cup Y)$. If distinct, the p entries of X and Y are ordered by γ_L; hence without loss of generality suppose that $\pi_p X \Longrightarrow \pi_p Y$ in γ_L. Since $\pi_p X$ is not a minima, some A from $X \cup Y$ must satisfy $A \longrightarrow \pi_p X$ in γ_L. But this event A precedes both $\pi_p X$ and $\pi_p Y$ in $\overline{\gamma}$—hence at least one of X, Y must not have been a consistent cut.

The case for $\max_{\gamma_L}$ is similar. □

On consistent cuts, the cut operations (minima and maxima in the localization) coincide with the timeslice operations (minima and maxima in the transitive global graph):

Lemma 9.20 Suppose parallel pair $(\mathbf{M}, \mathbf{M}')$ with localization $\mathbf{L}$ acts on graph α. Let $\gamma = \mathbf{M}(\alpha)$ and and $\gamma_L = \mathbf{L}(\alpha)$. If X and Y are consistent cuts, then

$$\begin{aligned} \min_{\overline{\gamma}}(X \cup Y) &= \min_{\gamma_L}(X \cup Y) \\ \max_{\overline{\gamma}}(X \cup Y) &= \max_{\gamma_L}(X \cup Y) \end{aligned}$$

Proof Since $\gamma_L \subset \overline{\gamma}$ and removing edges cannot cause an event to stop being minimal:

$$\begin{aligned} \min_{\overline{\gamma}}(X \cup Y) &\subset \min_{\gamma_L}(X \cup Y) \\ \max_{\overline{\gamma}}(X \cup Y) &\subset \max_{\gamma_L}(X \cup Y) \end{aligned}$$

Suppose $A \in (\min_{\gamma_L}(X \cup Y))$ but $A \notin (\min_{\overline{\gamma}}(X \cup Y))$. Then there exists a $B \in (X \cup Y)$ such that $B \longrightarrow A$ in $\overline{\gamma}$. Without loss of generality, suppose $A \in X$ and $B \in Y$, and $A|_{\overline{p}}$ exists. If $\pi_p Y \longrightarrow A$ in γ_L, then A could not have been minimal in γ_L. Hence $A \Longrightarrow \pi_p Y$ in γ_L and hence in $\overline{\gamma}$—in which case Y could not be mutually concurrent. The case for join is similar. □

Hence consistent cuts form a lattice.

Theorem 9.21 Suppose $(\mathbf{M}, \mathbf{M}')$ is a parallel pair. If nonempty, the set of consistent cuts in an $\mathbf{M}$ graph forms a lattice.

Proof By the Timeslice Lattice Theorem (Theorem 9.17), the set of timeslices form a lattice. Consistent cuts are a nonempty subset. By Lemma 9.19 and Lemma 9.20, this subset is closed under meet and join. □

As with vectors, the minima and maxima operations here are exactly meet and join. This only makes sense, as on consistent cuts, the operations coincide not only with the timeslice operations, but also with the vector operations. Precedence coincides as well.

Meets and joins also preserve event membership. Hence the set of consistent cuts containing some specified event set is a lattice.

Theorem 9.22 Suppose $(\mathbf{M}, \mathbf{M}')$ is a parallel pair, and a set of events X from an $\mathbf{M}$ graph is contained in at least one consistent cut. Then the set of all consistent cuts containing X forms a lattice.

Proof This follows directly from Theorem 9.21, and the above observation. □

(Part III)

Chapter 10

Timestamp Vectors and Rollback Vectors

As much previous research has noted, vectors play a fundamental role in representing distributed time structures. This chapter explores this role in terms of our time theory.

For each event from a graph from a parallel model, we introduce two special structures: the *timestamp vector*, containing the maximal events at each process that precede or equal the event, and the *rollback vector* containing the minimal.

Section 10.1 develops the definitions of these vectors, and Section 10.2 and Section 10.3 explore some properties of them (including their use as clocks).

10.1. The Definition

10.1.1. The Attempt

First, we present the definitions.

Definition 10.1 Suppose parallel pair $(\mathbf{M}, \mathbf{M}')$ acts on graph α. Let $\gamma = \mathbf{M}(\alpha)$ and $\beta = \mathbf{M}'(\alpha)$. For event A in γ, define its *timestamp vector* $\mathbf{V}(\gamma, \mathbf{M}, \mathbf{M}', A)$ to be the vector whose process p entry is the event B such that:

- $B \Longrightarrow A$ in $\overline{\gamma}$
- $B|_{\overline{p}}$ exists
- If $C \Longrightarrow A$ in $\overline{\gamma}$ and $C|_{\overline{p}}$ exists, then $C|_{\overline{p}} \Longrightarrow B|_{\overline{p}}$ in $\overline{\beta}$.

Define its *rollback vector* $\mathbf{R}(\gamma, \mathbf{M}, \mathbf{M}', A)$ symmetrically: the vector whose process p entry is the event B such that:

- $A \Longrightarrow B$ in $\overline{\gamma}$

- $B|_{\overline{p}}$ exists
- If $A \Longrightarrow C$ in $\overline{\gamma}$ and $C|_{\overline{p}}$ exists, then $B|_{\overline{p}} \Longrightarrow C|_{\overline{p}}$ in $\overline{\beta}$.

Usually the graph and the parallel pair are understood when we deal with these vectors. In these situations, we will condense the awkward parameter list and write simply $\mathbf{V}(A)$ and $\mathbf{R}(A)$, respectively.

An easy consequence of this definition is that an event precedes or equals everything in its rollback vector, and follows or equals everything in its timestamp vector.

Proposition 10.2 Suppose parallel pair $(\mathbf{M}, \mathbf{M}')$ acts on graph α. Let A be an event from $\gamma = \mathbf{M}(\alpha)$ and let $p \in$ **PROC-NAMES**. If $\pi_p\,\mathbf{V}(A)$ is defined, then $\pi_p\,\mathbf{V}(A) \Longrightarrow A$ in $\overline{\gamma}$. Similarly if $\pi_p\,\mathbf{R}(A)$ is defined, then $A \Longrightarrow \pi_p\,\mathbf{R}(A)$ in $\overline{\gamma}$.

Proof This assertion follows directly from Definition 10.1. $\square$

10.1.2. Unique Entries

Because the process components in a parallel pair are total orders, the process image of a vector entry is unique. That is, for event A from an $\overline{\mathbf{M}}$ graph and process $p \in$ **PROC-NAMES**, not more than one event from process p meets the criteria for the process p entry of $\mathbf{V}(A)$ and $\mathbf{R}(A)$. Because the process components are indeed components of $\mathbf{M}$, the vector entry itself—as an event in $\mathbf{M}$—is also unique.

10.1.3. Missing Entries

However, a problem with Definition 10.1 is that not all entries of these vectors are always defined. The number of qualifying events is never more than one—but it might be zero.

For parallel pair $(\mathbf{M}, \mathbf{M}')$, the process p entry of the $\mathbf{V}(A)$ vector is defined iff some B satisfying $B \Longrightarrow A$ in $\overline{\mathbf{M}}$ represents something at process p. A simple condition guarantees this property:

Proposition 10.3 Let $(\mathbf{M}, \mathbf{M}')$ be a parallel pair. If $\mathbf{M}$ is transitively bounded, then all entries in all timestamp and rollback vectors are defined.

Proof From Proposition 8.10, the global extrema represent the local extrema; hence the past of any event A touches each process component in at least one spot, as does the future. $\square$

Conveniently, the POT model is transitively bounded.

10.2. Properties Despite Missing Entries

We can prove a number of properties about timestamp and rollback vectors, even if we allow for vectors with undefined entries.

First, timestamp vectors and rollback vectors mark the influence horizons of events:

Theorem 10.4 Suppose parallel pair $(\mathbf{M}, \mathbf{M}')$ acts on graph α. Let A and B be events from $\gamma = \mathbf{M}(\alpha)$. Then

$$A \rightleftarrows B \text{ in } \overline{\gamma} \iff A \in \lceil \mathbf{V}(B) \rceil \iff B \in \lfloor \mathbf{R}(A) \rfloor$$

Proof Let $\mathbf{L}$ be the localization of $(\mathbf{M}, \mathbf{M}')$; let $\beta = \mathbf{M}'(\alpha)$ and $\gamma_L = \mathbf{L}(\alpha)$. Let $V = \mathbf{V}(B)$ and let $R = \mathbf{R}(A)$.

Suppose $A \rightleftarrows B \in \overline{\gamma}$. A must represent at least one process, so let that process be p. Then B has an ancestor representing part of the p component of $\overline{\beta}$. Hence by Definition 10.1, $(\pi_p V)|_{\overline{p}}$ exists, and $A|_{\overline{p}} \rightleftarrows (\pi_p V)|_{\overline{p}}$ in $\overline{\beta}$. Hence, by Definition 8.8, $A \in \lceil V \rceil$.

$A \in \lceil V \rceil$ implies there exists some p such that $A|_{\overline{p}}$ and $(\pi_p V)|_{\overline{p}}$ both exist and satisfy $A|_{\overline{p}} \rightleftarrows (\pi_p V)|_{\overline{p}}$ in $\overline{\beta}$. Hence $A \rightleftarrows \pi_p V$ in $\overline{\gamma}$, and by Proposition 10.2 and transitivity, $A \rightleftarrows B$ in $\overline{\gamma}$.

The case for the rollback vector is symmetric. □

The relation of an event at process p to the process p entries of its vectors satisfies a simple identity—an identity that is trivial for acyclic models.

Lemma 10.5 Suppose parallel pair $(\mathbf{M}, \mathbf{M}')$ acts on graph α. Let A be an event from $\gamma = \mathbf{M}(\alpha)$; let $\beta = \mathbf{M}'(\alpha)$ and $p \in$ **PROC-NAMES**. If $A|_{\overline{p}}$ exists in $\overline{\beta}$, then the process p entries of $\mathbf{R}(A)$ and $\mathbf{V}(A)$ exist in $\overline{\gamma}$, and their images in $\overline{\beta}$ bracket the image of A:

$$(\pi_p \mathbf{R}(A))|_{\overline{p}} \rightleftarrows A|_{\overline{p}} \rightleftarrows (\pi_p \mathbf{V}(A))|_{\overline{p}}$$

Proof A precedes or equals itself, so it must precede or equal its maximal improper ancestor at p. Similarly A must follow or equal its minimal improper descendant. □

Timestamp and rollback vectors are unique, up to $\mathbf{M}$ cycles:

Lemma 10.6 Suppose parallel pair $(\mathbf{M}, \mathbf{M}')$ acts on graph α. Let A and B be events from graph $\gamma = \mathbf{M}(\alpha)$, and let $\beta = \mathbf{M}'(\alpha)$. Then

$$A \leftrightarrows B \iff \mathbf{V}(A) = \mathbf{V}(B) \iff \mathbf{R}(A) = \mathbf{R}(B)$$

Proof Clearly if $A \overset{\longleftarrow}{\longrightarrow} B$ then the other two statements hold.

Suppose $\mathbf{V}(A) = \mathbf{V}(B)$. If $A|_{\overline{p}}$ exists, then the p entry of the A vector exists. Hence so does the p entry of the B vector, and these entries satisfy:

$$\pi_p \mathbf{V}(A) \quad = \quad \pi_p \mathbf{V}(B)$$

Lemma 10.5 establishes that

$$A|_{\overline{p}} \quad \Longrightarrow \quad (\pi_p \mathbf{V}(A))|_{\overline{p}}$$

Hence $A \Longrightarrow \pi_p \mathbf{V}(A)$. But Proposition 10.2 gives $\pi_p \mathbf{V}(B) \Longrightarrow B$. Hence $A \Longrightarrow B$, and similarly $B \Longrightarrow A$.

The case for rollback vectors is symmetric. □

10.3. Vector Clocks

Timestamp vectors have a natural use as $\mathbf{M}$-clocks for the transitive global model. If a process *timestamps* each event with its timestamp vector, then a simple comparison determines the $\overline{\mathbf{M}}$ relation of two events.[1] (Doing this comparison requires having all entries defined—which we have from Proposition 10.3.) Further, for well-behaved models like POT, calculating the timestamp vector for each event is very simple.

Rollback vectors describe the spread of influence of an event in a system. If A were instantaneously *rolled back*, the vector $\mathbf{R}(A)$ indicates the frontier of what needs to be undone. (But rollback vectors also function as clocks, although not necessarily very practical ones.)

The key result is that vector precedence (from Theorem 9.18) follows event precedence.

Theorem 10.7 *(Vector Clocks)* Suppose transitively bounded parallel pair $(\mathbf{M}, \mathbf{M}')$ acts on graph α. Let $\gamma = \mathbf{M}(\alpha)$.

For any two events A, B:

$$\mathbf{V}(A) \prec \mathbf{V}(B) \iff \mathbf{R}(A) \prec \mathbf{R}(B)$$

$$\iff (A \longrightarrow B \text{ in } \overline{\gamma} \ \wedge \ B \not\longrightarrow A \text{ in } \overline{\gamma})$$

Proof Suppose $A \longrightarrow B$ but $B \not\longrightarrow A$. Then Proposition 10.2 and transitivity give each $\pi_p \mathbf{V}(A) \longrightarrow B$. The definition of timestamp vector then gives

$$\pi_p \mathbf{V}(A) \Longrightarrow \pi_p \mathbf{V}(B)$$

[1] The vector clocks in the literature [StYe85, Fi88, Jo89, Ma89, JoZw90, SiKs90, Sm91, PeKe93, ReGo93, SmTy93] are special cases of this result.

for each p. Hence $\mathbf{V}(A) \preceq \mathbf{V}(B)$; $B \not\rightarrowtail A$ and Lemma 10.6 make this inequality strict.

Conversely, suppose $\mathbf{V}(A) \prec \mathbf{V}(B)$ and $A|_{\overline{p}}$ exists. Then Lemma 10.5 gives $A \rightleftarrows \pi_p\,\mathbf{V}(A)$. By hypothesis $\pi_p\,\mathbf{V}(A) \rightleftarrows \pi_p\,\mathbf{V}(B)$. Proposition 10.2 and transitivity then give $A \rightleftarrows B$. But Lemma 10.6 and the inequality of the vectors forces $A \neq B$ and $B \not\rightarrowtail A$.

The case for rollback vectors is symmetric. $\square$

Like much of the work here, this theorem applies to more general models than POT. For example, the theorem does not require that $\mathbf{M}$ be acyclic.

(Part III)

Chapter 11

Real Simultaneity

A timeslice from a computation graph is a set of logically concurrent events. Does this logical simultaneity imply real simultaneity? That is, how do timeslices correspond to the real instantaneous system states in the underlying physical computation? Clearly, a necessary condition is that the graph be produced by a concrete, grounding generator $\mathbf{M}$. This way, the underlying computation really exists and the components of a timeslice really do correspond to parts of this computation.

In this section we begin exploring this relationship for the parallel models we've constructed.

Section 11.1 forrmally defines our usage of the term *global state*: in physical computations, the system state at some instant; in computation graphs, the representation of a global state in *some* computation mapping to that graph.

Section 11.2 explores the relationship between timeslices and global states for the LINEAR model. However, Section 11.3 demonstrates how the partial order model POT fails to give the desired relationships.

11.1. Global States

Global States in Computations The term "global state" admits two interpretations: a static one (what's the local state everywhere right now?) and a dynamic one (what's everyone doing right now?).

Our work allows both interpretations. "Right now" presumably denotes some instant of real time—for computation $T = ((t_0, s_0), ..., (t_k, s_k))$, some instant t in the closed interval $[t_0, t_k]$. In terms of computation graphs, the most accurate picture we can obtain of the system at time t is the set of atoms for time t in the ground-level graph.

Definition 11.1 A *global state* in a trace $T = ((t_0, s_0), ..., (t_k, s_k))$ is the set of atoms in the ground-level graph of T representing system activity at some time $t \in [t_0, t_k]$.

For example, in Figure 11.1, the highlighted node and edge in the ground-level computation graph α constitute a global state, corresponding to real time t.

Global States in Modeled Computations A set $\mathcal{G}$ together with a grounding generator induces a map from atoms in $\mathcal{G}$ back to atoms in ground-level graphs. This induced map gives an easy way to define when sets of $\mathcal{G}$ atoms describe a true simultaneous moment in an underlying trace.

Definition 11.2 Let $\mathbf{M}$ be a grounding generator of set $\mathcal{G}$ and let $\beta \in \mathcal{G}$. A set S of atoms in β *represents a global state* when for some $\alpha \in \mathbf{M}^{-1}(\beta)$:

- $\langle\, \mathbf{M},\ \alpha \,\rangle(S)$ contains a global state in α
- but no proper subset of S does.

For example, in Figure 11.1, the highlighted nodes in the LINEAR graph β minimally represent the highlighted global state in the ground-level graph α.

Definition 11.2 includes two subtleties deserving special emphasis.

Global states do not necessarily occur A global state in a computation graph corresponds to a real global state in *some* physical computation that maps to that graph—not necessarily the physical computation really in progress.

Global states depend on the model Talking about how an event set in a graph corresponds to reality requires talking about how the graph corresponds to reality. Thus talking about global states requires specifying (at least implicitly) a grounding generator for that graph.

Whether we allow static global states or dynamic global states depends on whether our model allows passive events like *idle*, and whether we are exploring states as sets of atoms or strictly sets of events.

A Schema for Examining Models If timeslices are doing their job, then they should describe exactly the interesting global states in the graphs a model produces. Considering this issue yields some questions:

- What are the interesting global states?
- Do timeslices minimally represent these states?
- Do any other event sets minimally represent these states?
- Are there any of these states that cannot be minimally represented by event sets?

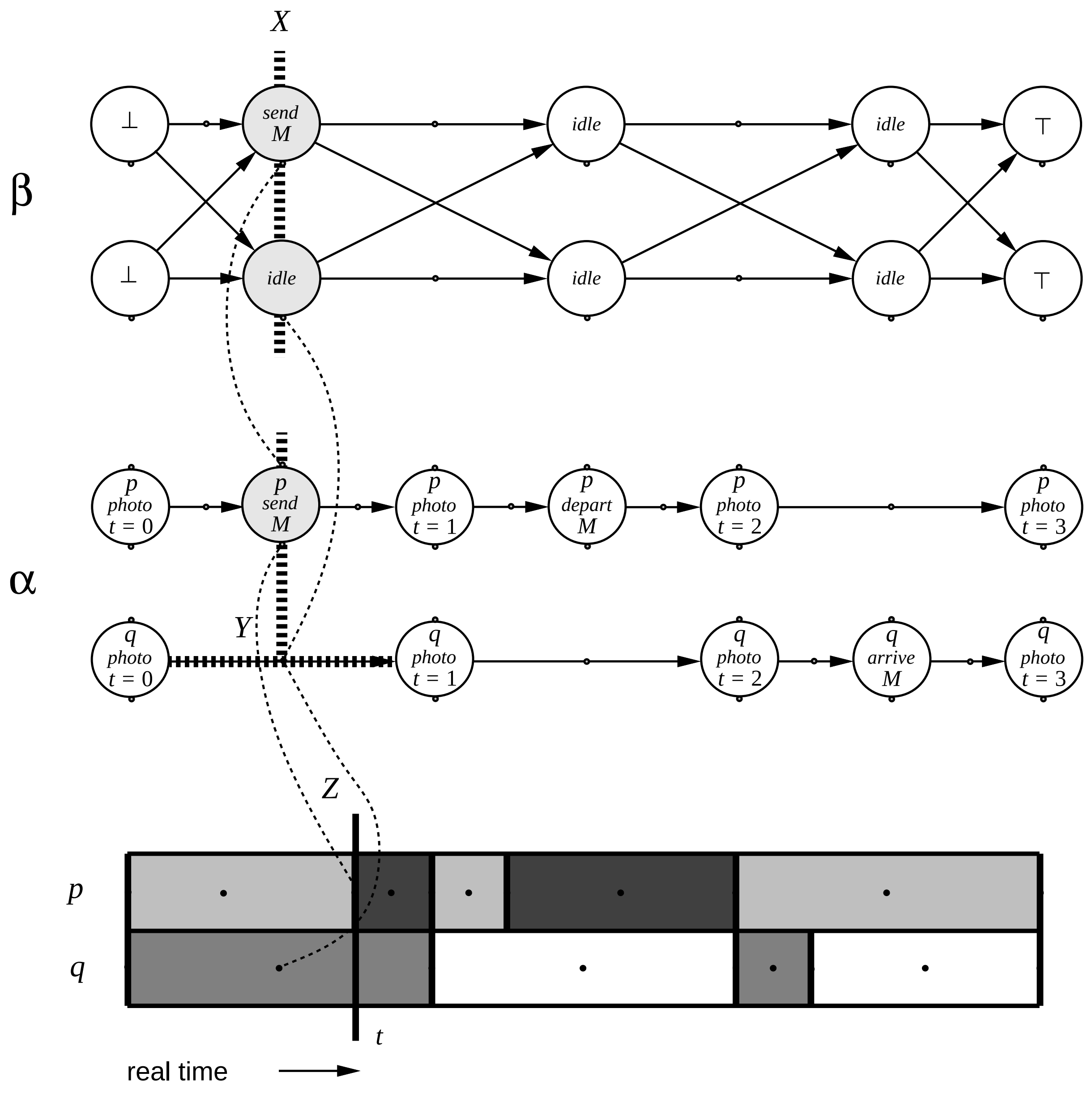

Figure 11.1 Global states arise from real simultaneity; event sets that minimally represent global states arise from the representational aspect of time models. The space-time region on the bottom describes some computation, whose ground level computation graph is α. Graph β is the image of α under the model LINEAR. Set Z is the region of space-time corresponding to the real time t; the edge and event of Y comprise the global state in α corresponding to Z. We follow the representation lines (dashed) to obtain X, the event set in β that minimally represents this global state.

If the model in question happens to be parallel, we have yet another question:

- Is each timeslice a consistent cut?

A global state is a simultaneous moment in a computation, perceived through the granularity of a ground-level computation graph. In this paper, we use the crude criteria that a global state is "interesting" when any part of its action is part of a "thing that happens" in a model. That is, if a model creates an event that represents part of the global state X, then X is relevant to that model. So the model should be be able to talk about it.

This criteria answers the first question from the above list. We can easily answer the last question: if a timeslice is not a consistent cut, then it represents no activity at some process p—hence it cannot represent a global state.

In Section 11.2 and Section 11.3 we explore whether the timeslices from LINEAR and POT describe exactly these interesting global states. The remaining questions from the above list form a schema for this exploration.

11.2. Timeslices and Global States in Linear Time

The case for LINEAR is extremely straightforward.

Theorem 11.3 Suppose LINEAR generates graph γ. If a set of events X in γ is a timeslice in $\overline{\gamma}$, then X minimally represents a global state.

Proof Consider a ground-level graph that is a pre-image of γ. Timeslices exactly represent either the initial global state, the final global state, or the global state between photos when something happens. (The definition of trace ensures that when two actions happen between photos, they happen simultaneously). $\square$

Theorem 11.4 Suppose LINEAR generates graph γ. If a set of events X from γ minimally represents a global state, then X is a timeslice in $\overline{\gamma}$.

Proof This follows from two facts. First, a strict subset of a timeslice cannot describe all processes. Second, if $A \longrightarrow B$ in $\overline{\text{LINEAR}}$ then the space-time regions of A and B have disjoint time ranges. So, such an X must touch all processes and be mutually concurrent. $\square$

Theorem 11.5 Let X be a global state from ground-level graph α. If an event in LINEAR(α) represents any part of X, then a timeslice in $\overline{\text{LINEAR}}(\alpha)$ minimally represents X.

Proof This fact is clear from the construction of $\overline{\text{LINEAR}}$ graphs. We create events in rows, one for each process, in accordance with the time periods. The timeslices are the rows. □

Theorem 11.6 In any graph produced by $\overline{\text{LINEAR}}$, each timeslice is a consistent cut.

Proof This follows directly from the proof of Theorem 11.5 above. □

11.3. Timeslices and Global States in Partial Order Time

Section 11.1 provides a schema to establish that timeslices express simultaneity in parallel models. This schema fails for POT.

Failure Consider an execution where process p sends a message to process q. Process q receives this message and returns a response to process p, who then receives the response. In the $\overline{\text{POT}}$ graph of this execution (see Figure 11.2), the singleton consisting of process q's *send* is a timeslice. But this timeslice cannot represent a global state because it says nothing about process p; no event set minimally represents the global state containing process q's *send*.

Limited Success We can establish some limited results. Mutually concurrent events in POT represent part of a global state: in some underlying computation, mutually concurrent events are simultaneous.

Lemma 11.7 Let γ be the POT image of a ground-level graph. If set of events X is mutually concurrent in $\overline{\gamma}$ then X minimally represents a subset of a global state.

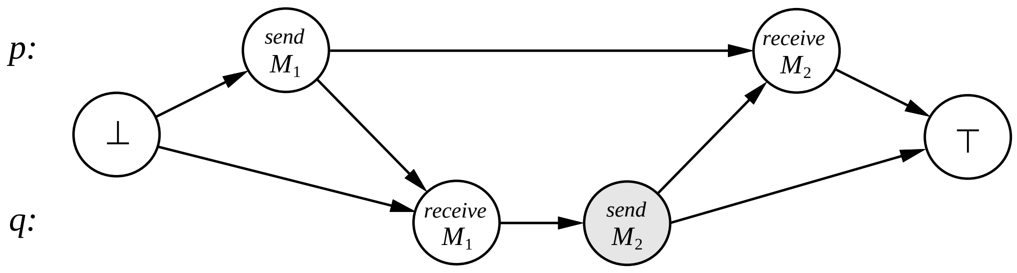

Figure 11.2 The shaded event is a timeslice in $\overline{\text{POT}}$; however, this timeslice does not represent a global state, as it says nothing about activity at process p.

Proof Obtain a ground-level graph in POT$^{-1}(\gamma)$ as follows. First, assign integers to the nodes in γ by setting each element of X to 0, setting each A following X to be one greater than the maximum values of its ancestors in POT, and each A before X to one less than the minimum value of its successors. Since POT is acyclic, this operation is well-defined. Secondly, add j to the value of the nodes, where $-j$ is the value on $\perp$. Let k be the resulting value on $\top$. By Axiom 3.1, a computation exists with $\perp$ photos at $t = 0$, $\top$ photos as $t = k$, and (for remaining nodes *foo*, marked with integer v) the appropriate *foo* actions occurring in the time interval $(v, v + 1)$.

Construct the ground-level graph for this computation. The events in X represent a subset of the ground-level graph events for $(j, j + 1)$. □

Thus, the timeslices that are consistent cuts in fact represent global states:

Theorem 11.8 Consistent cuts in $\overline{\text{POT}}$ minimally represent global states.

Proof Let γ be a POT graph, and let X be a consistent cut from $\overline{\gamma}$. By definition, X describes activity at every process. By Lemma 11.7, there exists a ground-level graph in POT$^{-1}(\gamma)$ in which X minimally represents a set of simultaneous events. Since this set must span all of space, it must be a global state. □

(Part III)

Chapter 12

View-Completeness

In Section 11.3, we saw that a very simple parallel pair will produce timeslices that neither represent global states nor derive from consistent cuts.

Consider again the POT graph from Figure 11.2. The graph fails because process p goes directly from the *send* event to the *receive* event. The p-*send* precedes the *receive* at process q; the p-*receive* follows it. Hence no event at p can be concurrent with the q-*receive*—even though process p actually "experienced" a moment of concurrency: the edge from the p-*send* to the p-*receive*.

In this section, we explore the deeper issues at work here. We develop the concept of *view-completeness*: if an atom at a process affords an external temporal view, then an event at that process affords the same view. In Section 12.1 we develop tools for dealing with ordering edges; in Section 12.2 we define view-completeness; and in Section 12.3 we explore the implications of view-completeness for timeslices.

12.1. Tools for Edges

Isolating Transition Edges We begin by introducing a tool to move from one event to its successor and to its predecessor:

> **Definition 12.1** Let A be an event from graph α. Suppose there exists a unique B in α such that $A \longrightarrow B$: we will denote this event by $\mathsf{next}(A)$.
>
> Define $\mathsf{prev}(A)$ similarly.

If an event has unique neighbors, then these neighbors must lie on all precedence paths:

> **Proposition 12.2** Let A be an event from graph α.
>
> 1. If $\mathsf{next}(A)$ exists, then for any B in the transitive closure $\overline{\alpha}$:
>
> $$A \longrightarrow B \quad \Longrightarrow \quad \mathsf{next}(A) \stackrel{=}{\longrightarrow} B$$

2. If $\mathsf{prev}(A)$ exists, then for any B in the transitive closure $\overline{\alpha}$:

$$B \longrightarrow A \quad \Longrightarrow \quad B \overset{\longrightarrow}{=} \mathsf{prev}(A)$$

Proof All outgoing paths from A in α must start with the edge $A \longrightarrow \mathsf{next}(A)$; similarly all incoming paths must end with $\mathsf{prev}(A) \longrightarrow A$. $\square$

In a parallel pair, only non-maxima in the multicomponent are guaranteed to have successors, since the general model may add cross-process edges. Nevertheless, since a non-maxima in the global model is the image of a nonempty set of non-maxima in the local model, we can still obtain successors by specifying which process component to consider.

Definition 12.3 Suppose $(\mathbf{M}, \mathbf{M}')$ is an parallel pair. Let A be an event from a graph that $\mathbf{M}$ generates, and let $p \in$ **PROC-NAMES**. If $A|_p$ exists, define

$$\begin{aligned} \mathsf{next}_p(A) &= (\mathbf{M}/\mathbf{M}')\ (\mathsf{next}(\ A|_p\ )) \\ \mathsf{prev}_p(A) &= (\mathbf{M}/\mathbf{M}')\ (\mathsf{prev}(\ (\ A|_p\ ))) \end{aligned}$$

Ordering on Edges We defined precedence only for the events in a computation graph. But the definition extends to the edges as well, if we pretend that a dummy event lies inside the edge.

Definition 12.4 Suppose E is an edge connecting node E_{in} to node E_{out} in graph α. For node A:

- Define $A \longrightarrow E$ when $A \overset{\longrightarrow}{=} E_{\text{in}}$.
- Define $E \longrightarrow A$ when $E_{\text{out}} \overset{\longrightarrow}{=} A$.

12.2. View-Complete Models

We now use the tools of Section 12.1 to develop *view-completeness*: when every edge has an event with the same view.

External Equivalence First, we define what it means for two atoms to have the same view.

Definition 12.5 Suppose $(\mathbf{M}, \mathbf{M}')$ is a multiprocess pair. Suppose A and B are atoms in an $\mathbf{M}$ graph, such that for some $p \in$ **PROC-NAMES**, both $A|_p$ and $B|_p$ exist. Then A and B are *externally equivalent at* p, written

$$A \overset{p}{\sim} B$$

when

1. In $\overline{\mathbf{M}}$, A is cyclic iff B is cyclic
2. For any $q \neq p$ and any event C such that $C|_{\overline{q}}$ exists:

$$\begin{array}{ccc} A \longrightarrow C \text{ in } \overline{\mathbf{M}} & \Longleftrightarrow & B \longrightarrow C \text{ in } \overline{\mathbf{M}} \\ C \longrightarrow A \text{ in } \overline{\mathbf{M}} & \Longleftrightarrow & C \longrightarrow B \text{ in } \overline{\mathbf{M}} \end{array}$$

Informally, $A \overset{p}{\sim} B$ when both are cyclic (never part of a timeslice) or acyclic, and both divide the atoms from other processes into the same "past" and "future" sets.

View-Complete Models View-completeness is simply the property of every edge having an externally equivalent event:

Definition 12.6 Suppose parallel pair $(\mathbf{M}, \mathbf{M}')$ acts on graph α. Graph $\gamma = \mathbf{M}(\alpha)$ is *view-complete* when for any edge $E \in \gamma$ and $p \in$ **PROC-NAMES**, if $E|_p$ exists, then there exists a node $A \in \gamma$ with $A \overset{p}{\sim} E$.

If $\mathbf{M}$ produces only view-complete graphs, then we say that parallel pair $(\mathbf{M}, \mathbf{M}')$ is *view-complete*.

Usually an Endpoint It would be convenient if in a view-complete model, the event externally equivalent to a given edge were always one of the endpoints of the edge. We can establish that for models meeting a fairly reasonable property, this will be the case.

We start by defining the property: when the model draws no back-edges or self-loops along process components.

Definition 12.7 A multiprocess pair $(\mathbf{M}, \mathbf{M}')$ with localization $\mathbf{L}$ is *locally acyclic* when for any A, B:

$$A \rightleftarrows B \text{ in } \mathbf{L} \quad \Longrightarrow \quad B \not\longrightarrow A \text{ in } \mathbf{M}$$

We call this property "locally acyclic" because cycles in such models must touch more than one process:

Lemma 12.8 Suppose locally acyclic multiprocess pair $(\mathbf{M}, \mathbf{M}')$ acts on graph α. If event A from $\gamma = \mathbf{M}(\alpha)$ satisfies

- $A \longleftrightarrow A$ in $\overline{\gamma}$
- $A|_p$ exists, for some $p \in$ **PROC-NAMES**

then there exists event B in γ satisfying:

- $A \longleftrightarrow B$ in $\overline{\gamma}$

- $B|_q$ exists, for some $q \neq p$

Proof If γ provides a path from A to A that does not touch such a B, γ must have an edge that contradicts Definition 12.7. $\square$

As promised, if the model is locally acyclic, then the event for an edge must be an endpoint.

Proposition 12.9 Suppose parallel pair $(\mathbf{M}, \mathbf{M}')$ is locally acyclic and view-complete. If edge E connects E_{in} to E_{out} in an $\mathbf{M}$ graph and $E|_p$ exists, then $E \overset{p}{\sim} E_{\text{in}}$ or $E \overset{p}{\sim} E_{\text{out}}$.

Proof If E is cyclic in $\overline{\mathbf{M}}$, then the conclusion easily holds. So, assume E is acyclic.

Since $(\mathbf{M}, \mathbf{M}')$ is view-complete, a node A must exist with $E \overset{p}{\sim} A$. Since the transitive p component is totally ordered, in $\overline{\mathbf{M}'}(\alpha)$ we have either $A|_{\overline{p}} \Longrightarrow E_{\text{in}}|_{\overline{p}}$ or $E_{\text{out}}|_{\overline{p}} \Longrightarrow A|_{\overline{p}}$.

Consider the case when $A|_{\overline{p}} \Longrightarrow E_{\text{in}}|_{\overline{p}}$. Assume atom C occurs at a different process: for some $q \neq p$, $C|_q$ exists.

- If $C \longrightarrow E$ then $C \longrightarrow A$ and hence $C \longrightarrow E_{\text{in}}$.
- If $C \longrightarrow E_{\text{in}}$ then $C \longrightarrow E$ by Definition 12.4.
- If $E \longrightarrow C$ then easily $E_{\text{in}} \longrightarrow C$.
- If $E_{\text{in}} \longrightarrow C$ then $A \longrightarrow C$ and hence $E \longrightarrow C$.
- If E_{in} is cyclic, then Lemma 12.8 gives us a B at another process with $B \longleftrightarrow E_{\text{in}}$; such a B precedes E and also follows E (since B follows A, and $A \overset{p}{\sim} E$. This violates our assumption that E is acyclic, thus E_{in} must be acyclic.

Thus $E \overset{p}{\sim} E_{\text{in}}$.

Similarly, if $E_{\text{out}}|_{\overline{p}} \Longrightarrow A|_{\overline{p}}$ then $E \overset{p}{\sim} E_{\text{out}}|_{\overline{p}}$. $\square$

12.3. Timeslices in View-Complete Models

View-completeness suffices to provide a very nice characterization of timeslices.

Preparation First, we establish that edges following an entry in the timestamp vector of an event cannot precede that event:

Lemma 12.10 Suppose parallel pair $(\mathbf{M}, \mathbf{M}')$ acts on graph α. Let $\gamma = \mathbf{M}(\alpha)$ and $\beta = \mathbf{M}'(\alpha)$.

Let A be an event in γ, let $p \in$ **PROC-NAMES**, and let edge E from γ have $E|_p$ nonempty. If $\pi_p\,\mathbf{V}(A)$ is defined, then

$$(\pi_p\,\mathbf{V}(A))|_{\overline{p}} \longrightarrow E|_{\overline{p}} \text{ in } \overline{\beta} \quad \Longrightarrow \quad E \not\longrightarrow A \text{ in } \overline{\gamma}$$

If $\pi_p\,\mathbf{R}(A)$ is defined, then

$$E|_{\overline{p}} \longrightarrow (\pi_p\,\mathbf{R}(A))|_{\overline{p}} \text{ in } \overline{\beta} \quad \Longrightarrow \quad A \not\longrightarrow E \text{ in } \overline{\gamma}$$

Proof Let $B = \pi_p\,\mathbf{V}(A)$, let E connect B to C, and let B', E' and C' be their process p images in $\overline{\beta}$. If $E \longrightarrow A$ then $C \rightleftarrows A$ (by Definition 12.4), but $B' \longrightarrow C'$ in the transitive p component. Hence B could not have been the p entry of the timestamp vector. The rollback case is similar. □

The Main Result With view-completeness, timeslices are exactly the consistent cuts.

Theorem 12.11 *(View-Completeness)* Suppose $(\mathbf{M}, \mathbf{M}')$ is a transitively-bounded view-complete parallel pair. Then $\overline{\mathbf{M}}$-timeslices are consistent cuts.

Proof Let $(\mathbf{M}, \mathbf{M}')$ act on graph α, with $\gamma = \mathbf{M}(\alpha)$ and $\beta = \mathbf{M}'(\alpha)$.

Clearly the bounding singletons are timeslices, and no other timeslice can contain a bounding node. So let X be a timeslice different from the bounding singletons. Since X is a partial cut, if X is not a consistent cut then some process p must not be represented.

We will now demonstrate that this can never be the case, by showing that if process p is not represented, then X could not have been a timeslice.

Assume process p is not represented in X. Define events A and B as follows:

$$\begin{aligned} A &= \pi_p\ (\sqcup_{C \in X} \mathbf{V}(C)) \\ B &= \pi_p\ (\sqcap_{C \in X} \mathbf{R}(C)) \end{aligned}$$

If $B \rightleftarrows A$, then X cannot be a timeslice. So it must be the case that $A|_{\overline{p}}$ properly precedes $B|_{\overline{p}}$ in the transitive p component. Hence A cannot be the maximum, so let E be the edge in $\overline{\gamma}$ connecting A to $\mathsf{next}_p(A)$. The edge $E|_p$ exists and satisfies

$$A|_{\overline{p}} \quad \longrightarrow \quad E|_p \quad \longrightarrow \quad B|_{\overline{p}}$$

in $\overline{\beta}$. By Lemma 12.10, E can neither precede nor follow any event in X.

Further, if E were cyclic, then $\mathsf{next}_p(A) \longrightarrow A$ (by Definition 12.4), hence $\mathsf{next}_p(A)$ would appear in the relevant timestamp vector rather than A. Thus E is not cyclic.

Since $(\mathbf{M}, \mathbf{M}')$ is view-complete, an acyclic event D must exist in γ with $D \overset{p}{\sim} E$, hence $X \cup \{D\}$ is mutually concurrent. Since D touches p (by definition of external equivalence), $D \notin X$. Thus X could not have been a timeslice. □

Chapter 13 repairs POT by forcing it to be view-complete. Chapter 14 considers more deeply the implications of the View-Completeness Theorem (Theorem 12.11).

(Part III)

Chapter 13

Real Simultaneity and View-Complete Partial Order Time

This chapter uses the results of Chapter 12 to revise the POT model so that it exhibits the correct timeslice behavior. We offer two approaches, both of which hinge on forcing POT to be view-complete. In Section 13.1, we restrict the input graphs so that only well-behaved graphs come out; in Section 13.2 we explicitly insert place-holder events. In Section 13.3, we demonstrate that these fixes work.

13.1. One Fix: Restrict the Domain

One approach is simply to restrict the domain of graphs to which POT applies.

Definition 13.1 Define the *restricted partial order time* model RPOT to be the model POT, restricted to ground-level graphs whose images are view-complete. That is, RPOT $=$ POT, on the domain

$$\mathcal{D} \quad = \quad \{\alpha \ : \ \text{POT}(\alpha) \text{ is view-complete}\}$$

13.2. Another Fix: Insert the Necessary Events

A more general technique to see that any transitively-bounded parallel model is view-complete is to insert place-holder events into the the edges between consecutive events at a process.

For parallel pair $(\mathbf{M}, \mathbf{M}')$, we need to do this insertion both in the $\mathbf{M}$ graph and in the $\mathbf{M}'$ graph. But since $\mathbf{M}'$, $\mathbf{M}'$ is trivially a parallel pair, we can use the same operator for both.

Definition 13.2 Suppose parallel pair $(\mathbf{M}, \mathbf{M}')$ acts on graph α. Let $\gamma = \mathbf{M}(\alpha)$ and $\beta = \mathbf{M}'(\alpha)$.

Define the *place-holding* model PH on such γ as follows. First, create a copy of γ, with each atom representing itself. Then, for each edge E in γ with $E|_{\mathbf{M}'}$ nonempty:

- Delete edge E from the copy.
- Add to the copy a new intermediate node $(A \parallel B)$ representing the deleted edge, where A and B are the nodes that E connects in γ.
- Add to the copy two ghost edges: $A \longrightarrow (A \parallel B)$ and $(A \parallel B) \longrightarrow B$

The place-holding model acts exactly as desired:

Theorem 13.3 If $(\mathbf{M}, \mathbf{M}')$ is a parallel pair, then

$$(\overline{\mathrm{PH}} \circ \mathbf{M}),\ (\overline{\mathrm{PH}} \circ \mathbf{M}')$$

is a view-complete parallel pair.

Proof This result follows directly from the definitions. The PH model just adds intermediate nodes and the appropriate edges, and does the same things both to $\mathbf{M}$ and to $\mathbf{M}'$. Any edge from $\mathbf{M}'$ now is split into two edges surrounding an intermediate node, and this new node is externally equivalent to these edges. □

Figure 13.1 illustrates this behavior.

(Strictly speaking, to make POT view-complete, we only need to insert intermediate nodes into TIMELINES edges from *send* events to *receive* events. Further, these insertions are sufficient but still not necessary—consider messages that carry no new ordering information.)

Applying $\overline{\mathrm{PH}}$ also preserves transitive bounding.

Proposition 13.4 $\overline{\mathrm{PH}} \circ \mathbf{M}$ is transitively bounded iff $\mathbf{M}$ is transitively bounded.

Proof Inserting intermediate events does not change the extrema. □

13.3. These Fixes Work

Once modified to be view-complete, the POT model exhibits the desired timeslice behavior.

Definition 13.5 Define the model PPOT to be the composition $\overline{\mathrm{PH}} \circ$ POT.

Theorem 13.6 Suppose PPOT or RPOT generate graph γ. If a set of events X in γ is a timeslice in $\overline{\gamma}$, then X minimally represents a global state.

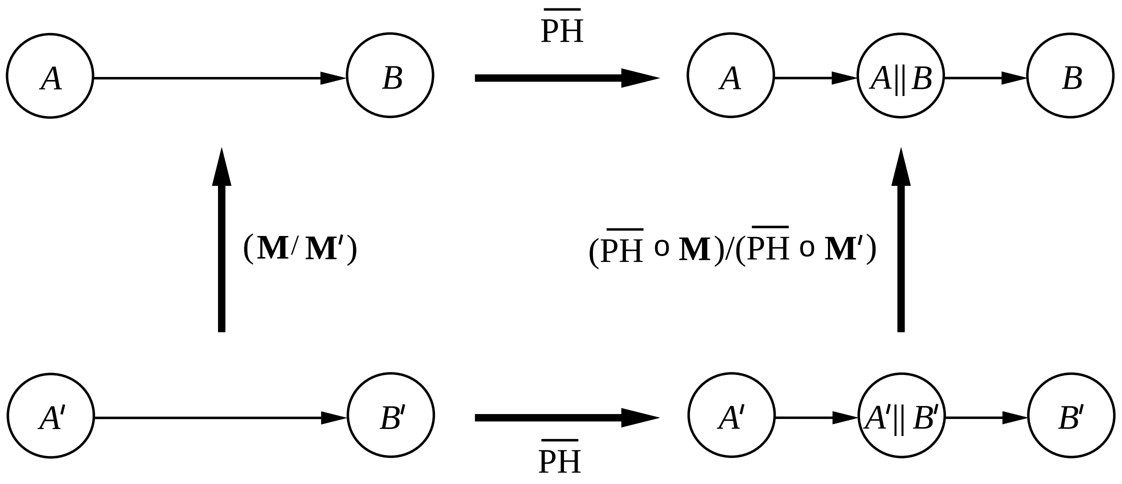
$\overline{\text{PH}}$
A
B
A
$A\|B$
B
$(\mathbf{M}/\mathbf{M}')$
$(\overline{\text{PH}} \circ \mathbf{M})/(\overline{\text{PH}} \circ \mathbf{M}')$
A'
B'
A'
$A'\|B'$
B'
$\overline{\text{PH}}$

Figure 13.1 We apply $\overline{\text{PH}}$ to a parallel pair $(\mathbf{M}, \mathbf{M}')$. This action preserves the parallelism: $\overline{\text{PH}} \circ \mathbf{M}$ is still parallel, with multicomponent $\overline{\text{PH}} \circ \mathbf{M}'$. Further, the action ensures view-completeness—each edge now has a place-holder edge.

Proof This fact follows directly from Theorem 11.8 and the View-Completeness Theorem (Theorem 12.11). □

Theorem 13.7 Suppose PPOT or RPOT generates graph γ. If a set of events X from γ minimally represents a global state, then X is a timeslice in $\overline{\gamma}$.

Proof A (nontrivial) global state maps to a full cut X in PPOT (or RPOT, respectively). Since MSG edges and PH $\circ$ TIMELINE edges go strictly forward in time, X must be consistent. □

Theorem 13.8 Let X be a global state from ground-level graph α. If an event in PPOT(α) represents any part of X, then a timeslice in $\overline{\text{PPOT}}(\alpha)$ minimally represents X; similarly for RPOT(α).

Proof This theorem follows directly from the proof for Theorem 13.7. □

(Part III)

Chapter 14

Timeslices and View-Completeness

This chapter explores the structure of timeslices in view-complete parallel pairs.

Section 14.1 introduces terms for two more varieties of multiprocess pairs. Section 14.2 presents a convenient *extendibility* property that follows from the View-Completeness Theorem (Theorem 12.11). Section 14.3 introduces an alternate form of timestamp and rollback vectors; Section 14.4 uses these alternate forms to characterize timeslices.

14.1. Two New Types of Models

Since this chapter will build on the View-Completeness Theorem (Theorem 12.11), we now define a short term summing up the conditions of that theorem:

> **Definition 14.1** If two models $(\mathbf{M}, \mathbf{M}')$ are a view-complete parallel pair, and $\mathbf{M}$ is transitively bounded, then we say that $(\mathbf{M}, \mathbf{M}')$ is a *consistent pair*.

The View-Completeness Theorem (Theorem 12.11) inspires this term: timeslices are *consistent* cuts.

Section 14.4 uses an additional property: that the process components appear independently in the global model.

> **Definition 14.2** A multiprocess pair $(\mathbf{M}, \mathbf{M}')$ is *independent* when, in any graph generated by $\overline{\mathbf{M}}$, any atom (except a bounding node) represents exactly one atom in exactly one component model in $\overline{\mathbf{M}'}$.

14.2. Extendibility

The View-Completeness Theorem (Theorem 12.11) yields the following as an easy consequence.

Theorem 14.3 *(Extendibility)* Suppose consistent parallel pair $(\mathbf{M}, \mathbf{M}')$ generates graph γ. Any set X of mutually concurrent events from $\overline{\gamma}$ extends to a consistent cut: that is, a consistent cut X' exists with $X \subset X'$.

Proof Such a set X extends to a maximal concurrent set X'. From the View-Completeness Theorem (Theorem 12.11), X' must a consistent cut. □

Hence any acyclic event is part of a consistent cut, and we can find timeslices using a greedy algorithm: just keep appending mutually concurrent events.

14.3. Extremal Timeslices

From the Extendibility Theorem (Theorem 14.3), we know that any acyclic event naturally extends to a timeslice. From the the View-Completeness Theorem (Theorem 12.11) we know that this timeslice will be a consistent cut. From Theorem 9.21 we know these consistent cuts form a lattice. Since things are finite, this set has a unique maximum and a unique minimum.

Section 14.3.1 constructs these extremal consistent cuts: they are the timestamp and rollback vectors, slightly adjusted. Section 14.3.2 proves that the event sets of these adjusted vectors are indeed the extremal cuts. (Section 14.4 will use these adjusted vectors to characterize arbitrary timeslices.)

14.3.1. Adjusted Timestamp Vectors and Adjusted Rollback Vectors

The timestamp vector of an event consists of the maximal event from each process that precedes or equals that event. Suppose acyclic event A occurs at exactly one place: process p. Then the p entry from $\mathbf{V}(A)$ equals A. We obtain the *adjusted timestamp vector* by, for each $q \neq p$, replacing the process q entry of $\mathbf{V}(A)$ with the minimal event equivalent to its local successor edge. Barring local cycles, we just take the $\pi_q \mathbf{V}(A)$ entry and adjust it one event forward. In the more general case that A occurs at multiple processes, then we only slide forward the events at the processes that A does not touch.

We define *adjusted rollback vectors* symmetrically by adjusting backward the entries from the rollback vector. For each $q \neq p$, we replace the process q entry of $\mathbf{R}(A)$ with the maximal event equivalent to its local predecessor edge.

Definition 14.4 Suppose consistent parallel pair $(\mathbf{M}, \mathbf{M}')$ acts on graph α. Let A be an acyclic non-bounding event from $\gamma = \mathbf{M}(\alpha)$; let $\beta = \mathbf{M}'(\alpha)$.

Construct the *adjusted timestamp vector* $\mathbf{V}^*(\gamma, \mathbf{M}, \mathbf{M}', A)$ from $\mathbf{V}(A)$ as follows. For each $p \in$ **PROC-NAMES** such that $A|_p$ does not exist:

- Let E be the edge from $\pi_p\,\mathbf{V}(A)$ to $\mathsf{next}_p((\pi_p\,\mathbf{V}(A)))$.
- Replace $\pi_p\,\mathbf{V}(A)$ by the p-minimum from

$$\{C\ :\ C \overset{p}{\sim} E\}$$

Define the *adjusted rollback vector* $\mathbf{R}^*$ similarly, using $\mathsf{prev}(p)\,(\pi_p\,\mathbf{R}(A))$ and the p-maximum. As with ordinary timestamp vectors, we will truncate the cumbersome parameter list whenever possible.

Definition 14.4 works: the entries of the adjusted timestamp vectors and adjusted rollback vectors exist.

Proposition 14.5 Suppose consistent parallel pair $(\mathbf{M}, \mathbf{M}')$ acts on graph α. Let A be an acyclic non-bounding event from $\gamma = \mathbf{M}(\alpha)$; let $\beta = \mathbf{M}'(\alpha)$. All entries of $\mathbf{V}^*(A)$ and $\mathbf{R}^*(A)$ are defined.

Proof If $\mathbf{V}(A)$ contains the $\overline{\gamma}$ maximum, then A must be the maximum. From this observation and from Proposition 8.10, if A is non-bounding, then no $(\pi_p\,\mathbf{V}(A))|_{\overline{p}}$ is the p-maximum in $\overline{\beta}$. Hence such an edge E exists, and because $(\mathbf{M}, \mathbf{M}')$ is view-complete, the set must be nonempty. By definition of $\overset{p}{\sim}$, each event in the set must touch the p component. The rollback case is similar. $\Box$

Figure 14.1 distinguishes between timestamp vectors and adjusted timestamp vectors.

Adjusting vectors only is complicated when events can occur at multiple processes or the view-complete event for an edge is not an endpoint. Since neither of these facts apply to PPOT or RPOT, adjusting vectors in these models is fairly simple.

14.3.2. The Extremal Timeslice Theorem

If event A is acyclic, then at every process where A does not occur, the timestamp vector entry must precede the rollback vector entry.

Let p be such a process. At p, the first edge after the timestamp entry and the last one before the rollback entry must both be concurrent with A. We select the $\mathbf{V}^*$ and $\mathbf{R}^*$ entries by finding the minimal and maximal externally equivalent events (respectively)—hence the $\mathbf{V}^*$ and $\mathbf{R}^*$ entries are respectively the minimal and maximal p events concurrent with A.

Consequently, the adjusted vectors give the bounds of the lattice of consistent cuts containing an event:

Theorem 14.6 *(Extremal Timeslices)* Suppose consistent parallel pair $(\mathbf{M}, \mathbf{M}')$ acts on graph α. Let $\gamma = \mathbf{M}(\alpha)$.

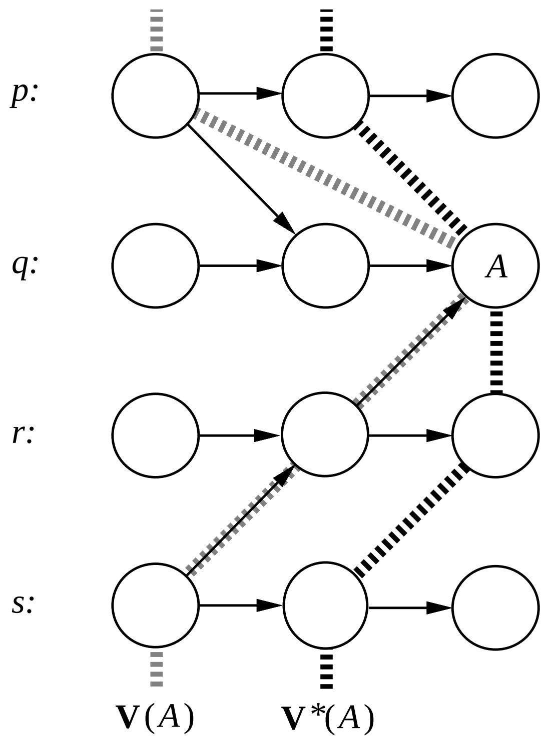

Figure 14.1 For an event A, the *timestamp vector* $\mathbf{V}(A)$ is its information horizon: the latest event, at each process, that A may have heard about. The *adjusted time-stamp vector* $\mathbf{V}^*(A)$ is just the timestamp vector, advanced one position everywhere except global event A.

If A is an acyclic non-bounding event, then

$$\begin{aligned}\{B \in \mathbf{V}^*(A)\} &= \sqcap\{X : X \text{ is a consistent cut containing } A\}\\ \{B \in \mathbf{R}^*(A)\} &= \sqcup\{X : X \text{ is a consistent cut containing } A\}\end{aligned}$$

Proof If A touches every process, then $\mathbf{V}^*(A) = \mathbf{R}^*(A)$ and every entry is A. This set is mutually concurrent because A is acyclic. Because the process components are totally ordered, this can be the only timeslice.

So, assume that there exists a process $p \in$ **PROC-NAMES** such that $A|_p$ does not exist. Let $B = \pi_p\,\mathbf{V}(A)$ and $C = \pi_p\,\mathbf{R}(A)$. From Theorem 10.4, no event D touching p and concurrent with A can have $D \Longrightarrow B$ or $C \Longrightarrow D$. If $C \Longrightarrow B$, then A cannot be acyclic. So it must be the case that $B|_{\overline{p}}$ properly precedes $C|_{\overline{p}}$ in the transitive p component. Let E_B be the edge connecting B to $\mathsf{next}_p(B)$ in γ, and E_C be the edge connecting $\mathsf{prev}_p(C)$ to C. Edge E_B is concurrent with A and and acyclic:

- *Concurrent.* If E_B precedes A, then the out node of $E_B|_p$ in β would be the p $\mathbf{V}(A)$ entry, rather than the in node. But E_B follows A, then by Definition 12.4 $B \Longrightarrow A$, which would give that $A \longleftrightarrow B$ and hence A is cyclic.
- *Acyclic.* If E_B were cyclic, then $\mathsf{next}_p(B) \longrightarrow B$ and thus B would not be the p entry in $\mathbf{V}(A)$.

Hence $\{\pi_p\,\mathbf{V}^*(A), A\}$ is a mutually concurrent set in $\overline{\gamma}$, and so can be expanded to a consistent cut, but no event preceding $\pi_p\,\mathbf{V}^*(A)$ in the p component can be part of a timeslice with A.

The case for $\pi_p\,\mathbf{R}^*(A)$ is similar. $\square$

Definition 14.4 defined adjusted timestamp vectors and adjusted rollback vectors as vectors: arrays of events. The Extremal Timeslice Theorem (Theorem 14.6) establishes that these vectors possess even more structure: their event sets are both cuts and timeslices.

Corollary 14.7 The event sets of $\mathbf{V}^*(A)$ and $\mathbf{R}^*(A)$ are consistent cuts.

Proof The $\sqcap$ and $\sqcup$ operations preserve consistent cuts. $\square$

Consequently, we can now regard $\mathbf{V}^*(A)$ and $\mathbf{R}^*(A)$ as simply event sets—since projection will never be ambiguous.

14.4. Characterizing Timeslices

The Extremal Timeslice Theorem (Theorem 14.6) tells us that the adjusted timestamp vector for an event gives us the minimal timeslice containing that event. One might conjecture that any timeslice can be obtained this way, but this conjecture is false. Figure 14.2 sketches a simple counter-example.

If this conjecture were to hold in general, then every timeslice X would have to possess some event A that forces of each remaining $B \in X$ to be part of X—where event A forces an event B when $B \neq A$ and $B \in \mathbf{V}^*(A)$.

The conjecture fails because acyclic events can be mutually concurrent without forcing each other. However, *we can express this forcing relation with a time model.* If our consistent parallel pair is acyclic and independent, then the forcing model will form an acyclic independent parallel pair.

This insight yields two results:

Unique Signatures Suppose X is a timeslice. It is not true in general that X is the adjusted timestamp vector of one entry. However, it is trivially true that X equals the join of the adjusted timestamp vectors, over all elements of X.

$$X \;=\; \sqcup\{\mathbf{V}^*(A) \;:\; A \in Y\}$$

If we removed elements from Y one by one, when would this relation stop holding? We can establish that there is a *unique* $Y \subset X$ such that the relation holds for Y, but does not hold for any proper subset of Y.

Meta-Timeslices A set of events is such a "timeslice" signature iff it is a mutually concurrent set in the forcing model. Consequently, a timeslice of k events in the forcing model expresses 2^k timeslices in the original model.

In Section 14.4.1, we define the FORCE model, to capture when an event forces another. In Section 14.4.2 we establish a series of lemmas about the FORCE model and the parallel pairs it induces. We use these lemmas to establish our main result in Section 14.4.3.

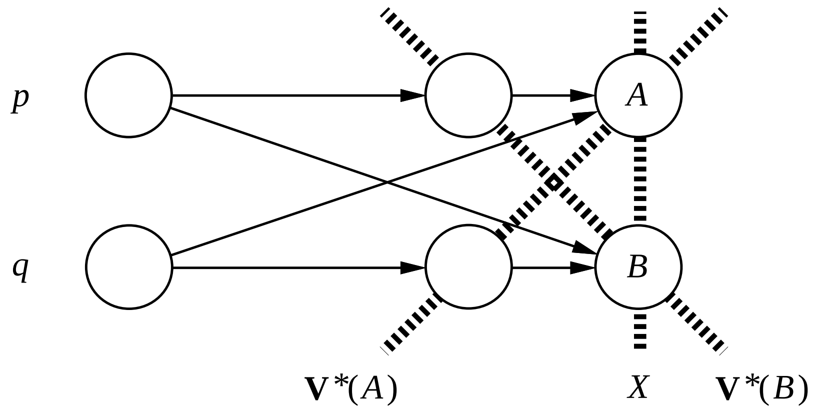

Figure 14.2 Timeslice $X = \{A, B\}$ equals neither $\mathbf{V}^*(A)$ nor $\mathbf{V}^*(B)$. This example disproves the conjecture that the Extremal Timeslice Theorem (Theorem 14.6) might characterize all timeslices.

14.4.1. A Model to Express Forcing

We define a model that copies each cross-process edge in the single-step global model and slides the in-node one event forward.

Definition 14.8 Suppose independent parallel pair $(\mathbf{M}, \mathbf{M}')$ acts on graph α. Let $\gamma = \mathbf{M}(\alpha)$ and $\beta = \mathbf{M}'(\alpha)$.

Define the model FORCE on γ as follows:

- copy γ; let each atom here represent itself.
- for each pair of non-bounding nodes A, B such that:
  - $A \longrightarrow B$ in γ
  - A occurs at process p but B does not.

  add a ghost edge from $\mathsf{next}_p(A)$ to B in the copy.

For example, to construct FORCE $\circ$ POT we copy the POT graph, and then for each *send* whose *receive* is at a different process, we draw an edge from the *successor* of the *send* to the *receive*.

The remainder of Section 14.4 establishes that FORCE captures the forcing discussed earlier. Figure 14.2 illustrated this fact: the counter-example timeslice X remains a timeslice even if we apply $\overline{\text{FORCE}}$—indicating that neither event forces the other.

14.4.2. Preparation

First, we show that applying FORCE preserves independence and parallelism.

Lemma 14.9 Suppose $(\mathbf{M}, \mathbf{M}')$ is an independent consistent parallel pair. Then $((\text{FORCE} \circ \mathbf{M}), \mathbf{M}')$ is an independent transitively-bounded parallel pair.

Proof This is clear from the definitions. The only tricky part is showing that $\overline{\text{FORCE}} \circ \mathbf{M}$ is bounded.

Suppose FORCE adds edge $\mathsf{next}_p(A) \longrightarrow B$. and $\mathsf{next}_p(A)$ was the global maximum. Let E be the edge from A to $\mathsf{next}_p(A)$. E is concurrent with B, but no node at p is. Hence $(\mathbf{M}, \mathbf{M}')$ could not be view-complete. $\square$

The FORCE model does not preserve consistency because the resulting model may not be view-complete. Consider the PPOT model. If only an intermediate event keeps a *send* from immediately preceding a *receive*, then FORCE will slide the in-node of the message edge up to the intermediate event. The edge from the intermediate event to the *receive* will then have no externally equivalent event.

We can use view-completeness to show that a path in $\overline{\gamma}$ can always be extended on the end with FORCE(γ) edges:

Lemma 14.10 Suppose independent consistent parallel pair $(\mathbf{M}, \mathbf{M}')$ acts on graph α. Let $\gamma = \mathbf{M}(\alpha)$ and $\beta = \mathbf{M}'(\alpha)$.

Suppose $A \longrightarrow B$ in $\overline{\gamma}$ and $B \longrightarrow C$ in $\overline{\text{FORCE}}(\gamma)$. Then $A \longrightarrow C$ in $\overline{\gamma}$.

Proof We establish the result assuming $B \longrightarrow C$ in FORCE(γ) but not γ. The more general result follows easily—just use this special result to have the $\overline{\gamma}$ path absorb each edge in the γ path.

Let event B occur at process p. Then C must occur somewhere else, and $\mathsf{prev}_p(B) \longrightarrow C$ in γ. Thus any event D at process p satisfies

$$D \longrightarrow C \text{ in } \overline{\gamma} \quad \vee \quad A \longrightarrow D \text{ in } \overline{\gamma}$$

If $A \not\longrightarrow C$ in $\overline{\gamma}$ then $A \not\longrightarrow \mathsf{prev}_p(B)$ and $B \not\longrightarrow C$. Let E be the edge connecting $\mathsf{prev}_p(B)$ to B. Then $A \not\longrightarrow E$ and $E \not\longrightarrow C$. Since $(\mathbf{M}, \mathbf{M}')$ is view-complete, there exists an event D at process p with $A \not\longrightarrow D$ and $D \not\longrightarrow C$. This violates the above condition. $\Box$

However, a path starting with a new FORCE edge only induces a $\overline{\gamma}$ path starting from the immediate predecessor of the path's first event.

Lemma 14.11 Suppose independent consistent parallel pair $(\mathbf{M}, \mathbf{M}')$ acts on graph α, and $\gamma = \mathbf{M}(\alpha)$.

Suppose non-bounding $\overline{\gamma}$ event A occurs at process $p \in$ **PROC-NAMES**. For any event B:

$$A \longrightarrow B \text{ in } \overline{\text{FORCE}}(\gamma) \quad \Longrightarrow \quad \mathsf{prev}_p(A) \longrightarrow B \text{ in } \overline{\gamma}$$

Proof If B is bounding, the result is trivial. So assume B is not bounding. Let q be the process of B. Suppose $A \longrightarrow B$ in $\overline{\text{FORCE}}(\gamma)$. Consider the path from A to B in FORCE(γ):

$$A \longrightarrow B_1 \longrightarrow .... \longrightarrow B_k = B$$

If $A \longrightarrow B_1$ in γ, then we easily have the result:

$$\mathsf{prev}_p(A) \longrightarrow B_1 \in \overline{\gamma}$$

Otherwise, Definition 14.8 gives the fact:

$$\mathsf{prev}_p(A) \longrightarrow B_1 \text{ in } \gamma$$

In either case, Lemma 14.10 gives the fact:

$$\mathsf{prev}_p(A) \longrightarrow B \text{ in } \overline{\gamma}$$

$\Box$

Consequently, FORCE preserves the acyclic property.

Lemma 14.12 Suppose consistent parallel pair $(\mathbf{M}, \mathbf{M}')$ is independent. If $\overline{\mathbf{M}}$ is acyclic then $\overline{\text{FORCE}} \circ \mathbf{M}$ is acyclic.

Proof Let γ be an $\mathbf{M}$ graph. If $\overline{\text{FORCE}}(\gamma)$ has a cycle, this must have come from a $\text{FORCE}(\gamma)$ edge, which all cross processes. So there exists A at process p and B at $q \neq p$ with $A \longleftrightarrow B$ in $\overline{\text{FORCE}}(\gamma)$. Lemma 14.11 gives $\mathsf{prev}_p(A) \longrightarrow B$ and $\mathsf{prev}_q(B) \longrightarrow A$ in $\overline{\gamma}$. Hence any event at p either precedes B or follows $\mathsf{prev}_q(B)$ in $\overline{\gamma}$. But the edge from $\mathsf{prev}_p(A)$ to A does neither, and the fact that $(\mathbf{M}, \mathbf{M}')$ is view-complete gives a contradiction. □

For events concurrent in the original model, FORCE precedence is equivalent to $\mathbf{V}^*$ forcing.

Lemma 14.13 Suppose independent consistent parallel pair $(\mathbf{M}, \mathbf{M}')$ acts on graph α, $\overline{\mathbf{M}}$ is acyclic, and $\gamma = \mathbf{M}(\alpha)$.

If A and B satisfy $A \not\longleftrightarrow B$ in $\overline{\gamma}$, then

$$A \in \mathbf{V}^*(B) \iff A \longrightarrow B \text{ in } \overline{\text{FORCE}}(\gamma)$$

Proof Let B occur at process p.

From Proposition 12.9 and Definition 14.4, we know that the p entry of $\mathbf{V}^*(B)$ is B, but for every $q \neq p$, the q entry is $\mathsf{next}_q((\pi_q \mathbf{V}(B)))$.

Let A occur at $q \neq p$. If $A \in \mathbf{V}^*(B)$, then $A \not\longrightarrow B$ in $\overline{\gamma}$ but $\mathsf{prev}_q(A) \longrightarrow B$ in $\overline{\gamma}$, hence $A \longrightarrow B$ in $\overline{\text{FORCE}}(\gamma)$.

If $A \longrightarrow B$ in $\overline{\text{FORCE}}(\gamma)$, Lemma 14.11 give $\mathsf{prev}_q(A) \longrightarrow B$ in $\overline{\gamma}$. Since $A \not\longrightarrow B$ in $\overline{\gamma}$, we have $\mathsf{prev}_q(A) = \pi_q \mathbf{V}(B)$. Hence $A \in \mathbf{V}^*(B)$. □

14.4.3. The Main Result

We now establish the main result: the $\overline{\text{FORCE}}$ maxima of a timeslice form the unique forcing subset of that timeslice.

Theorem 14.14 Suppose independent consistent parallel pair $(\mathbf{M}, \mathbf{M}')$ acts on graph α, and $\mathbf{M}$ is acyclic. Let $\gamma = \mathbf{M}(\alpha)$ and $\beta = \mathbf{M}'(\alpha)$.

Any $\overline{\gamma}$-timeslice X has a *unique* minimal subset Y such that

$$X = \sqcup\{\mathbf{V}^*(A) : A \in Y\}$$

Further, event set Y is a such a minimal subset iff it is a mutually concurrent set in graph $\overline{\text{FORCE}}(\gamma)$.

Proof Let X be a timeslice from $\overline{\gamma}$. Since $\overline{\text{FORCE}}(\gamma)$ is acyclic (from Lemma 14.12), let Y be the set of FORCE sinks in X.

$$Y \quad = \quad \min_{\overline{\text{FORCE}}(\gamma)}(X)$$

For some $p \in$ **PROC-NAMES**, let $B = \pi_p X$ and

$$B' \quad = \quad \pi_p \ (\sqcup\{\mathbf{V}^*(A) \ : \ A \in Y\})$$

Consider the two cases:

1. If $B \in Y$, then for any other $C \in Y$, we have $B \not\longleftrightarrow C$ in $\overline{\text{FORCE}}(\gamma)$. Lemma 14.13 and the fact that $B \not\longleftrightarrow C$ in $\overline{\gamma}$ also gives that $B \notin \lceil \mathbf{V}^*(C) \rceil$. Hence $B' = B$.
2. If $B \notin Y$, then by construction of Y there exists a $C \in Y$ with $B \longrightarrow C$ in $\overline{\text{FORCE}}(\gamma)$. By Lemma 14.13, $B \in \mathbf{V}^*(C)$. If $D \neq C$ from Y has $\pi_p \, \mathbf{V}^*(D)$ dominating B, then $B \longrightarrow D$ in $\overline{\gamma}$ and X could not be a timeslice. Hence $B' = B$.

Thus we establish the first part of theorem.

One direction of the second part is easy: by construction of Y, no two events can precede each other in $\overline{\text{FORCE}}(\gamma)$. For the other direction, observe that Y will be the set of sinks in the $\overline{\gamma}$ timeslice $\sqcup\{\mathbf{V}^*(A) \ : \ A \in Y\}$. $\square$

This theorem has another interesting aspect: it gives us our first example of a useful time model different from the standard LINEAR, POT collection.

Useful Applying $\overline{\text{FORCE}}$ to a view-complete version of POT yields a model whose timeslices represent large sets of timeslices in POT.

Different The LINEAR model follows real time. The POT model departs from real time, but still expresses a chronologically "reasonable" temporal order. However, the FORCE model explicitly expresses orderings not found in real time.

Chapter 15

Conclusion

Chapter 1 asserted that distributed systems with distributed information require a distributed notion of time. Chapter 2 through Chapter 14 then develop mechanisms for a theory of distributed time.

This paper concludes by returning focus to the original assertions. Section 15.1 summarizes the mechanisms we developed and the motivations behind them. Section 15.2 outlines future research directions: to demonstrate the power of this theory by using it as a framework for secure applications.

15.1. Summary

Distributed Time for Distributed Systems Natural intuition suggests that time is linear, and thus that we should organize experience into a nicely behaved linear sequence of moments. Recent thought suggests that this intuition fails for asynchronous distributed systems, where information is distributed but perception is delayed. Such distributed environments require a more distributed theory of time.

- Distributed time provides the best perceivable approximation of the underlying linear description, which asynchrony renders unknowable.
- Distributed time provides a more appropriate language for distributed systems concepts not expressible in the language of real time.

Abstracting away irrelevant physical details to some convenient notion of discrete event is a common tool. Distributed time formalizes the notion of abstracting away irrelevant temporal details as well. The tools extend further: to abstracting away irrelevant or inconvenient computational detail.

- Distributed time expresses the conceit that the computation that “really happens” differs from the computation that physically occurred.

Laying the Groundwork Our ultimate claim is that distributed time clarifies problems and solutions in distributed environments. This paper lays the groundwork for establishing that claim by building the formal mechanisms for a theory of distributed time.

- We built a standard *computation graph* format to talk about events and temporal precedence, and translated physical descriptions of computation into *ground-level computation graphs*.
- We developed a *time model* formalism to express abstraction: a time model transforms a computation graph to a more abstract one whose individual events and edges may represent events and edges in the original graph.
- We explored some properties of time models, and in particular how their functional nature allows us to compose them to build *hierarchies* of abstraction, and multiple routes to the same graph.
- We developed *parallel pairs* of time models, to provide two levels of description of parallel computation.
- We explored the structure of *timeslices*—event sets representing points of logical simultaneity. In particular, we showed how timeslices relate to global states in real computations, how timeslices form a lattice, and how to construct time models to provide certain timeslice properties.

15.2. Future Work

Establishing that distributed time is the appropriate framework for distributed systems requires formalizing distributed time; this paper provides that foundation. This section discusses how future work will round out the claim: Section 15.2.1 discusses the benefits of using distributed time, Section 15.2.2 quickly sketches some examples, and Section 15.2.3 outlines further research.

15.2.1. Using Distributed Time

Distributed time provides a general framework to think about problems (and solutions) relating to time in distributed systems. We highlight some of the advantages:

Orthogonality Distributed time introduces orthogonality between the clocks tracking temporal relations and the protocols using these relations. We can change clock implementations, perhaps due to security or efficiency requirements, without changing protocols.

Flexibility Framing protocols explicitly in terms of distributed time allows insight and extensions to the protocols.

Expressiveness Freed from realistically describing computation, distributed time models can express more convenient abstractions. The orthogonality between clocks and protocols extends to an orthogonality between clocks and temporal relations—we can change models without changing the way clocks are called and used.

Abstraction Hierarchies By providing for hierarchies of related time models, distributed time allows for using protocols with multiple models even within a single computation.

Encapsulation and Unification The orthogonality between clocks and protocols gains some additional advantages: we can solve once and for all the clock issues we otherwise need to solve separately for each protocol, and we can unify in a single framework protocols that separately affect distributed time.

15.2.2. Quick Sketches

As a preview of future publications, we quickly sketch some examples supporting how distributed time might achieve the benefits we catalog above.

For these sketches, we consider two application problems that lend themselves to distributed time.

Snapshots As Chandy and Lamport [ChLa85] point out, the problem of one process assembling a *distributed snapshot* of the global state at one instant is difficult when asynchrony prevents identifying an instant, but a *consistent* global state suffices. Consistent global states are just the timeslices from a view-complete version of $\overline{\text{POT}}$.

Rollback If a process wants to undo an event A and execute A' instead, all events that depend on A need to be undone. Distributed time is relevant on two levels: determining what needs to be undone reduces to detecting temporal precedence in a partial order model; establishing a computation where A' happened instead of A requires abstracting from a POT graph showing the rollback to one showing the "correct" computation.

Considering the problems of snapshots and rollback provides some simple examples of the advantages of distributed time.

Orthogonality If we obtain snapshots by using $\overline{\text{POT}}$-clock primitives to determine concurrency, then we can change from vector clocks to logging sites (to avoid the n entries in each vector) or to signed vectors (to gain some degree of security) without changing the protocol.

Flexibility Almost without exception, current snapshot protocols use marker-pushing and thus are limited to taking a single, roughly current snapshot. Phrasing the problem in terms of $\overline{\text{POT}}$ relations allows a protocol using $\overline{\text{POT}}$-clocks, which immediately gives variations for more general versions of the problem, such as off-line snapshots, multiple snapshots, and using snapshots to detect unstable properties.

Expressiveness Suppose we wanted to pretend that the only instantaneous global states were those where no messages were in transit. A simple extension of POT expresses this conceit: timeslices here are exactly the desired global states. A process can capture such a global state simply by using its favorite snapshot protocol with the new clock primitives.

Abstraction Hierarchies Processes might want to use multiple clock suites even within the same computation. A snapshot with $\overline{\text{POT}}$ clocks provides a global state; a snapshot with $\overline{\text{FORCE}} \circ \text{POT}$ clocks provides an exponential number of global states.

Encapsulation and Unification Rollback with modified replay changes history. The orthogonality of clocks and protocols along with the single time framework allows us to still take off-line snapshots *using the same snapshot protocols.* The hierarchy of models give further flexibility: a snapshot from the original graph traps for potential global states in the real physical computation; a snapshot from the revised graph traps for global states in the virtual physical computation.

15.2.3. Research Plan

Current research consists of formalizing the points raised in the quick sketches. This work explores three principal topics:

Distributed Time as a Framework for Applications We need to formally express application problems (such as snapshots and rollback) in terms of the distributed time framework.

Clocks for General Time Models Specifying clock behavior brings up some additional issues, such as what a process can know about the underlying computation (*knowability*) and how querying about temporal relations should affect the temporal relations (*observation effects*).

Security in Clocks and Protocols By departing from real time, we sacrifice the potential for easy hardware verification of clock values. Encapsulation and orthogonality arguments apply here too: distributed time raises security risks, and protocols that depend on distributed time (even tacitly) are liable to these risks.

Accuracy Do distributed time clocks accurately report temporal precedence? What happens if networks or processes fail—or act maliciously?

Confinement Distributed time involves distributing private information. Can malicious agents exploit this information?

This research project started with the first identification of these security issues [Sm91], and will culminate in a thorough exploration of security and distributed time. [Sm94]

Index of Notation

Notation	*Description*	*Page*
$\prec_\gamma$	the precedence relation on event sets induced by the graph γ	78
$\longrightarrow$	precedes	27
$\underline{\longrightarrow}$	precedes or equals	27
$\not\longrightarrow$	does not precede	27
$\longleftrightarrow$	mutually precedes	27
$\underline{\longleftrightarrow}$	mutually precedes or equals	27
$\not\longleftrightarrow$	concurrent	27
$\overset{p}{\sim}$	externally equivalent at p	100
$\top$	maximum or final event	32
$\bot$	minimal or initial event	32
$\sqcap$	*meet*: the greatest lower bound, usually used as a binary operation	79
$\sqcup$	*join*: the least upper bound, usually used as a binary operation	79
$\cup_P$	union of graphs relative to pairing P	39
$\cup$	union of models	41
$\cup_\emptyset$	disjoint union	39, 42
$\equiv$	graph identity	23
$\equiv_P$	graph identity, enumerated by pairing P	23
$\cong$	graph isomorphism	23
$\cong_P$	graph isomorphism, enumerated by pairing P	23
$\subset$	graph containment	22
$\tilde{\subset}$	model containment	48

$\overline{\subset}$	direct model containment: $\overset{\sim}{\subset}$, except the graphs are *identical*	49
$\overset{\sim}{\subseteq}$	strong model containment: $\overset{\sim}{\subset}$, except the constituencies are *equal*	49
$\overline{\subseteq}$	strong and direct model containment: $\overset{\sim}{\subseteq}$ and $\overline{\subset}$ simultaneously.	49
$\rhd$	refinement	54
$\overset{\sim}{\sqsubset}$	model component: $\overset{\sim}{\subset}$ and $\rhd$ simultaneously	56
$\overline{\sqsubset}$	direct model component: $\overset{\sim}{\sqsubset}$ with $\overline{\subset}$	56
$\overset{\sim}{\sqsubseteq}$	strong model component: $\overset{\sim}{\sqsubset}$ with $\overset{\sim}{\subseteq}$	56
$\overline{\sqsubseteq}$	strong and direct model component: $\overset{\sim}{\sqsubset}$ with $\overline{\subset}$ and $\overset{\sim}{\subseteq}$	56
$x\|_{\mathbf{M}'}$	what x represents in the multicomponent $\mathbf{M}'$	68
$x\|_{p}$	what x represents in the nontransitive process p component	68
$x\|_{\overline{\mathbf{M}'}}$	what x represents in the transitive closure $\overline{\mathbf{M}'}$ of the multicomponent	68
$x\|_{\overline{p}}$	what x represents in the transitive process p component	68
$\lceil S \rceil$	*local past closure*: the events that precede or equal these guys in the multicomponent	71
$\lfloor S \rfloor$	*local future closure*: the events that follow or equal these guys in the multicomponent	71
$\mathsf{next}(A)$	the event following A	99
$\mathsf{prev}(A)$	the event preceding A	99
$\mathsf{next}_p(A)$	the following event at process p	100
$\mathsf{prev}_p(A)$	the preceding event at process p	100
$(A \parallel B)$	the intermediate event between A and B	105
$\langle\, \mathbf{M},\ \alpha \,\rangle$	representation map from $\mathbf{M}(\alpha)$ to α	29
$\langle\langle\, \mathbf{M},\ \mathbf{M}',\ \alpha \,\rangle\rangle$	the containment map from $\mathbf{M}(\alpha)$ onto $\mathbf{M}'(\alpha)$, when $\mathbf{M}' \overset{\sim}{\subset} \mathbf{M}$	49
$\mathbf{M}/\mathbf{M}'$	the factoring model from $\mathbf{M}'$ graphs to $\mathbf{M}$ graphs	61

W^*	strings of items from W	11
α, β, γ	generic symbols for computation graphs; usually α transforms to β and β transforms to γ	22
$\overline{\alpha}$	transitive closure of graph α	35
δ	transition function	11
Σ	finite binary strings	11
$\pi_p W$	the p element of set W	66
acyclic	when a node is not on a cycle, or a graph has no cycles, or a model produces only graphs with no cycles	37
adjusted rollback vector	the rollback vector for an event, with with the entries for the other processes replaced by the last acyclic concurrent event—usually the predecessor	110
adjusted timestamp vector	the timestamp vector for an event, with with the entries for the other processes replaced by the first acyclic concurrent event—usually the successor	110
arrive	event type: message arrives at receive queue	24
atoms	the nodes and edges of a graph	22
bounded	possessing a *unique* minimum event and a *unique* maximum event	36
component	a model that produces a well-defined subgraph of another model	56
computation graph	a labeled directed graph, describing some given computation	21
compute	event type: change state; leave message queues untouched	12
CONFIGS	process configurations	11
concrete generator	a generator that produces no ghost events	38
concurrent	when two events are incomparable in a temporal relation; also, when a multiprocess pair has the property that extrema from different processes are concurrent	27, 71

DEV-CONFIGS	device configurations	14
consistent cut	a cut that is also a timeslice	77
consistent parallel pair	a parallel pair that is transitively bounded and view-complete	109
constituency	the atoms in α that some atom in $\mathbf{M}(\alpha)$ represents	29, 29
contain	a relation between models, indicating containment of graphs via isomorphism, and containment of constituencies	48
containment map	the bijection between a subgraph and its isomorphic image	49
cut	a set of events such that each process is touched exactly once	75
cyclic	when a node lies on a cycle or when a graph contains cycles	37
$\mathcal{D}$	generic symbol for domain of a model	29
decomposition	a model consisting of a disjoint union of components of another model, with some additional properties	59
decomposition set	the set of components that comprises a decomposition	59
direct containment	model containment, when the isomorphic graphs are actually *identical*	49
direct component	the component relation, when model containment is actually direct containment	56
domain of a model	the computation graphs on which a model is defined	29
depart	event type: message departs send queue	24
DEV-NAMES	I/O device names	11
event	a discrete "thing that happens," signified by a node in a computation graph	21, 27
externally equivalent	when two atoms at the same process appear the same to a different process	100
EXTREMA	a model to collapse extrema	37
factoring model	the new model taking graphs generated by a decomposition to graphs generation by a model	61

FORCE	model to slide in-nodes of cross-process edges forward one position, thus expressing when one event in a timeslices forces the membership of another	115
$\mathcal{G}$	generic symbol for set of computation graphs	22
generator of $\mathcal{G}$	a model taking a set of graphs to $\mathcal{G}$	38
ghost	an atom that represents nothing	28
global state	the part of the ground-level computation graph representing system activity at some instant in real time	93
ground-level computation graph	the computation graph version of a system trace	25
grounding generator	a generator that generates a set of graphs from a set of ground-level graphs	38
identical	two computation graphs that completely match: edges, nodes, labels	23
idle	event type: nothing happens	32
independent	when, in a multiprocess model, each non-bounding atom represents a single atom at a single process component	109
isomorphism	two computation graphs that match, when we ignore labels	23
L	generic symbol for localization	69
LINEAR	the linear order model	32
linear time order	a “total” order that allows simultaneous events	28
LINLINE	local timeline model from LINEAR	43
LINLINES	disjoint union of process LINLINE models	45
localization	model obtained by retaining only the edges coming from the transitive multicomponent	69
locally acyclic	when any cycle must leave involve at least two processes	101
$\mathbf{M}$	generic symbol for time model	29
$\overline{\mathbf{M}}$	transitive closure of model $\mathbf{M}$	36
$\mathbf{M}'$	generic symbol for a decomposition of a model $\mathbf{M}$	59
$\mathbf{M}(\alpha)$	the graph produced by $\mathbf{M}$ given α	29

$\mathbf{M}(\mathcal{G})$	apply $\mathbf{M}$ to each guy in $\mathcal{G}$	30
$\max_\gamma$	operation on an event set from γ that retains only the relative maxima	78
MESSAGES	possible messages	11
MSG	model that makes receives follow sends	45
$\min_\gamma$	operation on an event set from γ that retains only the relative minima	78
model	time model: a representational transformation on computation graphs	29
multicomponent	the set of process component models for a multiprocess model	65
multilinear model	a multiprocess model where each process component is linear	72
multiprocess model	a model consisting of a set of process models glued together	65
multiprocess pair	a multiprocess model together with a transitive reduction of its multicomponent	65
NAMES	process and I/O device names	11
NONIDLE	model that removes *idle* events	43
pairing between α_1 and α_2	a subset of $\alpha_1 \times \alpha_2$, pairing a subgraph of the one with a subgraph of the other	23
parallel model	a concurrent multiprocess model with straight-line process graphs	72
parallel pair	a parallel model together with its multicomponent	72
PH	model inserting place-holder events	105
photo	event type: photo event in ground-level computation graph	25
POT	model for partial order time	46
$\mathcal{P}(W)$	set of all subsets of W	14
PPOT	POT with intermediate events	106
PROC-NAMES	process names	11
Q	state set for processes	11
Q_{DEV}	state set for I/O devices	14
q_0	initial state	11

R	receive queue	11
$\mathbf{R}(\gamma, \mathbf{M}, \mathbf{M}', A)$	the rollback vector of event A in graph γ from parallel pair $(\mathbf{M}, \mathbf{M}')$	87
$\mathbf{R}(A)$	the rollback vector for event A, when the graph and models are understood	87
$\mathbf{R}^*(\gamma, \mathbf{M}, \mathbf{M}', A)$	the adjusted rollback vector of event A in graph γ from parallel pair $(\mathbf{M}, \mathbf{M}')$	110
$\mathbf{R}^*(A)$	the adjusted rollback vector for event A, when the graph and models are understood	110
receive	event type: receive a message	12
representative	the atom in $\mathbf{M}(\alpha)$ that represents an atom in α	29, 29
representation map	function taking atoms in one graph to what they represent in another	29, 29
rollback vector	cut containing the minimal event from each process that follows or equal a given event	87
RPOT	restricted POT: restrict the domain so that all graphs produced are view-complete	105
S	send queue	11
send	event type: send a message	12
strong containment	model containment, when the constituencies are actually equal	49
strong component	the component relation, when model containment is actually strong containment	56
strong direct containment	strong containment and direct containment simultaneously	49
strong direct component	strong component and direct component simultaneously	56
SYNC	model synchronizing equal length total orders	46
system trace	exhaustive physical description of a computation	18
TIMELINE	local timeline with no *idle* events	45
TIMELINES	disjoint union of process TIMELINE models	45
time model	a representational transformation on computation graphs	29
timeslice	maximal set of mutually concurrent events	76

timestamp vector	cut containing the maximal event from each process that precedes or equals a given event	87
TRANS	transitive closure model	36
transitively bounded	when the transitive closure is bounded	36
$\mathbf{V}(\gamma, \mathbf{M}, \mathbf{M}', A)$	the timestamp vector of event A in graph γ from parallel pair $(\mathbf{M}, \mathbf{M}')$	87
$\mathbf{V}(A)$	the timestamp vector for event A, when the graph and models are understood	87
$\mathbf{V}^*(\gamma, \mathbf{M}, \mathbf{M}', A)$	the adjusted timestamp vector of event A in graph γ from parallel pair $(\mathbf{M}, \mathbf{M}')$	110
$\mathbf{V}^*(A)$	the adjusted timestamp vector for event A, when the graph and models are understood	110
vector	an indexed set of events, one from each process	75
view-complete	when for every transition edge at a process, an event has the same external view	101

References

[AmJa93] Amman, P. and S. Jajodia. “Distributed Timestamp Generation in Planar Lattice Networks.” *ACM Transactions on Computer Systems.* To appear.

[BiJo87] Birman, K.P. and T.A. Joseph. “Reliable Communication in the Presence of Failures.” *ACM Transactions on Computer Systems,* 5: 47-76. February 1987.

[Ch89] Chandy, K.M. *The Essence of Distributed Snapshots.* Computer Science Technical Report CS TR 89-5, Caltech. March 1989.

[ChLa85] Chandy, K.M. and L. Lamport. “Distributed Snapshots: Determining Global States of Distributed Systems.” *ACM Transactions on Computer Systems.* 3: 63-75. February 1985.

[DaPr90] Davey, B.A. and H.A. Priestley. *Introduction to Lattices and Order.* Cambridge: Cambridge University Press, 1990.

[Fi88] Fidge, C.J. “Timestamps in Message-Passing Systems That Preserve the Partial Ordering.” *11th Australian Computer Science Conference.* 56-67. February 1988.

[Fi89] Fidge, C.J. “Partial Orders for Parallel Debugging.” *ACM SIGPLAN Notices.* 24: 183-194. January 1989.

[Fi91] Fidge, C.J. “Logical Time in Distributed Computing Systems.” *IEEE Computer.* 24 (8):28-33. August 1991.

[Gr75] Greif, I.G. *Semantics of Communicating Parallel Processes.* Ph.D. thesis, Massachusetts Institute of Technology. 1975.

[HLMW87] Herlihy, M.P., N. Lynch, M. Merritt and W. Weihl. *On the Correctness of Orphan Elimination Algorithms.* Computer Science Technical Report MIT LCS TM-329, Massachusetts Institute of Technology. 1987.

[Je85] Jefferson, D.R. “Virtual Time.” *ACM Transactions on Programming Languages and Systems.* 7: 404-425. July 1985.

[Jo89] Johnson, D.B. *Distributed System Fault Tolerance Using Message Logging and Checkpointing.* Ph.D. thesis, Rice University, 1989.

[JoZw90] Johnson, D.B. and W. Zwaenepoel. “Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing.” *Journal of Algorithms.* 11: 462-491. September 1990.

[La78] Lamport, L. “Time, Clocks, and the Ordering of Events in a Distributed System.” *Communications of the ACM*. 21: 558-565. July 1978.

[Ma87] Mattern, F. “Algorithms for Distributed Termination Detection.” *Distributed Computing.* 2: 161-175. 1987.

[Ma89] Mattern, F. “Virtual Time and Global States of Distributed Systems.” In Cosnard, et al, ed., *Parallel and Distributed Algorithms.* Amsterdam: North-Holland, 1989. 215-226.

[Ma93] Mattern, F. “Efficient Algorithms for Distributed Snapshots and Global Virtual Time Approximation.” *Journal of Parallel and Distributed Computing.* 18: 423-434. August 1993.

[PBS89] Peterson, L.L., N.C. Bucholz and R.D. Schlichting. “Preserving and Using Context Information in Interprocess Communication.” *ACM Transactions on Computer Systems*. 7: 217-246. August 1989.

[PeKe93] Peterson, S.L. and P. Kearns. “Rollback Based on Vector Time.” *12th Symposium on Reliable Distributed Systems.* IEEE, October 1993.

[Pr86] Pratt, V.R. “Modeling Concurrency with Partial Orders.” *International Journal of Parallel Programming.* 15 (1): 33-71. 1986.

[ReGo93] Reiter, M. and L. Gong. “Preventing Denial and Forgery of Causal Relationships in Distributed Systems.” *1993 IEEE Symposium on Research in Security and Privacy.*

[SiKs90] Singhal, M. and A.D. Kshemkalyani. *An Efficient Implementation of Vector Clocks.* Computer Science Technical Report TR OSU-CISRC-11/90-TR34, Ohio State University. November 1990.

[Sm91] Smith, S.W. *Secure Clocks for Partial Order Time.* Thesis proposal, School of Computer Science, Carnegie Mellon University. October 30, 1991. (See [SmTy91].)

[Sm94] Smith, S.W. *Distributed Time: Mechanisms, Protocols and Security.* Ph.D. thesis, School of Computer Science, Carnegie Mellon University. (In preparation, to appear in Summer 1994.)

[SmTy91] Smith, S.W. and J.D. Tygar. *Signed Vector Timestamps: A Secure Protocol for Partial Order Time.* Computer Science Technical Report CMU-CS-93-116, Carnegie Mellon University. October 1991; version of February 1993. (The majority of [SmTy91] is drawn verbatim from [Sm91].)

[SmTy93] Smith, S.W. and J.D. Tygar. *Sealed Vector Timestamps: Privacy and Integrity for Partial Order Time.* Submitted for publication. November 15, 1993.

[Sp89] Spezialetti, M. *A Generalized Approach to Monitoring Distributed Computations for Event Occurrences.* Ph.D. thesis, University of Pittsburgh, 1989.

[StYe85] Strom, R. and S. Yemini. “Optimistic Recovery in Distributed Systems.” *ACM Transactions on Computer Systems.* 3: 204-226. August 1985.

[TaLo91] Tay, Y.C. and W.T. Loke. *A Theory for Deadlocks.* Computer Science Technical Report CS-TR-344-91, Princeton University. August 1991.