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.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of either Wright Laboratory or the U.S. Government.

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.

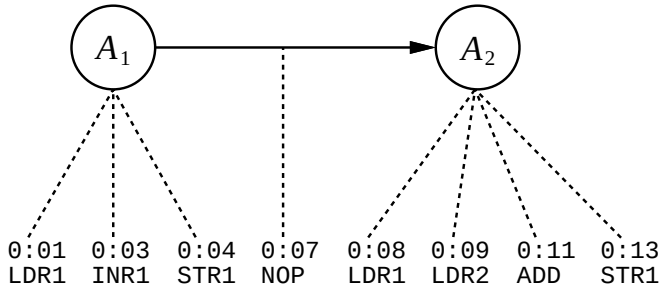

Figure 1.2 We may abstract away from the physical description by bundling basic physical events into computational events. The detailed machine code becomes "event A_1, then event A_2."

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.)

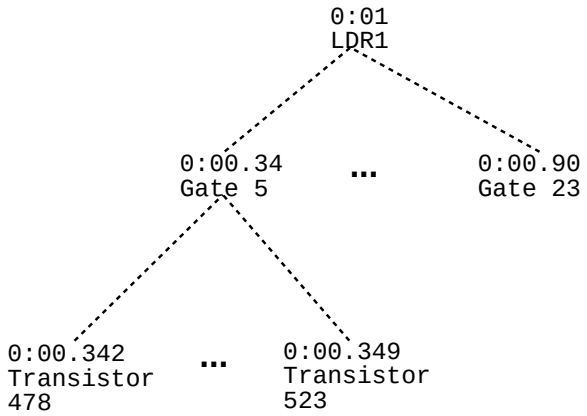

Figure 1.3 The physical description is itself an abstraction: each instruction may represent gate firings or transistor actions. The level of description we choose for our base is essentially arbitrary.

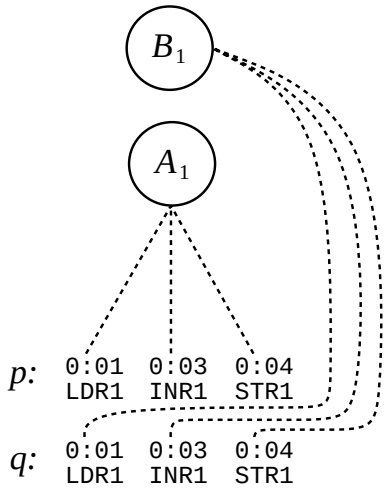

Figure 1.4 Abstract events A_1 and B_1 represent genuinely simultaneous computation; we regard these events as concurrent.

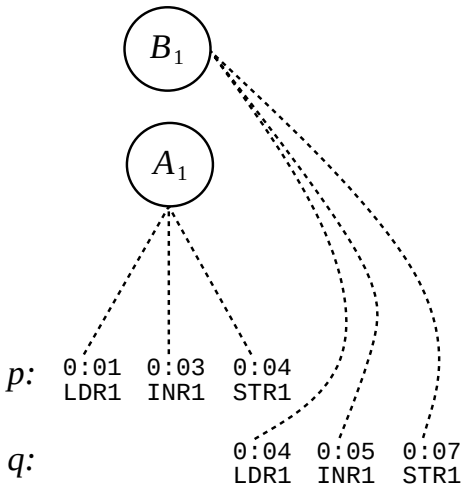

Figure 1.5 Abstract events A_1 and B_1 now represent computation that only "appears" simultaneous; nevertheless, we still regard these events as concurrent.

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.

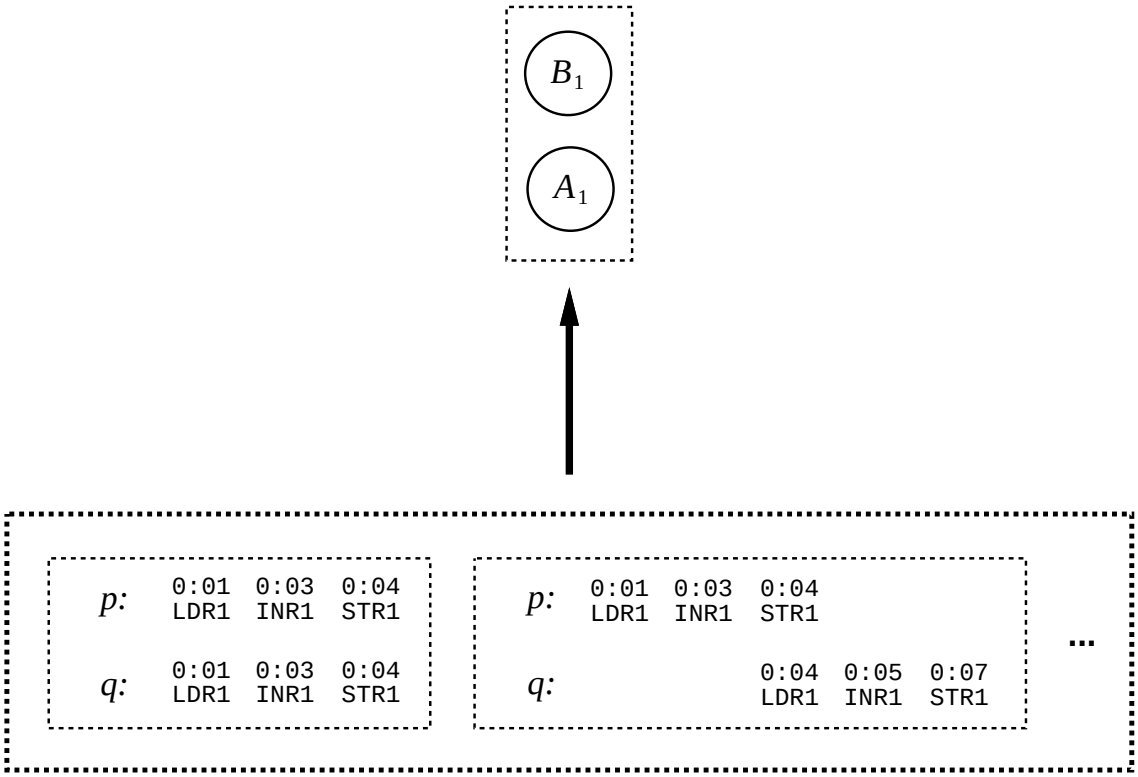

Figure 1.6 An abstract computation graph represents a set of possible physical computations. Once we abstract to the graph, we forget the presumably irrelevant physical detail.

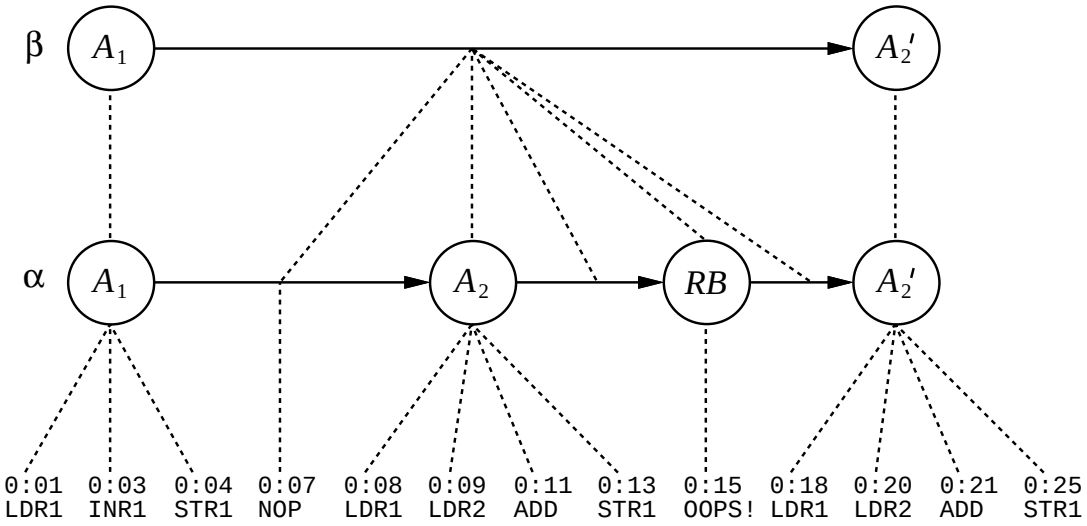

Figure 1.7 Rollback induces two levels of abstraction. We prune away irrelevant machine details to obtain description α; however, we presumably want to prune away irrelevant rollback details to obtain the "real" description β.

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 **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

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

such that

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

and

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

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:

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

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.

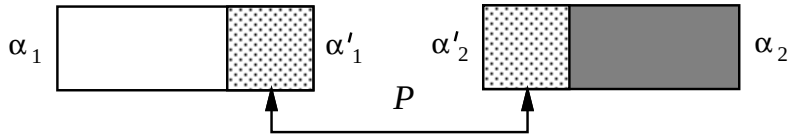

Figure 4.1 The pairing enumerating the identification between two identical subgraphs also specifies the subgraphs. Here, each $\alpha'_i \subset \alpha'_i$, and $\alpha'_1 \equiv_P \alpha_2$. However, P is a pairing not only between the subgraphs α'_i but is also a pairing between the graphs α_i. Given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

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

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

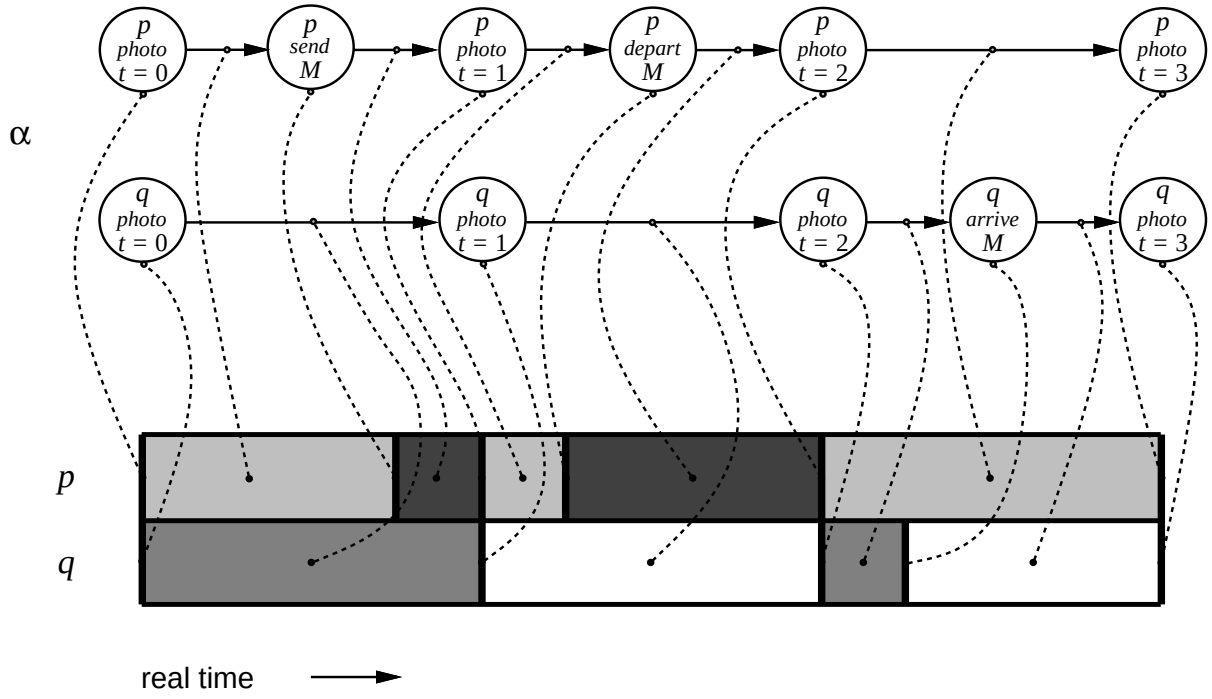

Figure 4.2 A ground-level computation graph represents computational space-time. Each atom in the ground-level graph α represents activity at process p or q at some point or interval of real time.

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:

- $A \rightleftharpoons B$ 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 $B \longrightarrow A$
- $A \Longleftrightarrow B$ 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.

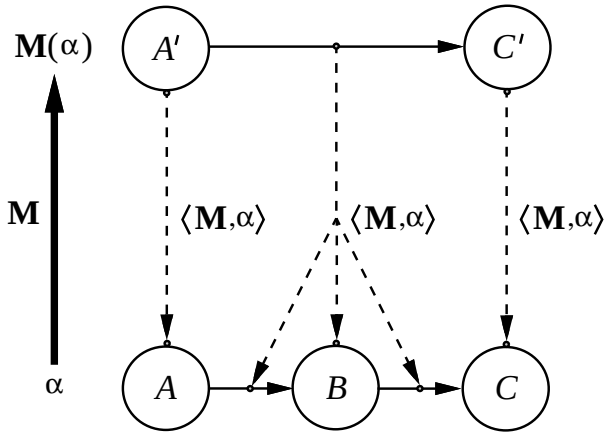

Figure 4.3 Model M transforms computation graph α to computation graph $\mathbf{M}(\alpha)$. The representation map $\langle\, \mathbf{M},\ \alpha \,\rangle$ takes each atom of $\mathbf{M}(\alpha)$ back to the set of atoms in α it represents. The bold arrow indicates the action of $\mathbf{M}$; the dashed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

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

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

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

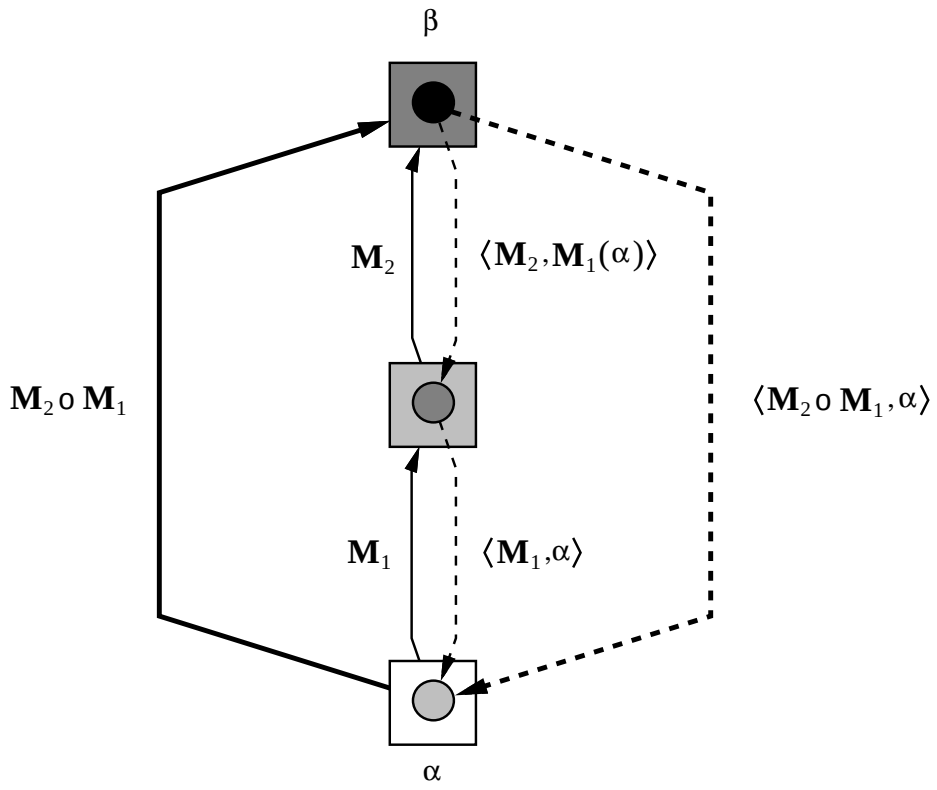

Figure 4.4 To obtain $\beta = (\mathbf{M}_2 \circ \mathbf{M}_1)(\alpha)$, we transform α according to $\mathbf{M}_1$, and then transform the result according to $\mathbf{M}_2$. To figure out what an atom of β represents in α, we obtain the set of atoms it represents in $\mathbf{M}_1(\alpha)$, and then 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, α $\rangle$ 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.

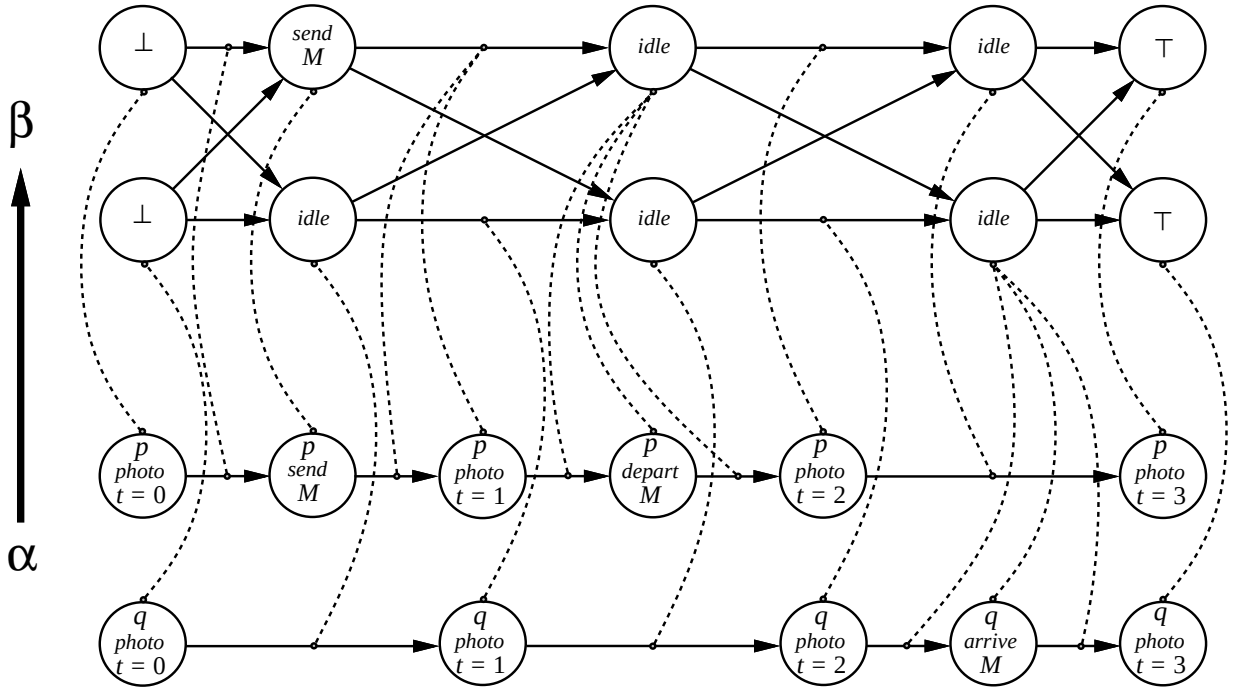

Figure 4.5 We obtain β by applying LINEAR to the simple ground-level computation graph α. Dashed lines connect each atom in β to the atoms it maps to under $\langle \text{LINEAR}, \alpha \rangle$.

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.

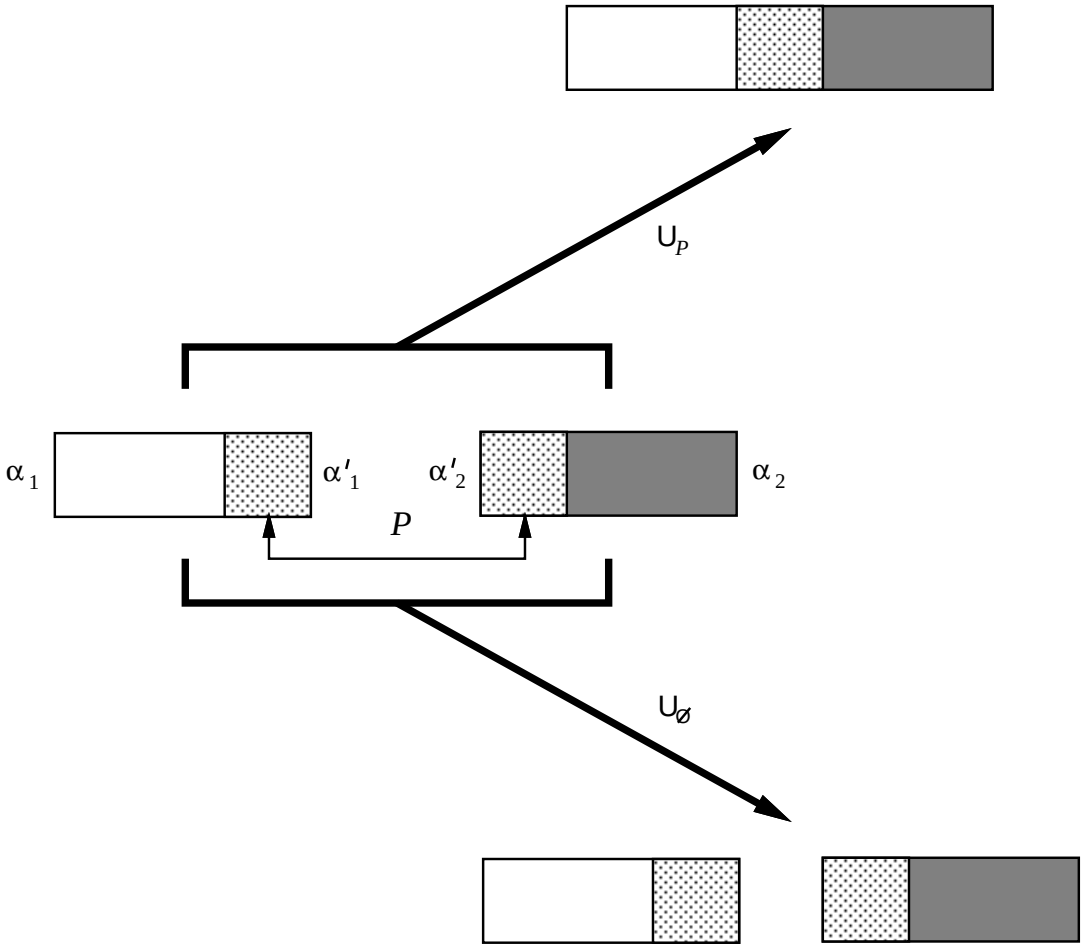

Figure 5.1 Suppose two computation graphs α_1 and α_2 have identical subgraphs (α'_1 and α'_2, respectively), matched by pairing P (left). We obtain the union $\alpha_1 \cup_P \alpha_2$ by merging the α'_i according to the pairing P (top right); we obtain the disjoint union $\alpha_1 \cup_\emptyset \alpha_2$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

(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.

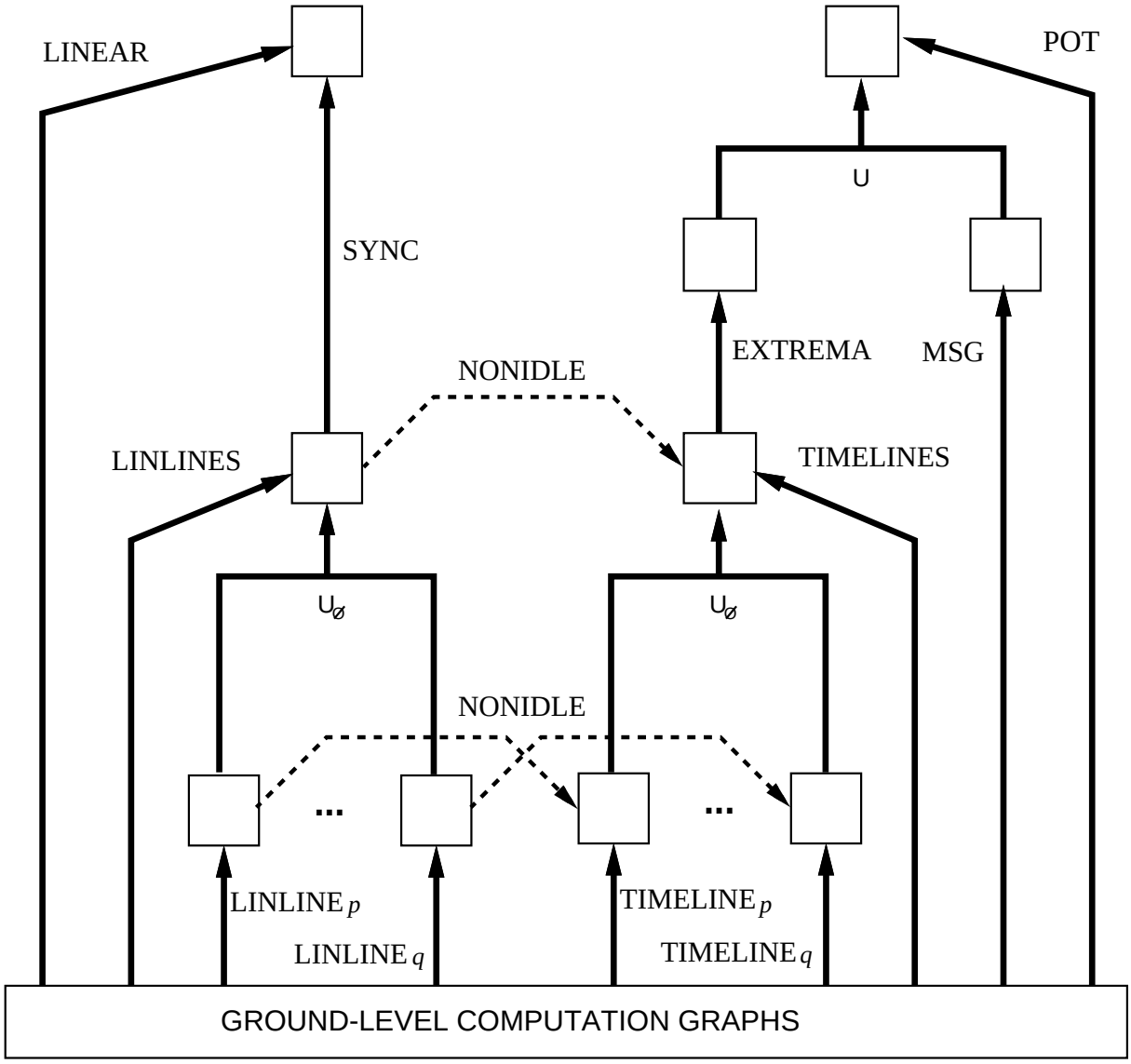

Figure 6.1 The models we discuss here fit into a composition hierarchy. The boxes indicate sets of computation graphs; an arrow **M** between two boxes indicates that model **M** is a surjection from the one set onto the other. What's more, the functional identities we illustrate here are also model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

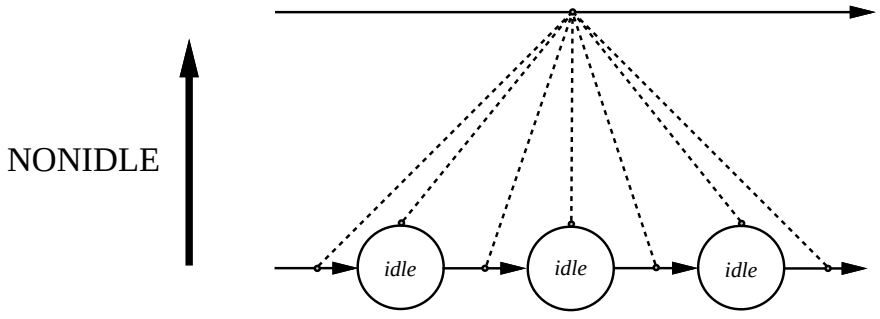

Figure 6.2 The model NONIDLE replaces maximal chains of *idle* nodes by a representative edge.

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

$$\mathbf{M}_1 \quad \tilde{\subset} \quad \mathbf{M}_2$$

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 ($\tilde{\subset}$) contains two elements, suggesting *isomorphism* ($\cong$) and *subgraph* ($\subset$). These concepts characterize containment: $\mathbf{M}_1 \tilde{\subset} \mathbf{M}_2$ 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 $\mathbf{M}_1 \tilde{\subset} \mathbf{M}_2$, 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 $\mathbf{M}_1 \widetilde{\subset} \mathbf{M}_2$.

- 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

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

- 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

$$\mathbf{M}_1 \quad \widetilde{\subseteq} \quad \mathbf{M}_2$$

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

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

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. $\mathbf{M}_1 \overline{\subseteq} \mathbf{M}_2 \implies \mathbf{M}_1 \widetilde{\subseteq} \mathbf{M}_2$
2. $\mathbf{M}_1 \overline{\subseteq} \mathbf{M}_2 \implies \mathbf{M}_1 \overline{\subset} \mathbf{M}_2$

Proof Condition 1′ implies Condition 1, and Condition 2′ implies Condition 2. □

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

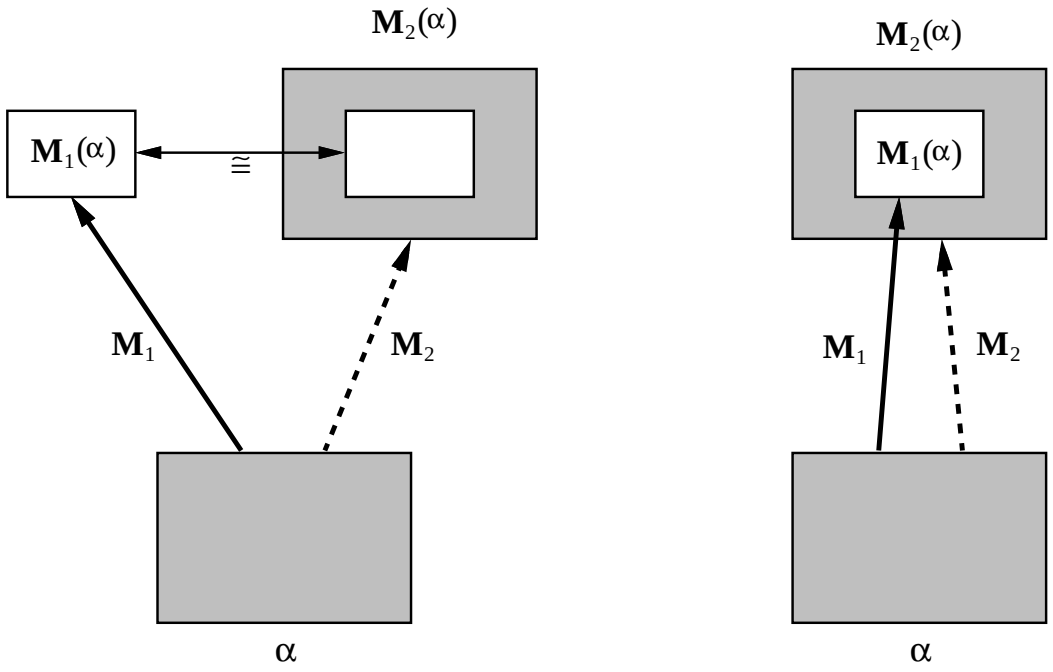

Figure 7.1 Containment of any form requires that one model always produces a graph isomorphic to a subgraph of what another model produces. However for direct containment, this isomorphism is in fact the identity. The left diagram shows ordinary containment: $\mathbf{M}_1 \tilde{\subset} \mathbf{M}_2$;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

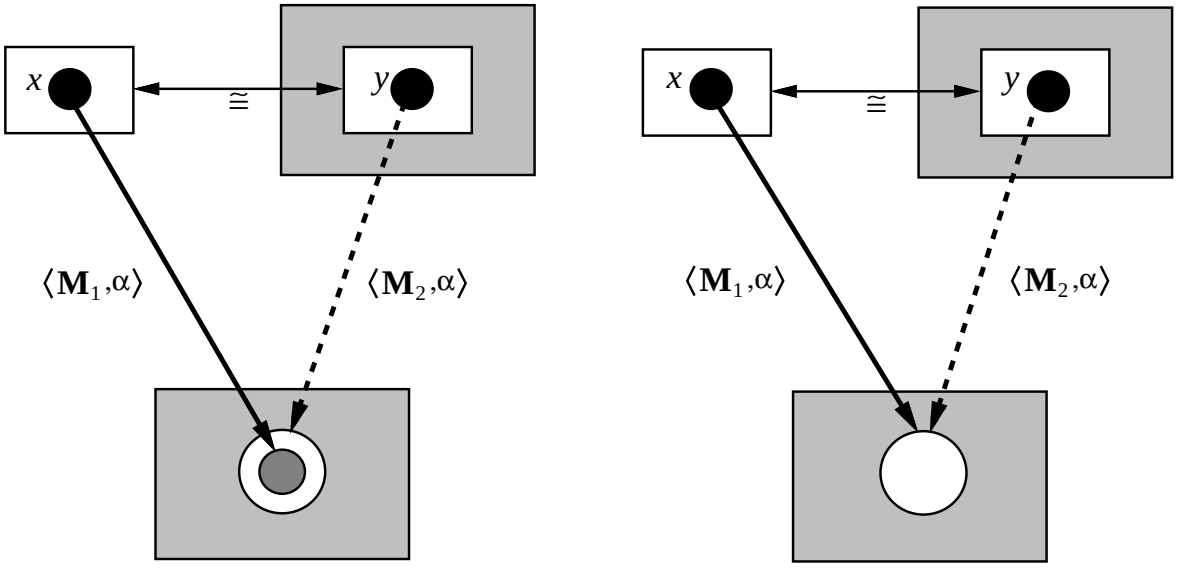

Figure 7.2 If $\mathbf{M}_1 \tilde{\subset} \mathbf{M}_2$, then the M_1 representation on any atom x yields a subset of what the M_2 representation yields on the matching atom y (left). For strong containment $\mathbf{M}_1 \tilde{\subseteq} \mathbf{M}_2$, the representations are equal (right).

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

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

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.

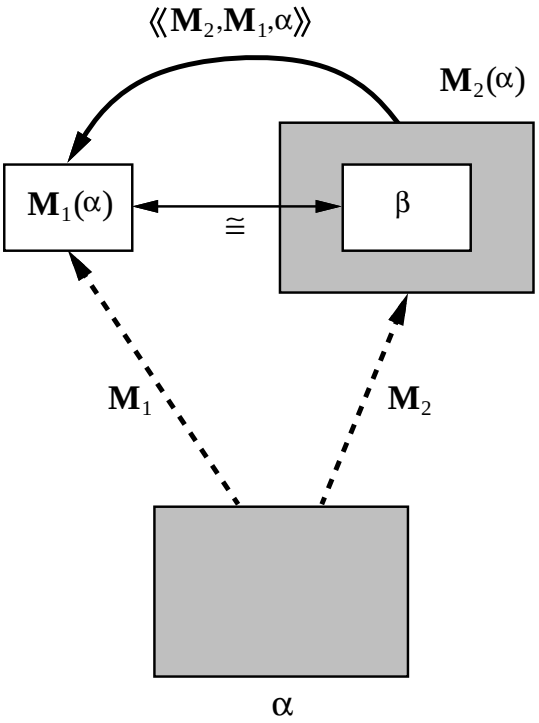

Figure 7.3 When one model contains another, a unique pairing connects the graphs they produce. Here we see that $\mathbf{M}_1 \cong \mathbf{M}_2$, so $\mathbf{M}_1(\alpha)$ is isomorphic to a subgraph β of $\mathbf{M}_2(\alpha)$. The containment map $\langle\langle\, \mathbf{M}_2,\ \mathbf{M}_1,\ \alpha\, \rangle\rangle$ 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 $\mathbf{M}_1 \widetilde{\subset} \mathbf{M}_2$ 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 $\mathbf{M}_1 \widetilde{\subset} \mathbf{M}_2$ 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:
>
> $$\begin{aligned} A_1 \rightleftharpoons B_1 \quad &\Longrightarrow \quad A_2 \rightleftharpoons B_2 \\ A_2 \not\leftrightarrow B_2 \quad &\Longrightarrow \quad A_1 \not\leftrightarrow B_1 \end{aligned}$$

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 $\mathbf{M}_1, \mathbf{M}_2$:
>
> $$\begin{aligned} \mathbf{M}_1 \widetilde{\subset} \mathbf{M}_2 \quad &\Longrightarrow \quad \overline{\mathbf{M}_1} \widetilde{\subset} \overline{\mathbf{M}_2} \\ \mathbf{M}_1 \overline{\subset} \mathbf{M}_2 \quad &\Longrightarrow \quad \overline{\mathbf{M}_1} \overline{\subset} \overline{\mathbf{M}_2} \end{aligned}$$

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**:

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

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 $\mathbf{M}_1 \triangleright \mathbf{M}_2$.

Clearly, refinement is transitive.

The relation $\mathbf{M}_1 \triangleright \mathbf{M}_2$ induces a function from $\mathbf{M}_1(\mathcal{D})$ to $\mathbf{M}_2(\mathcal{D})$): we write $\beta_1 \triangleright \beta_2$ 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 $\mathbf{M}_1, \mathbf{M}_2$:

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

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**:

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

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

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

7.3. Components

Suppose $\mathbf{M}_1 \widetilde{\subset} \mathbf{M}_2$, 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

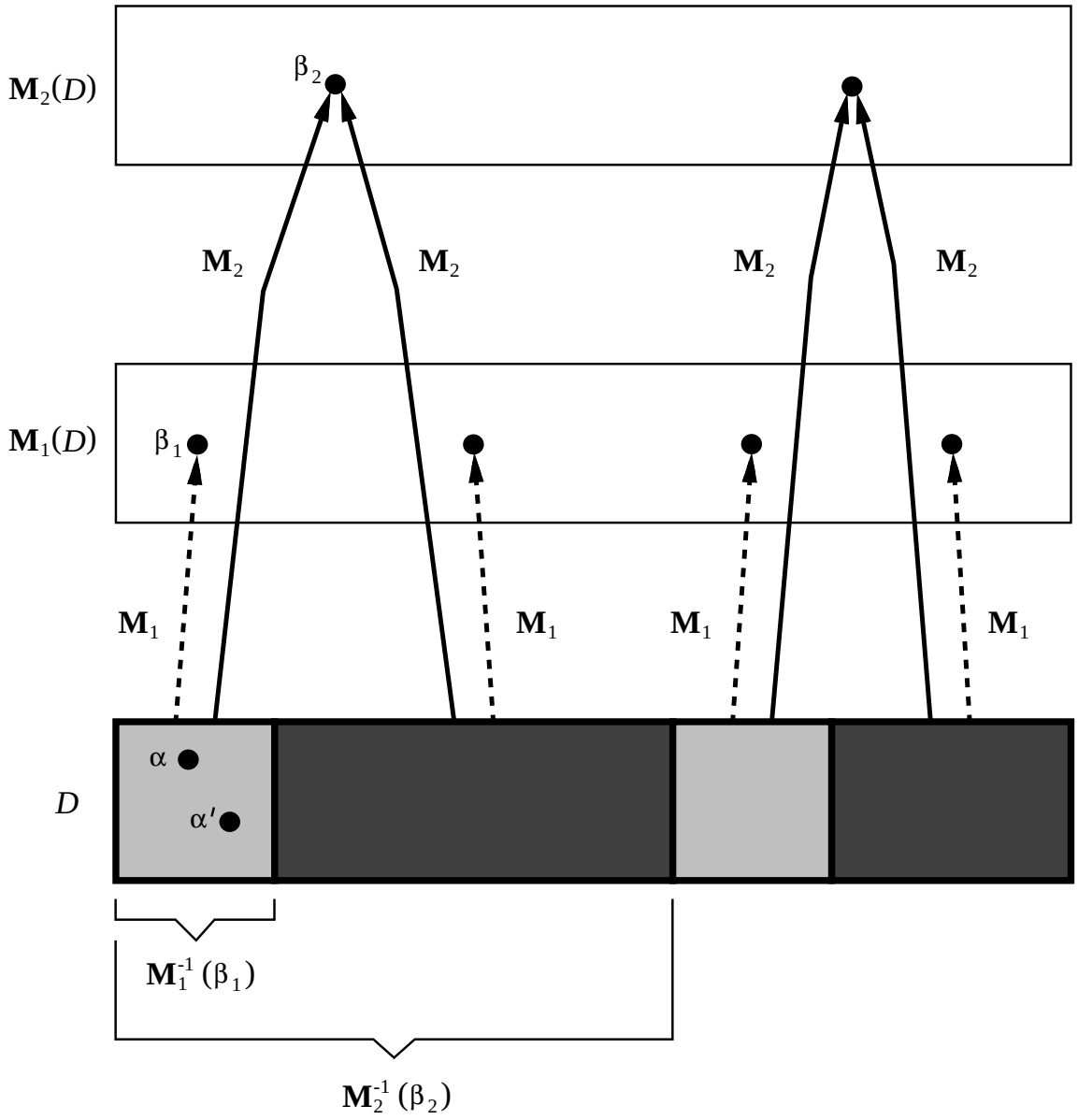

Figure 7.4 This diagram illustrates refinement: $\mathbf{M}_1 \triangleright \mathbf{M}_2$. The dashed arrows indicate the action of $\mathbf{M}_1$; the solid arrows indicate the action of $\mathbf{M}_2$. We see that when $\mathbf{M}_1$ identifies two graphs (for example, α and α') by taking them to 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 $\beta_1 \triangleright \beta_2$ to describe this relationship.

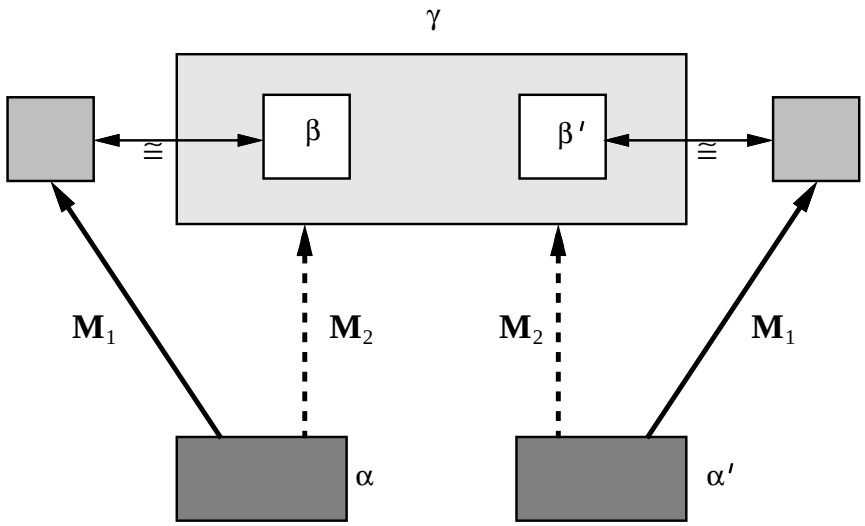

Figure 7.5 Containment does not guarantee well-defined components. Although $\mathbf{M}_1 \mathrel{\widetilde{\subset}} \mathbf{M}_2$ model $\mathbf{M}_1$ may be isomorphic to different subgraphs depending on the original graph. Here, $\gamma = \mathbf{M}_2(\alpha) = \mathbf{M}_2(\alpha')$ but β, the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

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

Talking unambiguously about the $\mathbf{M}_1$ component of a graph generated by $\mathbf{M}_2$ requires both containment and refinement:

Definition 7.12 Suppose $\mathbf{M}_1$ and $\mathbf{M}_2$ act on the same domain. $\mathbf{M}_1$ is a *component* of $\mathbf{M}_2$

$$\mathbf{M}_1 \quad \widetilde{\sqsubset} \quad \mathbf{M}_2$$

when $\mathbf{M}_1 \mathrel{\widetilde{\subset}} \mathbf{M}_2$ and $\mathbf{M}_2 \rhd \mathbf{M}_1$.

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

Definition 7.13 Suppose $\mathbf{M}_1$ and $\mathbf{M}_2$ act on the same domain.

- If $\mathbf{M}_1 \mathrel{\overline{\subset}} \mathbf{M}_2$ and $\mathbf{M}_2 \rhd \mathbf{M}_1$, then $\mathbf{M}_1$ is a *direct component* of $\mathbf{M}_2$:

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

- If $\mathbf{M}_1 \mathrel{\widetilde{\subseteq}} \mathbf{M}_2$ and $\mathbf{M}_2 \rhd \mathbf{M}_1$, then $\mathbf{M}_1$ is a *strong component* of $\mathbf{M}_2$:

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

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

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

Each of these relations is transitive.

Informally, $\mathbf{M}_1 \widetilde{\sqsubseteq} \mathbf{M}_2$ 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 $\mathbf{M}_1, \mathbf{M}_2$:

$$\begin{array}{lcl} \mathbf{M}_1 \widetilde{\sqsubseteq} \mathbf{M}_2 & \Longrightarrow & \overline{\mathbf{M}_1} \widetilde{\sqsubseteq} \overline{\mathbf{M}_2} \\ \mathbf{M}_1 \overline{\sqsubseteq} \mathbf{M}_2 & \Longrightarrow & \overline{\mathbf{M}_1} \overline{\sqsubseteq} \overline{\mathbf{M}_2} \end{array}$$

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**:

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

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"

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

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 $\widetilde{\sqsubset}$ POT because TIMELINES $\widetilde{\supset}$ 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 $\mathbf{M}_1 \widetilde{\sqsubset} \mathbf{M}$. 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.