On the Representation of Distributed Behavior

Christopher Shaver

On the Representation of Distributed Behavior

by

Christopher Daniel Shaver

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Edward A. Lee, Chair
Professor Marjan Sirjani
Adjunct Professor Stavros Tripakis
Professor John Steel

Fall 2016

On the Representation of Distributed Behavior

Abstract

On the Representation of Distributed Behavior

by

Christopher Daniel Shaver

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Edward A. Lee, Chair

Technologies pervasive today have enabled a plethora of diverse networked devices to proliferate in the market. Among these devices are sensors, wearables, mobile devices, and many other embedded systems, in addition to conventional computers. This diverse tapestry of networked devices, often termed the *Internet of Things* (IoT) and the *Swarm*, has the potential to serve as a platform for a new breed of sophisticated distributed applications that leverage the ubiquity, concurrency, and flexibility of these devices, often to integrate the similarly diverse information to which these devices have access. These kinds of applications, which we are calling *Highly Dynamic Distributed Applications* (HDDAs), are particularly important in the domain of automated environments such as smart homes, buildings, and even cities.

In contrast with the kind of concurrent computation that can be represented with petri nets, synchronous systems, and other rigid models of concurrent computation, applications running on networks of devices such as the IoT demand models that are more dynamic, semantically heterogeneous, and composable. Because these devices are more than just peripherals to central servers, themselves capable of significant amounts of computation, HDDAs can involve the establishment of direct connections between these devices to share and process information. From the perspective of the platform this will look like an ever-evolving network of dynamic configuration and communication that may have no clear sense of a center, or even a central point of observation. Computation in this fashion begins to look less like a sequence of coherent states and more like a physical interaction in space.

Designing and reasoning about these kinds of applications in a rigorous and scalable fashion will require the development of new programming language semantics, specification logics, verification methods, and synthesis algorithms. At the core of all of these components of a robust programming model will be a need for an appropriate mathematical representation of behavior, specifying precisely what happens in these spaces of devices during the execution of a HDDA. This behavioral representation must characterize an application in as much detail as is necessary, without having to introduce details regarding the realization of the execution on a particular platform, bound to a specific architecture of concrete pro-

cess and communication relationships. This representation must also be itself as scalable, modular, and composable as the distributed computations it describes.

The aim of this thesis is to identify the mathematical representation of behavior best fit to meet the challenges HDDAs pose to formal semantics and formal verification. To this end, this thesis proposes a new mathematical representation of concurrent computational behavior called an *Ontological Event Graph* (OEG), which generalizes, subsumes, or improves upon other representations of concurrent behavior such as *Mazurkiewicz Traces*, *Event Structures*, and *Actor Event Diagrams*. In contrast with many other representations of concurrent behavior, which are either graphs or partial orders, OEGs are, in essence, acyclic ported block diagrams. In these diagrams, each ported block represents an *event* and each connection between ports represents a direct *dependency* between events, while each port around a block represents the specific kind of information or influence consumed or produced in the *event*. OEGs have the advantage over other representations of concurrent behavior of being both modular and composable, as well as possessing a clear concept of hierarchy and abstraction.

The motivations and reasoning behind this choice of representation will be developed from two directions. The first will be an exploration of the context, consisting of networks of devices such as the IoT. The potential of these networks to serve as a distributed computational platform for HDDAs will be understood through the construction of a conceptual application model that will be called a *Process Field*. The demands of this application model and some concrete examples will be used to construct an intuitive picture of how the corresponding behavioral representation must look, and what it would need to serve the purposes of constructing semantics and performing verification in the context of HDDAs.

The second direction will reflect on the mathematical details of two classes of sequential representations of behavior, sequences and free monoids, and look at the ways these two classes generalize from the sequential case to the concurrent one. This will suggest, as potential generalizations of the two classes, *generalized sequences* and *free monoidal categories*. The former of these classes already has many instances prevalent in the literature. These existing representations based on generalized sequences, along with some other similar established means to represent concurrent behavior, will be reviewed in detail, and reasons will be given for why they do not meet the demands for modularity and composability needed for reasoning about HDDAs.

It will be concluded that the latter, far less investigated class of free monoidal categories offers a richer mathematical structure that is more appropriate for the demands of constructing formal models of HDDAs. In particular, free monoidal categories offer a very general form of composition involving both a parallel and a sequential product. These products, along with a collection of distinguished constants, form an algebra that can be used to construct semantics or to formulate specifications and prove properties of behaviors. Time will be taken to first review the concepts from category theory needed to define these structures, particularly focusing on *monoidal categories* and the concept of free structures and universal properties. The construction of free monoidal categories, and the particular variant of *free symmetric monoidal categories*, will then be detailed, and several important properties of these structures will be shown and discussed.

It will ultimately be established that spaces of OEGs defined in this thesis, along with their compositions, form free symmetric monoidal categories. Consequently, the study of OEGs can leverage all of the mathematical tools of monoidal category theory to study and better understand distributed behavior. A chapter will be devoted to showing specifically how the π-*calculus* can be given a particularly elegant and powerful behavioral semantics using OEGs.

To my Kitty.

Contents

List of Figures

Acknowledgments

I would like to extend my sincerest gratitude to everyone who supported me along the complex course that culminated in this thesis. I would first like to thank my two advisors, Professors Edward A. Lee and Stavros Tripakis. Upon my arrival at Berkeley, Professor Lee offered me a position in his research group, and since then, I have had the incredible opportunity to work with him researching the semantics of Models of Computation. Edward has always been supportive of my work and has taken the time to give me tremendously valuable feedback and to point me in the right direction. As a student of Edward's, I have been impressed by his curiosity, intensity, creativity, and prolific productivity as a researcher and a teacher.

I first met Stavros upon my entrance into Edward's research group. Stavros was a researcher at the time, and became the first person I worked closely with as a graduate research assistant. Stavros, in many ways, was the person who introduced me to the study of the semantics of Models of Computation. Over the subsequent years Stavros generously offered me many incredible opportunities to participate in research projects. For much of my time at Berkeley he has been the first person with whom I can discuss my ideas, particularly those that have been more difficult to communicate. I was honored to become Stavros's first official graduate student when he became a professor.

I am perhaps most greatly in debt to Professor Marjan Sirjani who inspired this work and believed in me while I pursued it through so many bouts of uncertainty and self-doubt. This work emerged from a tremendously powerful collaboration between Marjan and I, beginning during the summer of 2015, to better understand how to approach some challengingly complex systems of networked devices from a perspective of formal verification. Marjan encouraged me to pursue this work as the focus of my thesis, and has supported me throughout the process of writing it. I could not have done this without her.

I would like to thank my outside committee member Professor John Steel for teaching me computability theory. I took a class with him on the subject that influenced me greatly and made me feel comfortable enough with formal mathematics to pursue some of the more challenging work in this thesis. This was the first class I completed that required me to formally prove theorems in detail. This was not only an essential skill that I learned, but a powerfully deep and profound craft with which I have managed to gain some acquaintance.

I would like to thank all of the other faculty members who gave me opportunities to do compelling research. Among them, I would like to give a special thanks to Professor Alain Girault who offered me the opportunity to speak about my work on synchronous systems at Dagstuhl, and to do research with him at INRIA in Grenoble. I would also like to give a special recognition to Professor Reinhard von Hanxleden, who gave me the opportunity to do research at Kiel University.

I would also like to thank all of the other students, researchers, and faculty who have worked with me. I would like to thank Marten Lohstroh for working with me on several projects that set the stage for this work and pointing me in the direction of Hewitt's ideas on concurrency, which influenced me greatly. I would like to thank Christian Motika, Christos Stergiou, Michael Zimmer, David Broman, Matthew Weber, Armin Wasicek, and Ilge Akkaya for collaborating with me on projects, as well as fellow students and researchers with whom I had influential discussions and exchanges, namely Jan Reineke, Patricia Derler, Gil Lederman, Dai Bui, Ben Zhang, Hokeun Kim, Tomi Räty, Fabio Cremona, Marcell Vazquez-Chanlatte, Ben Caulfield, Alexandre Donze, Antonio Iannopollo, Pierluigi Nuzzo, Tommaso Dreossi, John Finn, and many others I have had the privilege to encountered at Berkeley and other institutions. A particular recognition is owed to Hugo Andrade, who always said good things about my presentations, despite others' convictions that they were difficult to follow.

Finally, I would like to thank all of my friends and loved ones for their support during my time as a graduate student. I would especially like to thank my partner Jules, who has shown me great love, and has given me strength and confidence in my work and in myself.

Chapter 1

Introduction

As they have become more pervasive and developed, computer networks have shifted from being centered around proper computers themselves, such as servers and desktops, to encompassing a plethora of diverse, heterogeneous computational and informational devices. Amongst these devices are sensors, actuators, smart phones, appliances, wearables, and many other kinds of embedded systems, particularly those endowing everyday objects with the sufficient faculties of intelligence and communication to be called and marketed as *smart*. Furthermore, the set of scales of more conventional computers, running typical operating systems and computing with traditional computer architectures, has become more diverse. Small form-factors of inexpensive, easily networked computers have made it possible for a rapid increase in the number of individual computing devices that could be found in a home or workplace, networked together, often through wireless networks.

The purposes of these ubiquitous devices are as diverse as they themselves are, but a prevalent example of such a purpose is to interact with the conditions of an environment, sensing or actuating things like light levels, temperature, the presence of people in various states of affairs, the state of doors and windows, the condition of utilities such as power and water, the state of appliances such as refrigerators and stoves. Wearable sensors can incorporate information about human bodies, monitoring health, movement, and even emotional affects. Indeed, these things can all affect each other, and often this is both the key aim and challenge of constructing an automated (smart) environment.

Some of these devices, prior to being direct participants in networks, were previously only peripherals to traditional computers or servers, and controlled or accessed through central processes on their hosts. This is particularly true of sensors and actuators, which may have only been available to a network via high-level processes running on a host computer. A typical application might involve the use of all of these elements, but the application itself would only be a traditional program running on a central host, communicating with these peripherals, and thereby mediating their interaction with each other. Interfacing with these devices, in this case, becomes a matter of constructing the system, having the right drivers, connecting cables, or picking up the right dedicated signals. The topology of communication that comes from such a system is a static one, and applications might assume in their semantics that, aside from errors, the connections remain consistent over the course of execution. Consequently, the kinds of applications written for such systems could be conceived of easily in familiar computational paradigms, languages, and semantics that were designed largely for sequential computation, perhaps with additions to accommodate simple forms of concurrency.

However, today many of these same kinds of devices are autonomous and have more significant computational resources of their own, and can often communicate via high-level protocols such as HTTP or UDP, or more specifically through RESTful interfaces and open APIs. Having these devices directly connected to networks, and broadly capable of forming connections with each other, opens the door to the possibility of highly distributed and dynamic applications that do not have a proper center to them. This new network of diverse, often autonomous, heterogeneous devices constitutes a new platform for computing which is being termed the IoT or the Swarm[35].

Utilizing the IoT or the Swarm (not to mention the "cloud" as well) as a platform for concurrent applications, developers can construct applications that are deployed into the Swarm, linking together a number of devices and coordinating their execution to perform some task that leverages the relationship that the involved devices have with the physical world. Many such distributed applications form *Cyber-Physical Systems* (CPSs) that process inputs from sensors and produce effects through a variety of actuators in a manner sensitive to space and time, involving feedback loops through the physical world. The challenge to application writers and computer scientists working on this frontier is to handle this broad decentralization of control, developing systems that do not rely on a central point of reference or synchrony to achieve complex aims, all while (and perhaps this is the most challenging dimension) having the components of these applications and their relationships dynamically changing over the course of execution. We will use the term *Highly Dynamic Distributed Application* (HDDA) to describe the kind of application we are envisioning running on such a platform.

1.1 Behavioral Representations

Distributed applications can readily be developed today for the IoT and the Swarm through ad-hoc methods, tailored to specific cases, and their behavior can reasoned out exhaustively or approximately. But, the increasing pervasiveness, scale, and complexity of applications that we are calling HDDAs demand a sophisticated system design model with appropriate languages, design tools, well-defined semantics, formal specifications, mathematical representations of behavior, analysis methods, verifications, and even synthesis algorithms. In other words, it will inevitably be necessary to write these applications in a manner that lends itself to establishing certain assurances about what ends up happening when they are run on such a complex and dynamic platform.

It is inevitable that many languages and tools will be created for this task, along with many formal methods. However, one might argue that the glue that ties all of these di-

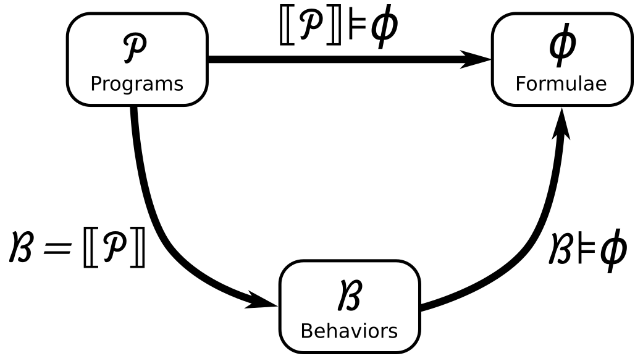

$\mathcal{P}$
Programs
$[\![\mathcal{P}]\!] \models \phi$
ϕ
Formulae
$\mathcal{B} = [\![\mathcal{P}]\!]$
$\mathcal{B} \models \phi$
$\mathcal{B}$
Behaviors

Figure 1.1: The relationship between programs, formulae, and behaviors.

mensions of a system design model together is the mathematical representation of behavior. This representation is what describes, in essence, what happens during an execution of a system. The importance of this particular piece in the greater puzzle of how to deal with computation on platforms like the IoT and the Swarm is highlighted by Abramsky in [3], while addressing the fundamental question *What do we compute?*. Abramsky points out that while traditional models of computation establish the meaning of a computing system as a function (for instance, $\mathbb{N} \to \mathbb{N}$), "in much of contemporary computing ... the purpose of the computing system is to exhibit certain behavior." To emphasize this, Abramsky asks provocatively "what does the internet compute?"

In interactive, let alone distributed platforms, behavior is central in understanding both the semantics of languages and the meaning of specifications. The relationship between *programs* in programming languages or models, *formulae* in specification logics, and *behaviors* in a behavioral representation can be summarized in the diagram shown in Figure 1.1, which depicts the relationship between these domains. Between programs and behaviors, the relationship is captured in a semantics, which determine for any given program $\mathcal{P}$ a behavior or set of behaviors $[\![\mathcal{P}]\!]$ that describe what happens when the program is executed. Between behaviors and formulae, the relationship is captured in the modeling relation, which determines for any given behavior $\mathcal{B}$ and formula ϕ whether the specification given by the formula holds for the behavior $\mathcal{B} \models \phi$.

The final relation between programs and formulae characterizes verification methods and other program analysis tools that confirm or deny that a particular program $\mathbf{P}$ has a property specified by a formula ϕ. On one hand, this relation can be defined by the other two, specifically in the form $[\![\mathcal{P}]\!] \models \phi$, and assuredly this must declaratively hold granted that the other two already exist. On the other hand, in a practical setting, an analysis algorithm can rarely proceed by first computing all behaviors, which might indeed be infinite, then using them to interpret the formula directly. Instead, many clever tricks and abstractions are involved in traversing this relation more directly. Nevertheless, the behavior remains key in giving a clear formal meaning to everything.

Moreover, it could be said that the behavioral representation constitutes a common currency amidst the plethora of different programming languages and models that may be created, as well as the variety of different logics that might be used. It is therefore the primary aim of this thesis, reflecting on the two questions of what languages and which logics are appropriate for dealing with HDDAs running on platforms such as the IoT and the Swarm, to first ask the question of what is the most fitting behavioral representation. How can we represent meaning in these platforms?

For a sequential system, it seems obvious that any representation of behavior has the general structure of a sequence of observations, usually either of states or events. What happens can simply be listed.

$$\mathbf{event}_1,\ \mathbf{event}_2,\ \mathbf{event}_3,\ \mathbf{event}_4,\ \ldots \tag{1.1}$$

This notion of steps in time is very intrinsic to a process being sequential.

However, for a concurrent system, such as the platforms we are considering here, how to represent behavior is far less obvious. Abramsky addresses this issue in [2], descriptively entitled *What are the fundamental structures of concurrency? We still don't know!*, which argues that, while there are a wide variety of proposals for such a system to represent concurrent computation, there is no clear definitive answer to this question. If the reader suspects this line of inquiry has been closed in the decade since this paper, Abramsky's discussion in the more recent [3] reflects the continuation of this inquiry asking "Is there a widely accepted universal model of interactive or concurrent computation?" and subsequently answering that "there is no such clear-cut notion of universality".

This thesis will not attempt to answer such a fundamental question. But it suffices to say that it is not obvious what an appropriate representation of concurrent behavior would be for HDDAs running on platforms such as the IoT. This thesis will propose a candidate for such a behavioral representation, and therefore endeavor to provide a ground for the development of programming languages, specification logics, verification tools, program analyses, and other components of a comprehensive system design model tailored to the development of these kinds of applications.

The behavioral representation that will be proposed will be called an *Ontological Event Graph* (OEG). This formal entity can most succinctly and intuitively be described as an acyclic ported block diagram. In this diagram, each block has a set of input and output ports and edges running from the input port of one block to the output port of another. These diagrams are reminiscent of dataflow diagrams and other kinds of diagrams used in component-oriented modeling languages where the blocks represent components and the edges represent channels of communication between them. These diagrams are ubiquitous in computer science and engineering and should be familiar to most readers. However the way they will be used here will be different. In an OEG each block will represent a behavioral *event* during the execution of a system. The incoming ports will represent the *dependencies* of the event, the pieces of information or influence necessary for it to happen, and the outgoing ports will represent dependencies that can be supplied to subsequent events; the results of the event. The edges connecting ports will then represent the passage of information or influence as a direct dependency between events.

There already exist a plethora of ways to represent concurrent processes, amongst them interleaved traces, partially ordered multisets, and varieties of event and task graphs. In all of these representations, events are merely vertices, and while they have incoming and outgoing dependency relations (direct or transitive) they do not possess a surrounding interface determining the order and the purpose of these incoming and outgoing dependencies. While adding this information might seem like a subtle enrichment of these existing models, it has a dramatic impact on modularity and composability. In particular, as we will see, it changes the mathematical theory around these entities radically. It changes behaviors from topological/relational entities to algebraic ones. Specifically, we will see that OEGs form free symmetric monoidal categories, algebraic structures that can be studied very directly using the tools of category theory. In more concrete terms, we will be able to construct OEGs with an algebraic language, and write formulae using this language. This provides a means to construct compositional OEG-valued semantics, and to reason about OEGs using the algebraic laws of free symmetric monoidal categories.

1.2 Outline of This Thesis

The structure of this thesis is depicted in Figure 1.2 as a dependency graph. Being that reading has been so far a sequential process, rather than a truly concurrent one, the reader will inevitably have to realize this structure as an interleaved trace. Nevertheless, the graph suggests several possibilities aside from the normative one suggested by the chapter enumeration. We will briefly summarize the chapters, then provide a couple paths the reader can take through the text.

Chapter 2

We begin looking in more detail at the motivating context for this work, centered around distributed platforms such as the IoT. In order to establish the scope of what is possible on these kinds of platforms, we imagine an application-level model built upon such a platform of devices and networks that will make it possible to write HDDAs. We will call this application model a Process Field and discuss how it can support a variety of *Models of Computation* (MoCs), allowing applications to make use of heterogeneous semantics. We will focus in particular on the Hewitt Actor Model as a superlative representative of the most flexible kind of semantics that one might expect on this platform and use it to identify important features that a behavioral representation must be able to handle. We will also give concrete examples of scenarios that will occur in HDDAs that illustrate these important features. The chapter will conclude attempting to sketch out a picture of behavior that emerges from these considerations, which we will term *Process Field Theories* through an analogy with physics.

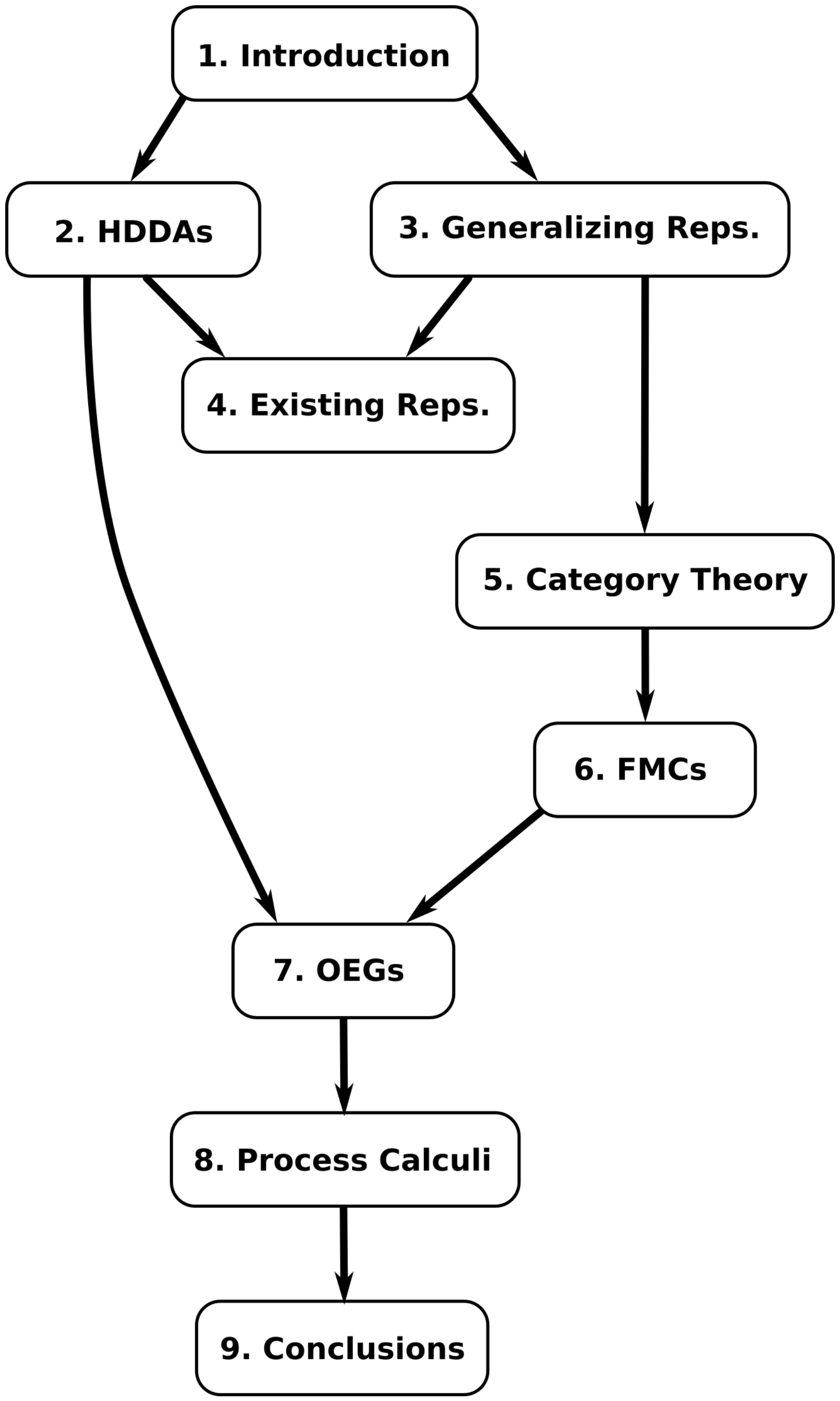


Figure 1.2: Flow of the thesis.

Chapter 3

Taking a different approach to the problem of determining the right kind of behavioral representation for distributed behavior, this chapter will start by considering two different ways of mathematically representing sequential behavior: proper order-theoretic *sequences*, and *free monoids*. These two methods will be shown to be conceptually different, but effectively equivalent, and thus easily conflated with one another in practice. However, we will argue that in generalizing each of these two sequential representations to their concurrent counterparts, *generalized sequences* and monoidal categories, respectively, this effective equivalence is lost and two radically different mathematical models of concurrent behavior emerge as candidates for fulfilling the demands of our context. We will conclude by giving preliminary conceptual arguments suggesting that the latter representation, rooted in monoidal category theory, captures computation in a more fundamental way.

Chapter 4

Reflecting on the other choice of *generalized sequences*, introduced in the previous chapter, we will address some of the more common instances of this kind of representation in the literature. Among these existing models are interleaved traces, Mazurkiewicz Trace, Event Structure, pomsets, and Actor Event Diagram, as well as other variants of these kinds of models. We will address the shortcomings of these models in addressing our demands for a behavioral representation.

Chapter 5

In order to build the mathematical tools needed to propose a new behavioral representation rooted in monoidal categories, we will first review and develop two important concepts from category theory: free constructions and monoidal categories themselves. The discussion of the former of these topics will involve an overview of the connection between *universal morphism*, free constructions, adjunctions, and monads. We will give the particular variant of the definitions for each of these concepts that we will use later and exemplify their use in the method of defining free constructions on monoids.

Chapter 6

Putting together the category theoretical concepts from the previous chapter, a definition for free monoidal categories will be given in detail, with a particular focus on the free symmetric monoidal categories variant of this construction. We will begin, first, defining *monoidal schemes*, which will be used to generate free monoidal categories. After giving the construction, we will go on to prove that the category of monoidal schemes is small co-complete and thus transfers its colimits via *Adjoint Functor Theorem* over to the corresponding category of free monoidal categories.

Chapter 7

This chapter will introduce our proposal for a new behavioral representation, which will be called an OEG. We will first present OEGs and their components conceptually, creating an *event ontology* for distributed behavior. We will then give formal definitions to *Ontological Event Schemes* (OESs), the languages of basic elements used to build OEGs, and then to OEGs themselves. Definitions will be given for parallel and sequential compositions on OEGs, as well as for special kinds of OEGs. It will then be shown that spaces of OEGs defined by a particular OES are free symmetric monoidal categories, and thus acquire the mathematical tools and theories around them developed in the previous chapter. The ramifications of this identity will be discussed and examples will be given of how the concepts from category theory manifest in the concrete use of these representations.

Chapter 8
As an example of how OEGs can be used in a sophisticated way to define the semantics of a concurrent language, an OEG semantics will be given for the π-calculus. We will discuss why this kind of semantics provides a particularly elegant basis for reasoning about π-calculus and carries intrinsically many of its important properties.

Chapter 9
Finally, conclusions will be drawn and an outline will be given of several directions of further work that can be done with OEGs.

In spite of the dependencies shown in the graph, a reader familiar with the basic concepts of category theory may jump to Chapter 5 to get a review of universal properties, free constructions, and monoidal categories, then proceed to Chapter 6 to read about the construction of free monoidal categories and about some of the properties that make them useful for our purposes here, but could just as well be useful elsewhere. As the graph suggests, a reader uninterested in the other representations of concurrent behavior, interested only in understanding OEGs, may skip Chapter 4, or at least defer it. A reader less concerned with the application space of OEGs, the IoT and the Swarm, could omit Chapter 2. Because Chapter 3 is primarily about motivations and intuitions, and perhaps a bit philosophical, this could also be skipped at the expense of not getting some intuitive ground leveraged later. Finally, for a reader that wishes to avoid category theory at all costs, but would still like to understand what OEGs are at a practical level, one could read Chapter 2 and jump straight to Chapter 7 and focus on the direct, set-theoretic construction of OEGs and their compositions.

1.3 Structures and Notations

There are a number of notations, structures, and conventions that will be used throughout the thesis. When possible, we will try and give the mathematical type of objects. In doing so, we will use the convention of dependent typing to indicate parametric families of objects. It is the contention of the author that this can clarify things, particularly when many of these objects might be constructed as programs. Specifically, the type

$$f \,:\, \prod a : A \,\cdot\, T(a)$$

indicates that f is a family of objects f_a, each of type $T(a)$, where a ranges over the type A. The type

$$x \,:\, \sum a : A \,\cdot\, T(a)$$

indicates that x is a pair of objects $(a,\, y)$ where $a \in A$ and $y \in T(a)$, noting that the type of the second element depends on the value of the first. The author is confident that the way these conventions are used will not introduce any significant foundational concerns.

Other various conventions and notations are listed as follows.

1. For declarative definitions the $\overset{\text{def}}{=}$ symbol will be used, while $=$ will be reserved for propositional equality.

2. The symbol 2 will be used to represent the boolean set {true, false}, and functions given the type $A \to 2$, where A is some set, will be taken as predicates over A.

3. The symbol $\bigstar$ will be used to represent the set containing a single element, which we will name $\star$. For any function $h \,:\, A \to B$, that does not already have $\star$ in its domain, the notation $h_\star$ will be used to represent the canonical lifting of h into a function of the type $A \cup \bigstar \to B \cup \bigstar$ defined

$$h_\star(x) \overset{\text{def}}{=} \begin{cases} \star & x = \star \\ h(x) & \text{otherwise} \end{cases}$$

4. For any set X, the symbol $\emptyset_X$ will be used to represent the unique function from $\emptyset$ into X (consisting of the empty mapping).

5. The operator $\amalg$ will be used to represent binary coproducts over sets, defined

$$A \amalg B \overset{\text{def}}{=} \{(a,\, 0) \mid a \in A\} \cup \{(b,\, 1) \mid b \in B\}$$

   This is a standard method to form disjoint unions by tagging elements from each set with a tuple marked with 0 or 1. To identify these tagged elements more clearly, the notations ι_L and ι_R will be used to represent the left and right injections, defined

$$\iota_L(a) = (a,\, 0)$$
$$\iota_R(b) = (b,\, 1)$$

For any given coproduct, these two notations will be overloaded to represent the particular injections corresponding to the operands of the coproduct. Given two functions

$$\begin{aligned} f &: A \to C \\ g &: B \to C \end{aligned}$$

the notation $f \nabla g$ represents the canonical coproduct of functions

$$f \nabla g : A \amalg B \to C$$

defined

$$(f \nabla g)(x) \overset{\text{def}}{=} \begin{cases} f(a) & x = \iota_L(a) \\ g(b) & x = \iota_R(b) \end{cases}$$

6. For any set X, and any equivalence relation $\cong$ over that set, $X/\cong$ will notate the quotient set, consisting of the equivalence classes of X under $\cong$. For any $x \in X$, $\langle x \rangle$ will be the equivalence class of the element x (the set of elements equivalent to x). Moreover, for any function or operator f over X, $\langle f \rangle$ will be the canonical lifting of f to a function or operator over elements of $X/\cong$ into elements of $X/\cong$.

7. The operators $\circ$ and $\bullet$ will both be used to represent categorical or functional composition, where $\circ$ corresponds to standard function composition and $\bullet$ is defined

$$g \bullet f \overset{\text{def}}{=} f \circ g$$

8. The operator $\bullet$ will also be used to represent monoid composition, along with the operator $\otimes$. The monoidal unit will be notated $\mathbb{1}$.

9. For any categorical object or set X, the notation $\mathbf{id}(X)$ will be used represent the identity morphism or function from X to itself. When the parameter X can be inferred from the context, we will shorten the notation to just **id**.

10. For any algebra $\mathcal{A}$, the notation $\|A\|$ will be used to represent the underlying set or structure of $\mathcal{A}$.

11. For any structure a with a well-defined notion of ordinal length, the notation $\|a\|$ will be used to represent that length. **Ords** will represent the collection of all ordinals.

12. For any category $\mathcal{C}$ representing algebras with homomorphisms, the notation $\mathcal{C}(X)$ will be used to represent the free algebra over underlying structure X (assuming that free algebras are defined for $\mathcal{C}$).

13. Throughout the text the symbol $\mathcal{A}$ will be used to represent alphabet-like sets and structures.

14. **Set** will be the category of sets and functions, and **Mon** will be the category of monoids and monoid homomorphisms. This convention will be maintained when possible.

15. $\bot_S$ will notate the bottom element of a partial order S (when such an element exists).

16. To simplify matters, operators and functions will sometimes be silently lifted to tuples of elements to which they apply. For instance, for some function $f : X \to Y$ it will be assumed that

$$f(x_1, \ldots, x_N) \overset{\text{def}}{=} (f(x_1), \ldots, f(x_N))$$

Chapter 2

Highly Dynamic Distributed Applications

Knowing what kinds of behaviors are possible in the applications we will be considering depends on the kinds of platforms on which these applications will be running. As we have stated, the kinds of platforms that this thesis will consider are those such as the IoT and the Swarm. But it can be seen from our description of these platforms, given in the introduction, that what characterizes them primarily are their constituting components, rather than an application or computational model that they give rise to. As the term suggests, the IoT, for instances, consists of *things*, a diverse collection of informatic devices with varying connections to the physical world, and the *internet*, a ubiquitous network providing the means for these devices to communicate. Likewise, the Swarm consists of a very large number of coordinated elements analogous to insects.

This leaves open the question of what kind of computational platform or system emerges from these components. What Models of Computation can be realized on these platforms? Assuredly many such models could be implemented by these platforms. However, considering that these platforms describe large open-ended networks, like the internet, rather than small highly controlled ones, like that of a distributed control system or even a network of communicating cores on a CPU, it may be the case that multiple models are operating on this platform at once, either independently or interacting with one another. These platforms will inevitably be *heterogeneous*, and thus what occurs on them will be a conjunction of different kinds of processes, each having been crafted in different semantics with a different set of design constraints in mind.

The question of how to handle semantic heterogeneity in modeling has been approached in the research efforts around *Ptolemy II*[15]. This component-oriented modeling platform facilitates the construction of systems in many different Models of Computation, such as *Kahn Process Network* (KPN), *Dataflow* (DF), *Synchronous Reactive* (SR), and *Discrete Event* (DE) models [57], as well as heterogeneous models that can combine these Models of Computation. While the individual Models of Computation implemented in Ptolemy II have been thoroughly researched individually, and all have precise semantics, the subject of combining these semantics in heterogeneous models has been an area of ongoing research. A theory of combining some of these models denotationally was constructed in [66], but this theory abstracted away from the behavioral dimension, leaving the question of how to combine the behaviors of models an open one. Instead, to reference Abramsky's comments recounted in the Introduction, the theory given in [66] provided a means to determine what function a heterogeneous model computed, without approaching the question of how this computation was performed.

In general, the temptation arising from semantic heterogeneity is to attempt to posit a overarching MoC for these platforms that is capable of subsuming all of the others, giving them a common compositional ground. However, without even determining whether this aim is a realistic one, a more fundamental question must first be answered, connected to the central theme of this thesis. Even if we were to posit such an overarching model formally, how would we express its behavioral semantics? If we knew it would all run on a single sequential machine, we might posit that it would be expressed as sequences of instructions on the machine, albeit at a loss of some abstract insight. But in our case, these applications will be running across a plethora of different machines. What underlying behavioral representation could serve as a common currency between the diverse models that may be running and interacting on these platforms, while abstracting away from the particularities of the devices and networks as much as possible? This is the question this thesis intends to answer in the form of a behavioral representation capable of codifying the semantics of various Models of Computation on platforms such as the IoT.

Nevertheless, determining the appropriateness of a behavioral representation requires at least an informal, intuitive concept of the application model that will be supporting HDDAs. Moreover, to even give a motivating example of an HDDA there will need to be a sense of how these may be structured. What kinds of events occur in an HDDA? What has to be accounted for in the behavioral representation? What features should it support?

This chapter will give such an intuitive but detailed characterization of the application platform made possible by the constitution of the IoT or the Swarm. A conceptualization of HDDAs will then be developed on this platform drawing on Hewitt's Actor Model[18]. Examples of these kinds of applications, illustrating their semantic complications, will be given in detail. We will conclude this chapter with a picture of the kinds of behaviors we would like our representation to represent.

2.1 The Process Field Platform

A significant amount of research has already been devoted to the nascent IoT and how to use modeling tools to develop distributed applications aimed at integrating its many diverse devices. The focus of these tools has been more oriented towards software engineering, encapsulation, reuse, composition, typing, and other forms of static analysis than formal semantics. The **Node-RED** [65] epitomizes this effort, providing a block diagram language for assembling components into *flows*, which bear resemblance to dataflow networks. The

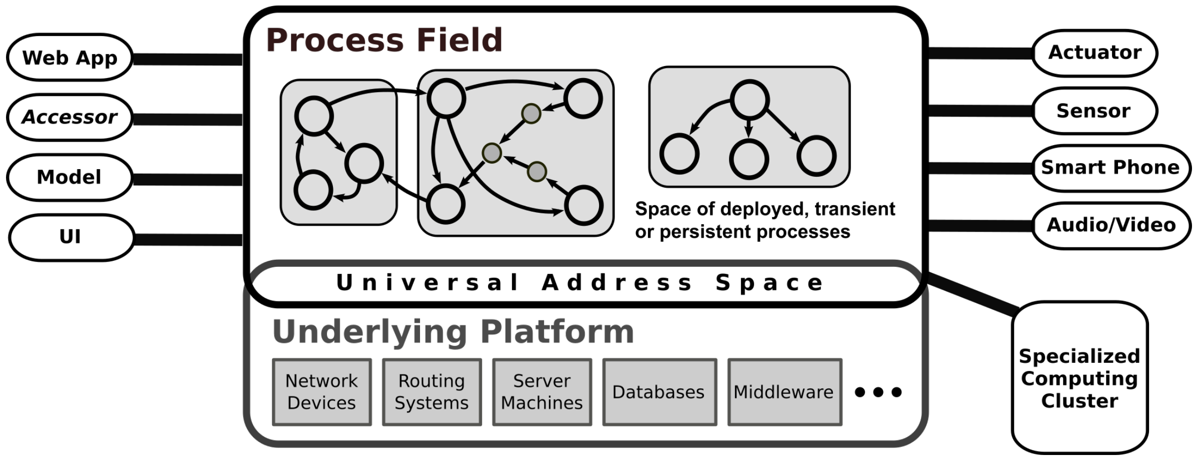


Figure 2.1: The Process Field

semantics for interaction and communication derive from the underlying semantics of the *ECMAScript* platform *NodeJS* [24] on which these networks run. Efforts have been made to give these *flows* other kinds of semantics as well, such as distributed dataflow[8].

Another line of research aimed at abstracting away the particularities of the devices and services that exist in these platforms, is developed by Latronico et al. in *A Vision of Swarmlets*[33]. This paper offers a picture of how applications can be constructed from *Acccessors*, which encapsulate devices and services in a component-oriented interface. These Accessors serve as a proxy layer that can facilitate the composition of devices and services in component-oriented modeling platforms like Ptolemy II. A *Swarmlet*, an application aimed at the Swarm, can be constituted of several such models communicating with each other through a variety of network protocols on the other side of the Accessor abstractions.

Drawing from these research efforts, it is clear that many techniques have been proposed or investigated for the purpose of abstracting away details of device interfaces and protocols, as well as details of the networks involved. This abstraction provides the application model with a much simpler picture of the IoT. Perhaps the most fundamental of these abstractions is the provision of a *universal address space*. This technique has been developed in the work of Kubiatowicz et al. in developing the *Global Data Plane* (GDP) [49], a space of datasets accessible from the application layer via universal addresses. What can be abstracted away in the GDP are the actual locations of this data, the routing necessary to access them, and details specific to the platforms that participate in this network. From the perspective of applications, access to the information *logs* that constitute this GDP is transparent, facilitated by a simple API.

In a series of presentations, [63][36], Shaver and Lohstroh elaborated on this idea of a GDP, suggesting a universally addressed space of *processes* – a *Global Process Plane*, drawing from Hewitt's Actor Model[18] for a sense of its semantics. This space, which we will call here a Process Field, consist of universally addressed concurrent processes. As is depicted in

Figure 2.1, the diverse entities that make up the "things" of the network are on the periphery of this space. Amongst these things are physical devices, such as sensors, actuators, and other components of CPSs; computational resources, such as databases, processors, and other kinds of specialized computation; or even other software systems, models, and user interfaces. These entities appear within this Process Field in the form of proxy processes that represent different means of interfacing with them. In addition, the Process Field consists of other transient and persistent computational processes. These processes all exist at particular addresses in the universal address space.

Beneath the application layer of the Process Field, and its universal address space, are all of the mechanisms responsible for realizing this abstraction. Amongst them are network devices and interconnects, physical computers, operating systems, middleware, data storage, etc... These mechanisms may be spread out across several physical devices, existing at different physical locations. They should make it possible for the processes at the application level to send messages to each other, create new processes, and migrate processes to new locations. As is the case in the GDP, the addresses may not correspond to particular devices or particular locations. The underlying location of a process running on a particular physical machine in a particular location may change without affecting the address visible from the application layer.

Considering the structure of this platform, we can imagine, with great generality, that the fundamental application level events we expect to see in HDDAs would appear on this kind of a platform. More specifically, the semantics of the Hewitt Actor Model provides a set of event types that one would expect to happen in a Process Field. We can imagine that processes can: (from [18])

- send a finite number of messages to other [processes]
- create a finite number of new [processes]
- designate the behavior to be used for the next message it receives.

And thus an informal application model we envisage for our Process Field can be conceived of as a variant of the Actor Model. Although we do not aim to pin down this model in complete generality and exclusivity, it can be seen that this is a reasonable starting point for identifying the kinds of behaviors that will be interesting on such a platform. We will proceed by diving into details about the Actor Model, assessing its fit to our circumstance.

2.2 The Actor Model

Hewitt's Actor Model[19] is, at its core, characterized foremost intuitively rather than formally. Though, several formal semantic accounts have been given of this model, such as that of Hewitt and Baker[6], Clinger[12], and Agha[4].

In this MoC, the atomic elements of computation are *Actors*. Each Actor exists at a particular address, in a space of addresses, and at any given moment in execution knows some of the addresses of other Actors. The most lucid and concise description of what an Actor can do may be that of Agha in [4]. To paraphrase: each Actor can respond to each message it receives by doing any combination of the following.

- Sending a finite set of messages.
- Creating a finite number of new Actors.
- Specifying a new behavior to handle subsequently received messages.

While the first two of these actions are clear, the third might require some further clarification, which we will come back to. We will refer to the specification of an *Actor* that maps received message to a collection of these actions as a *reaction rule*.

The execution of a particular Actor can be thought of as sequential, determined by the order in which received messages are handled. Messages are sent to it asynchronously from other Actors, and thus when an Actor receives them they are not necessarily inherently ordered. However, when messages are received, they are given an order, nondeterministically interleaving causally independent messages from different senders. These messages are then processed in this order creating a correspondingly ordered sequence of computational steps determined by reaction rules. This ordering is known as the *arrival ordering* of a Actor – we can state this order irrespective of whether we are talking about the order of actual message arrival or of the consequent reaction, since there is one message for every reaction.

The way that each message in the arrival ordering is handled might be different and depend on previous steps in the Actor's execution. Rather than supposing that each message is handled in the same fashion, by the same reaction rule, as in a stateless dataflow process, one of the consequence of handling each message is determining how the next message will be handled – the third possible action identified above. This could be framed precisely in many ways. The most familiar would be to simply say that the Actor has state, and each action can modify it, effecting the logic of the reaction rule. Another way of framing this would be to construct in the action the continuation for the Actor, consisting of a new reaction rule determining how the Actor will respond to the next message.

The details of the language in which reaction rules are specified can vary widely, and many such languages exist for programming Actors. The content of messages can vary widely, consisting of any kind of data. However, an essential piece of information a message must be able to convey is an address of an Actor. Alongside the fact that an Actor can only send messages to addresses it knows, which comprise part of its state, the communication of addresses between Actors is what facilitates the evolution of the network of possible communications between Actors. That is, through receiving messages containing addresses, an Actor learns which other Actors it can send messages to.

As a matter of fact, in its most abstract formulations, messages consist exclusively of addresses, and the relevant state of an Actor consists of addresses as well. This simplified Actor model serves a similar purpose as the λ-*calculus*, capturing the most significant primitive semantics mechanisms involved, out of which more complex computations can be built up

– albeit in tedious fashion similar to Church arithmetic in λ-calculus. Any practical variant on the Actor model endows Actors with the capacity to pass data in a variety of forms in messages, and to react to them in a way that performs any effective computation on their constituents. In other words, each reaction rule can involve an effective computation with the message as an input and the collection of messages to be sent, the collection of Actors to be created, and the continuation all computable functions of this input.

Nevertheless, an important observation of Hewitt regarding the abstract Actor model is that the messages could themselves be conceived of as Actors, since they contain addresses. Extending this to the more general case, this idea that messages are Actors suggests the notion of *mobile code*, if we suppose that the messages may contain both proper data as well as continuations containing reaction rules. If a Process Field is the abstract platform of universal addresses at which processes can concurrently operate, this model certainly captures much of what one could imagine happening on this platform.

2.3 A MoC for the Process Field

Although we must reiterate that the Actor model is not being put forward here as an ultimate proposal for a MoC running on a Process Field, there are many reasons why this model is arguably a realistic candidate for understanding semantics on this platform. Without fleshing out the exact details of this model, which differ over its many variations, enough can be extracted from it to point us in the right direction in our search for a behavioral representation. At the least, it is clear that such a representation should handle the Actor model, along with the other possible semantics defined on this platform, since this is a realistic model in which one would expect to define HDDAs.

At the least, it can be said that the Actor model is a particularly useful MoC amongst a heterogeneous spectrum of different models that may all simultaneously be at play in a Process Field. But Hewitt makes no hesitation in confidently arguing that this model is readily capable of subsuming a great many of other computing paradigms [19]. Many of these paradigms are the alternatives one could imagine occupying this space. Amongst them are simple sequential models, λ-calculus, functional languages, DF processes, control structures, and various mechanisms used in more traditional methodologies of concurrent programming such as semaphors and monitors. This makes a convincing case for supposing that a behavioral picture of the Actor Model could handle several other Models of Computation, even if these other models are not transliterated into the language of Hewitt Actors. In other words, if a behavioral representation can effectively codify Actor behavior, many other mechanisms one might find in HDDAs should also be covered.

Adding to this point, consider that the above definition for a single Actor, when given the state interpretation, amounts to that of a *Mealy Machine*. For an incoming message in the set M, and a state in space S that determines the reaction rule for the Actor, the action over the history of the Actor can be generally described denotationally by a function F

$$F : M \times S \rightarrow H \times W \times S \tag{2.1}$$

where the outputs of the function are a collection H of messages to send, a collection W of new Actors to create, and a new state determining the behavior of the Actor upon receiving the next message. If we consider M to be inputs, and $H \times W$ the outputs, it is easy to see that this fits the model of a Mealy Machine, both denotationally and operationally.

However, in general, how Actors are composed together into a model certainly goes beyond what is typically done with Mealy Machines in circuits and other similar kinds of models. In the Actor model the topology of connections between Actors is dynamic, let alone the fact that Actors are being created. These *outputs* and *inputs*, therefore, cannot always simply be connected with each other. Nevertheless, it is clear that Actors can be used to simulate many other models that have static networks. Reaction rules can be written to only send messages to a fixed collection of addresses, and to never create new Actors. More, since Actors only receive one stream of interleaved messages, the destination port of each message would have to be encoded in the messages, and the Actor would have to internally sort them out appropriately, potentially maintaining separate queues of data for each input as part of its state. The details of such an encoding go beyond the scope of this discussion, but they are merely suggested to add to the argument that something like the Actor model is high on the list of what must be considered for HDDAs running on a Process Field.

Another essential feature of platforms such as the IoT present in the Actor model is that of code mobility. Part of the execution of a HDDA might involve downloading fragments of code "on-the-fly", as web browsers do with ECMAScript in web applications. Distributing tasks may also involve sending their code, rather than simply sending a rigid set of parameters or a fixed pattern of data. An HDDA might perform a coordinated task, particularly in the context of the Swarm, by distributing fragments of its code to different entities that then proceed to interact with each other.

In spite of these reasons to suppose that the Actor Model might provide a specific application model for Process Fields, there are some features of Models of Computation that are not captured by the usual Actor Model. One of these, in particular, is simultaneity. Because the semantics of Actors demand that the arrival of messages be totally ordered, and thus potentially interleaved, nondeterministically, there can be no notion of two messages arriving at the same time as is the case in some DE models. One might argue that this is simply because there is no notion of time, itself, in the Actor Model, but even in DE models the notion of simultaneity is an artificial, rather than a physical one. Many simultaneous messages can arrive at a component from many different senders, in these kinds of models, during the same synchronous iteration. This is, in part, facilitated by the feature of components having multiple input channels, which Actors do not.

In the absence of multiple input channels, this problem with simultaneity can be generalized to the problem that incoming messages cannot be incomparable in their arrival. That is, the incoming messages cannot be given a less constrained partial ordering. In a DF

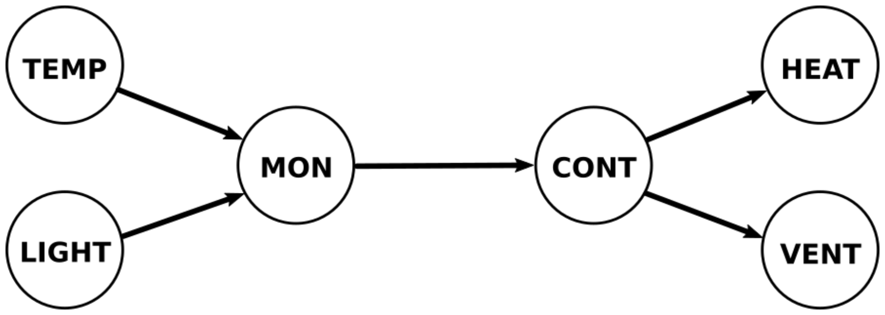


Figure 2.2: Climate control HDDA.

model, input channels to a dataflow component order incoming messages within the channel totally. But, between input channels, there is no ordering imposed by the semantics of the component itself. This is reflected in the operational details of the *Kahn Principle*[38]. Input events in these models commute if they are on different channels. The imposition of an *Arrival Ordering* prevents an Actor from directly representing components in these other models. This is not to say that a sufficiently complex Actor Model could not simulate DF or DE models, but rather that the semantics normally given to the Actor Model is not generally flexible enough to subsume these models directly.

Nevertheless, we can have in mind what could be described as a less constrained version of the Actor Model as an intuitive picture of the application model for HDDAs running on Process Fields, and proceed using this intuition to explore some examples of what can happen in HDDAs.

2.4 Examples of HDDAs

In order to give a concrete sense of what is possible in a HDDA, and to see what challenges it poses to the representation of behavior, we will look at two examples illustrating some important details about behaviors in Process Fields. These examples will be returned to once we propose our representation of behavior, showing how they would be modeled in this formalism. We will also refer to these examples in our consideration of existing models for representing behaviors.

Configuration

The first example will be taken from the domain of home automation, a prevalent theme in the IoT and the purpose of many of its marketed devices. The application we will consider, specifically, will be a simple climate control system, depicted in Figure 2.2 in the fashion of an Actor network. This system consists of two sensors: a temperature sensor **TEMP** and an ambient lighting sensor **LIGHT**. These sensors provide raw data to a persistent monitor process **MON** that will aggregate the data to form judgments about changes that should be made to climate conditions. These high-level judgments extrapolated from the low level sensor data are then forwarded as imperatives to a persistent controller process **CONT**. An example of such an imperative might be to raise the temperature by a certain amount gradually over a certain duration. This controller will respond to the high-level imperatives produce appropriate changes in climate through its communication with two actuators: a heater **HEAT** and a ventilation fan **VENT**. We assume that this system has been deployed into a Process Field that represents the application platform consisting of many concurrently running systems and containing other sensors, actuators, and devices.

Giving a semantics to this system would be straightforward if we consider the appropriate MoC for its functioning to be SR, *Process Network* (PN), or a DF variant. In all of these models, the set of Actors involved, and their topology of connections, are both static. Because this system is static and involves a fixed set of Actors, any of these might be reasonable. However, there is a more interesting challenge, for our purposes, in considering the *configuration* of this system. By configuration, we mean the process by which the two persistent processes **MON** and **CONT** are created and liked together with the necessary sensors and actuators. Representing the semantics of this better exhibits the complexities of HDDAs since it involves a dynamically changing network of components. This cannot be adequately modeled in any of the aforementioned Models of Computation, even if one of them suffices to determine the behavior once the system is configured.

We will take it to be the case that the semantics of configuration will generally be an important part of what happens in a Process Field. It is more common to relegate configuration to an *ad hoc* procedure that attempts to get everything in place before commencing the semantics. In a system that is constantly evolving, such as the IoT, imposing semantics only to stable circumstances that have been established would give us a disjoint, incomplete picture of behavior. Being able to reason about configuration, or reconfiguration of an HDDA, will allow these transitional interstices to be analysed, verified, or synthesize. While it might be easy to work out the configuration steps involved in setting up the application in this example by hand, when the scale of these applications increases vastly, the aide of formal methods will become greatly beneficial.

It may be presumed that in the Process Field we are imagining as the platform of this example some of the components that constitute this system already exist and have some connections. The network is in initial some state (or fragment of a state) and the process of configuration can evolve from this state through a collection of steps, creating Actors and sending messages, to the state depicted in Figure 2.2. For the purposes of this example, we will suppose that the initial state the system is as depicted in Figure 2.3. Therein we assume the existence of several Actors and preexisting relationships.

- The sensors, **LIGHT** and **TEMP**, as well as the actuators, **HEAT** and **VENT**, are all already presumed to be installed as devices and have these Actors as an interface in the Process Field for the physical devices to which they are attached.

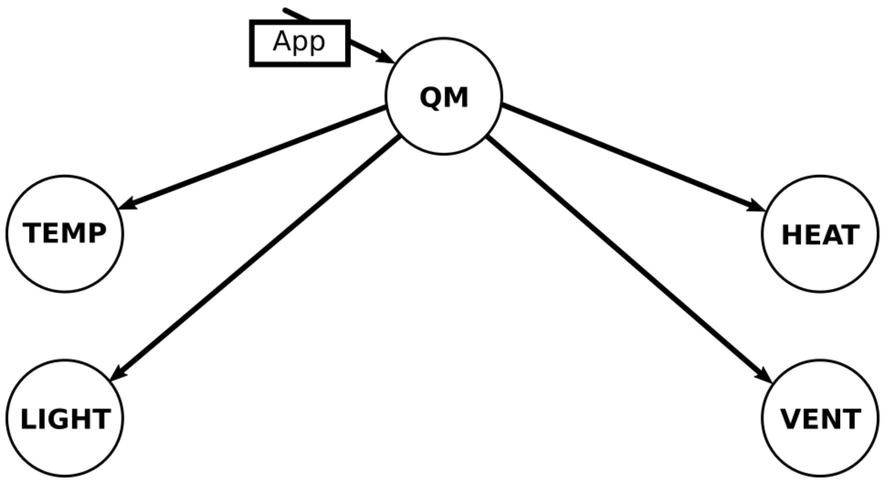


Figure 2.3: Initial configuration in the Process Field.

- There is a special Actor called a Query Manager, **QM**, which has persistent connections to all of the sensors and actuators. The job of this Actor is to facilitate connections to these devices so they can be directly utilized in applications.
- We assume that users of this system, themselves interfacing with the system through proxy Actors, have connections to the **QM**, and can send it a variety of requests involving the home automation system.
- For this example, we suppose that one of these users has sent the Query Manager a request message, containing the code for this climate control application. The request is to construct and deploy this application out into the Process Field.

Proceeding from this initial state, the configuration process can be imagined any number of ways, and we will not imagine the most simple of these, but rather one that more demonstrative. Given the **QM** must manage a potentially large volume of requests coming from different users to do different tasks, it will not perform the whole configuration itself, but instead create a new process to handle the configuration, freeing itself up quickly to process the next request. This style of spawning small asynchronous process to handle requests is reminiscent of the way modern web server systems such as NodeJS function, breaking the responding process up into "asynchronous" continuations that can be scheduled by an event handler.

The illustrated steps are as follows:

- The new process created by **QM** to configure the application will be called **AC** (Application Configurer). This step is shown in Figure 2.4a. The **QM** supplies the **AC** with the addresses of the sensors and actuators needed to build the application along with the application code. The result is shown in Figure 2.4b.

- The **AC** then proceeds to create the two persistent processes **Mon** and **Cont**, supplying the latter with the address of the two actuators. In contrast, **Mon** is not supplied with the addresses of the sensors because it does not send messages to them. It, rather, receives messages from them. This step is shown in Figure 2.4c (we hide **QM** in the next couple steps for simplicity) and the result is shown in Figure 2.4d.

- Confirmations from both **Mon** and **Cont** are then sent back to the **AC**, as is shown in Figure 2.4e. Consequently, the **AC** knows that both of these processes are ready to receive upstream information, and that the senders of this information can be given the addresses of these two processes. Because they are confirmed as ready, from a causal perspective, no additional handshaking (on the application level) is needed for the sensors to start sending data to **Mon** and for **Cont** to receive its commands.

- In its final step, **AC** forwards to the two sensors the address of **Mon**, as well as the instructions to start sending **Mon** data. It forwards the address of **Cont** to **Mon** so that the latter can start sending its course grain imperatives to the former. This is depicted in Figure 2.4f, resulting in the network shown in Figure 2.4f.

- Having properly configured the application, the **AC** process terminates, as is shown in Figure 2.4h. What is left behind is the intended network in Figure 2.3, along with the connections that remain between **QM** and the sensors and actuators.

What can be seen from detailing this configuration process is that it produces the intended network Figure 2.2, which could be taken as a kind of contract for this configuration process. At the least, the final network in Figure 2.5 contains the intended one Figure 2.3 as a subnetwork – this forms a kind of implication. It is likely that this is often what is desired when configuring a distributed application, that the final network of connections is guaranteed to be sufficient, modulo potential errors that can be handled. But more, in excess of the intended network is only the remaining connections of the **QM**, which were all assumed in the chosen initial network. There are no excess connections between the Actors involved in the application. By construction, **Cont** cannot send messages to **Mon**, **HEAT** and **VENT** cannot send messages to either **Mon** or **Cont**, and neither of the newly created processes can send messages to **TEMP** or **LIGHT**.

This configuration satisfies a much tighter constraint than simply containing the intended network, and there are many reasons why this kind of a constraint might be important. The absence of certain connections might be important for security. Whereas a home automation system may not be as concerned with this, one can easily imagine many cases where it is important that communication channels are only one way, and that entities that may transitively share information cannot do so directly. Another reason for this might be reliability. In the final configuration of this application, it is notable that there are no cycles. In more complicated cases where exact verification cannot be done, eliminating cycles might be useful in avoiding emergent properties unexpectedly taking hold of a system.

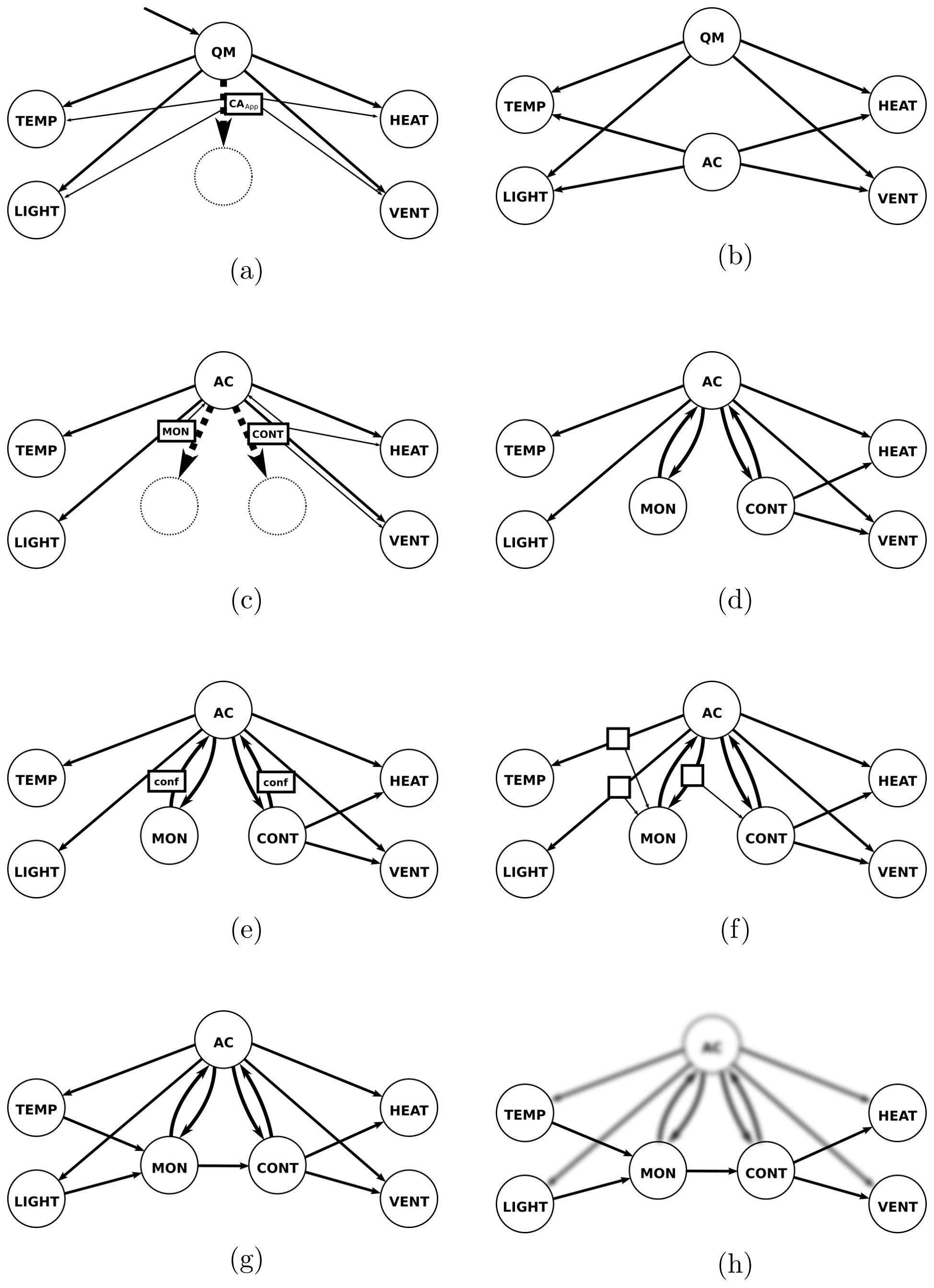


Figure 2.4: A home automation swarmlet.

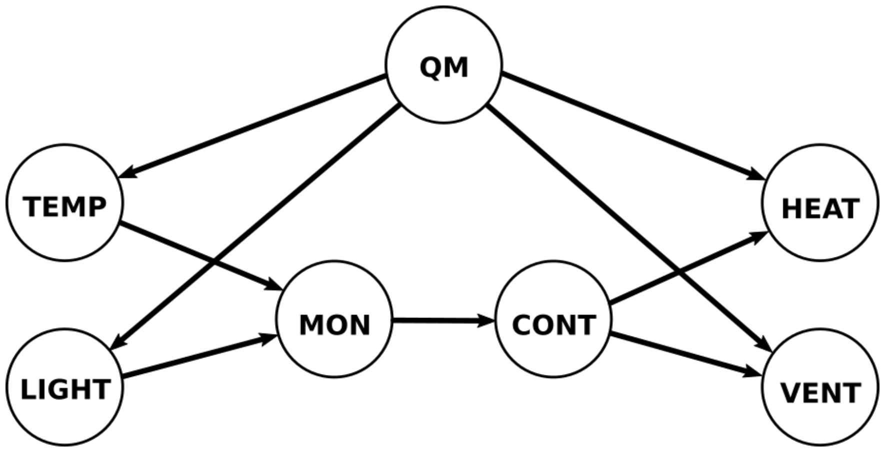


Figure 2.5: Final configuration in the Process Field.

Following these configuration steps, the HDDA commences with the behavior intended for its static network. This behavior could be defined in a DF MoC, where tokens are sent along connections between Actors. However, this MoC most certainly cannot accommodate the above process of configuration. Something along the lines of the Actor model can, as witnessed in the above informal characterization, but immersing the simple DF semantics for the application into the Actor model forgoes the advantages of designing the application in a constrained MoC that might provide a degree of analyzability and formal constraints that cannot be sustained through this immersion into a much broader paradigm. Therein lies the benefit of semantic heterogeneity in HDDA. We might wish to express the configuration of the application and its running behavior in different models, then compose them sequentially, such that the ultimate outgoing consequences of the configuration process feed properly into the incoming dependencies of the running process.

The challenge this example poses to semantics is a need for a behavioral representation capable of handling applications such as this one along with their configuration, which involves the dynamic creation and configuration of processes. The history expressed in this representation must be able to run through both of these stages. This is a particular challenge if these stages are semantically heterogeneous. This behavioral representation must be composable in a way that facilitates the joining of the configuration behavior with the running behavior of the application. The behavior should also allow certain questions to be meaningfully asked about the application. As we discussed, it may be important to establish that the result of the configuration behavior is the intended network or something implied by it. These questions are not questions of events but questions regarding the state of the process between events – here, specifically, it is the state at the end of the above events. However, we must recognize that while a sequence of informal steps were given for the above configuration process, the Process Field on which it runs is not synchronous, and we are not necessarily justified in regarding the above steps as formal ones. This raises the important

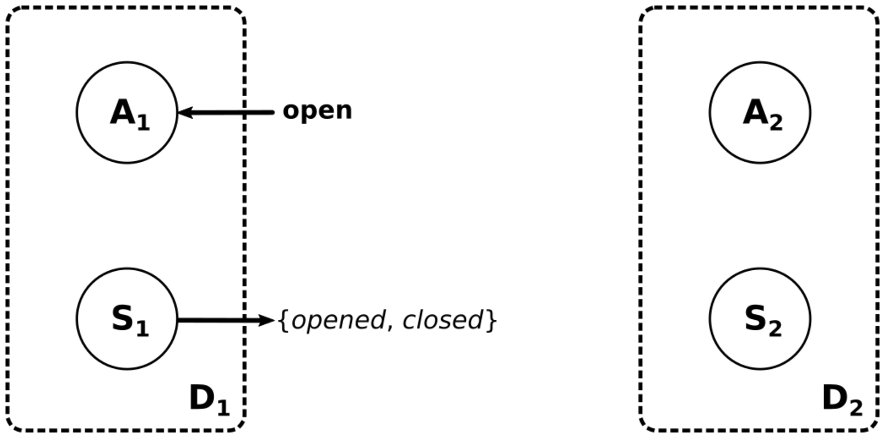


Figure 2.6: Initial configuration in the Process Field.

question: what is state in the behavioral representation?

Perspective

The second example will be one that could, in some form, easily appear in many contexts. We will use the framing of smart buildings to make the example as intuitive and illustrative as possible, although we will mention other framings as well. In this example, we will imagine a secure space in a building such as an office in which there are two entry doors. These doors will be called D_1 and D_2. Each door D_k has an actuator $\mathbf{A}_k$ that can open it, and a sensor $\mathbf{S}_k$ that can report whether the door is *opened* or *closed*. If the actuator $\mathbf{A}_k$ is sent the message *open*, the door D_k will subsequently open. The opening of the door will trigger the sensor $\mathbf{S}_k$ to report to whom it is connected that the door has been *opened*. Once the door returns to being closed, $\mathbf{S}_k$ will report *closed*.

This example will focus on a basic property ϕ that we are interested in maintaining for this system:

$$\phi = \text{ Only one door may be open at any time.} \tag{2.2}$$

One might suppose that this is for security reasons. The two doors could be an outer door and an inner door, and in the interest of maintaining secrecy, the outer door might need to be shut before the inner door opens and discussions behind the inner door are heard. It could also be for environmental reasons, to not let out heat or to keep out contaminants. It could also be to prevent a person from sneaking into a space while the door is closing. If we move back from the particulars a bit, this problem can be generalized to two of many cyber-physical objects that are actuated to be "opened" or "closed", and for which this property might be important.

If we suppose that these 4 elements, $\mathbf{A}_1$, $\mathbf{S}_1$, $\mathbf{A}_2$, and $\mathbf{S}_2$, are all present in a Process Field as Actors different scenarios can be imagined in which other Actors interact with them, as

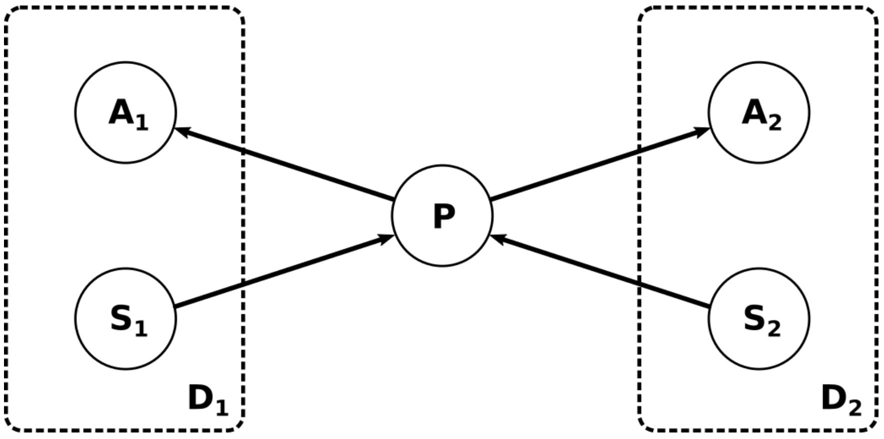


Figure 2.7: Simple system with one managing actor P.

is shown in Figure 2.6. In these scenarios we can ask questions about whether or not the above property ϕ can be maintained in the scenario. More specifically, if we assume a static network of connections between the 4 elements and a number of additional Actors, that can potentially open the doors, can we construct a protocol in the reaction rules that guarantees the property ϕ.

In this discussion of the example, we are defining ϕ informally, and can draw some intuitive conclusions about whether or not a certain network with a certain distributed protocol amongst the Actors successfully maintains this guarantee. To do all of this formally, a formal semantics must be given to the model yielding a representation of its behavior or behaviors, and further ϕ will have to be given in a formal logic that can be modeled by the behavioral representation.

To make this concrete, consider the simplest case in which there is only one additional Actor P interacting with both doors, their sensors and actuators. The topology of this network is depicted in Figure 2.7. As can be seen, we assume that P must be able to receive messages from the sensors and send commands (open) to the actuators. We assume behaviors of this configuration begin when P is sent a high-level message from some outside entity to open either door. Given this configuration, it should be fairly obvious at an intuitive level that maintaining ϕ can be done easily, since P can maintain the state of the doors and defer or deny any request that cannot maintain ϕ.

But how do we formally show this? What additional information or assumptions do we need? How do we represent the behaviors in a way that accurately portrays the situation, without making potentially fragile assumptions? We are assuming that this system is asynchronous, and thus we do not assume that the system has a single, well-defined sequence of global states. We assume that it can take time for the message sent to the actuator, telling it to open the door, to travel from P to $\mathbf{A}_k$. Therefore, it cannot be concluded *from the perspective of* P that the door has been opened subsequent to sending this message. P must wait for the confirmation from the corresponding sensor $\mathbf{S}_k$. Can we then be sure that it is open? One can safely conclude, assuming that the hardware, drivers, middleware, etc... all worked, that it had been opened. Perhaps, it could be closed again, or closed and reopened several times.

More assumptions clearly need to be added. Given the network in Figure 2.7, and assuming the door only opens when the corresponding actuator receives an **open** message, from the perspective of P, it might be safe to conclude that after sending the message to open the door, and receiving the message that it had been opened, that it might be either still open, or closed, and the sensor update has simply not yet reached P. On the other hand, if P receives a message, reporting the door as having **closed**, it can safely assume that it will remain closed until it sends another message to the actuator, telling it to **open**. Given P can known that a door is closed, it can safely decide to open the other without violating the property ϕ. It would take little effort to sketch out this protocol, and if we can give it a behavioral representation, we can ask precise questions about the runs of this protocol.

We can see already how this example begins to deviate from the conventional Actor model, which was devised originally to deal with closed systems. The sensors and actuators in the example fire as a consequence of influence outside of the Process Field (the collection of known Actors). This outside influence could not be adequately modeled by making the Actors nondeterministic, because this influence is highly organized. There is, indeed, a *cyber-physical* dimension to this model that involves the *cyber* details of the Actor model in conjunction with the *physical* details of how the door responds to the actuator, and what effects it can consequently have on the sensor. This *physical* behavior must be incorporated with the *cyber* in some fashion when constructing the behaviors of the runs of this system.

One might contend that the Actor model could be preserved if we assume that the sensors and actuators are themselves sending and receiving messages, to and from the outside. But there still must be a way to model the possible "runs" of these external messages as a behavior, and connect the representation of this behavior with that of the model. We are faced again with a kind of semantic heterogeneity in reasoning about our model, but in this case heterogeneity between the cyber and the physical. We can term this *cyber-physical heterogeneity.*

All of these considerations do not even get to the more challenging cases, though setting the ground for them. Consider the case of two independent processes P_1 and P_2 being in control of D_1 and D_2, respectively, having a topology of connections shown in Figure 2.8. If both processes can receive a request message to open their respective doors, it is fairly clear that the property ϕ cannot be guaranteed. Some additional communication channels must be available between these two otherwise isolated submodels to ensure ϕ. Consider the configuration in Figure 2.9, where each process P_k can also get sensor updates from the opposite sensor $\mathbf{S}_{3-k}$. Is this sufficient?

Again, using intuition, one can reason out convincingly that this is not the case. From the perspective of P_1, knowing D_2 has been closed from the sensor $\mathbf{S}_2$ cannot guarantee that P_2 has not since sent it another **open** message. Hence, P_1 cannot be sure it can open D_1 and preserve ϕ. If this is not adequate, then what is? Do P_1 and P_2 need to communicate with each other directly, or through a third entity that represents a mutex?

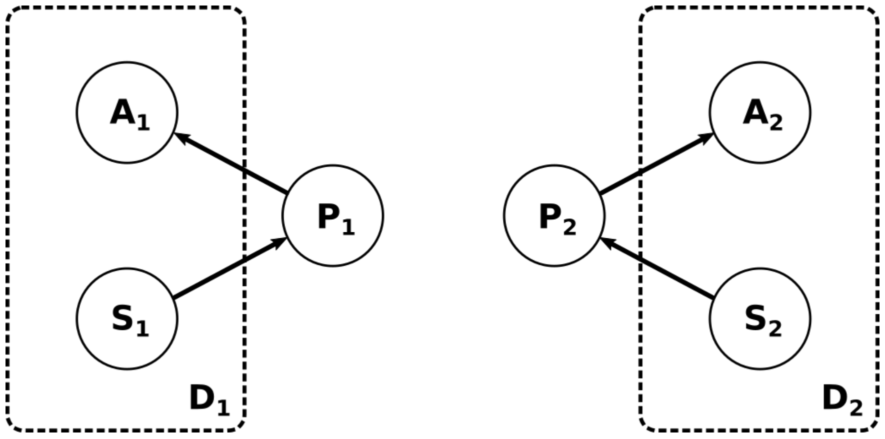


Figure 2.8: A more complex system with two independent processes managing the doors.

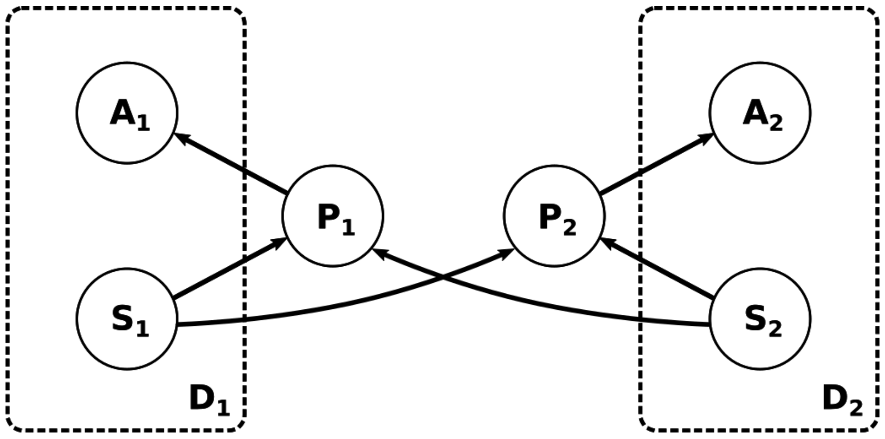


Figure 2.9: A possible set of connections between the two processes.

Answering these questions in a definite way is less important here than knowing how to ask them, given our aim is not to craft particular analyses and solve particular problems, but instead to construct a behavioral semantics that places all of these things into a common coherent representational model. The challenges presented by this example to a behavioral representation are a need to model properties, a need to define a notion of *perspective*, and again heterogeneity, this time *cyber-physical heterogeneity*.

2.5 Process Field Theory

The Actor Model and our examples provide us with enough of a provisional sense of a MoC to imagine what kinds of behaviors we will see, and hence we will need to represent, in the platforms we are calling Process Fields. The above examples use these semantics to help build this picture, identifying particular properties we would like a behavioral representation to have in order to capture the executions of HDDAs. The properties we are looking for include true concurrency, the ability to represent the creation and configuration of entities, composability, modularity, and the ability to serve as a common language between different Models of Computation in heterogeneous models.

We can begin to approach a sense of what kind of behavioral model we are looking for by elaborating more on the semantics we have given for the Actor model, getting to something more fundamental about behavior on this platform. In doing this we can unpack what we are suggestively implying, calling our platforms Process Field. Why a "field" of processes?

A couple of condensations can first be made to the Actor model that are important in this context. One is an observation stated by Agha in his presentation of the model [4] (though it was probably known much earlier). Given an Actor consists most generally of a reaction rule, containing a knowledge of Actor addresses, replacing the reaction rule with the continuation, specifying its new behavior, can be framed as simply replacing the Actor with a new Actor. We can elaborate on this in a manner that seems to the author to go beyond what Agha is stating. From this perspective, the meaning of an Actor can be reduced to a single state in the process in which it is involved, and what is thought of as an Actor, acting on a series of received messages by performing actions, can be reframed as a series of Actors at a single address, each reacting to a single message. Moreover, since the continuation is another Actor, as is a message, the replacement with the continuation can be interpreted as the Actor sending a message to its own address, containing the continuation.

Another condensation can be made, in light of the above. The act of creating a new Actor can be framed as sending an Actor to an empty address, instantiating the Actor as the first state to exist at that address. The consequence of these two condensations is that the list of three possible actions that could arise from a reaction can be compressed to the first: sending a finite set of messages. After these condensations, executions in the Process Field look more like messages being sent around a space of persistent reaction sites along addresses, splitting and intersecting at them. This is illustrated in Figure 2.10, which shows such an execution. This particular picture of the Actor model, as we will argue, is one that serves well the purpose of giving a general MoC to a Process Field, constituting the basis for HDDAs.

We can now unpack the intuitive meaning of a Process Field, drawing from an analogy to physics. Hewitt cites physics as an important influence behind the Actor model. In this analogy, Actors are like particles, while messages carry influence between them that must travel through space in a causal fashion. Indeed, as in particle physics, this very influence is itself mediated by particles. For instance, two electrons collide when one sends the other a message in the form of a photon, mediating electrical repulsion. We can depict this as the *Feynman Diagram* shown in Figure 2.11. A quantum mechanical view of this would be that an Actor, such as an electron, goes through a series of states (positions, momenta, spins) changing under the influence of forces, mediated by messages. The persistent ontological entities in this system are therefore the Actors, and the dynamics consist of these Actors going asynchronously through a series of state changes – the *world-line* of each Actor in this

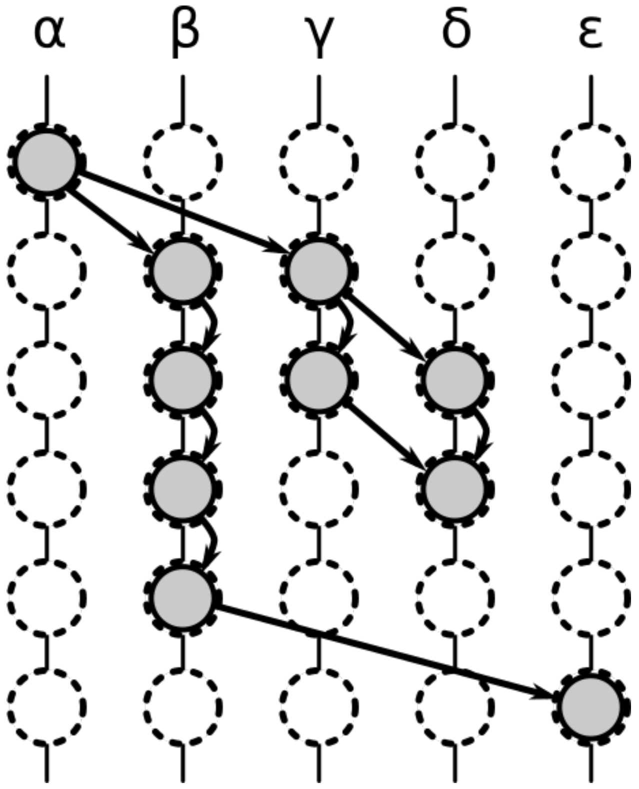


Figure 2.10: Execution in a Process Field.

analogy is its linear arrival order.

This analogy can be taken through a radical shift, incorporating the vast condensation constructed above, from quantum mechanics to quantum field theory. In quantum field theory, there is an ontological shift from viewing the particles as fundamental entities to viewing the particles as themselves states of fields, which themselves become the fundamental ontological entities of the system. As can be seen, we have made a similar ontological shift in our above considerations. Actors with addresses can be shifted from the fundamental entities of the model, which persist through changes of state over the course of reactions to messages, to simply being states themselves of the persistent field of addresses. One can think of the Process Field as a field of generic containers at particular addresses, as in the Figure 2.10. In this field, an Actor becomes the state of a container which changes each time the Actor reacts to a message. Even more radically, we can state that the message and the Actor, which are in essence two actors, simply collide at the address, ultimately producing a collection of outgoing Actors (particles), going in different directions, one of which may be stationary, remaining at the same address.

Along these lines, the Actor model can be transformed into what might be called a Process Field Theory. As in quantum field theory, in this model, every event is a collision

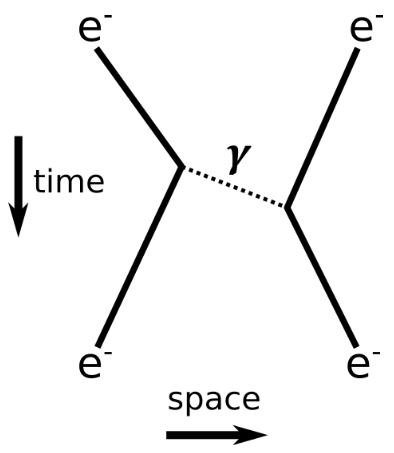


Figure 2.11: Feynman diagram of electron scattering (polarized).

consisting of a collection of incoming and outgoing particles. Here the particles are Actors, consisting of both data and continuations specifying reactions to data. What determines the consequence of the collision is the field itself. In our computational counterpart, this is the general computational platform, which at any address can perform a generic evaluation.

A final radical step is suggested in this interpretation, which we can argue is deeply important in reasoning about computations in the platforms we are considering. One of the most important advancements of quantum field theory beyond its predecessor theory of quantum mechanics, is that quantum field theory consistently integrates special relativity – as a matter of fact, this is one of the most important impetuses of its development [67]. Given that both an Actor A stationary at an address and a message M it receives are both Actors, putting aside any notion of *active* and *passive* that might throw the interaction into asymmetry, one might suppose, taking special relativity through our analogy into a discrete form, that the addresses constitute a frame of reference for this interaction. In this frame of reference, A is remaining stationary at a single address α while M is traveling between addresses – A is still at α and M is moving to this address. This is depicted in Figure 2.12a. It would follow, by this analogy, that we could just as well construct a different frame of reference in which M is still at address β and A is moving, as shown in Figure 2.12b. If we are operating under the interpretation that each reaction produces a new Actor, which in Figure 2.12 is called A', there is no longer a fixed notion of which of the interacting Actors is "continued" by the resulting continuation, here A'. In general, the continuation carries parts of both interacting entities. This is particularly true when M contains mobile code that is then run to produce A'. The situation could even be framed as depicted in Figure 2.12c so that the new actor A' is being sent as a message away from the address γ of M, and thus it would appear from this frame that M receives a message A, sends one A' immediately, then terminates. Of course, in this notion of a frame of reference there is no obvious correlate to *inertial.*

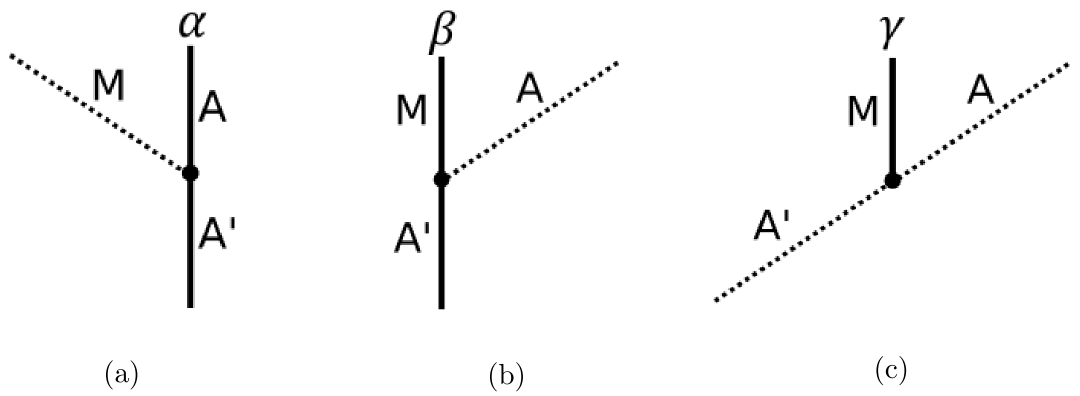


Figure 2.12: The same *interaction* from three *frames of reference.*

One could get anthropomorphic with this whole analogy, if it seems too peculiar, and imagine on one hand a letter M being mailed to a person at home A, who must actively open it up, read it, and psychologically incorporate its contents yielding in the person a new state of being A'. This is the archetypical analogy for the Actor model and other asynchronous protocols. But suppose instead that what is being sent is a TV repairman A, carrying a mobile phone, to a house. Prior to leaving the repairman answers the phone and receives a message telling him to go and perform the repair. After traveling to the house, while on site, the repairman acts on the TV M, changing its state. Perhaps while he is there, he gets a call from the owner, or speaks to someone in the home A'.

This latter scenario metaphorically illustrates the idea that the active component could just as well be the mobile one, and the passive data (the TV) could be stationary. Moreover, one could relativistically look at this same scenario in the repairman's frame of reference. He is stationary relative to himself, and in his current state he has his address at which he is waiting to receive the house, or perhaps a phone call. In some respect, that is no less counterintuitive than relativity (Galilean or Einsteinian) is to physics of Aristotle, in the repairman's frame of reference, the house does indeed come to him through space and time. Likewise, from the letter's frame of reference its addressee's mailbox arrives at the letter, having traveled through space and time. In fact, both of these examples can be restructured to concretely illustrate this principle by simply changing surrounding details. Suppose the repairman stays in his repair shop and the TV is brought to him. Similarly, the addressee of the letter travels to the post office to acquire it. Some semantics might regard these alternatives as equivalent, as they involve the same set of objects in the same causal relationships.

Like relativity, we can ask here what the invariant is in the behavior. What does not depend on the frame of reference? If we consider the addresses and the platform to constitute this frame of reference, what we are left with are the interactions between these Actors as the important ontological events that identify the behavior of the entire system. These interactions involve a set of incoming Actors upon which the event depends, and a set of

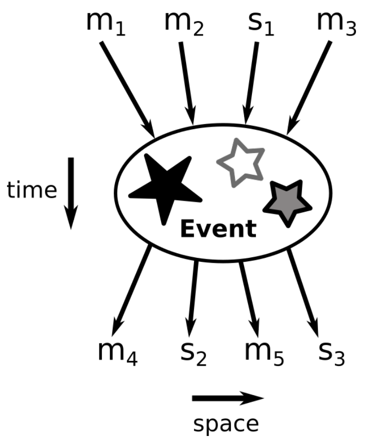


Figure 2.13: Events as collisions of computational data such as messages or states.

outgoing Actors produced as a consequence of this event. As can be seen in Figure 2.13, we now have a model of an event that looks like a collision, or more specifically a *scattering*.

This train of reasoning has provided us with a potential ontology out of which we can build representations of behaviors. Our *events* are these collisions between one collection of entities producing another as a result. The *dependencies* that connect these *events* are not just binary relations but are the entities themselves, containing particular information constituting a kind of state, message, continuation, or other kind of influence. We will see in Chapter 7 how this ontology is assembled into a formal model of behavior. However, we will first approach the task of developing a behavioral representation from a different angle, suggesting an underlying mathematical structure that will be expanded on in great detail. In Chapter 7, these two angles will converge on the OEG.

Chapter 3

Generalizing Representations of Sequential Behavior

A computing system could be conceived of so broadly as to pull into its category almost any physical system, which can be said to, in some respect, compute its trajectory from its initial conditions – its 'input'. A trajectory through a series of quantifiable or qualifiable observation is indeed the ostensible *behavior* of the physical system. It is an account of what happened in its evolution. In a computational system we can very generally define a *behavior* to be an account of what *will have* happened in this system while it is computing.

In contrast with this notion of behavior as semantics, the mathematical legacy of semantics in computing systems has often reduced computations to the partial function they compute from inputs to terminating outputs. For systems that aim to compute specific functions or perform delimited tasks, it might suffice to ask what function a program computes to understand what it does, and relegate behavioral concerns to qualities of the program related to the practicalities of running the program on a machine that will take time and use resources in a particular fashion that can be altered without fundamentally changing the meaning of the program.

But in the face of reactive and distributed systems pervasive today, which do not simply compute an output or diverge, but instead interact, the notion of *behavior* has become more central to the notion of semantics. As Abramsky argues in [3], regarding the notion of a partial function as semantics:

> This idea also served computer science well for many years: it is perfectly natural in many situations to view a computational process in terms of computing an output from an input. This computation may be deterministic, non-deterministic, random, or even quantum, but essentially the same general paradigm applies.
>
> However, as computation has evolved to embrace diverse forms and purposes: distributed, global, mobile, interactive, multi-media, embedded, autonomous, virtual, pervasive, . . . the adequacy of this view has become increasingly doubtful.

> Traditionally, the dynamics of computing systems their unfolding behaviour in space and time has been a mere means to the end of computing the function which specifies the algorithmic problem which the system is solving. In much of contemporary computing, the situation is reversed: the purpose of the computing system is to exhibit certain behaviour. ([3])

Therefore in today's reactive and distributed systems, particularly running on platforms such as the IoT and the Swarm, the essential semantic details of an application must be captured in an appropriate, precise mathematical picture of behavior – a behavioral representation. And this raises the primary question of this thesis: what kind of behavioral representation is appropriate for HDDAs running on the distributed platforms we have termed Process Fields?

Insofar as we know, every computing system we study as a candidate for realization into the physical world is realized as a physical system, and thus the horizon of realizable computing systems lies in the inherent properties of physical systems. It would therefore be reasonable to attempt to describe what happens in computing, computational *behavior*, as an abstraction of what happens in a physical system. After all, every useful abstract model of a physical system is one ultimately validated through a kind of computation, an unfolding of a model of physics, that must correspond to a collection of empirical observations. This would suggest that we look to physics for an idea of how to model computational behavior, as we did towards the end of Chapter 2.

The gap, however, between the diversity of physical processes and the most primitive computational ones is that the kinds of computation developed in the first models of computation such as Turing Machines, Universal Register Machines, Lambda Calculus, and Post Systems, do not involve any notion of *space*. Instead, these fundamental archetypes of computing systems all involve a notion of behavior rooted in single progression of events in *time*. More specifically, these events are discrete, countable events rather than continuously evolving ones, and finally, have a clear beginning, a first event. Cutting behavior down from the possibilities of physics by all of these constraints leaves only a discrete well-ordered notion of time and no notion of space. A behavior in these systems, and in all of the elaborations on them, can therefore be abstracted into a sequence, either of what *events* happened in the system, or of what *states* the system occupied.

As one moves away from these traditional models of computation into reactive and concurrent models this gap begins to close. Correspondingly, some of the assumptions that would lead one to represent the behavior of such a system as a sequence become lifted, and a broader representation of behavior becomes necessary. When concurrency amounts to a constrained static collection of systems, all of which are themselves sequential, the sequential form of a behavior can be retained through considering interleavings of the sequential behaviors of each component. The cost of this retention, though, is a conflation between concurrency and nondeterminism, which we will expand on in Chapter 4. Rather than *abstracting fewer details away from* physical systems, leaving details associated with space intact in the behavioral description, this approach to concurrency instead *adds more details to* sequential systems.

Nevertheless, the introduction of the IoT, the Swarm, and other platforms for distributed applications that we would regard as Process Fields, make it possible to write applications and design systems that do not factor as easily into a concurrent collections of sequential components. The dynamic creation and reconfiguration of processes, as well as code mobility, deviate more significantly from the assumptions that lie beneath sequential models of computation. There is an inevitable need for something like *logical space* to complement the *logical time* at the basis of sequential computation.

While this reasoning motivates a departure from sequential models of behavior in order to fulfill the demands of highly dynamic platforms for distributed computation, there are, nevertheless, properties of and analyses around sequential representations of computation (even of concurrent computation) that would be advantageous to preserve in a non-sequential representation. Amongst these properties are notions of *concatenation* and *subsequency* that do not have immediately obvious analogs when discharging the notion of a sequence, for which they are both very obvious.

At the level of analysis, it would also be advantageous to preserve some semblance of sequential specification logics such as *Linear Temporal Logic* (LTL) and *Computational Tree Logic* (CTL) in a similar logic aimed at writing specifications for HDDAs. While one might respond critically to this characterization of logics like LTL as sequential specifications, since they are applied to, and arguably even designed for, concurrent systems, these logics are nevertheless defined in terms of linear sequences (or trees), and extended to concurrency only insomuch as a set of sequences can be defined to model an LTL formula when each individual sequence does. That is, the structures that model formulae in these logics have no sense of logical space. If a different behavioral representation were formulated, departing from sequential models, this would also open the door for new logics that furnish theories for these models.

This chapter will approach the pursuit of an appropriate representation of behavior in Process Fields from a different direction than our discussion in Chapter 2. Here we will first understand the different ways sequential behaviors can be represented. While this may seem intuitively obvious, there are many superficially unimportant technical details that become relevant when we try to generalize from the sequential to the concurrent. We will show here that there are two (and potentially more) approaches to modeling sequential behavior that look and behave the same, modulo these technical details, which seem to be little more than issues of bookkeeping. But when we take these two different framings and generalize them, we arrive at radically different mathematical structures: *generalized sequences* and *free monoidal categories.*

Approaching our fundamental question this way will give us a sense of what schematic possibilities we have to draw from in making our choice of a behavioral representation. Two distinct branches of generalization will emerge from this discussion, one of which, as we will discuss in Chapter 4, encompasses many of the existing models of concurrent behavior. And we will give reasons in that discussion why this branch will not ultimately be our choice. As we will see, the other branch of generalizations, free monoidal categories, is the one that we will build into spaces of OEGs. We will expand on this second branch in Chapters 5 and 6.

3.1 Representing Sequential Systems

There are at least two routes that could be taken to representing sequential behaviors as mathematical objects (perhaps more even), and amongst these views many variants. The two views are that of proper *sequences* (which we will just call *sequence*) and that of *free monoids*. In both cases, the intuitive concept of a behavior that we aim to represent is that a set of events, representing things that have happened in the system, happen one after another, recounting the *history* of the system.

Intuitively, we might render the picture as follows

$$a \to b \to a \to c \to b \to b \to \dots \tag{3.1}$$

suggesting a sequential system in which the types of events a, b, c, etc... can occur. For the purposes of this discussion, we will consider a system in which the possible types of events form a set $\mathcal{A} = \{a,\, b,\, c,\, \dots\}$, sometimes called an *alphabet* or a set of *actions*. We will look at the way both approaches take the picture in 3.1 and forge it into a more precise mathematical description.

Sequences

The first approach to representing a sequential behavior involves the construction an abstract space of events to index individual instances of event types in $\mathcal{A}$ occurring. It is in this space that the sequential structure can be established **underneath** the alphabet $\mathcal{A}$. A natural choice for such a space is a linearly ordered set $E = (\|E\|,\, \leq)$ of events $\|E\|$, which merely function as points in the space.

$$e_1 \leq e_2 \leq e_3 \leq e_4 \leq e_5 \leq e_6 \leq \dots$$

Since we are constructing this space for the purpose of indexing the identity of events is not important, and any isomorphic ordered set would serve the same purpose.

We then map each event in E to an event type in $\mathcal{A}$ with some function

$$\bar{\alpha} \,:\, \|E\| \to \mathcal{A}$$

giving us the pair $\alpha = (E,\, \bar{\alpha})$ as a representation of behavior.

$$e_1 \mapsto a \leq e_2 \mapsto b \leq e_3 \mapsto a \leq e_4 \mapsto c \leq e_5 \mapsto b \leq e_6 \mapsto b \leq \dots$$

If we are dealing with a well-ordered sequence, and by that we mean one with a well-ordered space of events, we can always represent the space of events canonically as an ordinal κ, and thus we have a very common definition [25] of a sequence over $\mathcal{A}$

$$\alpha = (\kappa,\, \bar{\alpha} : \kappa \to \mathbf{A})$$

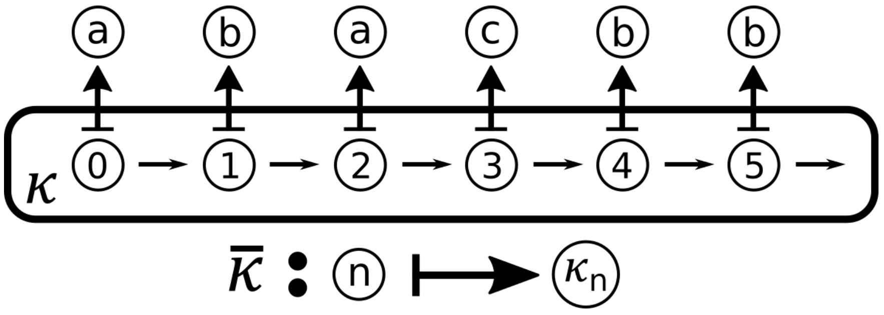


Figure 3.1: Sequential behavioral representation.

We can then call the projection of the first component the *length*[1], $\|\alpha\| = \kappa$, and index the events $\alpha_k = \bar{\alpha}(k)$ for $k < \kappa$. When dealing with behaviors involving finite numbers of events our ordinal κ is simply a natural number, but we may also want to include behaviors that do not terminate as ω-length sequences, or have other reasons to use infinite ordinals.

In the finite case, one would be justified in writing the behavior down as a simple list.

$$(a,\, b,\, a,\, c,\, b,\, b)$$

We will write the empty list down as (). This notation is consistent with the notion that a sequence α of length $\|\alpha\|$ over $\mathcal{A}$, is defined by a function $\|\alpha\| \to \mathcal{A}$, which is isomorphic to $\mathcal{A}^{\|\alpha\|}$, an $\|\alpha\|$-tuple.

The ordered set of events, or ordinal, constitutes a model of *logical time*, and thus our behavioral representation consists of a logical timeline along with a labeling for each moment. This is depicted in Figure 3.1 – the reader may find this Figure, by itself, superfluous as a means of understanding the given mathematical specification, but the graphic will be taken in comparison to another as an illustration of the contrast that is central to this chapter. This resembles a typical trajectory in physics which might be a function $\mathbb{R} \to X$ except that it is conventional in physics to either defined trajectories over all of time $\mathbb{R}$, or at least to be rather informal about the relevant intervals. Here, it is more relevant to consider behaviors of different lengths, which will be an important feature for the way that we will use them later.

If an observer were to watch the events occurring during a computation up to any point before the computation terminates, the behavior the observer sees up to this point in the logical time of the computation is a *prefix* of its complete behavior, as well as longer prefixes that will be may be observed later. This observer, in observing the process, therefore, sees what could be thought of as a sequence of events or an increasing sequence of prefixes.

[1]It should be noted that in the transfinite case, the *length* is not a cardinality, but instead the specific ordinal defining the space, since a cardinal would conflate many infinite ordinals and thereby the structure of the behaviors defined over them. If we wish to discuss the cardinality of the sequence this will not be referred to as *length*, but instead *size*

Giving an ordering to sequences, we can define this notion of a prefix formally. For any two sequences α and β

$$\alpha \leq \beta \overset{\text{def}}{=} \|\alpha\| \leq \|\beta\| \wedge \bar{\alpha} = \bar{\beta} \restriction_{\|\alpha\|}$$

This ordering simply adds to the ordering of the underlying ordinals the additional stipulation that the two sequences agree on the domain of the shorter sequence. For instance,

$$\begin{aligned}
&() \leq (a,\, b,\, a,\, c,\, b,\, b)\\
&(a,\, b,\, a) \leq (a,\, b,\, a,\, c,\, b,\, b)\\
&(a,\, b,\, a) \leq (a,\, b,\, a,\, c,\, a)\\
&() \leq (a,\, b,\, a)
\end{aligned}$$

It should be noted, however, that unlike the ordering in the ordinals, which is total, this ordering is partial. In the second and third example $(a,\, b,\, a)$ is less than two different sequences, which are incomparable to each other. When $\alpha \leq \beta$, we call α a *prefix* of β.

The space of our sequences $Seq(\mathcal{A})$ over $\mathcal{A}$ can then be defined with a dependent sum as follows

$$Seq(\mathcal{A}) \overset{\text{def}}{=} \sum \kappa : \mathbf{Ords} \cdot \kappa \to \mathcal{A}$$

That is, the space of these pairs, where the first element, length, constrains the second, defining a mapping of this length. A poset category $\mathbf{Seq}(\mathcal{A})$ can be formed from this space along with the above ordering as the set of morphisms. The initial element, or bottom, of this poset is

$$\bot_{Seq(\mathcal{A})} \overset{\text{def}}{=} (0,\, \emptyset_{\mathbf{A}})$$

making it a pointed poset. And here we are close to the epitome of sequential behavioral representations: the *domain*. What remains is to cap the length of the sequences at some ordinal (usually a limit) ρ, giving us an pointed ρ-complete partial order. If this ρ is specifically ω, the first infinite ordinal, we then have a *domain*.

The consequence of this construction is that sequential behavior is often studied in semantics from the perspective of domain theory, as first developed by Scott [60]. However, as it was initially conceived, the elements of the domain were not the behaviors themselves, but rather the consequences of the behaviors on the value ultimately being computed. The behavioral prefixes, in this case, corresponded to approximations of the ultimate value that these behavioral prefixes computed. It is not a stretch of the imagination to see that these domains of approximation can be constructed as the images of monotonic functions over the behaviors themselves, assigning to each behavior, each prefix of a computation, the approximation reached at that point. This will be discussed further at the end of the chapter, but it is important to begin setting the stage here.

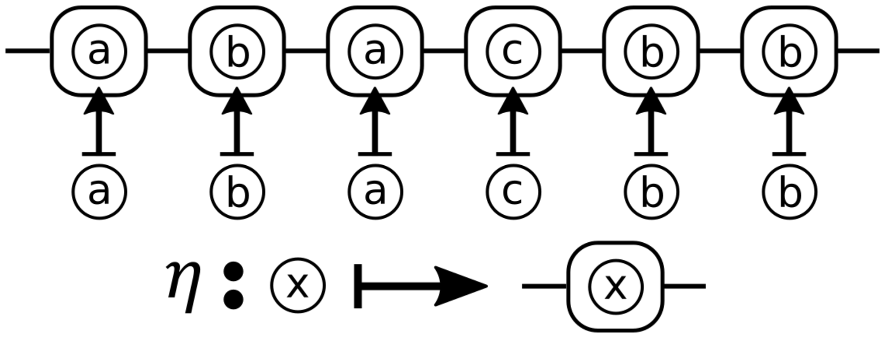


Figure 3.2: Free monoidal behavioral representation.

In summary, the approach of representing sequential behaviors as sequences in $Seq(\mathcal{A})$ can be thought of as a "topological" or "relational" one, building behaviors out of spaces. To emphasize the intuition one can roughly characterize the representation as

$$\text{Logical Time} \Rightarrow \text{Event Types}$$

Free Monoids

In contrast to this view of a sequential behavior as a labeled order of observation instances, sequential behaviors can also be represented as members of the free monoid $\mathbf{Mon}(\mathcal{A})$ generated by $\mathcal{A}$. Perhaps the most illustrative and direct way of framing $\mathbf{Mon}(\mathcal{A})$ is that it is a monoid $(\|\mathbf{Mon}(\mathcal{A})\|, \bullet, \mathbb{1})$ built upon an underlying set $\|\mathbf{Mon}(\mathcal{A})\|$ containing every finite string that can be built from the alphabet $\mathcal{A}$. These strings are often called *words*, and are often written as such. For instance, where $\mathcal{A}$ are the elements a, b, c, ..., $\|\mathbf{Mon}(\mathcal{A})\|$ contains

$$abaabc,\ bbbb,\ c,\ \text{etc...}$$

including the empty string. In fact, this underlying set is familiar to most computer scientists in the form of the Kleene star of the set $\mathcal{A}$, typically denoted $\mathcal{A}^*$. What is added to this set, as in any monoid, are an associative product operation $\bullet$ and a unit element $\mathbb{1}$. Specifically, the action of the product in the free monoid is to simply concatenate words together, while the unit is the empty word. The axioms of monoids, stating that $\bullet$ is associative and that $\mathbb{1}$ is the identity of $\bullet$ are obvious in this case.

However, a subtlety that normally might be glossed over, particularly in the treatment of $\mathcal{A}^*$, will actually be an aid to intuition in this case. We will take the time to make it clear. The proper construction of a free monoid does not necessarily include the elements of $\mathcal{A}$ directly into $\|\mathbf{Mon}(\mathcal{A})\|$. Instead, more generally, one defines an injection $\eta^{\mathcal{A}}$ that maps each generating element $a \in \mathcal{A}$ into a one-letter word $\langle a \rangle$, which is kept distinct from the generating element itself. That is, $\eta^{\mathcal{A}}(a) = \langle a \rangle \neq a$. To a programmer, this distinction should come as no surprise, as a character in a language like **C** is distinct from a string of length 1 containing just that character, even though there is a bijection between the two. This should be no more a surprise to a linguist, who would distinguish the letter 'a' with the English word "a".

Why then bother with the formality of the injection, when, as with all injections, the domain is isomorphic to the image? In a sense, this injection wraps the event type in an interface that allows it to be algebraically concatenated with other words, whereas the generating set $\mathcal{A}$, as a mathematical structure, possessed no such capability. This is depicted in Figure 3.2. Each event type is endowed with a rudimentary structure that facilitates the construction of chains, and therefore, in some sense, a model of time that emerges from the construction rather than being assumed before it. One can now compare Figure 3.2 with Figure 3.1 to see an illustration of the key intuitive difference in these approaches.

In these terms, our example behavior looks like the following:

$$\langle a\rangle \bullet \langle b\rangle \bullet \langle a\rangle \bullet \langle c\rangle \bullet \langle b\rangle \bullet \langle b\rangle \bullet \dots$$

As in the case of a sequence, here we also get a sequential ordering of observations. Each word in the free monoid is a history of observations that can be constructed through concatenations. It is therefore not just an ordered representation, but more a composable one. As can be seen, the mathematical glue that connects the observations together can connect the observations directly, without forming a separate set of instances to which they are attached. While the glue in the context of total orderings is relational, acting only on individual elements, in free monoids the glue is combinational and closed over all histories; an algebraic operation.

A technical feature of a free monoid that is will also be relevant is that it possesses what is known in algebra as a *universal property* among the set of all monoids. Although we will go over the details of this in great depth in Chapter 5, here we will give a brief account of what this means intuitively.

The first important consequence of this property is that any element α of the underlying set $\|\mathbf{Mon}(\mathcal{A})\|$ can be expressed as an algebraic term in the language of monoids, using only the images of the generating elements $\langle a\rangle$, $\langle b\rangle$, ... along with the operator $\bullet$ and constant $\mathbb{1}$. Even more significantly, any two terms expressing the same element can be shown to be equivalent using only the axioms of monoids. For $\mathbf{Mon}(\mathcal{A})$,

$$\begin{array}{l} \langle a\rangle \bullet (\mathbb{1} \bullet \langle b\rangle) \\ (\langle a\rangle \bullet \langle b\rangle) \bullet (\mathbb{1} \bullet \mathbb{1}) \\ \langle a\rangle \bullet \langle b\rangle \end{array}$$

all represent the same word, which we could write directly as ab, and can all be proven equivalent using only the associativity and identity axioms.

In contrast, consider the monoid $(\mathbb{Z}, +, 0)$ of integers with the operation of (standard) addition and the unit 0. One could chose two generators 1 and -1 in the underlying set, and for sure any integer can be expressed using only these two elements. But the two terms

$$1 + (-1) \qquad \text{and} \qquad (-1) + 1$$

both equal to each other, and to 0, cannot be shown to be equivalent using the axioms of monoids. There are additional equations in this monoid that go beyond those that can be proven from the axioms. In this sense, the free monoid is the most general, placing the fewest equivalence constraints on its terms; only the ones necessary in all monoids.

If we extended the integers to include $-\infty$ and ∞, the first aforementioned consequence of the universal property would not hold either. No finite term built from the generators could express either of these. In this respect, the free monoid is the most economical, only consisting of what can be generated from its generating elements and nothing more.

The second important consequence, which is actually central to defining the universal property of $\mathbf{Mon}(\mathcal{A})$, is that for any other monoid M, if there exists a function $h : \mathcal{A} \to \|M\|$ (not necessarily an injection), mapping elements of $\mathcal{A}$ into the underlying set of M, this function can be transformed in a canonical fashion into a monoid homomorphism $h^\sharp : \mathbf{Mon}(\mathcal{A}) \to M$. In other words, $h^\sharp$ is generated from, or a *lifting* of, h. This more complicated property will be explicated fully in Chapter 5. However, the way it words is intuitively simple and reminiscent of the **map-reduce** pattern in functional programming. Given h and any term representing an element of $\mathbf{Mon}(\mathcal{A})$, say

$$\langle a \rangle \bullet (\mathbb{1} \bullet \langle b \rangle)$$

$h^\sharp$ simply applies h to each generating element, then interprets $\bullet$ and $\mathbb{1}$ in the monoid M, as $\bullet_M$ and $\mathbf{id}_M$. Thus

$$h^\sharp(\langle a \rangle \bullet (\mathbb{1} \bullet \langle b \rangle)) = \langle h(a) \rangle \bullet (\mathbb{1} \bullet \langle h(b) \rangle)$$

A concrete, and relevant, example of this property can be seen if we choose our M to be the monoid $(\mathbb{N}, +, 0)$ of natural numbers with addition. Consider the mapping h from $\mathcal{A}$ to $\|M\|$ that simply maps every element to 1.

$$f : \mathcal{A} \to |M|$$
$$f(x) \overset{\text{def}}{=} 1$$

The lifting $h^\sharp$ then takes any term and replaces each $\bullet$ with $+$, each $\mathbb{1}$ with a 0, and each generating element with 1

$$h^\sharp(\langle a \rangle \bullet (\mathbb{1} \bullet \langle b \rangle)) = 1 + (0 + 1) = 2$$

In essence, $h^\sharp$ gives an effective length for any given term, enumerating the number of generator instances needed to construct it. For any $\alpha \in \mathbf{Mon}(\mathcal{A})$, we can therefore define the *length*

$$\|\alpha\| \overset{\text{def}}{=} h^\sharp(\alpha)$$

As is the case with sequences, a partial ordering can also be defined over the words of the free monoid.

$$\alpha \leq \beta \overset{\text{def}}{=} \exists\, \gamma : \mathbf{Mon}(\mathcal{A}) \ \cdot\ \beta = \alpha \bullet \gamma$$

Here, it is worth noting that this relation is expressed very directly as a simple formula in the theory of the monoid and, if true, has a specific witness in the free monoid itself. This ordering definition could be extended to all monoids with the consequence that it would in general form a preorder rather than a partial order (lacking antisymmetry). The monoids that are partially ordered under this preorder definition are often referred to as *pomonoids* (and in some of these pomonoids the witness to an ordering may not be unique).

Similar to sequences, the collection of words less than α can be called its prefixes, and as in the case of sequences, a chain in this ordering characterizes a sequence of observations of a computation in progressing. There is a key intuitive difference here though. As opposed to the model of time that lies underneath a sequence, and is exposed in the ordering of its prefixes, in the case of a word in a free monoid, the model of logical time defined by the ordering emerges from the theory of the free monoid.

With this in mind, more broadly, each monoid can itself be represented as a category, and the collection of monoids with monoid homomorphisms forms a 2-category **Mon**. The free monoids are not themselves isomorphic in **Mon** but unique up to isomorphism for each set of generators. Therefore each set of generators has a corresponding free monoid in **Mon**, and these free monoids form a subcategory of **Mon**. Following from universality, somewhat analogous to the case of ordinals, the homomorphisms in this subcategory are all monoidal embeddings. This category can be constrained further to one of pomonoids with monoid homomorphisms (which end up being all monotonic in the induced ordering).

We can summarize the contrast between these two approaches to representing sequential behavior as follows. With sequences, we construct a model of logical time and use it as a backbone for constructing a sequence of events as a mapping.

$$\text{Logical Time} \Rightarrow \text{Event Types}$$

With free monoids, we wrap each event type in an algebraic interfaces and assemble sequences of events. Intuitively,

$$\text{Constructions on Event Types} \Rightarrow \text{Logical Time}$$

The important point that these two diagrams emphasize is that the model of time is presupposed in the former case, but in the latter case emerges from the algebraic construction.

Operations on Behaviors

These two pictures of sequential behaviors have an apparent difference that is subtle, buried in the mathematical details, and both ultimately result in representations of behaviors that we could think of as lists of one sort or another. However, as has already been eluded to, the difference becomes far more substantial once one tries to generalize away from sequential systems. Before approaching this generalization, there are still more differences that can be illustrated in these subtleties, and in discussing them we can explain why these two pictures are so often treated as interchangeable in the sequential case.

Specifically, we will look at the operations that should be possible on behaviors. What should we be able to do with them that is essential to constructing semantics or logics on them. First, consider the operations of appending to or concatenating behaviors, and how these two operations are codified differently in the two approaches of representation. During a computation, if we observe it progressing, insomuch as a behavior is a history, we are looking at the extension of a behavior. Unless one is dealing in the most controversial corners of quantum mechanics, we would be safe for practical purposes with the physical assumption that what has happened already will not change in the future. In other words, computation is at least *causal*.

Suppose a behavior were to have a new event $c \in \mathcal{A}$ appended, the simplest case. We can define this operation $\mathbf{Append}(c,\, \alpha)$ for a behavior α. For the free monoid approach the definition is simple because the structure is already equipped with concatenation as its fundamental algebraic composition.

$$\begin{aligned}&\mathbf{Append} \,:\, \mathcal{A} \to \mathbf{Mon}(\mathcal{A}) \to \mathbf{Mon}(\mathcal{A})\\ &\mathbf{Append}(a,\, \alpha) \overset{\text{def}}{=} \alpha \bullet \eta^{\mathcal{A}}(a)\end{aligned}$$

The new event is simply lifted into the monoid and concatenated on the end.

The corresponding definition for well-ordered sequences makes use of the fact that we have a successor function $S \,:\, \mathbf{Ords} \to \mathbf{Ords}$ that is total. We can use this to extend the timeline by one step, then assign the new event to the new step.

$$\begin{aligned}&\mathbf{Append} \,:\, \mathcal{A} \to \mathit{Seq}(\mathcal{A}) \to \mathit{Seq}(\mathcal{A})\\ &\mathbf{Append}(a,\, (\kappa,\, \bar{\alpha})) \overset{\text{def}}{=} (S(\kappa),\, \bar{\alpha} + \{\kappa \mapsto a\})\end{aligned}$$

Rather than append in a direct algebraic fashion, here the technique is to perform a specialized extension with respect to the underlying model of time, notwithstanding the fact that the result is quite similar.

Appending can be broadened to concatenation in which two behaviors are composed, one after another. Similarly, we can define the operation $\mathbf{Concat}(\alpha,\, \beta)$ which will represent the events in α followed by those in β. In the case of free monoids, this operation is again natural, being simply the monoid product.

$$\begin{aligned}&\mathbf{Concat} \,:\, \mathbf{Mon}(\mathcal{A}) \to \mathbf{Mon}(\mathcal{A}) \to \mathbf{Mon}(\mathcal{A})\\ &\mathbf{Concat}(\alpha,\, \beta) \overset{\text{def}}{=} \alpha \bullet \beta\end{aligned}$$

In the case of sequences, we can again take advantage of the successor operation already defined on the underlying ordinals to extend the space. However, here we will do it define it recursively in terms of appending.

$$\begin{aligned}
&\mathbf{Concat} \,:\, \mathit{Seq}(\mathcal{A}) \to \mathit{Seq}(\mathcal{A}) \to \mathit{Seq}(\mathcal{A}) \\
&\mathbf{Concat}(\alpha,\, (\rho,\, \bar{\beta})) \overset{\text{def}}{=} \begin{cases} \alpha & \|\beta\| = 0 \\ \mathbf{Append}(\bar{\beta}(\rho),\, \mathbf{Concat}(\alpha,\, (\rho,\, \bar{\beta}|_\rho)) & \|\beta\| = S(\rho) \\ \bigvee_{m<\|\beta\|} \mathbf{Concat}(\alpha,\, (m,\, \bar{\beta}|_m)) & \|\beta\| \text{ is a limit} \end{cases}
\end{aligned}$$

In the above, the limit case takes the least upper bound of the chain of concatenations of prefixes. Note that the particular definition of the successor case appends from the outside in, since doing the opposite would involve shortening the remainder from the front, which cannot be done in any meaningful way if the successor is past any limits. This construction draws from the standard definition of addition on ordinals that can be found in [25].

In this comparison, appending and concatenation, although intrinsic to the monoidal construction, can be defined for the well-ordered sequences as well. But the latter is possible because of the existence of a total successor function on the ordinals. Given addition can be defined over the ordinals, and is associative, the ordinals already have a monoidal structure. As we will discuss later, generalizations of well-ordered sequences lose this notion of a successor, and thus lose a direct analog to this kind of appending and concatenation.

Consider then the operation of indexing the behavior, which is more intrinsic to the well-ordered sequences. We can define this operation $\mathbf{Index}(\alpha,\, n)$ where $n < \|\alpha\|$ (we have defined ordinal valued length for both of our cases). The definition for well-ordered sequences is immediate.

$$\begin{aligned}
&\mathbf{Index} \,:\, \sum \alpha : \mathit{Seq}(\mathcal{A}) \,\cdot\, \|\alpha\| \to \mathcal{A} \\
&\mathbf{Index}(\alpha,\, n) \overset{\text{def}}{=} \alpha(n)
\end{aligned}$$

In contrast, indexing the elements of the free monoid requires a construction that makes use of its universal property and the lifting we discussed earlier. We will first define the function

$$\begin{aligned}
&\mathbf{sample} \,:\, \mathcal{A} \to (\mathbf{Ords} \times \mathbf{Ords} + \mathcal{A}) \to (\mathbf{Ords} \times \mathbf{Ords} + \mathcal{A}) \\
&\mathbf{sample}(a,\, x) \overset{\text{def}}{=} \begin{cases} (S(n),\, m) & x = (n,\, m) \wedge n < m \\ a & x = (m,\, m) \\ x & \text{otherwise} \end{cases}
\end{aligned}$$

which takes in a "counter" $(n,\, m)$ and increments its first value, unless its has reached its limit m. In this case it outputs the value a. If a single value is sent into the function it simply passes it through. If this function, applied to a sequence of values a_1, a_2, a_3, ..., the partially applied functions can be composed together into a chain. Consequently, the composition can be passed a counter that moves along the chain, grabbing a value a_n when the counter reaches its limit, then passing this value through the rest of the chain. That is,

$$\mathbf{sample}(a_3) \circ \mathbf{sample}(a_2) \circ \mathbf{sample}(a_1) \circ \mathbf{sample}(a_0)$$

applied to $(0, 2)$ would return a_2, (1, 1) would return a_0, and (2, 14) would return (6, 14). To make use of this in this fashion, we take note that **sample** is a morphism in **Set** from $\mathcal{A}$ to the underlying set of the $\mathbf{Endo}^{op}(\mathbf{Ords} \times \mathbf{Ords} + \mathcal{A})$ monoid of endomorphisms with reverse composition (to preserve the direction of our ordering in the free monoid). Using the universal property of the free monoid, we can then lift **sample** into a homomorphism $\mathbf{sample}^\sharp$.

Before defining the indexing function using this construction, we will prove a simple lemma about it, that will be used later.

Lemma 1. *For $\alpha \in \mathbf{Mon}(\mathcal{A})$, and some ordinal n such that $\|\alpha\| \leq n$*

$$\mathbf{sample}^\sharp(\alpha)(0,\, n) = (\|\alpha\|,\, n)$$

Proof. This can be proven by induction over the words of the free monoid, which can be well-ordered by their length. Given some $\alpha \in \mathbf{Mon}(\mathcal{A})$

Case $\alpha = \mathbb{1}$

In this case $\|\alpha\| = 0$. $\mathbf{sample}^\sharp(\mathbb{1}) = \mathbf{id}_{\mathbf{Fun}}$ and $\mathbf{id}_{\mathbf{Fun}}(0,\, n) = (0,\, n) = (0,\, \|\alpha\|)$.

Case $\alpha = \alpha' \bullet \eta^{\mathcal{A}}(a)$

In this case

$$\|\alpha\| = \|\alpha' \bullet \eta^{\mathcal{A}}(a)\| = \|\alpha'\| + 1$$

Using the definition of the universal lifting

$$\begin{aligned}&\mathbf{sample}^\sharp(\alpha)\\ &= \mathbf{sample}^\sharp(\alpha' \bullet \eta^{\mathcal{A}}(a))\\ &= \mathbf{sample}^\sharp(\eta^{\mathcal{A}}(a)) \circ \mathbf{sample}^\sharp(\alpha')\\ &= \mathbf{sample}(a) \circ \mathbf{sample}^\sharp(\alpha')\end{aligned}$$

Using the inductive assumption, assuming that $\|\alpha^\sharp\| \leq n$,

$$\mathbf{sample}^\sharp(\alpha')(0, n) = (\|\alpha^\sharp\|,\, n)$$

hence, assuming that $\|\alpha\| < \|\alpha\| \leq m$

$$\begin{aligned}&\mathbf{sample}^\sharp(\alpha)(0,\, m)\\ &= \mathbf{sample}(a) \circ \mathbf{sample}^\sharp(\alpha')(0,\, m)\\ &= \mathbf{sample}(a)(\|\alpha'\|,\, m) = (\|\alpha\|,\, m)\end{aligned}$$

□

This establishes, as would be expected, that sending the counter through the chain of sampling functions yields a counter incremented to the length of the word if the limit is higher than or equal to this length.

We can then define the indexing function.

$$\mathbf{Index} : \sum \alpha : \mathbf{Mon}(\mathcal{A}) \cdot \|\alpha\| \to \mathcal{A} \tag{3.2}$$

$$\mathbf{Index}(\alpha, n) \overset{\text{def}}{=} \mathbf{sample}^{\sharp}(\alpha)(0, n) \tag{3.3}$$

We can be sure that the codomain is in fact $\mathcal{A}$ because the dependent type of the domain constrains the index n to be within the length of the monoid. This will be proven.

Proposition 1. *The above defined* **Index** *indeed has* $\mathcal{A}$ *as its codomain.*

Proof. This can be proven by induction over the length of α.

Case $\alpha = \mathbb{1}$
In this case, since $\|\alpha\| = \emptyset$, $\mathbf{Index}(\mathbb{1})$ is simply the empty function $\emptyset_{\mathcal{A}}$ by definition.

Case $\alpha = \alpha' \bullet \eta^{\mathcal{A}}(a)$ In this case $\|\alpha\| = \|\alpha'\|+1$. Let $m \in \|\alpha\|$, that is, $m < \|\alpha\|$. Expanding the definition

$$\mathbf{sample}^{\sharp}(\alpha)(0, m) = \mathbf{sample}^{\sharp}(\alpha' \bullet \eta^{\mathcal{A}}(a))(0, m)$$
$$= [\mathbf{sample}^{\sharp}(\alpha') \bullet \mathbf{sample}(a)](0, m)$$

Either $m < \|\alpha'\|$ or $m = \|\alpha'\|$. In the former case, using the induction hypothesis

$$\mathbf{sample}^{\sharp}(\alpha')(0, m) \in \mathcal{A}$$

hence

$$[\mathbf{sample}^{\sharp}(\alpha') \bullet \mathbf{sample}(a)](0, m) \in \mathcal{A}$$

In the latter case, using Lemma 1

$$[\mathbf{sample}^{\sharp}(\alpha') \bullet \mathbf{sample}(a)](0, \|\alpha'\|) = \mathbf{sample}(a)(\|\alpha'\|, \|\alpha'\|) = a \in \mathcal{A}$$

□

In summary, we have defined length, **Append**, **Concat**, and **Index** for both kinds representations. Having **Index** defined for words in free monoids, we are justified in using the same notation α_n to notate $\mathbf{Index}(\alpha, n)$, although during this chapter the more explicit notation will be maintained for continuity.

Interchangeability of Representation

Given that we can capture the same details in both of these kinds of representation, and can define the same basic operations on both of them, it is unsurprising that they are often conflated. One may apply the Kleene Star to a set $\mathcal{A}^*$ and proceed to append, concatenate, or index its members. The justification for this conflation can be made explicit by constructing transformations between the two spaces $\mathit{Seq}(\mathcal{A})$ and $\mathbf{Mon}(\mathcal{A})$.

In the direction of free monoids to sequences, the transformation follows directly from the indexing function 3.2.

$$\mathbf{toSeq} : \mathbf{Mon}(\mathcal{A}) \to \mathit{Seq}(\mathcal{A})$$
$$\mathbf{toSeq}(\alpha) \overset{\text{def}}{=} (\|\alpha\|, \mathbf{Index}(\alpha))$$

The corresponding transformation from sequences to free monoids can be constructed inductively, however with a notable caveat.

$$\mathbf{toMon} : \mathit{Seq}(\mathcal{A}) \to \mathbf{Mon}(\mathcal{A})$$
$$\mathbf{toMon}(\alpha) \overset{\text{def}}{=} \begin{cases} \mathbb{1} & \|\alpha\| = 0 \\ \mathbf{toMon}(\alpha \restriction_{\kappa}) \circ \eta^{\mathcal{A}}(\alpha(\kappa)) & \|\alpha\| = S(\kappa) \\ \bigvee_{m<\kappa} \mathbf{toMon}(\alpha \restriction_{\mathbf{m}}) & \|\alpha\| \text{ is a limit} \end{cases}$$

The caveat to this definition is that although prefix ordering is defined over words in the free monoid, in the standard construction we did not include transfinite-length words, and thus the least-upper bound of transfinite chains do not exist. If we restrict our sequences to finite ones, we can ignore the final transfinite case and have a legitimate definition. But, if we add transfinite words into the free monoid, this definition could be used to map into them from the corresponding transfinite preimages. However, we will defer this consideration to a later time.

Notwithstanding the transfinite case, it can be shown that these two transformations are inverses. To prove this, first we will prove a couple Lemmas. The first shows that the last element of the word can be extracted using the predecessor to the length.

Lemma 2. *For $\alpha \in \mathbf{Mon}(\mathcal{A})$ and $a \in \mathcal{A}$,*

$$\mathbf{Index}(\alpha \bullet \eta^{\mathcal{A}}(a)) = a$$

Proof. Firstly,

$$\|\alpha \bullet \eta^{\mathcal{A}}(a)\| = \|\alpha\| + 1 > \|\alpha\|$$

Expanding the definition

$$\begin{aligned} &\mathbf{Index}(\alpha \bullet \eta^{\mathcal{A}}(a))(\|\alpha\|) \\ &= \mathbf{sample}^\sharp(\alpha \bullet \eta^{\mathcal{A}}(a))(0, \|\alpha\|) \\ &= \mathbf{sample}^\sharp(\eta^{\mathcal{A}}(a)) \circ \mathbf{sample}^\sharp(\alpha)(0, \|\alpha\|) \\ &= \mathbf{sample}(a) \circ \mathbf{sample}^\sharp(\alpha)(0, \|\alpha\|) \end{aligned}$$

Using Lemma 1

$$\mathbf{sample}(a) \circ \mathbf{sample}^\sharp(\alpha)(0, \|\alpha\|) = \mathbf{sample}(a)(\|\alpha\|, \|\alpha\|) = a$$

□

The next two show what happens when the indexing domain of the **Index** function is restricted.

Lemma 3. *Let* $\alpha \in \mathbf{Mon}\mathcal{A}$.

$$\mathbf{sample}^\sharp(\alpha) \restriction_{\mathcal{A}} = \circ_{\mathcal{A}}$$

Proof. This can be proven by induction over words $\alpha \in \mathcal{A}$.

Case $\alpha = \mathbb{1}$
$\mathbf{sample}^\sharp(\mathbb{1}) = \circ_{\mathcal{A}}$

Case $\alpha = \alpha' \bullet \eta^{\mathcal{A}}(a)$
Expanding the definition

$$\mathbf{sample}^\sharp(\alpha' \bullet \eta^{\mathcal{A}}) = \mathbf{sample}(a) \circ \mathbf{sample}^\sharp(\alpha')$$

Using the induction hypothesis, for $x \in \mathcal{A}$

$$\mathbf{sample}(a) \circ \mathbf{sample}^\sharp(\alpha')(x) = \mathbf{sample}(a)(x) = x$$

□

Lemma 4. *For* $\alpha, \beta \in \mathbf{Mon}(\mathcal{A})$,

$$\mathbf{Index}(\alpha \bullet \beta) \restriction_{\|\alpha\|} = \mathbf{Index}(\alpha)$$

Proof. Let $\kappa < \|\alpha\|$. Expanding the definition

$$\begin{aligned}&\mathbf{Index}(\alpha \bullet \beta)(\kappa)\\ &= \mathbf{sample}^\sharp(\beta) \circ \mathbf{sample}^\sharp(\alpha)(0, \kappa)\\ &= \mathbf{sample}^\sharp(\beta) \circ \mathbf{Index}(\alpha)(\kappa)\end{aligned}$$

From the Proposition 1 it is assured that $\mathbf{Index}(\alpha)(\kappa)$ is in $\mathcal{A}$. Then Lemma 3 can be applied, completing the proof. □

From these lemmas, two additional important facts can be derived in the following proposition.

Proposition 2.

- $\mathbf{Index}(\mathbf{id}) = \emptyset_{\mathcal{A}}$
- *For* $\alpha \in \mathcal{A}$

$$\mathbf{Index}(\alpha' \bullet \eta^{\mathcal{A}}(a)) = \mathbf{Index}(\alpha) + \{\|\alpha\| \mapsto a\}$$

Proof.

- This follows from the dependent type of the **Index** function.
- Given that the domain of $\mathbf{Index}(\alpha' \bullet \eta^{\mathcal{A}}(a))$ is $\|\alpha' \bullet \eta^{\mathcal{A}}(a)\| = \|\alpha\| + 1$ the function can be decomposed

$$\begin{aligned}&\mathbf{Index}(\alpha' \bullet \eta^{\mathcal{A}}(a))\\ &= \mathbf{Index}(\alpha' \bullet \eta^{\mathcal{A}}(a)) \restriction_{\|\alpha\|} + \{\|\alpha\| \mapsto \mathbf{Index}(\alpha' \bullet \eta^{\mathcal{A}}(a))(\|\alpha\|)\}\end{aligned}$$

Applying Lemma 4 to the first term and Lemma 2 to the second completes the proof.

□

Using all of the lemmas, and the above proposition, it can be shown that **toMon** and **toSeq** are inverse functions (restricting to finite sequences).

Theorem 1. *The functions* **toMon** *and* **toSeq** *are inverses.*

Proof. This will be shown in each direction.

Case toMon ∘ toSeq

Expanding the definition, on an arbitrary element $\alpha \in \mathbf{Mon}\mathcal{A}$

$$\mathbf{toMon} \circ \mathbf{toSeq}(\alpha) = \mathbf{toMon}(\|\alpha\|, \mathbf{Index}(\alpha))$$

Proceeding by induction over words

Case $\alpha = \mathbb{1}$

Since $\|\alpha\| = 0$ $\mathbf{toMon}(0, \ldots) = \mathbb{1}$

Case $\alpha = \alpha' \bullet \eta^{\mathcal{A}}(a)$

Since $\|\alpha\| = \|\alpha'\| + 1$

$$\begin{aligned}&\mathbf{toMon}(\|\alpha\|, \mathbf{Index}(\alpha' \bullet \eta^{\mathcal{A}}(a))\\ &= \mathbf{toMon}(\|\alpha'\|, \mathbf{Index}(\alpha' \bullet \eta^{\mathcal{A}}(a) \restriction_{\alpha'}) \bullet \eta^{\mathcal{A}}(\mathbf{Index}(\alpha' \bullet \eta^{\mathcal{A}}(a)(\|\alpha'\|)))\\ &= \mathbf{toMon}(\|\alpha'\|, \mathbf{Index}(\alpha')) \bullet \eta^{\mathcal{A}}(a)\\ &= \alpha' \bullet \eta^{\mathcal{A}}(a)\end{aligned}$$

where the penultimate equivalence uses Lemmas 4 and 2, and the last equivalence uses the induction hypothesis.

Case toSeq ∘ toMon

This case will be shown by induction over the underlying ordinal of an arbitrary sequence α.

Case $\|\alpha\| = 0$

In this case

$$\begin{aligned}&\mathbf{toSeq} \circ \mathbf{toMon}(\alpha)\\ &= \mathbf{toSeq}(\mathbb{1}) = (0, \emptyset_{\mathcal{A}})\end{aligned}$$

Case $\|\alpha\| = \kappa + 1$

In this case

$$\begin{aligned}&\mathbf{toSeq} \circ \mathbf{toMon}(\alpha)\\ &= \mathbf{toSeq}(\mathbf{toMon}(\alpha \restriction_{\kappa}) \bullet \eta^{\mathcal{A}}(\alpha(\kappa)))\\ &= (\|\mathbf{toMon}(\alpha \restriction_{\kappa})\| + 1, \mathbf{Index}(\mathbf{toMon}(\alpha \restriction_{\kappa}) \bullet \eta^{\mathcal{A}}(\alpha(\kappa))))\end{aligned}$$

Using the induction hypothesis on $\alpha \restriction_{\kappa}$, since

$$\|\alpha \restriction_{\kappa}\| < \|\alpha\|$$

it follows that

$$\|\mathbf{toMon}(\alpha \restriction_{\kappa})\| = \kappa$$

Since the domain of

$$\mathbf{Index}(\mathbf{toMon}(\alpha \restriction_{\kappa}) \bullet \eta^{\mathcal{A}}(\alpha(\kappa)))$$

is $\kappa + 1$ we can consider evaluating this function on κ and on $m < \kappa$. In the former case, using Lemma 2

$$\mathbf{Index}(\mathbf{toMon}(\alpha \restriction_{\kappa}) \bullet \eta^{\mathcal{A}}(\alpha(\kappa)))(\kappa) = \alpha(\kappa)$$

In the latter case, we can restrict the function to κ and use the Lemma 4

$$\begin{aligned}&\mathbf{Index}(\mathbf{toMon}(\alpha \restriction_{\kappa}) \bullet \eta^{\mathcal{A}}(\alpha(\kappa)))(m)\\ &= \mathbf{Index}(\mathbf{toMon}(\alpha \restriction_{\kappa}) \bullet \eta^{\mathcal{A}}(\alpha(\kappa))) \restriction_{\kappa} (m)\\ &= \mathbf{Index}(\mathbf{toMon}(\alpha \restriction_{\kappa}))(m)\end{aligned}$$

Then using the induction hypothesis

$$\mathbf{Index}(\mathbf{toMon}(\alpha \restriction_{\kappa}))(m) = \alpha(m)$$

□

This establishes that our two transformations constitute a bijection between finite sequences and words in the corresponding free monoid, but this only establishes by itself that there are equal numbers of both representations. What more must be established to show that these are isomorphic for the purposes for which they are used as behavioral representations, we must show that this bijection commutes with all of our basic operations. We will prove this.

Theorem 2. *Sequences and Free Monoids are Isomorphic*
The bijection **toSeq** *between* $\mathbf{Mon}(\mathcal{A})$ *and* $Seq(\mathcal{A})$ *(along with its inverse* **toMon***) commute with or preserve* **Append***,* **Concat***, and* **Index***.*

Proof. In order to distinguish between the two versions of each function, we will use superscripts M for monoid and S for sequence.

We will start with the simplest case, showing

$$\mathbf{Index}^M = \mathbf{Index}^S \circ \mathbf{toSeq}$$

For any word $\alpha \in \mathbf{Mon}(\mathcal{A})$

$$\mathbf{Index}^S(\mathbf{toSeq}(\alpha)) = \mathbf{Index}^S(\|(\|\alpha),\, \mathbf{Index}^M(\alpha)) = \mathbf{Index}^M(\alpha)$$

Next we will show that

$$\mathbf{Append}^S(a,\, \mathbf{toSeq}(\alpha)) = \mathbf{toSeq}(\mathbf{Append}^M(a,\, \alpha))$$

Expanding the LHS

$$\begin{aligned}&\mathbf{Append}^S(a,\, \mathbf{toSeq}(\alpha)) = \mathbf{Append}^S(a,\, (\|\alpha\|,\, \mathbf{Index}^M\alpha))\\&= (\|\alpha\| + 1,\, \mathbf{Index}^M(\alpha) + \{\|\alpha\| \mapsto a\})\end{aligned}$$

Expanding the RHS

$$\begin{aligned}&\mathbf{toSeq}(\mathbf{Append}^M(a,\, \alpha)) = \mathbf{toSeq}(\alpha' \bullet \eta^{\mathcal{A}}(a))\\&= (\|\alpha' \bullet \eta^{\mathcal{A}}(a)\|,\, \mathbf{Index}^M(\alpha' \bullet \eta^{\mathcal{A}}(a)))\\&= (\|\alpha\| + 1,\, \mathbf{Index}^M(\alpha) + \{\|\alpha\| \mapsto a\})\end{aligned}$$

where the last equivalence uses Proposition 2.

Finally we will show that

$$\mathbf{Concat}^S(\mathbf{toSeq}(\alpha),\, \mathbf{toSeq}(\beta)) = \mathbf{toSeq}(\mathbf{Concat}^M(\alpha,\, \beta))$$

Expanding the RHS

$$\begin{aligned}&\mathbf{toSeq}(\mathbf{Concat}^M(\alpha,\, \beta)) = \mathbf{toSeq}(\alpha \bullet \beta)\\ &= (\|\alpha \bullet \beta\|,\, \mathbf{Index}^M(\alpha \bullet \beta))\\ &= (\|\alpha\| + \|\beta\|,\, \mathbf{Index}^M(\alpha \bullet \beta))\end{aligned}$$

Expanding the LHS

$$\mathbf{Concat}^S(\mathbf{toSeq}(\alpha),\, \mathbf{toSeq}(\beta)) = \mathbf{Concat}^S((\|\alpha\|,\, \mathbf{Index}^M(\alpha)),\, (\|\beta\|,\, \mathbf{Index}^M(\beta)))$$

Therefore we must show that

$$\mathbf{Concat}^S((\|\alpha\|,\, \mathbf{Index}^M(\alpha)),\, (\|\beta\|,\, \mathbf{Index}^M(\beta))) = (\|\alpha\| + \|\beta\|,\, \mathbf{Index}^M(\alpha \bullet \beta))$$

which will be proven by induction over words for $\beta \in \mathbf{Mon}(\mathcal{A})$.

Case $\beta = \mathbb{1}$
Since $\|\beta\| = 0$

$$\mathbf{Concat}^S((\|\alpha\|,\, \mathbf{Index}^M(\alpha)),\, (0,\, \ldots)) = (\|\alpha\|,\, \mathbf{Index}^M(\alpha))$$

Case $\beta = \beta' \bullet \eta^{\mathcal{A}}(b)$
Since $\|\beta\| = \|\beta'\| + 1$

$$\begin{aligned}&\mathbf{Concat}^S((\|\alpha\|,\, \mathbf{Index}^M(\alpha)),\, (\|\beta\|,\, \mathbf{Index}^M(\beta)))\\ &= \mathbf{Append}^S(\mathbf{Index}^M(\beta' \bullet \eta^{\mathcal{A}}(b))(\|\beta'\|),\\ &\qquad \mathbf{Concat}^S((\|\alpha\|,\, \mathbf{Index}^M(\alpha)),\, (\|\beta'\|,\, \mathbf{Index}^M(\beta' \bullet \eta^{\mathcal{A}}(b)) \restriction_{\|\beta'\|})))\\ &= \mathbf{Append}^S(b,\, \mathbf{Concat}^S((\|\alpha\|,\, \mathbf{Index}^M(\alpha)),\, (\|\beta'\|,\, \mathbf{Index}^M(\beta'))))\end{aligned}$$

Continuing with the induction hypothesis

$$\begin{aligned}&= \mathbf{Append}^S(b,\, (\|\alpha\| + \|\beta'\|,\, \mathbf{Index}^M(\alpha \bullet \beta')))\\ &= (\|\alpha\| + \|\beta'\| + 1,\, \mathbf{Index}^M(\alpha \bullet \beta') + \{\|\alpha\| + \|\beta'\| \mapsto b\})\\ &= (\|\alpha\| + \|\beta\|,\, \mathbf{Index}^M(\alpha \bullet \beta' \bullet \eta^{\mathcal{A}}(b)))\\ &= (\|\alpha\| + \|\beta\|,\, \mathbf{Index}^M(\alpha \bullet \beta))\end{aligned}$$

□

What follows from this is that sequences under this concatenation operator, and with the empty sequence as an identity, forms a monoid isomorphic to the free monoid.

Corollary 1. $\mathbf{Mon}(\mathcal{A}) \cong (\mathit{Seq}(\mathcal{A}),\, \mathbf{Concat},\, (0,\, \emptyset_{\mathcal{A}}))$

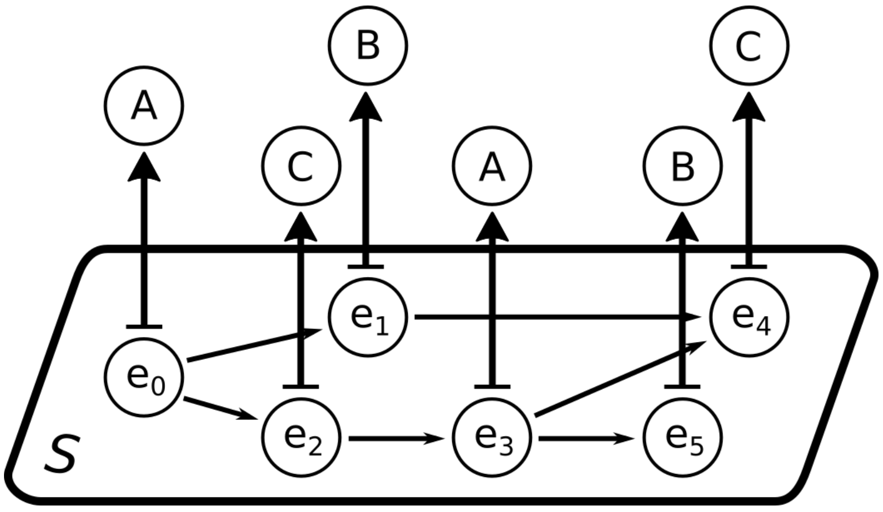


Figure 3.3: Generalized sequence.

Proof. Using Proposition 2, the unit is preserved by **toSeq**, and by the above theorem the product is as well, thus **toSeq** is a monoid homomorphism. Since **toSeq** is a bijection, it is more a monoid isomorphism. □

In other words, these two systems, free monoids and spaces of sequences, can indeed be used interchangeably in representing the behavior of sequential systems. And perhaps this is a clear reason why a greater distinction was not drawn between the two. Only when we move to a more general kind of computation do these two approaches diverge into generalizations that are most certainly not isomorphic in this same fashion.

3.2 Generalizing

Moving away from these sequential systems and their correspondingly sequential behaviors, our aim is to represent the behaviors of HDDAs, which are concurrent. In this chapter, our aim is first to get a sense of what kind of mathematical object could serve this purpose, generalizing from those representing sequential behaviors. What are the generalizations of sequences and of free monoids?

The answer to the former case is the one that appears most frequently in the literature. Generalizing the sequence means replacing the underlying model of sequential time with a less constrained one. The most obvious choice is to go from a well-order, or ordinal space of events to a partially ordered one. That is, a representation of behavior as pairs

$$\alpha = (S, \hat{\alpha} : \|S\| \to \mathcal{A})$$

where S is a poset. This is depicted in Figure 3.3. Other similar candidates for S include acyclic graphs, and pointed variations of the like such as domains (pointed ω-CPOs) if a

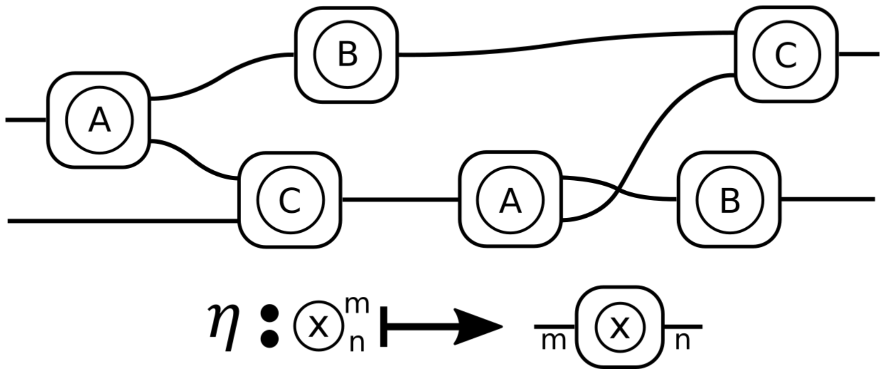


Figure 3.4: Diagram in a monoidal category.

foundation is needed. Time is these models can be thought of as relativistic or asynchronous, not embedded into a global linear timeline. Although such an embedding could be done as a matter of realizing the execution in a Newtonian notion of global time, it certainly cannot be done uniquely. We will call this class of representations *generalized sequences*.

The answer in the latter case is a generalization of a free monoid that offers a notion of concurrency. Such a generalization can be found in category theory in the form of monoidal categories. It is this generalization, rather than the former, that we will develop into our proposed behavioral representation. In order to develop this approach to modeling concurrent behavior we will go into great detail defining and characterizing the specific kind of monoidal categories we would like to use, namely a free symmetric monoidal categories, which will be done in Chapter 6. This will involve giving clear general definitions first for monoidal categories and there variants, and for the free construction in Chapter 5.

Here we will give just enough of a preview of monoidal category to make an argument about why these two generalizations are no longer as interchangeable as their sequential counterparts. In a monoidal category, the structures that we will be dealing with will be more like acyclic ported block diagrams. Reflecting on the picture of a ordinary monoid as a chain, shown in Figure 3.2, the elements of a monoidal category have both the ability to be linked sequentially, a role now played by the composition operator $\circ$, and joined in parallel with an additional product $\otimes$. This is illustrated in Figure 3.4. Although the term *monoidal* emphasizes the additional parallel composition operation $\circ$ the in our analogy here the monoidal product is replaced by categorical composition $\circ$.

Although these generalizations are both graph-like, insomuch as their sequential counterparts are both linear, it must be emphasized how manifestly different these representations are. Whereas the general topological models have events as points and simple relations connecting them, the elements of monoidal categories, which we will call diagrams, place a specific interface around each event determining how many incoming and outgoing edges connect to them, and moreover in which order. Even more specifically, the interface can be typed and thus constraints can be embedded in each event determining which other events

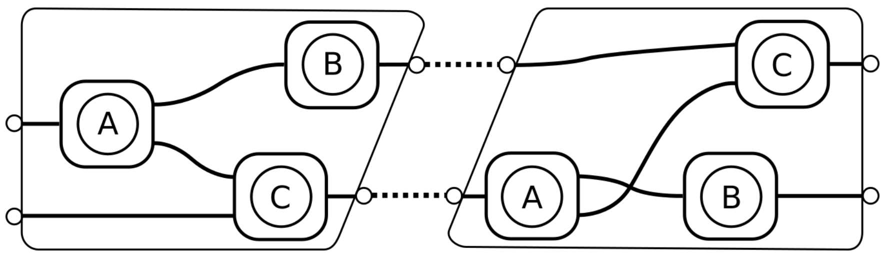


Figure 3.5: Composition in a monoidal category.

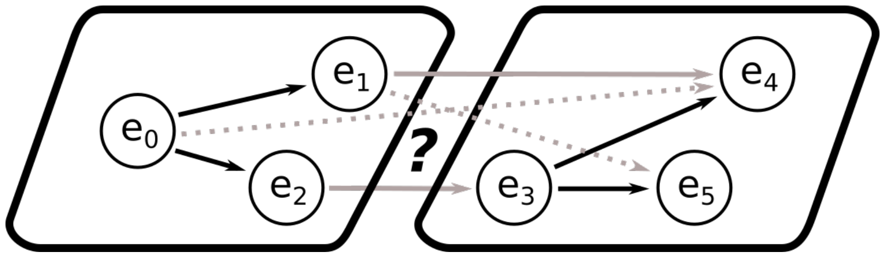


Figure 3.6: Ambiguity in composing generalized sequences.

can be connected to it. An entire diagram also may have unconnected incoming and outgoing ports determining how it may be connected to other diagrams in a larger context. Standard partial orders and graphs do not have these features, and thus cannot make distinctions articulated by them. In contrast with the sequential case, the generalized representations contain manifestly different information.

Furthermore, the interfaces of events in diagrams in monoidal categories is what makes it possible to compose them. The existence of a clearly defined composition, made possible by the categorical composition operator $\circ$, means that the operations of appending and concatenation can be defined in monoidal category just as directly as they can in free monoids. This is illustrated in Figure 3.5. In contrast, there is no obvious way to define concatenation over relational structures without additional information beyond points and relations being added to the structure, or without parameterizing the operation by explicit instructions on how exactly to compose two graphs or posets. This difference is illustrated in Figure 3.6, showing that it is not generally clear which events from each graph to connect with a relation or ordering.

Which Generalization?

Granted that these two concurrent generalizations are most certainly different, and have different features, the question then can be raised as to which of them is more appropriate for representing the behaviors of HDDAs running in Process Fields. In Chapter 4 we will discuss particular kinds of generalized sequences found in the literature, and will argue that these approaches do not provide the kind of representation we need, pushing us in the direction of the other approach, free monoidal categories. But we can begin here to distinguish between these approaches through more general considerations regarding what is essential to the representation of computational behavior.

As opposed to the sequential case, in which we define appending and concatenation, as well as indexing, on both sequences and words in free monoids, in the generalizations this is no longer true. Even in the sequential case we can see that the notion of appending and concatenation are, in a sense, more direct in the monoidal representation. These operations are algebraic ones, determining compositions. Concatenation is carried through the generalization into free monoidal categories, and provides the same means of composition. Likewise, indexing is more direct in the sequence representation, since indexing is spatial, a mapping from one space, in this case an ordinal, to another, a set of event types. In generalized sequences the idea of indexing is still retained in the form of a more general space. Given that having both of these features becomes complicated in the generalization we are compelled to ask an important question. Which of these operations, appending/concatenating or indexing, is more crucial to meaning when we are concerned with establishing it upon a computational system?

Reflecting on the fundamental intuitions behind computation, one cannot help but see a bias towards appending and concatenating as more essential to formulating a behavioral semantics. Arguing why this is the case will require some elaboration. Causality immediately comes to mind as one of these fundamental intuitions, and causality alone would not be enough to side on this debate. Although, there is a case to be made that causality depends on a notion of an underlying space in which one can state for any event, which other events upon which it can depend. With an ordered space, a familiar condition of causality can be stated; one which might look like the following.

$$\bar{\alpha}(e) = F(\{\bar{\alpha}(e') \in \|S\| \mid e' < e\})$$

Here, the behavior at e depends, via some function F, on the strict principal downset of e in poset S. That is, what happens can only depend on what happened before it. This is usually what we understand causality to mean, and it is nearly always formulated in a manner that uses indexing rather than concatenation.

However, this kind of causality would only seem to be enough in the sequential finite case, where we have a clear notion of a predecessor or strict prefix (apart from the empty behavior). As can be seen in work such as that of Matsikoudis [41], strict causality may be too broad a notion when generalizing to non-sequential or infinite domains (his point leans more towards the latter). That a behavior, at some moment in a model of time, depends on past behavior is inadequate to establish that the behavior can be constructed iteratively; or, in short, computed. What is more specifically necessary in defining the semantics of computational behaviors is *constructiveness*, the property that the behavioral representation can be reached through induction, albeit potentially transfinitely, from some initial behavioral fragment. This is a somewhat elaborate way of stating that a computation moves forward, pushing into the future, rather than pulling from the past[2]. Rather than simply exist as a function of its history, a computation builds on it, progressing, and adding to it. This suggests the importance of concatenation over indexing.

The most familiar instance of this constructiveness appears in the *least fixed-point* semantics given to iterative constructs to define their denotations [61]. While this kind of semantics was initially devised by Scott, Strachey, and others to denotationally construct the functions computed by programs, it also became used for the construction of computational histories. The prototypical example of this is the semantics given to KPNs by Kahn and MacQueen[30], in which the semantics defined for a network of processes, subject to certain important constraints, is given as the history of tokens passed through all of the channels connecting these processes. In particular, this history is defined to be the least fixed point of a function representing the network operating on these histories.

Generally, in this fixed-point schema, computation is modeled as a monotone endomorphism on an ordered structure, such as a domain D

$$\begin{aligned}&F\,:\,D\rightarrow D\\&\forall\,a,\,b:D\,\cdot\,a\leq b\;\Rightarrow\;F(a)\leq F(b)\end{aligned}$$

This function computes a history in D from another history in D and thus the conventional denotation associated with the computation modeled by F is the least history x satisfying the fixed-point equation

$$x = F(x)$$

For a general function of this sort, without further stipulations, such a least fixed points may or may not exist. Moreover, fixed points other than the least fixed points may exist for the function. There are several additional criteria that can secure the existence of a least fixed point. If the domain has a finite height, it must exist. If the domain is ω-complete (or directed complete) and F is Scott-Continuous, it must exist. While this is the most frequently used case, as Park points out in [52], when the domain is complete up to a sufficient limit ordinal κ and F commutes with limits at κ, this is also sufficient.

What is the connection between fixed-points, monotonicity, and constructiveness then? Kleene's fixed-point theorem uses monotonicity to construct the least fixed-point iteratively from the bottom element $\bot$ of the domain D, through the repeated application of F. This particular fixed-point is important precisely because it is reachable via iteration, hence constructive. The proof that this constructive fixed-point exists uses the following two consequences of monotonicity

$$\begin{aligned}&\bot\leq F(\bot)\\&\alpha\leq F(\alpha)\;\Rightarrow\;F(\alpha)\leq F(F(\alpha))\end{aligned}$$

[2]I credit this analogy to Gil Lederman

to construct an increasing orbit starting at $\bot$

$$\bot \leq F(\bot) \leq F^2(\bot) \leq F^3(\bot) \leq \dots$$

which models, in some respect, the computation itself inductively extending its semantic value. That is, over this orbit

$$\alpha \leq F(\alpha)$$

This could be interpreted straightforwardly as the notion that a computation produces behavior as a consequence of state. Given the state of a system can be concretely represented as a function of (or relation on) its past events, this amounts to stating that a computation extends behavior. To this end, it is more essential that F is increasing on its $\bot$ orbit, and converging to some limit, than it is that F is monotonic. The latter is simply used as a sufficient condition for the former.

Constructiveness is therefore a more essential intuition than causality, because it emphasizes the representation of the semantics as being inductively reachable through iteration, and if we think of a computation as extending its behavior, the notion of appending or concatenating to it certainly captures this idea. This would suggest a monoidal representation of behavior, rather than one that is only a generalized sequence.

From this perspective, when concatenation is possible, a behavioral semantics can alternatively be defined by a function **next** that determines for any behavior, any history of computation so far, what to appended to it.

$$\alpha_{\kappa+1} = \alpha_\kappa \bullet \mathbf{next}(\alpha_\kappa)$$

We can then derive an endofunction

$$F(\alpha) = \alpha \bullet \mathbf{next}(\alpha)$$

which is increasing on its entire domain by definition (though not necessarily monotonic). Hence, if we have a free monoidal representation of behavior, we should always, in principle, be able to define such a **next** function. This is a kind of “forward” or “progressive” causality.

Indeed, in cases where a domain is already monoidal, and has a well-defined notion of concatenation that is consistent with its ordering, given an endomorphism F

$$\forall\, x \leq F(x) \,\cdot\, \exists\, \gamma \,\cdot\, F(x) = x \bullet \gamma$$

as follows from the ordering definition. Consequently, we can Skolemize the above in order to define **next** over the domain of postfixed points.

$$\exists\, \mathbf{next} \,\cdot\, \forall\, x \leq F(x) \,\cdot\, F(x) = x \bullet \mathbf{next}(x)$$

This is the case in models such as KPN or DF, since the domains in the fixed-point semantics of these models are indeed monoids (up to their infinite members) and their prefix order over histories is consistent with concatenation.

But in this kind of formalism, how does one deal with limits and convergence? For a complete behavior x, beyond which there is no progress, $\mathbf{next}(x) = \mathbb{1}$. Clearly, when this is the case,

$$F(F(x)) = F(x \bullet \mathbf{next}(x)) = F(x \bullet \mathbb{1}) = F(x)$$

Therefore, $\mathbf{next}(x) = \mathbb{1}$ implies that x is a fixed point of the derived F.

To address limits, we need to use the ordering on the monoid to define the limit

$$F^\omega(x) \overset{\text{def}}{=} \bigvee_{k<\omega} F^k(x)$$

The condition that the represented computation goes no further than ω can then be recast as the condition that

$$\mathbf{next}(F^\omega(x)) = \mathbb{1}$$

Then, of course, $F^\omega(x)$ is a fixed point.

Let us formalize this.

Definition 1. *Let M be an $\aleph_0$-complete* pomonoid *($\aleph_0$-CPOM) if M is defined as the tuple*

$$M = (\|M\|, \bullet, \mathbb{1}, \leq)$$

where $(\|M\|, \bullet, \mathbb{1})$ is a monoid and $(\|M\|, \leq)$ is an $\aleph_0$-CPO, under the condition that for all a, $b \in \|M\|$

$$a \leq b \Leftrightarrow \exists\, c \cdot b = a \bullet c$$

Again, the condition cannot be generally be taken as a definition for the ordering since it only generally defines a preorder; that is, antisymmetry must be established. We are choosing closure here under all countable limits so as to accommodate an extended free monoid that can have ordinal length words. Less may be necessary. We can then define a condition under which a function **next** gives rise to an iteration function with a least fixed-point. This condition is defined as follows.

Definition 2. *Given a $\aleph_0$-CPOM M, a function* $\mathbf{next} : \|M\| \to \|M\|$ *is* finitely supported *iff for all $x \in \|M\|$ such that x is not finite (in the order-theoretic sense)* $\mathbf{next}(x) = \mathbb{1}$.

Here, by x being *finite*, we are using the order-theoretic definition, that x is *way-below* itself, $x << x$. That **next** is *finitely supported* simply means that it drops off to $\mathbb{1}$ at all of the non-finite points.

It can be proven, then, that this is a sufficient criteria for the iterator defined by **next** function having a constructive fixed-point.

Theorem 3. *If* **next** *is finitely supported, then for all $m \in \|M\|$ the function defined*

$$F(x) \stackrel{def}{=} x \bullet \mathbf{next}(x)$$

has a fixed point $\hat{m}$ such that

$$\hat{m} = F^{\omega}(m)$$

Proof. Suppose that $y = F^{\omega}m$ is finite. Since the chain

$$m \leq F(m) \leq F^2(m) \leq F^3(m) \leq \dots$$

has y as its limit

$$y = \bigvee_{k<\omega} F^k(m)$$

it follows from $y << y$ that there must be a $n < \omega$ such that $y \leq F^n(m)$. But it is also the case that $y \geq F^k(m)$ for all $k < \omega$. Therefore, $y = F^n(m)$. Since, F is increasing, $y \leq F(y)$. But since

$$F(y) = F(F^n(m)) = F^{n+1}(m)$$

and $n + 1 < \omega$, it also follows that

$$y \geq F^{n+1}(m) = F(y)$$

Therefore, $F(y) = y$. If, otherwise, y is not finite, then

$$F(y) = y \bullet \mathbf{next}(y) = y$$

by the finitely supported property. □

The consequence of this theorem is that we can replace a semantics based on continuous endomorphisms over domains with a semantics based on finitely supported endomorphisms over pomonoids (such as free monoids).

Given that when concerned with most all practical computations there is an interest in defining the behavioral consequences of a finite histories, extending a **next** function to be $\mathbb{1}$ on infinite histories should be possible most of the time. Whereas when dealing with a continuous physical process, behavior can certainly depend on an infinite history. However, the order-theoretic structure of the continuum precludes even defining a clear notion of a monoid or of a **next**. This is true as well for many generalized sequences, for which it can be said that no clear **next** function can be defined, just as the ordered space itself possesses no notion of a successor analogous to that of well-orders. In these cases, one is most likely confined to dealing with fixed points in the traditional fashion.

All of this would suggest that for computations (as opposed to all physical systems) a monoidal representation of behavior, even concurrent behavior, captures something more essential about computation than one based on a generalized sequence. Nevertheless, in the next chapter we will explore several existing representations of concurrent behavior, none of which go fully down the path of monoidal categories we will explore, leading to our definition of OEGs

Chapter 4

Existing Representations of Distributed Behavior

The study of non-sequential computing systems has a long and diverse history that reflects the similarly diverse series of circumstances that motivated work in this area. While the core of the theory of computation was built upon the notion of computational steps forming a sequence of actions or transformations, that ultimately conclude in a final state or diverge, as Winskel and Nielsen argue in [69]

> ... in reality, few computational systems are sequential. On all levels, from a small chip to a world-wide network, computational behaviours are often distributed, in the sense that they may be seen as spatially separated activities accomplishing a joint task. Many such systems are not meant to terminate, and hence it makes little sense to talk about their behaviours in terms of traditional input-output functions. Rather, we are interested in the behaviour of such systems in terms of the often complex patterns of stimuli/response relationships varying over time. ([69])

This further emphasizes the role of *behaviors* as semantics, particularly in the shift from studying sequential to non-sequential systems.

Given what we argued in the beginning of Chapter 3, developing a model of distributed behavior may involve both generalizing away from sequential ones as well as abstracting fewer details away from the physical realities of the systems being studied. And insomuch as these systems have varied greatly, from small clusters of processing devices with highly regularized communication mechanisms to the IoT, the appropriate models of the systems and their behavior have varied as well, providing answers to the key questions associated with a particular kind of system or platform.

Therefore, while it is the case that a plethora of models for parallel, concurrent, and distributed computing exist throughout the literature, the demands of the particular platforms we have in mind, and have explicated in Chapter 2, have lead us to develop a new behavioral representation rather than use one of the many existing ones, or even a simple variant. Of the demands we have placed on a representation of behavior, perhaps the most important, for the purposes of contrasts, are modularity, composability, and true concurrency.

In Chapter 3, we considered a spectrum of possible behavioral representations, starting with the equivalent approaches of sequences and free monoids, and expanding outward into the inequivalent branches of generalized sequences and free monoidal categories. We argued in the end of the Chapter why our demands suggested favoring the last of these possibilities, free monoidal categories. In this Chapter, we will review existing representations of concurrent behavior and discuss where they fall in this spectrum, and why they lack the properties we seek for reasoning about HDDAs in Process Fields.

4.1 Which Kinds of Systems?

Before reviewing some existing approaches to modeling concurrent behavior, it is worth first discussing the particular kinds of systems that influenced their development and identify ways in which these systems are different from the ones in which we have taken interest, such as the IoT. In contrast with the representation of behavior itself, which serves to define the semantics of a parallel, concurrent, or distributed system, there are the models of systems. This category consists of programming languages, process calculi, transition systems, and other *Models of Computation* (MoCs).

As we have argued in Chapter 2, of the existing Models of Computation the Hewitt Actor Model provides one of the more realistic pictures of what execution looks like in the Process Fields. In contrast with this very dynamically structured model, many system or platform models that have influenced the development of concurrent representations of behaviors simply multiply a sequential computational platform, creating one with several sequential components. These components are often then given a fixed topology of communication connections through which they share information or have access to shared resources.

Early process calculi containing a concurrency primitive, such as *Communicating Sequential Processes* (CSP), are rooted in the intuition of a system consisting of a fixed network of sequential components. Hoare makes this clear in an early paper on CSP [21], citing that

> The programs expressed in the proposed language [CSP] are intended to be implementable, both by a conventional machine with a single main store, and by a fixed network of processors connected by input/output channels... It is consequently a rather static language... ([21])

This is reflected in *Calculus of Communicating Systems* (CCS) as well, which could not yet send the names of channels. Process network models such as that of Kahn and MacQueen [30], and dataflow models such as that of Dennis [13], are also based in the systemic model of a statically networked collection of otherwise sequential processes. On the side of logic, Pnueli introduces *Linear Temporal Logic* (LTL) in [55] as a logic for reasoning about both sequential and concurrent systems, with the model of concurrent systems again a fixed collection of sequential processors modeling "n programs being concurrently run by n processors"[55].

In contrast with these early models of concurrency, the Hewitt Actor Model[19] makes far more radical assumptions about the system, lacking a fixed network of sequential computing entities. But Actors are nevertheless still internally sequential entities, albeit dynamically created and configured. One consequence of this is the interleaving of received messages. This structural dynamism is reflected in Milner's π-calculus[48], but this calculus (at least in its first conception) similarly imposes a sequentiality on message reception, and thus bears a similar retention of a sequential primitive beneath the concurrency.

4.2 Traces

While we identified two potential types of generalizations of sequential behavioral representations in Chapter 3, a third measure to accommodate non-sequentiality is to simply flatten it down into sequences, retaining the same essential forms of sequential behavioral representations. This approach goes generally under the name of *trace theory*. The intuition behind this approach is that although operations may be happening concurrently, if observed, they will ultimately fall into a particular sequential order for the observer. Hoare summarizes this view in [22]

> Imagine there is an observer with a notebook who watches the process and writes down the name of each event as it occurs. We can validly ignore the possibility that two events occur simultaneously; for if they did, the observer would still have to record one of them first and then the other, and the order in which he records them would not matter. ([22])

This model of concurrent behavior consists for a particular system, therefore, of all interleaved traces of its events. Winskel refers to this model, particularly if it is closed under prefixes, as *Hoare Traces* or *Hoare Languages*. The advantage of this system is that it maintains the simplicity of dealing with either sequences or words in free monoids, and therefore is a composable representation at the least. However, the key problem with this model is that without additional mechanisms a single run or execution of a concurrent system could generate many such traces and there is no distinction in these traces between interleaving and non-deterministic choices.

Moreover, composition becomes problematic with this representation. While on one hand, composing two individual traces α and β is a clearly defined mathematical operation $\alpha \bullet \beta$, in the context of the interleaving of concurrent events composing two collections of traces A and B by composing their members may exclude possible interleaving that would occur if the operations modeled by A and B could overlap with each other concurrently.

For instance, suppose one has a simple system that could perform actions in the alphabet $\{a, b, c\}$, where actions b and c both depend on a having happened, but can occur concurrently with each other; additionally, suppose a can only happen again after b. If set $A = \{abc, acb\}$ constitutes the possible traces of the system with one invocation, two invocations in succession could be composed $A \bullet A$ resulting in the set

$$A \bullet A = \{abcabc, acbacb, abcacb, acbabc\}$$

of compositions of traces. But, unless each iteration of A is synchronous, this composition misses traces *abacbc* and *ababcc*, which interleave events from the two iterations. Therefore, in general, there is not enough information in interleaved traces themselves to determine how to compose them.

Finally, interleaved traces do not scale well as a representation. To completely characterize the executions of a system with a number of concurrent components, the permutations of interleaved orderings may increase exponentially with this number. This problem is discussed in the context of verifying programs in a *threaded* MoC in [34]. This is a particularly significant problem when addressing platforms such as the IoT, which may consist of very large numbers of devices. The idea of representing their behavior by permuting every ordering of every one of them is at the least cumbersome and awkward, not to mention the two other issues.

These problems have all been well-known for a long time, but they are important to restate, as they prototype the challenges associated with a representation of concurrent behavior. Nevertheless, because LTL, or one of its close cousins, have remained the *sine qua non* of concurrent system verification, and interleaved traces serve as the model for LTL [55], traces have not been easy to dispense of. Two approaches to dealing with some of these problems are either to include with traces more information about the system that produced them, or to abandon sequential representations and move to one of the two generalized forms we discussed in Chapter 3.

We will first look at the former of these approaches, which attempt to abate the problems with traces while maintaining their basic form.

Mazurkiewicz Traces

In many concurrent systems, particularly ones with fixed structures, there are certain causality relationships that can be established between types of events. Observing this, one can take any individual interleaved trace of behavior and construct rules determining which subsequent events could be reordered, specifically on account of the fact that they occur concurrently, and thus independently. For instance, if events a and b always occur independently, the two traces $ac\mathbf{ba}dca$ and $ac\mathbf{ab}dca$ can be identified as two different interleavings of the same essential process.

A formal theory around this observation was formulated by Mazurkiewicz in [43], which represented traces specifically as words in a *partially commutative* monoid. These monoids, as opposed to free ones generated from an alphabet $\mathcal{A}$, $\mathbf{Mon}(\mathcal{A})$, have in addition to the axioms a set of commutativity rules for select pairs of elements. To continue the above example, if a and b commute, that is $a \bullet b \cong b \bullet a$, then

$$a \bullet c \bullet \mathbf{b} \bullet \mathbf{a} \bullet d \bullet c \bullet a \cong a \bullet c \bullet \mathbf{a} \bullet \mathbf{b} \bullet d \bullet c \bullet a$$

In general, every commuting pair a and b will generate an equivalence class

$$\alpha \bullet \mathbf{b} \bullet \mathbf{a} \bullet \beta \cong \alpha \bullet \mathbf{a} \bullet \mathbf{b} \bullet \beta$$

for all words α and β. Any set of commuting pairs from the alphabet, thereby partitions the free monoid into clusters of possible traces reachable from each other by exchanges.

Using this partially commutative monoid, a run of a system can be represented by any representative trace t from one of its equivalence classes $\langle t \rangle$. This representation is known as a Mazurkiewicz Trace and is defined in [43], and further elaborated in [44], [45], and [1]. In [1], which gives the most comprehensive treatment of Mazurkiewicz Trace amongst these references, a Reliance Alphabet characterizes the behavioral possibilities of a particular system, defining for this system, and its behavioral traces, its partially commutative monoid.

Definition 3. *Reliance Alphabet*
A Reliance Alphabet H is defined

$$H \overset{def}{=} (\mathcal{A}, \mathcal{I})$$

where the alphabet $\mathcal{A}$ *is a set of event types and the* independence relation $\mathcal{I}$ *is a symmetric irreflexive binary relation over $\mathcal{A}$.*

Aalbersberg and Rozenberg, in [1], also include the complement of $\mathcal{I}$, the *dependence relation*, in the Reliance Alphabet, but this is simply a derivative of $\mathcal{I}$. The purpose of the independence relation is simply to define which pairs of event types in $\mathcal{A}$ commute in the words of $\mathbf{Mon}(\mathcal{A})$. To be specific, for each $(a, b) \in \mathcal{I}$, $a \otimes b = b \otimes a$, and thereby any two traces that can be related by this exchange are equivalent in H.

From $\mathcal{I}$ a set of commutativity axioms along with substitution generate a relation R_H over words in $\mathbf{Mon}(\mathcal{A})$, and the partially commutative monoid $\mathcal{M}_H$ constructed can be given the underlying set $\mathbf{Mon}(\mathcal{A})/R_H$. The product over this set can be defined as the equivalence class of the free product over any two representative words.

$$\langle t_1 \rangle \otimes \langle t_2 \rangle \overset{\text{def}}{=} \langle t_1 \otimes t_2 \rangle$$

This product is clearly well-defined as the canonical lifting of the free product through the equivalence class projection. The identity is then the equivalence class $\langle \mathbb{1} \rangle$ of the empty word $\mathbb{1}$, which contains only itself $\{\mathbb{1}\}$. Consequently, with respect to the equivalence classes H, it suffices to identify any element $\langle t \rangle$ by choosing a word t, and it suffices to simply concatenate these representatives to identify the composition of the runs they represent.

It is in this manner that Mazurkiewicz Traces attempt to solve problems that we identified with interleaved traces. The most obvious of these is that the partially commutative monoid enables a much greater economy of representation, because only one concrete trace is necessary to identify the entire set of traces that arise from the possible behaviors of a system. The more technically significant problem this formalism solves is that it is able to provide an unambiguous means to compose behavioral representations asynchronously. The structure of the equivalence class takes care of the permutations left out of a simple set of trace concatenations.

To see this consider an example similar to the one given previously involving a system with an alphabet $\mathcal{A} = \{a, b, c\}$. Suppose we include in the independence relation a single symmetric pair (b, c). Take then the trace $\langle bc \rangle = \{bc, cb\}$. If we take the pointwise product this set by itself, we get

$$\{bc, cb\} \otimes \{bc, cb\} = \{bcbc, bccb, cbbc, cbcb\}$$

However, the trace product $\langle bc \rangle \otimes \langle bc \rangle = \langle bcbc \rangle$ contains the two additional elements *bbcc* and *ccbb* left out of the pointwise product, which, in a sense, places a synchronization barrier between the two behaviors, only permitting a second b or c to occur after one of each has happened.

More powerfully even, a further lifting of the partially commutative monoid can be achieved with the underlying set $\mathcal{PM}_H \setminus \emptyset$, sets of Mazurkiewicz Traces. Here the pointwise product over sets makes more sense. We can define the product

$$A \otimes B \overset{\text{def}}{=} \{a \otimes b \mid a \in A, b \in B\}$$

between two sets of traces, and define the unit as the set $\{\langle \mathbb{1} \rangle\}$. We can then consider two distinct traces $\{\langle abc \rangle, \langle bac \rangle\}$ which cannot be transformed into one another through commutativity (owing to the fact that an equivalence class partitions a set). If we multiply this by itself in the naive setwise fashion we would be taking the product

$$\begin{aligned} &\{abc, acb, bac\} \otimes \{abc, acb, bac\} \\ &= \{abcabc, abcacb, abcbac, acbabc, acbacb, acbbac, bacabc, bacacb, bacbac\} \end{aligned}$$

and missing *abbcac* and *babcac*. Instead, using trace composition,

$$\{\langle abc \rangle, \langle bac \rangle\} = \{\langle abcabc \rangle, \langle bacabc \rangle, \langle abcbac \rangle, \langle bacbac \rangle\}$$

and thus the missing two permutations are covered in the third and fourth classes, respectively.

This would seem to solve the issue of composition, as well as achieve a distinction between concurrency and nondeterminism. However, this solution comes at a cost, paid in the limitations of what can be faithfully modeled by an independence relation. As can be seen in [43], and much of the literature on the subject, Mazurkiewicz had in mind, again, fairly rigid, static concurrent systems. In particular, Mazurkiewicz uses *Petri Nets* as his prototype of a system that can be modeled in this fashion. Event types are either always concurrent or always dependent.

If we go back to our original example illustrating the issue with composing traces, we give a fairly simple, intuitive scheme for the dependencies between the events a, b, and c.

Yet this scheme cannot be encoded as an independence relation. This can be easily checked via exhaustion. Since the independence relation is symmetric, and we know we would like b and c to be independent. If we stopped there, our example set $\{abc, acb\}$ would be the single trace $\langle abc\rangle$, and the product with itself would be the single trace $\langle abcabc\rangle$ consisting of only the four variants we deemed initially to be incomplete. If we make a and c independent then the trace $\langle abc\rangle$ picks up the variant cab, which is already too broad, because we want c to follow a. Specifically, we would like the c to commute with subsequent as but not antecedent ones.

This limitation places a clear bound on the complexity of the systems that can be represented behaviorally by traces, or otherwise forces more complex ones to be abstracted. Sassone, Nielsen, and Winskel observe this same limitation in [58], comparing them to other true concurrent representations of behavior such as Event Structures:

> Mazurkiewicz trace languages are too abstract to describe faithfully labelled event structures. Clearly, any trace language with alphabet $\{a, b\}$ able to describe such a labelled event structure must be such that $ab \cong ba$. However, it cannot be such that $aba \cong aab$. Thus, we are forced to move from the well-known model of trace languages. ([58])

As a response to this limitation, Sassone et al. propose *generalized* Mazurkiewicz Traces [58] which extend simple commutativity rules with general substitution to a more complex set of context sensitive commutativity rules. The single independence relation $\mathcal{I}$ is replaced by a function

$$\mathfrak{I} : \mathbf{Mon}(\mathcal{A}) \to 2^{\mathcal{A}\times\mathcal{A}}$$

mapping each word t to a symmetric irreflexive relation $\mathfrak{I}(t)$ satisfying several unsurprising consistency and coherence axioms. Ultimately, the equivalences generated by $\mathfrak{I}$ are those for each word $\alpha \in \mathbf{Mon}(\mathcal{A})$ and $(a, b) \in \mathfrak{I}(\alpha)$ such that

$$\alpha ab\beta \cong \alpha ba\beta$$

where β is any word in $\mathbf{Mon}(\mathcal{A})$. Constructing this relation, one can derive a quotient monoid in the same fashion as before.

This formalism clearly generates a richer variety of monoids than those defined by a single independence relation. However, what has been lost again in this very general lifting is the economy of the representation. Indeed, we could encode our example in this language, constructing an appropriate monoid. But in doing so, we would need to specify that, based on our rules, the context α such that $\alpha ac \cong \alpha ca$ are specifically those α in which there are fewer cs than as, so that the particular c in the ac fragment is the instance of c following an earlier a, such as was the case for $abacbc \cong abcabc$ in our original example.

Characterizing the commutativity rules is no longer simple and places a more formidable weight onto this model of the underlying system. It would be more direct in a representation

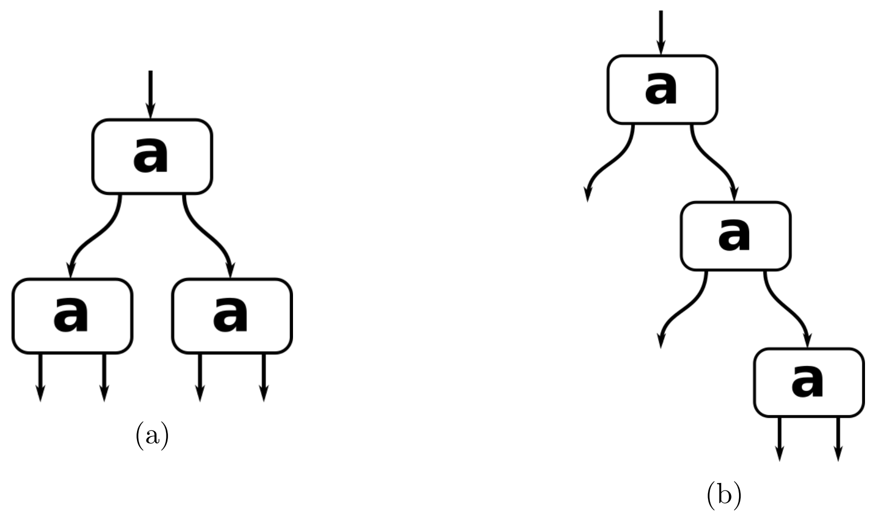


Figure 4.1: Two different processes that look the same to a sequential observer.

to simply identify which a and which c are connected. And this all still leaves unresolved the issue of concurrency amongst the same event type. Two a events, for instance, in these systems are always sequential. Two clear reasons exist for this. One epistemological reason is that so sequential observer will ever know if two instances of a ever commute between two runs. The other reason is ontological. The systems imagined in the development of these representations, again, are often topologically static networks of sequential elements operating concurrently. The archetype of an event type is an abstract sequential step taken on one of these components that may or may not be concurrent with steps taken by other components.

Our notion of representing behavior in a Process Field contrasts strongly with this picture, both in the ontology of a HDDA executing, as well as in our aim to devise a representation that is ontological in nature, rather than one that approximates ontology through a collection of projected perspectives. Without additional information we cannot distinguish between the two fragments of execution depicted in Figure 4.1, both of which would look like aaa to any sequential observer. Trace languages, even the most general, cannot easily distinguish between these without adding more contextual information.

4.3 True Concurrency

Throwing fully aside the constraints and complications of anchoring behavioral representation to a linear structure, albeit perhaps with commutativity rules or other means to aggregate ontologically equivalent interleavings of concurrent behavior, we arrive at a set of non-linear mathematical structures that inherently represent two concurrent events as non-sequential, and thus unordered. As opposed to interleaving representations, these non-sequential structures that represent concurrency inherently are aggregated under the approach of *true concurrency.*

Of these *true concurrent* representations, the subcategory of these mathematical models that has dominated the literature falls under the branch of representations we referred to in Chapter 3 as *generalized sequences.* These representations consist of an abstract space of events along with a relational structure such as a graph or partial order, labeled by event types. The relation between two events represents a causal relation, either direct or transitive, defining a concurrent model of logical time. In contrast with Mazurkiewicz Traces, which factor out the distinction between possible observers through commutativity rules, these representations identify the behavior itself in abstract terms that can be projected back into sequences.

The distinction is a little like the two different interpretations of vector spaces, one being that vectors and linear transformations are represented in particular coordinate systems, but have invariant properties and relationships that are conserved through coordinate transformations (which have a certain constrained form and structure). The other is that vectors are geometric objects in abstract vector spaces that can be coordinated various ways. Of course, the former is analogous to trace languages like Mazurkiewicz Traces, that provide a collection of coordinate transformations along with their vector as a tuple. The latter finds its analog in the more geometric forms of generalized sequences, which can likewise be coordinated by various topological orderings.

Lamport Clocks

An early notion of events as being inherently partially ordered in distributed systems, rather than totally ordered, is reflected in Lamport[32], who in contrast with Mazurkiewicz has in mind a system that is indeed spatially distributed, like the IoT. In fact, in this paper, Lamport identifies distributed systems specifically as

> [consisting] of a collection of distinct processes which are spatially separated, and which communicate with one another by exchanging messages. A network of interconnected computers, such as the ARPA net, is a distributed system. ([32])

One might recognize *ARPA net* as the nascent internet. For sure, Lamport's model was a rudimentary kind of IoT, where the "things" were nevertheless conventional computing systems.

Indeed, Lamport proceeds to construct a model of behavior under the same aforementioned assumption of a concurrent composition of sequential processes:

> We begin by defining our system more precisely. We assume that the system is composed of a collection of processes. Each process consists of a sequence of events... In other words, a single process is defined to be a set of events with an *a priori* total ordering. ([32])

In this model, a fixed collection of sequential processes send asynchronous messages to one another. And given each sequential process is behaviorally sequential, modulo the messages that are interacting with it from outside of this sequence, the totally events of each individual process can be modeled as well-ordered sequences, like those at the beginning of Chapter 3. The events of each process can therefore be represented as labeled ordinals $\alpha = (\kappa, \bar{\alpha})$. The underlying model of well-ordered time for each process constitutes for Lamport a *logical clock*, such that each event happening on any process happens at some logical time in that process.

Lamport defines an irreflexive partial ordering on the collection of events across all processes. Two events a and b are ordered $a < b$ when a happens before b in the same process, or when a represents the sending of a message from one process that is received as an event b in another process. This ordering of events by itself constitutes a partially-ordered model of behavior in the systems Lamport is considering. However, Lamport constrains these models to those in which the events are always also linearly ordered along one of a fixed collection of sequential processes. What follows from this model, which superimposes a partial order with a collection of total orderings are particular analyses relevant to the kind of system Lamport has in mind. The specific question asked in the paper that introduces this model is the question of whether, given first the collection of sequential orderings and the ordered pairs of message sending and receiving events, a strict partial ordering of events can be constructed; indicating that the collection of sequences and message pairs are consistent with each other.

For Lamport, the answer to this question comes in the form of an assignment of a single global index to each event which places them in a linear order that respects the underlying partial order, and gives a possible ordering for a sequential observer. These indexes are considered logical timestamps and constitute a *Lamport clock*, giving the whole collection of events a sequential model of logical time. The idea of a *vector clock*, often mis-attributed to this work by Lamport appears nearly a decade later in the work of Fridge[16] and Mattern[42], who elaborate on the work in [32].

A vector clock precisely recovers Lamport's original idea of an underlying partial ordering from the collection of event sequences and message pairing, indexing each event with a N-tuple of natural numbers, where N is the number of sequential processes. Getting into the details of both of these systems, while interesting, would be a departure from the aim of our discussion, but it suffices to say that both Lamport clocks and vector clocks are methods of taking a fixed collection of communicating sequential processes modeled as a combination of sequences and messaging pairs, and establishing the very existence of a true concurrent, partially-ordered model, the latter differing in that it explicitly constructs this model. In other words, the aim is less to study and reason about the behaviors then to simply show that they exist for the systems in question – that the behavioral semantics are, in a sense, definable. Consequently, there is little discussion in this work about how to work with these partial orders, particularly in the earlier case where Lamport does not even presume to have them directly.

Nonetheless, modeling this particular kind of concurrency with appropriate behavioral representations is still a relevant are of interest as can be seen in [7]. Any decent behavioral representation aimed at systems like the IoT should also be able to cover this ground, albeit a narrowly constrained subspace of what is possible in the systems we have in mind. We shall see in Chapter 7 an example of how OEGs can indeed subsume this kind of behavior into its more broadly applicable potentials.

Partial-Ordered Models of Behavior

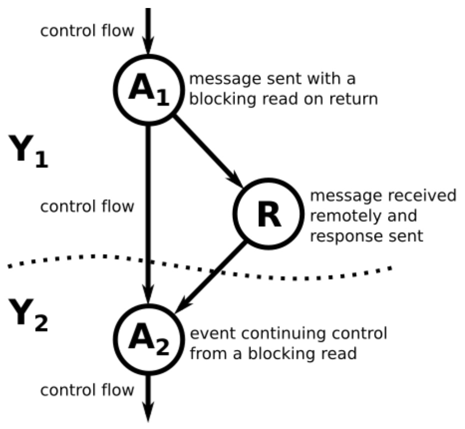
The epitome of a generalized sequence, and perhaps the beginning of what is considered true concurrency research, is found in *Event Structures*, first presented in [50], then further expanded on in Winskel's thesis[68]. In the latter, Winskel builds upon the work of Lamport, Hewitt, and Petri, constructing a sophisticated theory around partially-ordered representations of behaviors. In contrast with Lamport, the kind of systems Event Structures were devised to model were Petri Nets, much as in the work of Mazurkiewicz.

An *elementary* Event Structure is simply a poset $S = (E, \leq)$, consisting of a set of events E and a partial ordering. These events can be labeled by a function into some alphabet of event types $\mathcal{A}$ amounting to a *labeled* Event Structure, and in this we find our generalized sequence $\alpha = (S, \hat{\alpha})$ as a representation of behavior. In contrast to the work of Lamport, for which the partial ordering of events exists to witness the causal consistency of the relationship between sequential processes, Event Structure give the semantics of processes like Petri Nets that are more concurrent at their foundation.

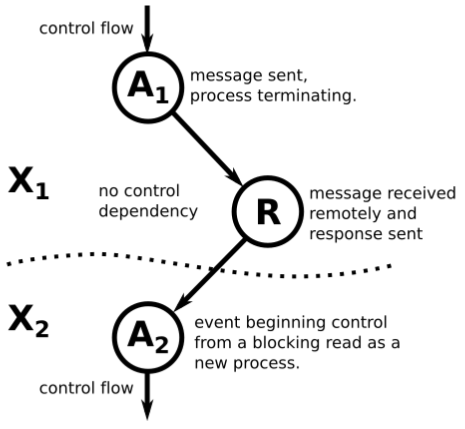
Consequently, Winskel et al. develop the mathematical foundations of these posets more than Lamport, intending to use them as a full denotational semantics, rather than a simply using them as a verification method. The focus of these developments in both [50] and [68] is showing how spaces of Event Structure can form Scott domains, in the same manner as the spaces of partial computable functions in the denotational semantics of sequential systems. A highly concurrent system with a complex causal network of events can be thereby be described as a directed set of approximations that characterize how computation in these systems progresses. In the earlier paper, this is described succinctly, identifying Event Structures as the "intermediate between nets and domains."[50]

While this work provides a compelling basis for using partial orderings as representations of behaviors, and certainly accomplishes the goal of bridging the gap between a broad range of concurrent systems, typically rooted in operational semantics, and the mathematics of denotational semantics, what is lost in the generality is a clear means to compose behaviors. This lack indeed reflected in the domain theoretic framing. As was discussed in Chapter 3, the domain theoretic means of expressing progress is extension by means of a monotonic function (or functional) operating over an increasing orbit, rather than an algebraic concatenation onto the history since no such concatenations exist for Event Structures. A fragment of execution represented as an Event Structures is therefore only meaningful as, and insomuch as it is a prefix of a longer execution. There is no coherence to the idea of cutting a piece of an execution out of context and treating it as a modular element that could be inserted somewhere else.

To see the complications that arise with both composition and modularity, consider an example shown in Figure 4.2. In this example, there are a pair of events A_1 and A_2, the former


control flow
A1
message sent with a blocking read on return
Y1
control flow
R
message received remotely and response sent
Y2
A2
event continuing control from a blocking read
control flow

(a) An event graph.


control flow
A1
message sent, process terminating.
X1
no control dependency
R
message received remotely and response sent
X2
A2
event beginning control from a blocking read as a new process.
control flow

(b) A different event graph with the same transitive closure.

Figure 4.2: Graphs of events indistinguishable under transitive closure.

of which sends a message to a third event R, and the latter of which receives a response from R. In the version shown in Figure 4.2a, A_2 depends on both the reception of the message from R, and on control flow following from A_1. A_1 and A_2 might be, for instance, a send command followed by a blocking read command constrained to the same sequential process. In the second version shown in Figure 4.2b, there is no control dependency between A_1 and A_2. Although these two graphs have the same transitive closure, and thus the events have the same causality ordering in both cases, nonetheless the graphs give different ontological information about the connections between the events.

If this additional information did not serve a purpose in answering important questions about the behaviors being represented, it could indeed be left out. And, in the contexts at which Event Structure is aimed the representation succeeds to provide enough information to determine questions regarding the causal ordering of events. However, in contrast with models of computation such as Petri Nets, in which the structure of the system is static, HDDAs consist of structures that can (and often will) change during execution. Consequently, it is important in the context of HDDAs to represent behaviors as free standing, modular fragments of behavior that can be composed.

We must consider the example in Figure 4.2 from a perspective of composition and modularity. Dividing the two graphs along the dotted line to form execution fragments gives two different dependency interfaces, characterizing the flow of effects from the upper part of the graph to the lower part. In Figure 4.2a, there are two dependencies crossing the dotted line, and thus when divided into two fragments Y_1 and Y_2, the lower fragment Y_2 depends on both the reception of a message and the passing of control happening prior to the fragment. While Y_1 provides these dependencies, X_1, the upper fragment shown in Figure 4.2b, does not. Therefore, one could compose Y_1 with Y_2 and get precisely the execution in Figure 4.2a, whereas composing X_1 with Y_2 would not fulfill the dependencies of Y_2. Instead, the result

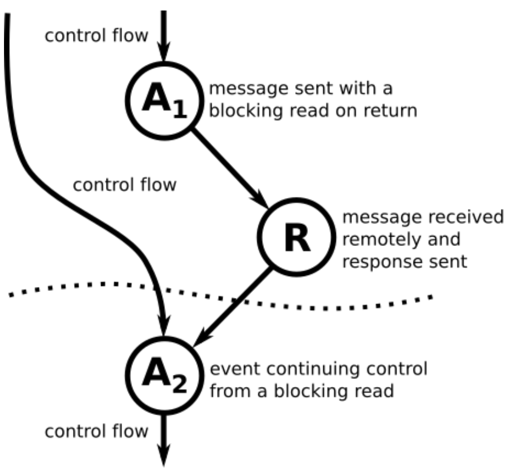

control flow
A1
message sent with a blocking read on return
control flow
R
message received remotely and response sent
A2
event continuing control from a blocking read
control flow

Figure 4.3: A potential composite of parts of two different graphs.

of a composition between X_1 and Y_2 might look more like what is shown in Figure 4.3, in which the additional dependency in Y_2, unfulfilled by X_1, becomes an incoming dependency of the composition.

One might argue, to the contrary, that direct dependency could be lost in the interior of the execution fragment, since the kind of composition shown above happens at the boundaries, but this would limit reasoning about these fragments to only additively composing them. Maintaining the direct dependencies makes the operation of composition simpler, but more importantly, opens the possibility to define propositions about interior fragments. For instance, one could state that there exists a suffix common to the fragments in Figure 4.2a and Figure 4.3, namely Y_2, but Y_2 is not a suffix of the fragment in Figure 4.2b.

Ironically, some of the information missing in a partially ordered representation of behavior, as illustrated in the above example, that would be necessary to give a meaning to algebraic composition over them is the kind of information that was lost from causal Petri Nets in their transformation to Event Structure. Specifically, the cites at which behaviors could be joined together in a meaningful way, in the manner we have illustrated above and suggested in the notion of monoidal categories, are represented to some degree by the *places* or *conditions* in Petri Nets, projected out in their transformation to Event Structures. In [50], it is made clear that Event Structures lose information in Petri Nets when projecting out conditions, which is why the relation between the two spaces is an adjunction, but not an isomorphism. It is argued that this conflation of Petri Nets in their representation as Event Structures is an acceptable one for the purposes of reasoning about domain theory, which may very well be true. But it is clear how the existence of peripheral places lost in the transformation have have abundantly clear implications on a notion of composition.

Partially-Ordered Multisets

Although works introducing Event Structures do not attempt to form an algebra of out of these representations, it is not impossible to form a kind of algebra out of posets. Pratt, in [56], introduces an algebra of *pomsets*, partially-ordered multisets, which are simply labeled poset. Of course, as we have seen, there are many precedents for using labeled posets as representations of behavior, however Pratt contributes to these precedents by introducing an algebra. In doing so, an attempt is made to retain some of the algebraic advantages of sequential representations of behavior while moving to a true concurrent representation like that of Winskel et al.

Specifically, an important advantage of sequential models of behavior, and their extension to concurrency through the mechanism of interleaving, that Pratt identifies as contributing to their prominence against true concurrent models (at the time of writing, but perhaps enduring for much longer after) is that they can be used to form *Kleene Algebras.* These algebras build off of a notion of concatenation in the representation of individual behaviors, lifting this notion into operations on sets of behaviors. If individual runs are individual sequences, or traces, processes can be described as the set of all possible behaviors that could arise, the multiplicity arising specifically out of the nondeterministic choices in the system (including the abstraction away of the possibility of a multiplicity of inputs from the environment). If one starts with concatenation, or similar kinds of algebraic compositions in the individual behaviors, one can then build a Kleene Algebra on the level of processes and thereby algebraically reason about processes.

To this end, Pratt defines an algebra for pomsets. For comparative simplicity, we will transliterate Pratt's definitions into the language of generalized sequences that we have already introduced in Chapter 3. A pomset, in this language, is defined as follows.

Definition 4. *Given an alphabet of actions $\mathcal{A}$, a pomset p over $\mathcal{A}$ is defined*

$$p \overset{def}{=} (S, \bar{p})$$

where S is a poset and $\bar{p}$ is a function

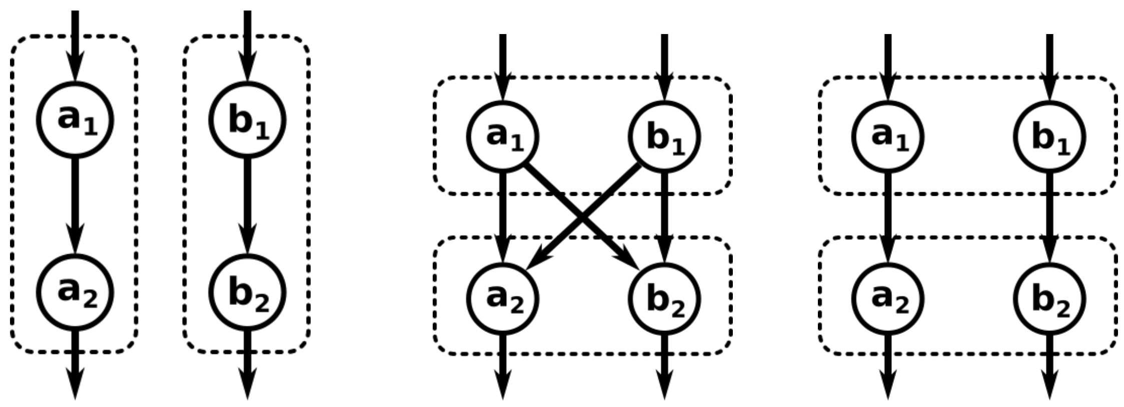
$$\bar{p} : \|S\| \to \mathcal{A}$$

To be more specific, Pratt defines these structures only up to order isomorphism so that any two structures can be given posets with disjoint underlying sets. Using this definition, two compositions, *concurrence* and *concatenation* can be defined as follows.

Definition 5. *Given an alphabet of actions $\mathcal{A}$, and two pomsets*

$$p = ((S_p, \leq_p), \bar{p})$$
$$q = ((S_q, \leq_q), \bar{q})$$

over $\mathcal{A}$, with disjoint S_p and S_q



(a) $(a_1 \,;\, a_2) \,||\, (b_1 \,;\, b_2)$ (b) $(a_1 \,||\, b_1) \,;\, (a_2 \,||\, b_2)$ (c) $(a_1 \,||\, b_1) \bullet (a_2 \,||\, b_2)$

Figure 4.4: Compositions of pomsets, the dotted rectangle indicates the first grouping.

1. *the concurrence $p \,||\, q$ of p and q is defined*

$$p \,||\, q \overset{def}{=} ((S_p \cup S_q,\ \leq_p \cup \leq_q),\ \bar{p} \cup \bar{q})$$

2. *the concatenation $p \,;\, q$ of p and q is defined*

$$p \,;\, q \overset{def}{=} ((S_p \cup S_q,\ \leq_p \cup \leq_q \cup S_p \times S_q),\ \bar{p} \cup \bar{q})$$

where $\bar{p} \cup \bar{q}$ is the coproduct of the two functions.

Descriptively, concurrence combines the events of two pomsets, making the events from one incomparable with the other. This operation corresponds to combining two independent concurrent behaviors. Concatenation, on the other hand, combines p and q such that for every event $e_p \in p$ and $e_q \in q$, $e_p \leq e_q$ in the composition. In short, every event in q is dependent on every event in p.

Without going into details about the Kleene algebra over these operations, it can be seen that these two operations form a kind of algebra over pomsets that does indeed have a notion of concatenation, which seems to provide the possibility of what we are looking for in a behavioral representation. However, this concatenation is too restrictive, suffering from the very same issue introduced initially in the case of interleaved sets. There is no way in this algebra of concatenating two pomsets while only introducing selective dependencies.

Take the extreme instance of two independent sequential components A and B, each which produces sequences of event types a and b respectively (we will give the event types subscripts to clarify which individual events they label). If we take a single event behavior from each component and combine them concurrently, we get $a \,||\, b$ (abusing the notation a bit to conflate single events with single-event pomsets). If we take each component and sequentially repeat each behavior we get $a_1 \,;\, a_2$ and $b_1 \,;\, b_2$, which combined concurrently forms

$(a_1\,;\,a_2)\,||\,(b_1\,;\,b_2)$. If we, instead, concatenate two copies of $a\,||\,b$, the result $(a_1\,||\,b_1)\,;\,(a_2\,||\,b_2)$ has more dependencies than $(a_1\,;\,a_2)\,||\,(b_1\,;\,b_2)$. This is illustrated in Figure 4.4, in which the latter ordering of compositions in 4.4a can be compared to the former in 4.4b. In short,

$$(a_1\,;\,a_2)\,||\,(b_1\,;\,b_2) \neq (a_1\,||\,b_1)\,;\,(a_2\,||\,b_2)$$

in spite of our sense that there should be some form of composition $\bullet$ where we can have

$$(a_1 \bullet a_2)\,||\,(b_1 \bullet b_2) \neq (a_1\,||\,b_1) \bullet (a_2\,||\,b_2)$$

as is shown in Figure 4.4c, we would need some kind of additional information to determine that a_2 depends only on a_1 and b_2 depends only on b_1, let alone any number of alternatives, such as b_2 depending on a_1, but b_1 remaining independent of a_2.

One way to allow more flexibility, while maintaining the general algebraic form of concatenation is to parameterize the operator. In [23], Hudak uses pomsets to represent the behavior of evaluation in functional programming languages. In his treatment of the pomset algebra, in addition to concurrence and concatenation, he defines a *restricted concatenation* as follows.

Definition 6. *Given an alphabet of actions $\mathcal{A}$, two pomsets*

$$p = ((S_p,\, \leq_p),\, \bar{p})$$
$$q = ((S_q,\, \leq_q),\, \bar{q})$$

over $\mathcal{A}$, with disjoint S_p and S_q, and a predicate $P \subseteq S_q$, the restricted concatenation $p \bullet_P q$ of p and q is defined

$$p \bullet_P q \overset{def}{=} ((S_p \cup S_q,\, \leq_p \cup \leq_q \cup \leq_P),\, \bar{p} \cup \bar{q})$$
$$\leq_P = \{(e_p, e_q) \in S_p \times S_q \;|\; P(e_q)\}$$

While this provides a family of compositions that interpolate between concurrence and concatenation, this still does not completely cover the interstitial territory. Nevertheless, if the unary predicate that parameterizes Hudak's restricted concatenation were widened to a binary one, explicitly stating which event in q depends on which event in p, only then would we have a completely expressive notion of concurrent behavioral concatenation. But this kind of a composition would no longer form a straightforward algebra with a concrete set of operators that observe a clear set of axioms.

We find ourselves with the same problem of needing somewhere to put the extra information necessary for complete composition. Mazurkiewicz Traces, and their generalization, place this additional information in the partially commutative monoid, generated by the independence relation. If we did take the approach of parameterizing pomset algebras as aggressively as we have suggested, the information is then put into the operators in the form of parameters. A third alternative would be to put this information in the representations themselves, which is precisely what we intend to do, and to this end, posets are insufficient. A richer representation is needed to meet our aims.

4.4 Other Representations

Thus far we have focused on two families of behavioral representation that are pervasive in the literature, trace languages such as Mazurkiewicz Traces and labeled posets such as pomsets and Event Structures. Other kinds of representations of concurrent behavior can be found, but they mostly bear similarity to what has already been addressed, offering no clear solutions to our demands for composition and modularity. We will discuss some of these more briefly.

Canonical Dependency Graphs

It is emphasized in [1] that Mazurkiewicz Traces can be alternatively represented in the form of *directed acyclic graphs* (DAGs). In contrast with the trace languages, these graphs have the advantage of internalizing the details of the partially commutative monoid. Rather than representing behavior as the conjunction of a representative (in the sense of an equivalence class) sequence and the systemic details of the independence relation, this conjunction can be transformed into a unique DAG, called a *canonical dependency graph*. This transformation is performed by weakening the linear relation of sequentiality in the representative sequence wherever two neighboring events are independent in the independence relation. In [1], a simple and intuitive algorithm is given for this.

While this would seem to place additional information in the representation, and given that DAGs are more general than posets, since they do not have to be transitively closed, one might suppose that canonical dependency graphs are more expressive than labeled poset representations, this turns out not to be the case. It is shown in [1] that canonical dependency graphs are indeed isomorphic to their transitive closures. The reason that this is possible is that the set of canonical dependency graphs, like posets, is a proper subset of the entire space of labeled DAGs that could be formed from the same alphabet of event types. This is due specifically to the fact that any two labeled events in the canonical dependency graph can only be concurrent if they share an edge in the independence graph, and more must be concurrent if they are in the independence graph and do not have an interposing event between them. In other words, the partially commutative monoid is still present in the restriction of possible DAGs that could be canonically dependency graphs.

Furthermore, because canonical dependency graphs contain the same essential information as Mazurkiewicz Traces, the possibility of defining an algebraic concatenation of canonical dependency graphs still relies on the independence relation to determine how to connect the events between two graphs being composed. Hence, the internalization of the independence relation does not provide enough means to determine composition, as it cannot generally be inferred from any given graph nor could it be guaranteed *a posteriori* that two canonical dependency graphs are compatible with each other. For a single representation, the independence relation can be abstracted away, but for the algebra to make sense, it cannot be dispensed with.

Another way of looking at this would be to ask what the relationship is between Mazurkiewicz Traces, and thus canonical dependency graphs, with labeled Event Structures or pomsets. Sassone et al. answer this question in [59], proving that pomsets (or in their words deterministic semilanguages), deterministic Event Structure, and generalized traces are all isomorphic. Given that generalized traces are a strict superset of Mazurkiewicz Traces, it can be concluded that this representation, and its true concurrent counterpart, canonical dependency graphs, are both more constrained than the labeled poset representations we have discussed.

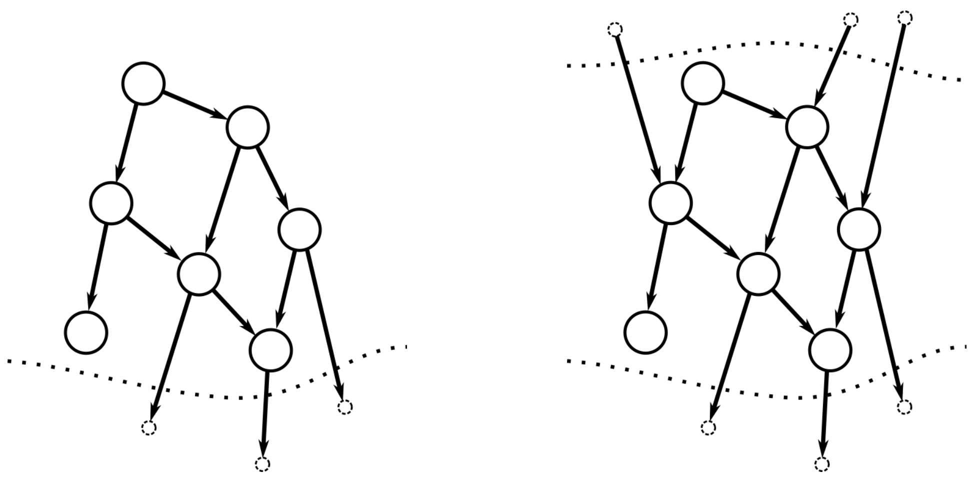
Actor Event Diagrams

Given that in investigating existing representations of concurrent behavior, we have stressed the kind of models each representation had been developed to characterize, and that the models that influenced their development were significantly different from those we have in mind, it would make sense to look at how the behaviors of systems most closely related to those we are interested in are represented. As we discussed in Chapter 2, HDDAs bear a much closer resemblance in behavior to the Hewitt actor model, than a Petri Nets or static networks of communicating sequential processes. For this reason, it would be important to consider efforts to behaviorally represent the Hewitt actor model.

In [6], Hewitt and Baker suggest that the appropriate model for representing the behavior of actor models is that of an DAG. In this graph, each event is the arrival of a message at an actor, and the edges arise out of two relationships between arrivals: the *arrival ordering* and the *activation ordering*. The former of these corresponds to the causal relationship between two successive message arrivals being witnessed by the same actor. The latter corresponds to the relationship between the arrival of one message at one actor, and as a consequence of a message being sent in response to this message, the arrival of this sent message at its destination. In this manner, a DAG of arrivals could be derived from the execution of a particular model.

This representation is expanded on significantly in the thesis of Clinger [12], who calls these graphs of arrival events Actor Event Diagrams. Clinger formally works out some of the suggestions in [6] and ultimately reaches an end similar to Winskel et al. in giving actor models a Scott-like semantics through the construction of a domain of Actor Event Diagrams. Insomuch as the semantics of the actor model can be codified in the form of a continuous function over this domain, the behavior of the model can be identified as the least fixed-point of this function. Much like the domain-oriented semantics of Petri Nets, given as Event Structures in [50] and [68], the approach taken to computational progression is again extension rather than concatenation. Therefore, like Event Structures, Actor Event Diagrams also lack a clear general notion of composition.

On the other hand, the technique Clinger uses suggests something more than just a graph. The way that extensions are determined in Actor Event Diagrams is that included with the graph itself are the pending messages that are produced in the execution, but have not yet arrived at the actors to which they are being sent. The result is a construction that



(a) An Actor Event Diagram with pending messages forming a boundary.

(b) An open event diagram with both pending and incoming messages.

Figure 4.5: Event diagrams with boundaries.

combines an DAG with a notion of a boundary of open ended forward dependencies. This is illustrated in Figure 4.5a in which the dotted line depicts a boundary along which there are three pending messages. These pending messages provide an interface that determines how the graph can be extended, since there are clear cites to which further events can be attached.

If a step further were taken, and not only pending messages, but initial messages were incorporated into the graph, as is illustrated in Figure 4.5b, there would then be in two such graphs a pair of boundaries determining the possibility of concatenation. In fact, something along these lines is proposed by Talcott in [64], who builds on Clinger's Actor Event Diagrams. Recognizing the importance of representing modular fragments of behavior in a composable fashion, Talcott introduces *Open Event Diagrams* as a modular behavioral representation for Actor Models. She indeed argues, as we have here, that the semantics originally posited by Hewitt and Baker in [6] does not have a clear concept of composition because it only considers complete executions of models in isolation. And further, although Clinger's model includes incomplete behaviors, these behaviors are still isolated, unable to represent incoming dependencies from the surrounding environment, or other behaviors for that matter.

Although Talcott's representation is very likely the closest to being a monoidal categorical one, it nevertheless still lacks a simple binary operator for sequential composition. This is a consequence of its dependence on names rather than order to govern composition. Instead of employing a combinational algebra, Open Event Diagrams are connected via a contraction over particular names. In essence, this is a feedback operation rather than a sequential composition. Consequently, it is more general operation that does not generally preserve the acyclic property of the diagrams. In order to preserve this essential property the operation must be made a partial one contingent on the transitive dependency relations of pending messages on initial messages for each diagram. Talcott's system is therefore close to being a *traced symmetric monoidal category*[29], though missing a number of critical details. Nevertheless, Open Event Diagrams do provide for a means of establishing composition and modularity and are structurally the closest representation discussed so far to what we will define later as OEGs.

Kahn Histories

A final representation worth considering, although it is not often though of as a representation of behavior, is that of channel histories in the denotational semantics of Kahn processes [30] - what we might call *Kahn histories*. In order to describe these adequately, one must first set the scene of KPNs, and thereby also identify the kind of system that influenced the formulation of this particular behavioral representation. A KPNs is a constituted of a collection of deterministic sequential processes that communicate tokens of information with each other through queued channels. Each queue connects an output port of one process to an input port of another. The processes then function within this structure concurrently. Each individual process sequentially performs blocking reads from the queues connected to its input ports, and performs non-blocking writes to the queues connected to its output ports.

Although this system can be described in the operational manner we have given above, it is not obvious from this description that such a system will always produce the same sequence of tokens on all of its queues. In other words, remarkably, this concurrent system is deterministic in the tokens it produces. Kahn establishes this fact in [30], making use of an ingenious representation of the network. Each channel in the system is given the denotational value of a history of tokens that pass through it. Supposing these tokens are all part of a set V, these histories are either words finite words in the alphabet of V, that is members of $\mathbf{Mon}(V)$, or infinite words that complete extending sequences of words in $\mathbf{Mon}(V)$.

To be more precise about this, as we mentioned in Chapter 3, the free monoid can be given a natural prefix ordering.

$$a \leq b \overset{\text{def}}{=} \exists\, c \cdot a \bullet c = b$$

We referred to this before as a pomonoid (partially-ordered monoid), since this definition indeed satisfies the axioms for being a partial order for a free monoid (although for a general monoid it does not). Given this ordering, and therefore the existence of ordinal-indexed chains of words, the underlying set can be augmented with the limits of ω-indexed chains (longer chains could in principle be included as well, but they are often left out because they are not considered useful in most cases). The result, often notated V^{**} is an ω-complete partial order (ω-CPO), and given that the unit word $\mathbb{1}$, representing an empty history, is the bottom $\bot$ of this ordering, V^{**} is a domain (in the Scott sense).

It follows from basic results in order theory that if a finite product of V^{**}, $(V^{**})^N$ for some finite N, given the pointwise product ordering, also forms a domain. For a system with

N channels, the domain $(V^{**})^N$ can denotationally represent the states of all of the channels at an arbitrary point in the computation of the network.[1] Thus, the progress of computation in an N-channel KPN can be represented as a chain in $(V^{**})^N$. Supposing that this chain is at most of length ω (since it is conventionally assumed that all computations are), the computation of the KPN can be represented denotationally with the limits of these chains, which are already included in $(V^{**})^N$ by construction. Therefore, $(V^{**})^N$ serves as a kind of behavioral representation for these models. One might argue, of course, that there is an abstraction away from many behavioral details, and thus the use of the term *behavior* is a bit debatable since these histories are those of tokens produced and not operational events. Nonetheless, being a history, this representation is capable of expressing something about the trajectory of the process, and not just its ultimate results; that is to say, histories, at least, are accumulative.

However, what is needlessly squandered in the semantics that builds off of this representation is the notion of concatenation. We started off with a free monoid $\mathbf{Mon}(V)$, which has a clearly defined concept of concatenation, and even used this to construct the prefix ordering. This notion of concatenation can easily be extended pointwise to the product $\mathbf{Mon}(V)^N$. In taking the ω-completion of the underlying set of the monoid, the notion of concatenation is left aside as a mere interstitial mechanism used to bootstrap the ordering. But through the same transfinite methods of defining ordinal addition, as we showed in Chapter 3, the concatenation operation can be extended to ordinal length sequences, and by extension to tuples of them. There are other possibilities for dealing with the technical details of this situation that go beyond the scope of this discussion.

It suffices to say that in this behavioral representation for the collective channel history of a KPNs, there is still a well-defined notion of concatenation. Moreover, given the history α at any point in the execution of the model, any further progress can be expressed as both an extension of α to a history β such that $\alpha \leq \beta$, and as the concatenation of some history γ to form $\alpha \bullet \gamma$, an extension of α. Following from our definition of the order, we should always be able to find such a γ to represent the progress made from α to β.

The suggestion being made here is that $\mathbf{Mon}(V)^N$ (and perhaps its limit-completed counterpart), is a monoidal representation of behavior that has some intrinsic sense of concurrency, albeit a somewhat limited one. From the perspective of the channels, tokens being added constitute events, and adding tokens to separate channels are independent events. In fact, as we will discuss later in Chapter 7, these representations can be turned into the morphisms of a free symmetric monoidal category.

On the other hand, it is well-known that this elegant representation is indeed not behavioral enough to deal with non-deterministic process networks that extend those of Kahn with the possibility of non-blocking reads from input channels, and non-deterministic outputs. The Brock-Ackermann anomaly [10] demonstrated that some operational details beyond the histories of channels were necessary to reason about the semantics of non-deterministic pro-

[1]This could be generalized straightforwardly to channels with heterogeneous types of tokens, but this would not add much more than notational complications to this particular discussion.

cess networks in a fully-abstract fashion. Such an operationally augmented representation was given as a resolution to this anomaly by both Kok and Jonsson, in [31] and [26], respectively. Both of these solutions were proved to be fully abstract, and later to be equivalent. For a more complete history of this, the reader may refer to [20], which also suggests an interpretation of behavior in these systems as Profunctors.

For the reader familiar with the mathematics of KPNs, it might come as a surprise that continuous functions and fixed-points have yet to be mentioned, but it is important for the sake of what is being discussed here to give an account such as we have without needing to bring these up. In part this is to emphasize that monotonicity and fixed-points, while being important technical concepts in denotational semantics, are not central to the notion of behavior. This point was argued at the end of Chapter 3, and it is reflected in the above and following discussions. Nevertheless, as a matter of completeness or for the sake of interest, these concepts will be addressed briefly. For the reader who wishes to move on, understanding the rest of this chapter is not essential.

The key to showing determinacy for KPNs using channel histories is that each individual process can be represented as a Scott-Continuous function from the histories of its input channels to the histories of its output channels. Monotonicity, a weakend form of Scott-Continuity, arises naturally from these processes, because extending the histories of input channels can only extend those of the output channels; an output channel cannot have its history reversed or the past parts of it changed as a consequence of more input. Scott-Continuity extends monotonicity to limits over ω-indexed chains, asserting that the limits are preserved by the function representing the process. If each process can be represented in this fashion, the whole network can be represented as a product of these functions, which is also Scott-Continuous. It follows from Kleene's fixed-point theorem that this function has a least fixed-point, and that this fixed-point can be reached via the ω orbit of the function from $\perp$. That is, for such a function F, $F^{\omega}(\perp)$ is the least fixed-point of F.

Chapter 5

Key Concepts from Category Theory

In Chapter 3, we identified the concurrent generalization of free monoidal representations of sequential behavior as free monoidal categories, and gave reasons why such a representation would be preferable to a generalized sequence. Having considered in Chapter 4 the alternative of generalized sequences in detail, addressing critically many such representations that are pervasive in the literature, we will proceed instead towards the development of a representation of concurrent behavior based on free monoidal categories. Before defining exactly what a free monoidal category is, and showing how they are constructed, in this chapter, we will first review the key concepts from category theory that will be used in the construction. Amongst these concepts are the two that appear manifestly in the term itself: that of a *free* object in a category of algebras, and that of a monoidal category.

While monoidal category will be introduced without much complication, providing sufficient intuitions, interpretations, and examples to get the basic idea, the subject of free objects will be elaborated on more significantly. A free object is constructed in category theory from a universal property, witnessed by a universal morphism. It is a widely held view that category theory is, in fact, the study of universal properties. By taking time to introduce this concept, and to disambiguate any idiosyncrasies in our construction of it, we will be getting to the heart of category theory in the process. It will be important to emphasize this key concept and to show how we will be using it.

Furthermore, well-known to the cognoscenti of category theory is the connection between universal properties and adjunctions, which have two equivalent but usefully distinct formulations. We will go over this connection and explain why the defining mechanisms of adjunctions will be interesting for our purposes. We will then state and explain the significance of *Adjoint Functor Theorem*, which will later confer an important and useful property on our system of behavior representations.

It will be assumed that the basics of category theory are known to the reader. Amongst these basics, are the definitions of categories, functors, and natural transformations. For the reader interested in familiarizing themselves with the basics of this subject, one might refer to MacLane's canonical text on the subject, although this reference is aimed mostly at algebraists. A less technical introduction might be found in Awodey [5] or Pierce [53].

However, it should be mentioned that the definition we will use for a category here will the form in which the homsets themselves are part of the categorical structure rather than a single set of morphisms. This is a stylistic choice, but one that makes some of the reasoning more straightforward as the categorical composition can be defined as a dependently-typed function over pairs of homsets rather than a partial function over all morphisms.

Someone with a strong familiarity in category theory might skip this chapter entirely, but it might be valuable, on the other hand, to look at the particular manner in which these concepts are being defined in this thesis. The style preferred for the definitions and proofs in this thesis is a Skolemized one in which some of the witnesses to properties of objects are included in the definition of those objects explicitly. Moreover, the proofs will attempt to be as constructive as possible, using derivation and induction on constructed witnesses when possible. The advantage of this approach, in the opinion of the author, is that it makes it easier to realize the structures computationally, or at least see how they would function as programs.

5.1 Monoidal Categories

A monoidal category extends the notion of a category, adding to it a *monoidal* product over both the objects and the morphisms, along with identities for both. This kind of category, sometimes referred to as a *tensor category*, was initially developed as a generalization of spaces with tensor products and other kinds of products that are more general than internal Cartesian products (products that may lack projections, for instance, but still retain other multiplicative properties). Many of the basic properties of these categories are developed in [40] and are detailed in [39].

Generally, in monoidal categories the monoidal axioms of associativity and identity do not hold up to equality on the objects and morphism, but instead up to natural isomorphism. In the simple case of the Cartesian product over sets and functions, for instance, it is not the case that $((a, b), c) = (a, (b, c))$, but instead there is simply a canonical, parameterized procedure for moving between them; which, of course, is captured by a natural isomorphism. In the case where the monoidal axioms do hold up to equality, the monoidal category is regarded as a *strict* one. In this work, the focus will be on *strict* monoidal categories, so it will be assumed from here on (unless explicitly mentioned) that the monoidal categories will indeed be *strict*. This does not limit generality much because every monoidal category is indeed isomorphic to a strict one[39].

The typical definition given of a monoidal category follows the description fairly directly, adding to a category $\mathcal{C}$ a bifunctor $\otimes : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ and a monoidal unit object, along with natural associativity and identity transformations in the non-strict case. However, because of how we will be generating monoidal categories from underlying structures, it will make more sense for our purposes to define a (strict) monoidal categories with an underlying monoid. The bifunctor will therefore be broken up into its components, $\otimes$ on objects and $\otimes_{hom}$ on morphisms.

The formal definition of a monoidal category can be given as follows

Definition 7. *Monoidal Category*
A (strict) monoidal category $\mathcal{C}$ is defined

$$\mathcal{C} \overset{def}{=} (\mathcal{O},\, \mathbf{hom},\, \circ,\, \mathbf{id},\, \otimes,\, \mathbb{1},\, \otimes_{hom},\, \mathbb{1}_{hom})$$

with the typing

$$\begin{aligned}
\mathcal{O} &: \mathbf{Mon} \\
\mathbf{hom} &: \mathcal{O} \times \mathcal{O} \to \mathcal{U} \\
\circ &: \prod_{a,\, b,\, c\,:\,\mathcal{O}} \mathbf{hom}(b,\, c) \times \mathbf{hom}(a,\, b) \to \mathbf{hom}(a,\, c) \\
\mathbf{id} &: \prod_{a\,:\,\mathcal{O}} \mathbf{hom}(a,\, a) \\
\otimes &: \mathcal{O} \times \mathcal{O} \to \mathcal{O} \\
\mathbb{1} &: \mathcal{O} \\
\otimes_{hom} &: \prod_{a,\, b,\, c,\, d\,:\,\mathcal{O}} \mathbf{hom}(a,\, c) \times \mathbf{hom}(b,\, d) \to \mathbf{hom}(a \otimes b,\, c \otimes d) \\
\mathbb{1}_{hom} &: \mathbf{hom}(\mathbb{1},\, \mathbb{1})
\end{aligned}$$

where the following axioms, named **AxiomsMonCat**, *hold*

$$f \circ (g \circ h) = (f \circ g) \circ h \tag{5.1}$$
$$\mathbf{id} \circ f = f \circ \mathbf{id} = f \tag{5.2}$$
$$a \otimes (b \otimes c) = (a \otimes b) \otimes c \tag{5.3}$$
$$\mathbb{1} \otimes a = a \otimes \mathbb{1} = a \tag{5.4}$$
$$f \otimes_{hom} (g \otimes_{hom} h) = (f \otimes_{hom} g) \otimes_{hom} h \tag{5.5}$$
$$\mathbb{1}_{hom} \otimes_{hom} f = f \otimes_{hom} \mathbb{1}_{hom} = f \tag{5.6}$$
$$(f \otimes g) \circ (h \otimes k) = (f \circ h) \otimes (g \circ k) \tag{5.7}$$
$$\mathbf{id}(a \otimes b) = \mathbf{id}(a) \otimes_{hom} \mathbf{id}(b) \tag{5.8}$$
$$\mathbb{1}_{hom} = \mathbf{id}(\mathbb{1}) \tag{5.9}$$

with the free variables in the above quantified universally to everything conforming with the above typing, and with the **id** *element over the object implied in context.*

In this definition, $(\mathcal{O},\, \mathbf{hom},\, \circ,\, \mathbf{id})$ forms a category under the first two axioms, $(\mathcal{O},\, \otimes,\, \mathbb{1})$ forms a monoid under the third and forth, and $(\mathbf{hom},\, \otimes_{hom},\, \mathbb{1}_{hom})$ forms a multisorted monoid-like structure that conforms with the monoid over objects. The two products $\otimes$ and $\otimes_{hom}$ will both be refered to as $\otimes$, unless it is necessary to make the distinction explicit,

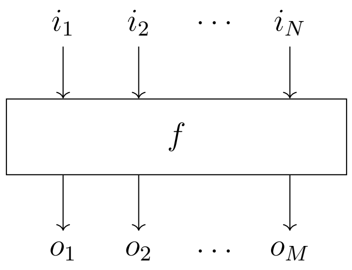


Figure 5.1: A graphical representation of a morphism in a monoidal category.

and likewise for $\mathbb{1}$ and $\mathbb{1}_{hom}$. The final three axioms connect the categorical product $\circ$ and identity **id** with the monoids $\otimes$. The axioms 5.7 and 5.8 are, in particular, the condition under which $\otimes$ and $\otimes_{hom}$ together form a bifunctor.

These categories were originally conceived as a means to represent vector spaces with linear transformations, or other algebras with tensor products, where the monoidal product is the tensor product and the composition of morphisms is the composition of linear transformations. However, these categories can also be used to interpret the structure of block diagrams such as those found in circuits, control systems, and graphical programming languages [17]. This correspondence with diagrams began with diagrammatic methods used to perform calculations in linear algebra, initiated by Penrose [ref] and others. This correspondence was then rigorously established by Joyal, Street, et al. in [27, 28, 29], connecting various variants of monoidal categories correspond to topological graphs with various features. This correspondence is summarized in an encyclopaedic form in [62], and is what has motivated the approach in this thesis to move from graph or order oriented representations of behavior to ones based in monoidal categories.

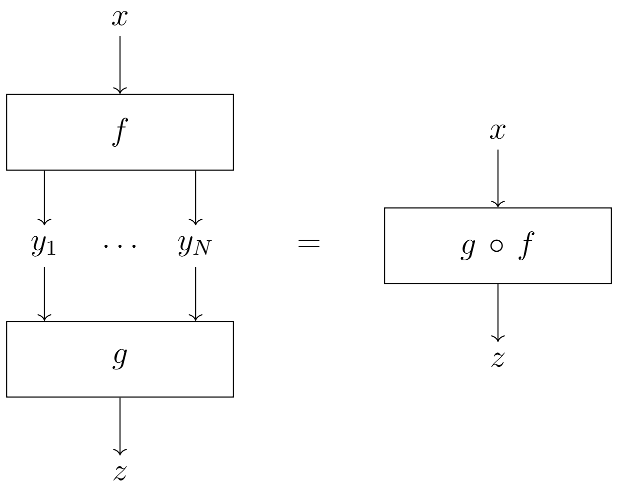
In this graphical correspondence, each morphism can be thought of as an oriented ported block, with input ports on one side and output ports on the other. This is depicted in Figure 5.1 as a morphism f. Each input port is marked with an object $i_k \in \mathcal{O}$ and each output port with an object $o_k \in \mathcal{O}$, resulting in a block

$$f \,:\, \mathbf{hom}(i_1 \otimes i_2 \otimes \ldots \otimes i_N,\, o_1 \otimes o_2 \otimes \ldots \otimes o_M)$$

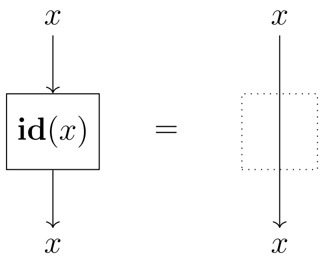
Each object in the monoidal category is then represented in this interpretation as a directed channel connecting to an output port of a block on one end and an input port on the other. Categorical composition $\circ$ in the diagram is then connecting blocks together by their channels. Specifically, for two blocks

$$f \in \mathbf{hom}(x,\, y_1 \otimes \ldots \otimes y_N),\; g \in \mathbf{hom}(y_1 \otimes \ldots \otimes y_N,\, z)$$

the composition $g \circ f \in \mathbf{hom}(x,\, z)$ is depicted in Figure 5.2a. When an identity element $\mathbf{id}(x) \in \mathbf{hom}(x, x)$ is used, which by this interpretation has the same type of input and output ports, rather than representing it as a block, it is absorbed into the channel as in Figure 5.2b. Consequently, composition with the identity is made diagrammatically transparent, simply



(a) Categorical composition of morphisms.



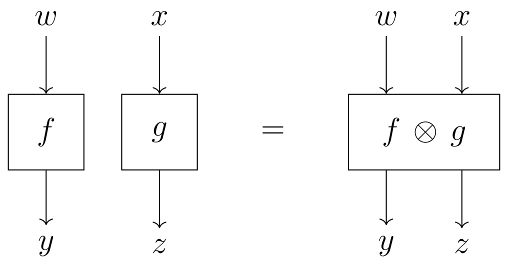
(b) Identity elements.

Figure 5.2: Graphical depiction of composition and identities.

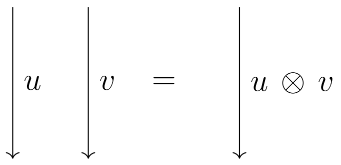
replacing the identity with the channel. For this reason, channels may appear with either or both ends left disconnected and be themselves interpreted as identity morphisms.

The monoidal product $\otimes$ in the monoidal category is depicted by the juxtaposition of either channels, as in Figure 5.3b, or of blocks, as in Figure 5.3a. The monoidal identity object $\mathbb{1}$ is, in a sense, a nonexistent channel, while the monoidal identity morphism $\mathbb{1}_{hom}$ can similarly be omitted from the diagram. From a diagrammatic perspective, the monoidal identity morphism has little significance, while the identity object can be considered the input or output type of a block with no inputs or no outputs, respectively. For instance, a morphism $h \in \mathbf{hom}(\mathbb{1}, x)$ can be represented as a "source" block.

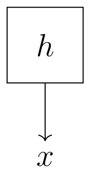
Given that these diagrams do not involve the clustering of blocks, it is clear that the associativity of composition and the two monoidal products corresponds to the diagrammatic absence of these groupings. The identity axioms correspond in the diagram to the ability to replace an identity morphism with a channel and to the ability to leave out monoidal identities altogether. The axiom 5.7 corresponds to an absence of any groupings between the two compositions in the diagrams. This is depicted in Figure 5.4 in which the dotted boxes



(a) The monoidal composition of morphisms.



(b) The monoidal product of objects.



(c) A 'source' morphism.

Figure 5.3: More graphical elements of monoidal categories.

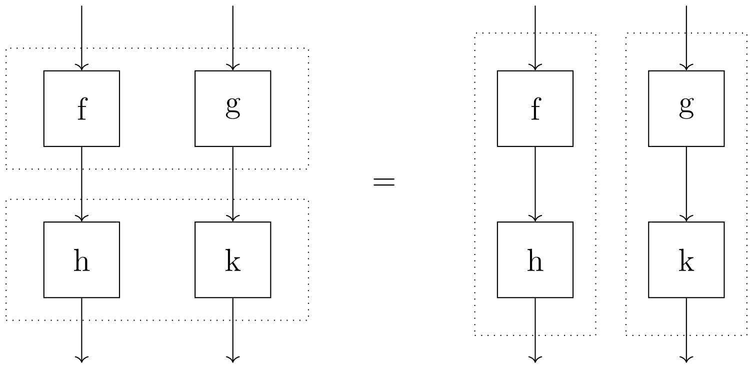


Figure 5.4: $(f \otimes g) \circ (h \otimes k) = (f \circ h) \otimes (g \circ k)$

indicate the order of construction of the two terms. On the left, $f \otimes g$ and $h \otimes k$ are taken first, then composed to form $(h \otimes k) \circ (f \otimes g)$, whereas on the right, $h \circ f$ and $k \circ g$ are formed first, then $\otimes$-composed into $(h \circ f) \otimes (k \circ g)$.

While these two basic composition operations seem superficially constrained to build diagrams in a fairly rigid fashion, since all outputs from one block must be inputs to the block composed with it, combined with identities, these compositions can be used to represent

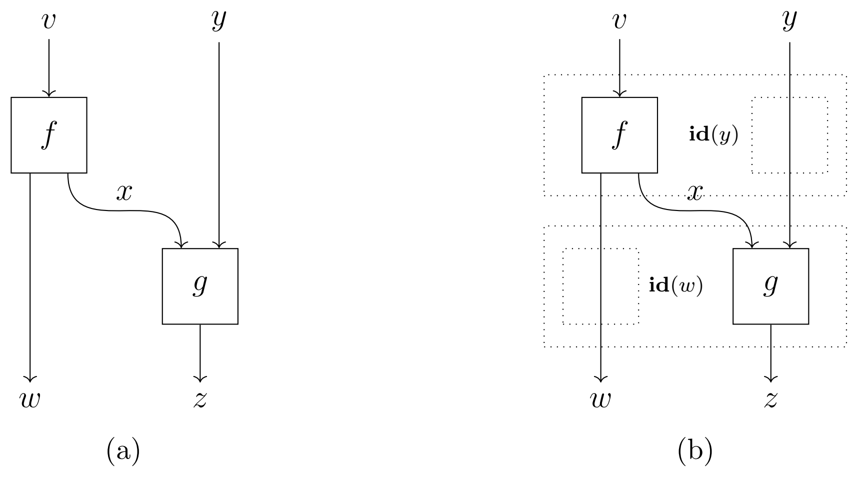


Figure 5.5: A more complex composition.

block diagrams composed in more intricate ways. Suppose one wanted to compose two blocks

$$f \in \mathbf{hom}(v,\, w \otimes x)$$
$$g \in \mathbf{hom}(x \otimes y,\, z)$$

only connecting the x output of f with the x input of g in the manner shown in Figure 5.5a, letting the other input and output bypass the two blocks. Inserting identities and grouping $\otimes$-products, as shown in Figure 5.5b, one can construct the diagram with the term

$$(\mathbf{id}(w) \otimes g) \circ (f \otimes \mathbf{id}(y)) \in \mathbf{hom}(v \otimes y,\, w \otimes z)$$

This is a common enough construction that one can define it with the *diagonal* operator $\rhd_x$, along with its mirror image $\lhd_x$, and several other operators.

Definition 8. *Let the following derived operators be defined for a monoidal category $\mathcal{M}$:*

- $$\rhd \;:\; \prod_{v,w,x,y,z\,:\,\mathcal{M}} \mathbf{hom}(v,\, w \otimes x) \times \mathbf{hom}(x \otimes y,\, z) \to \mathbf{hom}(v \otimes y,\, w \otimes z)$$
  $$f \rhd_x g \overset{def}{=} (\mathbf{id}(w) \otimes g) \circ (f \otimes \mathbf{id}(y))$$

- $$\lhd \;:\; \prod_{v,w,x,y,z\,:\,\mathcal{M}} \mathbf{hom}(v,\, x \otimes w) \times \mathbf{hom}(y \otimes x,\, z) \to \mathbf{hom}(y \otimes v,\, z \otimes w)$$
  $$f \lhd_x g \overset{def}{=} (g \otimes \mathbf{id}(w)) \circ (\mathbf{id}(y) \otimes f)$$

These operators are, of course, like categorical composition, not total over all homsets. Note also that only the x must be explicitly noted in the operator since the rest of its parameters can be inferred.

To summarize the above features of the graphical correspondence between monoidal categories and ported block diagrams, the corresponding diagrammatic language established in [27] is specifically that of acyclic block diagrams confined to a two dimensional surface. The axioms of monoidal categories are the isometries of these diagrams. Blocks can therefore be moved around, so long as they do not cross over channels, and the diagram will remain the same morphism in the category. Crossing will be incorporated into the diagrammatic language through the definition of *symmetric monoidal category*, which add additional algebraic structure to the standard monoidal category. Moving around blocks to syntactically group elements into algebraic terms can be utilized as a calculation technique.

In order to demonstrate this kind of diagrammatic reasoning, and to establish some additional algebraic structure, some properties of the diagonal operators can be asserted and proved using the above graphical reasoning, justified in [27].

Proposition 3. *The following identities hold for all compatible morphisms:*

$$f \lhd_x (g \lhd_y h) = (f \lhd_x g) \lhd_y h \tag{5.10}$$
$$f \rhd_x (g \rhd_y h) = (f \rhd_x g) \rhd_y h \tag{5.11}$$
$$f \rhd_x (g \lhd_y h) = (f \rhd_x g) \lhd_y h \tag{5.12}$$
$$f \lhd_x (g \rhd_y h) = (f \lhd_x g) \rhd_y h \tag{5.13}$$
$$f \lhd_{\mathbb{1}} g = f \rhd_{\mathbb{1}} g = f \otimes g \tag{5.14}$$

For any morphisms,

$$f : \mathbf{hom}(y, x),\ g : \mathbf{hom}(x, z)$$

the following identity holds:

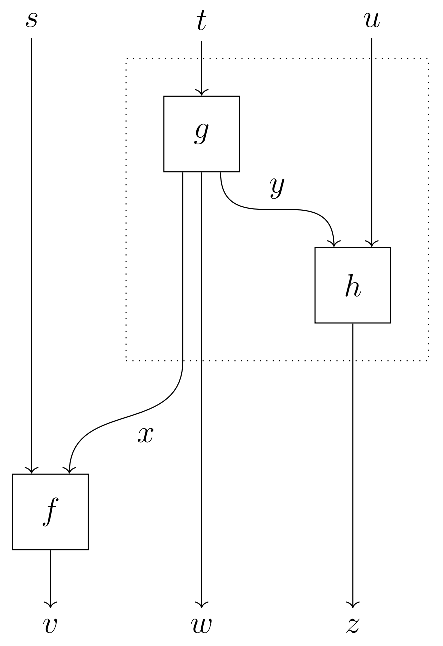
$$f \rhd_x g = g \lhd_x f = g \circ f$$

Proof. All of the identities can be verified diagrammatically. We will show one of the less obvious of these in Figure 5.6 corresponding to identity 5.13. In the last identity, involving the monoidal identity in the operator, i.e. $\rhd_{\mathbb{1}}$, there connection between the two operands is $\mathbb{1}$ and can thus be removed leaving $f \otimes g$, as shown in Figure 5.7a. Likewise, in the final identity, the connection x is the entire domain of g and codomain of f, consituting a composition $g \circ f$, as shown in Figure 5.7b. □

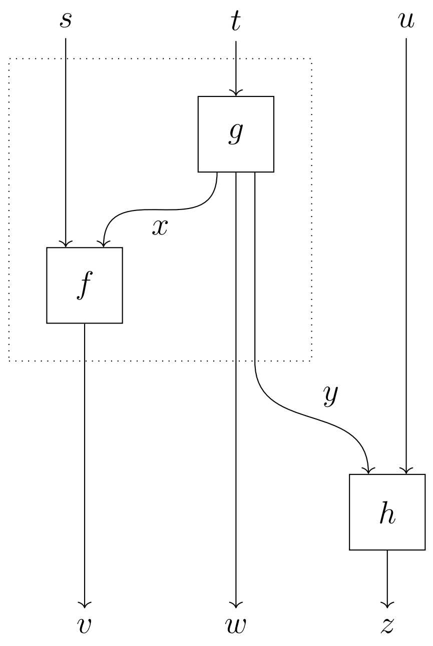
By compatible here, it is meant that both sides of the equation exist. This may not always be the case.

Given these identities, we are justified in removing parenthesis from expressions involving $\otimes$, $\rhd$, and $\lhd$ operators (with their parameters named). Hence, expressions like

$$f \rhd_x g \lhd_y h \otimes k \lhd_z m \lhd_w n$$

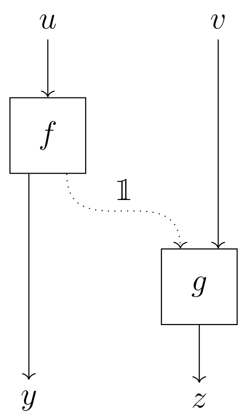


(a) $f \lhd_x (g \rhd_y h)$

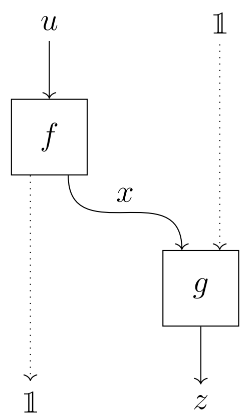


(b) $(f \lhd_x g) \rhd_y h$

Figure 5.6: Graphical proof of associativity.

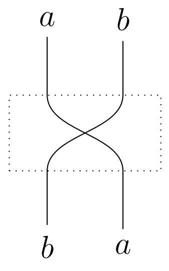


(a) $f \rhd_{\mathbb{1}} g = f \otimes g$

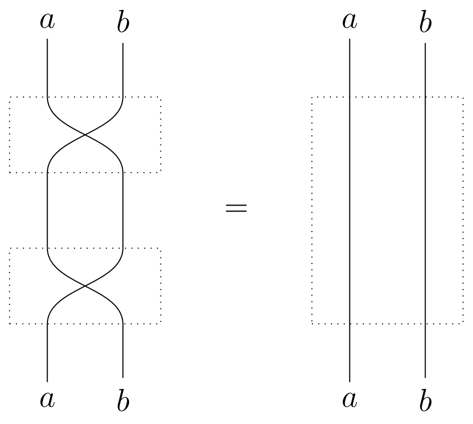


(b) $f \rhd_x g = g \circ f$

Figure 5.7: More graphical properties.



(a) $\beta(a, b)$



(b) $\beta(b, a) \circ \beta(a, b) = \mathbf{id}(a \otimes b)$

Figure 5.8: Braidings in symmetric monoidal categories.

are unambiguous. Though, one cannot arbitrarily insert parenthesis, since the composition formed might not involve compatible operands. It should be noted that the final identity does not mean that categorical composition $\circ$ will, by itself, associate with any of these operators. This is because categorical composition does not name its parameter, whereas the diagonal operators do. Therefore, one can readily translate any composition into a diagonal operator, but only special cases of diagonals to compositions.

Symmetric Monoidal Categories

In [27] the monoidal category variant that corresponds to acyclic block diagrams, free from the constraint of a two dimensional surface, is a symmetric monoidal category. This is the familiar language of abstract acyclic block diagrams we are looking for in a behavioral representation such as OEGs. This variant adds to monoidal categories a collection of constant morphisms called *braidings* for each pair of objects, which commutate the monoidal product over this pair via composition. That is, for any two objects $a, b \in \mathcal{O}$, there is a corresponding braiding

$$\beta(a, b) \in \mathbf{hom}(a \otimes b, b \otimes a)$$

that represents a crossing of channels in the diagrammatic language. This is depicted in Figure 5.8a.

In the more general structure of *braided* monoidal category, this crossing is oriented and can lead to channels non-trivially twisting around each other. However, in the case of symmetric monoidal categories, two consecutive crossings of channels by two braidings reduces to the identity, as shown in Figure 5.8b. In formal terms,

$$\beta(b, a) \circ \beta(a, b) = \mathbf{id}(a \otimes b)$$

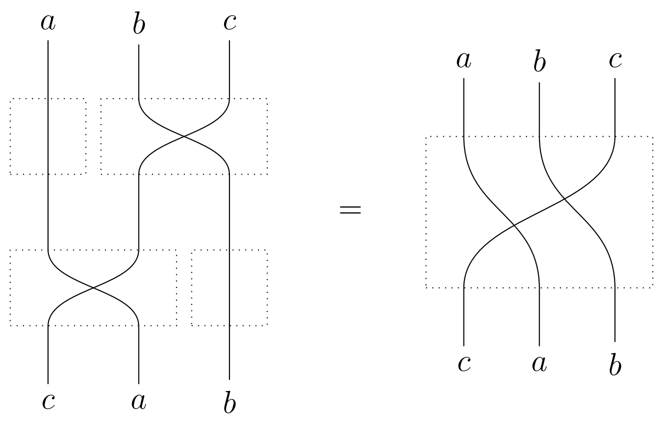


Figure 5.9: $\beta(a \otimes b, c) = (\beta(a, c) \otimes \mathbf{id}(b)) \circ (\mathbf{id}(a) \otimes \beta(b, c))$

hence the qualifier *symmetric*.

This kind of monoidal category can be defined formally as follows:

Definition 9. *Symmetric Monoidal Category*
A (strict) symmetric monoidal category $\mathcal{C}$ is defined

$$\mathcal{C} \overset{def}{=} (\mathcal{O}, \mathbf{hom}, \circ, \mathbf{id}, \otimes, \mathbb{1}, \otimes_{hom}, \mathbb{1}_{hom}, \beta)$$

where the first 8 components form a monoidal category, the last component has the type

$$\beta : \prod_{a,\, b \,:\, \mathcal{O}} \mathbf{hom}(a \otimes b, b \otimes a)$$

and the following additional axioms hold

$$\beta(b, a) \circ \beta(a, b) = \mathbf{id}(a \otimes b) \tag{5.15}$$
$$\beta(a \otimes b, c) = (\beta(a, c) \otimes \mathbf{id}(b)) \circ (\mathbf{id}(a) \otimes \beta(b, c)) \tag{5.16}$$
$$\beta(a, b \otimes c) = (\mathbf{id}(b) \otimes \beta(a, c)) \circ (\beta(a, b) \otimes \mathbf{id}(c)) \tag{5.17}$$
$$\forall f : \mathbf{hom}(a, b);\, g : \mathbf{hom}(c, d) \cdot \beta(b, d) \circ (f \otimes g) = (g \otimes f) \circ \beta(a, b) \tag{5.18}$$

Collectively, the axioms are named **AxiomsSymMonCat**.

We have already addressed the first axiom, which was depicted in Figure 5.8a. The second and third axioms allow braidings to be composed with each other and are usually consolidated under the name of the *hexagon* axioms (a name that reflects the commutativity diagrams they form with the associator in the non-strict case). These are depicted in Figures 5.9 and 5.10 The fourth axiom 5.18, depicted in Figure 5.11, allows blocks to slide through a braiding, commuting them.

Using the braiding operators, the order of block inputs and outputs can be permuted. This is shown in Figure 5.12. This is the last syntactic mechanisms necessary to facilitate the

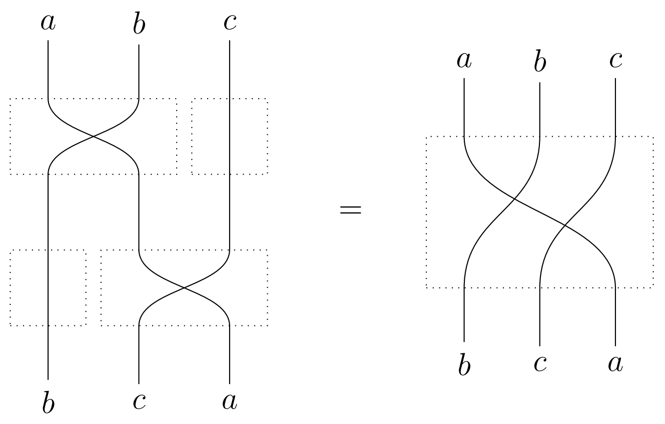


Figure 5.10: $\beta(a,\, b \otimes c) = (\mathbf{id}(b) \otimes \beta(a,\, c)) \circ (\beta(a,\, b) \otimes \mathbf{id}(c))$

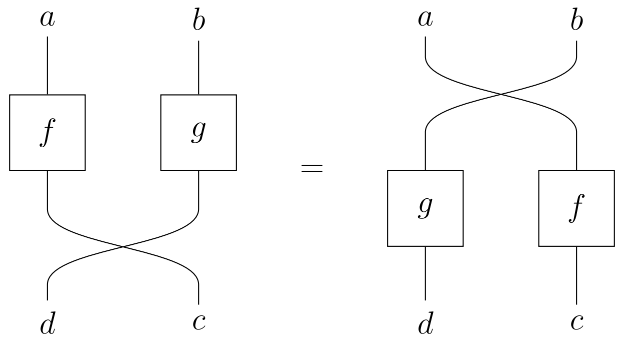


Figure 5.11: $\beta(b,\, d) \circ (f \otimes g) = (g \otimes f) \circ \beta(a,\, b)$

construction of the complete language of acyclic block diagrams. While it is the case that the completeness of this language for these diagrams is shown in [27], where the diagrammatic space itself is defined in topological terms, in Chapter 7, this completeness will also be shown for OEGs specifically.

Armed with only braidings and identity elements, any permutation of channels (objects) may be constructed. That is to say, more precisely, given any product of objects $x_1 \otimes \ldots \otimes x_n$ and any permutation σ of n elements, one can use only braidings and identities to construct a morphism

$$Perm_\sigma\, :\, \mathbf{hom}(x_1 \otimes \ldots \otimes x_n,\, x_{\sigma(1)} \otimes \ldots \otimes x_{\sigma(n)})$$

If we think of braidings as simply crossing pairs of channels in our visual representation, these permutation morphisms cross them iteratively to the effect of achieving their permutation.

This can be shown inductively through the use of braidings to exchange individual for an arbitrary product of objects, which are known to generate all permutations. This is depicted in Figure 5.12, which shows the simple, but nontrivial, permutation $\sigma = (3\,2\,1)$ of channels between two blocks. Being able to construct these permutations allows blocks to

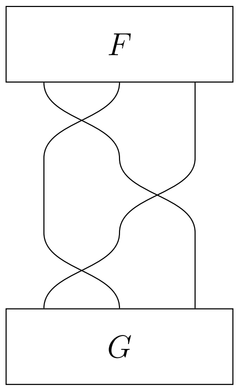


Figure 5.12: $G \circ (\mathbf{id} \otimes \beta) \circ (\beta \otimes \mathbf{id}) \circ (\mathbf{id} \otimes \beta) \circ F$

be connected sequentially in arbitrary order. Recalling our earlier discussion in Chapter 4 on the inability to define general algebraic sequential compositions on generalized sequences, this is how this aim is accomplished in the context of symmetric monoidal categories. It should be intuitively convincing that with identity elements and permutations, arbitrary acyclic block diagrams can be constructed. Rather than parametrizing compositions, we instead have in the braidings and identities a sufficient sublanguage to factor the mapping between outgoing edges of one graph and incoming edges of another out of the operators. This consequently makes the algebraic laws fairly simple and succinct.

Nevertheless, an important fact about permutations will be needed to complete this picture, which was proven by MacLane in [40]. Although it is simple to see that any permutation can be built from only braidings and identity elements, and that there are generally many ways to do this that are syntactically distinct, what is far more complicated is to show that all of these different ways of building the same permutation are equivalent in the category. That this is indeed the case was proven by MacLane, and is called *coherence*. In other words, for any two terms expressing the same permutation, there exist a proof, using the above axioms, that can show that they are equivalent. This will be of use later in reasoning about OEGs.

In general, that several terms constructed from algebra of the category can be proven equivalent, if constructed from a collection of constant morphisms (or morphism variables), implies that one can consider the corresponding diagram, in a sense, to be the geometric ground truth of behind all of the different constructions. In a free symmetric monoidal category, as we shall see later, more specifically, each morphism can be identified with a particular graph combining the primitive morphisms that are used to generate the category modulo permutations to the incoming and outgoing channels of the whole graph. If we pin down the ordering of these incoming and outgoing channels, however, we get a strict correspondence between the diagrams and the collection of all provably equivalent constructions. This amounts to what was proven by Joyal and Street in [27]. This property will be precisely what we need for a representation of behavior that can be called *ontological*.

5.2 Free Objects and Derived Structures

We will first review the concept of a *free* object in a category, defining all the constituting mechanisms necessary to construct them, then showing how the structures of adjunctions and monads be derived from them. As we have mentioned, rather than resting on definitions that are formulated in terms of existence, we will Skolemize the definitions, giving constructions whenever possible. At the least, these constructions will give a clear sense of which formal data are necessary to substantiate some of the important properties we will discuss, but in many cases, the constructions will provide the basis for effective procedures that may be used in computations involving these constructs.

In many categories where the objects are algebraic structures, and the morphisms are the corresponding homomorphisms between these structures, there is a notion of a *free* object in this category generated by some particular collection of basis elements. This element is a categorical generalization of algebraic structures such as free monoids, free groups, free vector spaces, and the like. This object is called *free* because it is, in a sense, the least constrained instance of the algebra within the category that include the generating elements. More specifically, it is constrained only by the axioms all members of the algebra fulfill, and consisting of only elements algebraically expressible in terms of the generators.

One gets the underlying elements of free structure, typically, by forming every term possible from the operations and constants of the algebra using the generators, then taking to be equivalent only the terms that can be proven equivalent under the axioms of the algebra (commutativity, associativity, etc...). The underlying set of elements is then the equivalence class over the set of generated terms. Every other algebra (not isomorphic to the free one) either models additional formulae, or includes elements not equivalent to a generated term. The free monoid $\mathbf{Mon}(\mathcal{A})$ of some set $\mathcal{A}$, for instance, consists of equivalence classes of *monoidal terms*. These monoidal terms are built inductively out of elements of $\mathcal{A}$ along with a purely syntactic binary product and a syntactic identity element. An example of this was carried out in Chapter 3, but here we will show the construction in categorical terms so as to exemplify our formal mechanisms.

Constructing Free Elements

The general categorical formulation of a *free* element of a category of algebraic structures $\mathcal{S}$ is expressed as a *universal property* of the category $\mathcal{S}$. This formulation involves both the category of algebraic structures $\mathcal{S}$ along with a category of the underlying structures $\mathcal{G}$, consisting of the objects of which are the underlying collections of elements for an algebra in $\mathcal{S}$. A functor between these categories $\mathcal{U} : \mathcal{S} \to \mathcal{G}$, called a *forgetful functor*, maps each algebraic structure in $\mathcal{S}$ to its underlying structure in $\mathcal{G}$, forgetting the algebraic details such as operators and constants. For instance, the forgetful functor on monoids, groups, rings, and the like maps each of these single-sorted structures into its underlying set of elements. A multisorted structure, in contrast, might have a tuple of sets as its underlying structure. In our case, that of free monoidal category, the underlying structure will be more complicated than sets or tuples of sets.

Addressing monoids specifically as an example, we take a monoid, as usual, to be a triple $M = (\|M\|,\ \otimes,\ \mathbb{1})$, where the underlying structure is the set $\|M\|$. These monoids form the category **Mon** with monoid homomorphism as its morphisms. The forgetful functor from **Mon** into the category **Set** containing its underlying sets is defined as follows.

Definition 10. *Monoid forgetful functor*
Given the category of monoids **Mon** *and the category of sets* **Set***, the forgetful functor for* **Mon** *is defined*

$$\begin{aligned}
&\mathcal{U}_{\mathbf{Mon}} : \mathbf{Mon} \to \mathbf{Set}\\
&\mathcal{U}_{\mathbf{Mon}}(\|M\|,\ \otimes,\ \mathbb{1}) \overset{def}{=} \|M\|\\
&\mathcal{U}_{\mathbf{Mon}}(F) \overset{def}{=} \|F\|
\end{aligned}$$

where F is a monoid homomorphism and $\|F\|$ is its underlying function.

Over objects, the functor simply maps the monoid to its underlying set, while over morphisms the functor maps the homomorphism between two monoids into the set-function between the two underlying sets.

Using this forgetful functor, a universal morphism from an object $\mathcal{G}$ to the forgetful functor $\mathcal{U}$ can be defined, characterizing a *universal property* in $\mathcal{S}$.

Definition 11. *Universal Morphism*
Given categories $\mathcal{S}$ and $\mathcal{G}$, and a functor

$$\mathcal{U} : \mathcal{S} \to \mathcal{G}$$

a universal morphism **Univ** *of object $g \in \mathcal{G}$ is defined as a tuple*

$$\mathbf{Univ}_g \overset{def}{=} (f^g,\ \eta^g,\ \zeta^g)$$

with the typing

$$\begin{aligned}
f^g &: \mathcal{S}\\
\eta^g &: \mathbf{hom}_{\mathcal{G}}(g, \mathcal{U}(f))\\
\zeta^g &: \prod_{s \in \mathcal{S}} \mathbf{hom}_{\mathcal{G}}(g, \mathcal{U}(s)) \to \mathbf{hom}_{\mathcal{S}}(f^g,\ s)
\end{aligned}$$

and the derived function

$$\begin{aligned}
&\theta^g : \prod_{s \in \mathcal{S}} \mathbf{hom}_{\mathcal{S}}(f^g,\ s) \to \mathbf{hom}_{\mathcal{G}}(g, \mathcal{U}(s))\\
&\theta^g(k) \overset{def}{=} \mathcal{U}(k) \circ \eta^g
\end{aligned}$$

defined such that ζ^g is the inverse of θ^g.

This definition can be summarized in the commutativity diagram

$$\begin{tikzcd}
g \arrow[r, "\eta^g"] \arrow[dr, "h = \theta^g(k)"'] & \mathcal{U}(f^g) \arrow[d, "\mathcal{U}(k)"] & f^g \arrow[d, "k = \zeta^g(h)"] \\
& \mathcal{U}(d) & d
\end{tikzcd}$$

in which for any object $d \in \mathcal{S}$ and morphism h in $\mathcal{G}$, there is a unique morphism $\zeta(h)$ in $\mathcal{S}$ such that the diagram on the right commutes. The more typical formulation of a universal morphism, such as in [39], uses the above uniqueness property for the definition. It can be seen, however, that this is equivalent to the requirement that ζ be the inverse of θ.

Proposition 4. *Given a tuple (f^g, η^g, ζ^g) with the typing of a universal morphism, the two properties are equivalent:*

1. *ζ^g is the inverse of θ^g.*
2. *For all $d \in \mathcal{S}$ and h : $\mathbf{hom}(g, \mathcal{U}(d))$ there exist a unique morphism $h^\sharp$: $\mathbf{hom}(f, d)$ such that $h = \theta(h^\sharp)$.*

Proof.

(1) $\Rightarrow$ (2)
Inverses are unique, therefore, if ζ^g exists and is the inverse of θ^g, there is a unique $h^\sharp = \zeta(h)$ corresponding to h, and it fulfills $h = \theta^g(h^\sharp)$.

(2) $\Rightarrow$ (1)
Skolemizing the statement that there exists an $h^\sharp$ for all h, one constructs a function $h^\sharp = \zeta(h)$. One side of the inverse is thereby already established. The remaining side $k = \zeta^g(\theta^g(k))$, is established by considering that for $h = \theta(k)$ there must be a unique $h^\sharp = \zeta^g(\theta^g(k))$ satisfying $h = \theta^g(h^\sharp)$, but k itself statisfies this, hence $\zeta^g(\theta^g(k)) = k$.

□

The constructive definition emphasizes the important fact that ζ^g and θ^g witness a (natural) set isomorphism

$$\begin{aligned}
\mathbf{hom}_{\mathcal{S}}(f^g, d) &\cong \mathbf{hom}_{\mathcal{G}}(g, \mathcal{U}(d)) \text{ witnessed as} \\
\zeta^g &: \mathbf{hom}_{\mathcal{G}}(g, \mathcal{U}(d)) \to \mathbf{hom}_{\mathcal{S}}(f^g, d) \\
\theta^g &: \mathbf{hom}_{\mathcal{S}}(f^g, d) \to \mathbf{hom}_{\mathcal{G}}(g, \mathcal{U}(d))
\end{aligned}$$

for all $d \in \mathcal{S}$.

Although the concept of a universal property in $\mathcal{S}$ extends beyond what are considered to be free structures, in the case where $\mathcal{S}$ is a category of algebraic structures and $\mathcal{G}$ is a category of underlying components, well call the first component f^g of a universal morphism from g into $\mathcal{U}$ the *free* $\mathcal{S}$-structure *generated by* g. The approach we will take to constructing these free structures will be to define not only the structure f^g itself, but also the components η^g and ζ^g that witness the free structure as a universal property; specifically by showing that our definition for ζ^g is the inverse of the canonical θ^g.

The Construction of Free Monoids

We will continue the example of a free monoid, and construct its corresponding universal morphism. Later the construction of free monoidal categories will follow along the same lines as this construction, involving only a more intricate instance of the steps involved. Moreover, the later definition of free monoidal categories will depend on that of a free monoid, hence the details of this construction – the particular way we do it in this thesis – need to be made explicit.

As we have established, witnessing a universal morphism involves defining the three elements: f^g, η^g, and ζ^g. The first of these, the free object itself, will be constructed from the term language of monoids. For any given set S of generating elements, this language can be constructed inductively to form a set of terms $\mathbf{MonTerms}(S)$. The language construction is a follows.

Definition 12. *Monoidal Terms*
Given a set S, the set of monoidal terms $\mathbf{MonTerms}(S)$ is constructed inductively by the following inference rules

$$\frac{x \in S}{x \in \mathbf{MonTerms}(S)} \qquad \frac{}{\underline{\mathbb{1}} \in \mathbf{MonTerms}(S)} \qquad \frac{a,\, b \in \mathbf{MonTerms}(S)}{a \mathbin{\underline{\otimes}} b \in \mathbf{MonTerms}(S)}$$

We must also define the *linear contexts* (one-hole contexts) for terms.

Definition 13. *Linear Monoidal Contexts*
Given a set S, the set of linear monoidal contexts $\mathbf{MonContexts}(S)$ is constructed inductively by the following inference rules

$$\frac{}{\bigcirc \in \mathbf{MonContexts}(S)}$$

$$\frac{\gamma \in \mathbf{MonContexts}(S),\ a \in \mathbf{MonTerms}(S)}{\gamma \mathbin{\overline{\otimes}} a \in \mathbf{MonContexts}(S)}$$

$$\frac{\gamma \in \mathbf{MonContexts}(S),\ a \in \mathbf{MonTerms}(S)}{a \mathbin{\overline{\otimes}} \gamma \in \mathbf{MonContexts}(S)}$$

*Each element $\gamma \in$ **MonContexts**(S) is a function **MonTerms**$(S) \to$ **MonTerms**(S). These are defined, recursively, for any a, $b \in$ **MonTerms**(S)*

$$\begin{aligned} \bigcirc\,(a) &\overset{def}{=} a \\ (\gamma \,\overline{\otimes}\, a)(b) &\overset{def}{=} \gamma(b) \,\underline{\otimes}\, a \\ (a \,\overline{\otimes}\, \gamma)(b) &\overset{def}{=} a \,\underline{\otimes}\, \gamma(b) \end{aligned}$$

These linear contexts, in essence, substitute a term into a single position within another term.

Using these term languages an equivalence relation can be defined over the monoidal terms using the axioms of monoids.

Definition 14. *Monoidal Equivalence*
The equivalence relation

$$\overset{\mathbf{Mon}}{\Longleftrightarrow} \subseteq \mathbf{MonTerms}(S) \times \mathbf{MonTerms}(S)$$

is defined inductively by the following inference rules

$$\frac{}{a \otimes (b \otimes c) \overset{\mathbf{Mon}}{\Longleftrightarrow} (a \otimes b) \otimes c}$$

$$\frac{}{a \otimes \mathbb{1} \overset{\mathbf{Mon}}{\Longleftrightarrow} a} \qquad \frac{}{\mathbb{1} \otimes a \overset{\mathbf{Mon}}{\Longleftrightarrow} a}$$

$$\frac{a \overset{\mathbf{Mon}}{\Longleftrightarrow} a'}{\gamma(a) \overset{\mathbf{Mon}}{\Longleftrightarrow} \gamma(a')}$$

Let the function

$$\langle - \rangle_{\mathbf{Mon}} : \mathbf{MonTerms}(S) \to \mathbf{MonTerms}(S)/ \overset{\mathbf{Mon}}{\Longleftrightarrow}$$

*be the canonical projection induced by $\overset{\mathbf{Mon}}{\Longleftrightarrow}$, where **MonTerms**$(S)/ \overset{\mathbf{Mon}}{\Longleftrightarrow}$ is the canonical quotient.*

Putting these definitions together, if we suppose a set $S = \{u, w\}$, then the following would be examples of terms in **MonTerms**(S):

$$\begin{aligned} &[\![u \,\underline{\otimes}\, ((w \,\underline{\otimes}\, \underline{\mathbb{1}}) \,\underline{\otimes}\, w)]\!] \\ &[\![(u \,\underline{\otimes}\, w) \,\underline{\otimes}\, w]\!] \\ &[\![u]\!] \\ &[\![\underline{\mathbb{1}}]\!] \end{aligned}$$

We will enclose the formal terms in double brackets $\llbracket - \rrbracket$ to separate them from their surrounding context. The parenthesis in the concrete representation of the terms (the one written out) are not part of the abstract syntax, but rather markers of precedence (as is typical in programming language texts).

Examples of monoidal contexts and their application to terms would be:

$$\llbracket u \,\overline{\otimes}\, (\bigcirc \,\overline{\otimes}\, w)\rrbracket(\llbracket w \,\underline{\otimes}\, u\rrbracket) = \llbracket u \,\underline{\otimes}\, ((w \,\underline{\otimes}\, u) \,\underline{\otimes}\, w)\rrbracket$$
$$\llbracket \bigcirc \,\overline{\otimes}\, \underline{\mathbb{1}}\rrbracket(\llbracket(\underline{\mathbb{1}} \,\underline{\otimes}\, u) \,\underline{\otimes}\, u\rrbracket) = \llbracket((\underline{\mathbb{1}} \,\underline{\otimes}\, u) \,\underline{\otimes}\, u) \,\underline{\otimes}\, \underline{\mathbb{1}}\rrbracket$$

Finally, some examples of term equivalences are

$$\llbracket u \,\underline{\otimes}\, ((w \,\underline{\otimes}\, \underline{\mathbb{1}}) \,\underline{\otimes}\, w)\rrbracket \overset{\mathbf{Mon}}{\Longleftrightarrow} \llbracket(u \,\underline{\otimes}\, w) \,\underline{\otimes}\, w\rrbracket$$
$$\llbracket u \,\underline{\otimes}\, \underline{\mathbb{1}}\rrbracket \overset{\mathbf{Mon}}{\Longleftrightarrow} \llbracket u\rrbracket$$

which one could write equivalently

$$\langle u \,\underline{\otimes}\, ((w \,\underline{\otimes}\, \underline{\mathbb{1}}) \,\underline{\otimes}\, w)\rangle_{\mathbf{Mon}} = \langle (u \,\underline{\otimes}\, w) \,\underline{\otimes}\, w\rangle_{\mathbf{Mon}}$$
$$\langle u \,\underline{\otimes}\, \underline{\mathbb{1}}\rangle_{\mathbf{Mon}} = \langle u\rangle_{\mathbf{Mon}}$$

in terms of the canonical projection.

The free monoid for a set S can be defined in terms of these above elements.

Definition 15. *Free Monoid*
Given a set S of elements, the free monoid $\mathbf{Mon}(S)$ is defined

$$\mathbf{Mon}(S) \overset{def}{=} (\mathbf{MonTerms}(S)/ \overset{\mathbf{Mon}}{\Longleftrightarrow}, \langle \underline{\otimes} \rangle, \langle \underline{\mathbb{1}} \rangle)$$

In this monoid, the product of equivalence classes of formal terms is simply the equivalence class of the formal product. That is

$$\langle a \rangle \otimes \langle b \rangle = \langle a \,\underline{\otimes}\, b \rangle$$

In computing such a product, we must choose a representative of each of the equivalence classes.

And with this, we can define the universal morphism corresponding to our construction and show that it is indeed a free element.

Theorem 4. *Given a set S, the triplet $(\mathbf{Mon}(S), \eta^S, \zeta^S)$ defined*

$$\begin{aligned}
&\eta^S : S \to \mathcal{U}_{\mathbf{Mon}}(\mathbf{Mon}(S)) \\
&\eta^S(x) \overset{def}{=} \langle x \rangle \\
&\zeta^S : (S \to \mathcal{U}_{\mathbf{Mon}}(M)) \to (\mathbf{Mon}(S) \to M) \\
&\zeta^S(h)(\langle a \rangle) \overset{def}{=} W_h(a), \textit{ where} \\
&\quad W_h(a \,\underline{\otimes}\, b) = W_h(a) \otimes W_h(b) \\
&\quad W_h(\underline{\mathbb{1}}) = \mathbb{1} \\
&\quad W_h(x \in S) = h(x)
\end{aligned}$$

witnesses a universal morphism.

Proof. First, in the definition for $\zeta^S(h)$, a recursive definition was used for W_h. This produces a well-defined function via structural induction over the syntactic terms it operates on.

We must then show that $\zeta^S(h)$ defines a monoid homomorphism $\mathbf{Mon}(S) \to M$. From the definition, the two criteria of this can be verified.

$$\begin{aligned}&\zeta^S(h)(\langle a\rangle \otimes \langle b\rangle) = \zeta^S(h)(\langle a \underline{\otimes} b\rangle)\\ &= W_h(a \underline{\otimes} b) = W_h(a) \otimes W_h(b) = \zeta^S(h)(\langle a\rangle) \otimes \zeta^S(h)(\langle b\rangle)\\ &\zeta^S(h)(\mathbb{1}) = \zeta^S(h)(\langle \underline{\mathbb{1}}\rangle) = W_h(\underline{\mathbb{1}}) = \mathbb{1}\end{aligned}$$

It must then be shown that ζ^S and θ^S are inverses. Starting with $\theta^S \circ \zeta^S$ and letting $h : S \to \mathcal{U}_{\mathbf{Mon}}(M)$, we expanding the definition

$$\theta^S \circ \zeta^S(h) = \mathcal{U}_{\mathbf{Mon}}(\zeta^S(h)) \circ \eta^S$$

Then operating on an arbitrary $x \in S$

$$\begin{aligned}&\theta^S \circ \zeta^S(h)(x) = \mathcal{U}_{\mathbf{Mon}}(\zeta^S(h)) \circ \eta^S(x) = \mathcal{U}_{\mathbf{Mon}}(\zeta^S(h))(\langle x\rangle)\\ &= \zeta^S(h)(\langle x\rangle) = W_h(x) = h(x)\end{aligned}$$

thus $\theta^S \circ \zeta^S = \mathbf{id}_{\mathbf{Fun}}$.

For $\zeta^S \circ \theta^S$, operating on an element $k : \mathbf{Mon}(S) \to M$, and this itself operating on an arbitrary element $\langle a\rangle \in M$, the definition is expanded

$$(\zeta^S \circ \theta^S(k))(\langle a\rangle) = W_{\theta^S(k)}(a)$$

What needs to be shown here is that

$$W_{\theta^S(k)}(a) = k(\langle a\rangle)$$

This can be proven by structural induction over the syntax of a. The base cases are

$$\begin{aligned}&W_{\theta^S(k)}(\underline{\mathbb{1}}) = \mathbb{1} = k(\mathbb{1}) = k(\langle \underline{\mathbb{1}}\rangle)\\ &W_{\theta^S(k)}(x) = (\theta^S(k))(x) = (\mathcal{U}_{\mathbf{Mon}S}(k) \circ \eta^S)(x) = (\mathcal{U}_{\mathbf{Mon}S}(k))(\langle x\rangle) = k(\langle x\rangle)\end{aligned}$$

where $x \in S$. The inductive case is

$$\begin{aligned}&W_{\theta^S(k)}(a \underline{\otimes} b) = W_{\theta^S(k)}(a) \otimes W_{\theta^S(k)}(b) = k(\langle a\rangle) \otimes k(\langle b\rangle)\\ &= k(\langle a\rangle \otimes \langle b\rangle) = k(\langle a \underline{\otimes} b\rangle)\end{aligned}$$

which uses in the second step the inductive assumption. This establishes the $\zeta^S \circ \theta^S = \mathbf{id}_{\mathbf{Fun}}$ side of the inverse, completing its verification. $\square$

It is therefore the case that $\mathbf{Mon}(S)$ is the free monoid generated by S, witnessed by this definition of a universal morphism. Furthermore, the two witnessing functions that come along with this free element are more than of just technical significance in establishing that $\mathbf{Mon}(S)$ is a free monoid. These both have a practical and intuitive purpose.

The function η^S embeds elements of S into the underlying set of elements of $\mathbf{Mon}(S)$. In some definitions, this embedding is taken to be an outright injection. Since there is a clear bijection between $x \in S$ and $\langle x \rangle \in \mathcal{U}_{\mathbf{Mon}}(\mathbf{Mon}(S))$, one can clearly replace each of the latter with the former creating a set that is isomorphic to $\mathcal{U}_{\mathbf{Mon}}(\mathbf{Mon}(S))$. Indeed, some treatments do just this. However, here we have an important reason to not to conflate the explicit element x with those of $\langle x \rangle$. Distinguishing these two allows an explict notion of hierarchy to be added to the construction. Consider a two-level free monoid $\mathbf{Mon}(\mathcal{U}_{\mathbf{Mon}}(\mathbf{Mon}(S)))$. This builds terms out of equivalence class of terms from the one-level $\mathbf{Mon}(S)$. For example,

$$\begin{array}{l} \langle \langle u \underline{\otimes} w \rangle \underline{\otimes} \langle u \rangle \rangle \\ \langle \underline{\mathbb{1}} \underline{\otimes} \langle \underline{\mathbb{1}} \underline{\otimes} u \rangle \rangle \\ \langle \langle u \rangle \rangle \end{array}$$

The key is reflected in the third of these, that $\langle \langle u \rangle \rangle \neq \langle u \rangle$, and thus we can distinguish clearly between both of these levels (and thus any number of them).

Nevertheless, there is a similarly explicit mechanism available to flatten this hierarchy, following along the lines of the aforementioned bijection by which

$$\ldots \longrightarrow \langle \langle \langle a \rangle \rangle \rangle \longrightarrow \langle \langle a \rangle \rangle \longrightarrow \langle a \rangle \longrightarrow a$$

We will encounter this mechanism in the next section. Later we will also see this hierarchy put to use explicitly in the context of OEGs, and its importance will become clear. In short, we can say that η^S is the explicit mediator of this hierarchy.

The function ζ^S is similarly important in a way that bears some familiarity to all programmers that have worked in the style of *functional programming.* If we consider another monoid M, with a different product and identity from those of the free one, the lifting defined by ζ^S takes, as its domain, a function h from S into the underlying set of M, interpreting each element of S in M. The image of ζ^S over k is a monoid homomorphism h that interprets any element α of $\mathbf{Mon}(S)$ in M. This interpretation takes any term representation of α and crawls along its syntax, replacing each formal $\underline{\otimes}$ operator with $\otimes_M$, each formal $\underline{\mathbb{1}}$ with $\mathbb{1}_M$, and each base element $x \in S$ with its interpretation $h(x)$. The result is an expression that can be evaluated in M.

This is, indeed, the *map-reduce* pattern familiar in functional programming; the *map* part being the application of h to each base element of a term, and the *reduce* part being the interpretation of the monoid M that provides a means to reduce all of the interpretations of the base elements into a single element of M. In particular, we get the familiar *map* function by itself when we consider a function $f : S \to S'$ which is then composed with the embedding $\eta^{S'}$ associated with the free monoid $\mathbf{Mon}(S')$. This composite $\eta^{S'} \circ f$ is lifted by ζ^S into a homomorphism between two free monoids. Hence, one could state that **map** is defined

$$\mathbf{map}(f) \overset{\text{def}}{=} \zeta^S(\eta^{S'} \circ f)$$

which we will generalize in the next section.

Likewise, the *reduce* component by itself can be distilled. Here one must consider a free monoid where the base elements are already the underlying elements of another monoid M. In this case, the set in question is $\mathcal{U}_{\mathbf{Mon}}(M)$, and the function to be lifted through $\eta^{\mathcal{U}_{\mathbf{Mon}}}$ is simply the identity $\mathbf{id}_{\mathbf{Fun}}(\mathcal{U}_{\mathbf{Mon}}(M))$. This results in a homomorphism between $\mathbf{Mon}(\mathcal{U}_{\mathbf{Mon}}(M))$ and M itself. Hence, one could define **reduce**

$$\mathbf{reduce}_M \overset{\text{def}}{=} \zeta^{\mathcal{U}_{\mathbf{Mon}}(M)}(\mathbf{id}_{\mathbf{Fun}}(\mathcal{U}_{\mathbf{Mon}}(M)))$$

This, as well, will give rise to a generalization described in the next section.

Derived Structures

Having defined a free element f^X in a category of structures $\mathcal{S}$ for each underlying structure X in a category $\mathcal{G}$, and therefore having η^X and ζ^X for each of these as well, further structures significant to category theory can be derived. Namely, an adjoint pair of functors between $\mathcal{G}$ and $\mathcal{S}$ can be derived from the parameterized set of universal morphisms. It will be seen that these two derived structures build on free elements in a useful way, generalizing some of the structures defined for free monoids in the previous section. Casting free elements into the formulation of adjoints also provides an additional advantage in allowing us to invoke the well-known *Adjoint Functor Theorem*, a central theorem in category theory which we will state at the end of this section.

Given two categories $\mathcal{G}$ and $\mathcal{S}$, and a forgetful functor $\mathcal{U}_{\mathcal{S}} : \mathcal{S} \to \mathcal{G}$, if we have a universal morphism (f^X, η^X, ζ^X) for every $X \in G$ we can extend the particular mapping $X \mapsto f^X$ between objects in the respective categories to a functor $\mathbf{f}$ between them; that is $\mathcal{G} \to \mathcal{S}$. We will call this functor the *free functor*. We can then show that this functor is a *left adjoint* to $\mathcal{U}_{\mathcal{S}}$. We give this construction, including both forms of the adjunction, in a theorem that is derived from Theorem IV.2 in [39].

Theorem 5.
Given a universal morphism (f^X, η^X, ζ^X) from X into $\mathcal{U}_{\mathcal{S}}$ for each $X \in \mathcal{G}$, the following can be defined:

- *a functor $f : \mathcal{G} \to \mathcal{S}$ defined*

$$f(X) \overset{def}{=} f^X$$
$$f(w : \mathbf{hom}(X, Y)) \overset{def}{=} \eta^Y \circ w$$

forming an adjunction with $\mathcal{U}_{\mathcal{S}}$

$$(\theta,\, \zeta) \,:\, \mathcal{F} \dashv \mathcal{U}$$

where θ and ζ are natural transformations with θ^X and ζ^X as their respective components for each X, forming a natural isomorphism witnessing the adjunction.

- *an adjoint unit/counit pair $(\eta,\, \epsilon)$ where the unit η is the natural transformation with η^X as its components, and counit is defined as a natural transformation with the components*

  $$\epsilon_d \overset{def}{=} \zeta^{\mathcal{U}_{\mathcal{S}}(d)}(\mathbf{id}_{\mathcal{U}_{\mathcal{S}}(d)})$$

  also witnessing $\mathcal{F} \dashv \mathcal{U}$.

Proof.
First, f must be shown to be a functor. For this the uniqueness property of the image of ζ will be used. For the identity,

$$f(\mathbf{id}(X)) = \zeta(\eta^X \circ \mathbf{id}(X)) = \zeta(\eta^X)$$

and

$$\theta^X(\mathbf{id}(f(X))) = \mathcal{U}_{\mathcal{S}}(\mathbf{id}(f(X))) \circ \eta^X = \mathbf{id}(\mathcal{U}_{\mathcal{S}}(f(X))) \circ \eta^X = \eta^X$$

hence

$$\mathbf{id}(\mathcal{U}_{\mathcal{S}}(f(X))) = \zeta^X(\theta^X(\mathbf{id}_{f(X)})) = \zeta^X(\eta^X) = f(\mathbf{id}(X))$$

For composition, consider two morphisms $u \,:\, \mathbf{hom}(X,\, Y)$ and $v \,:\, \mathbf{hom}(Y,\, Z)$

$$\begin{aligned} &f(v) \circ f(u) = \zeta^X(\theta^X(f(v) \circ f(u))) \\ &= \zeta^X(\mathcal{U}_{\mathcal{S}}(f(v) \circ f(u)) \circ \eta^Y) = \zeta^X(\mathcal{U}_{\mathcal{S}}(f(v)) \circ \mathcal{U}_{\mathcal{S}}(f(u)) \circ \eta^Y) \end{aligned}$$

and

$$\begin{aligned} &\mathcal{U}_{\mathcal{S}}(f(v)) \circ \mathcal{U}_{\mathcal{S}}(f(u)) \circ \eta^Y = \mathcal{U}_{\mathcal{S}}(f(v)) \circ \theta^X(f(u)) \\ &= \mathcal{U}_{\mathcal{S}}(f(v)) \circ \theta^X(\zeta^X(\eta^Z \circ u)) = \mathcal{U}_{\mathcal{S}}(f(v)) \circ \eta^Y \circ u \\ &= \theta^Y(f(v)) \circ u = \theta^Y(\zeta^Y(\eta^Z \circ v)) \circ u \\ &= \theta^Y(\zeta^Y(\eta^Z \circ v)) \circ u = \eta^Z \circ v \circ u \end{aligned}$$

thus

$$f(v) \circ f(u) = \zeta^X(\eta^Z \circ v \circ u) = f(v \circ u)$$

The naturality of θ in both indices is confirmed by verifying the naturality squares for each. For the first parameter d, the contravariant square for $p : \mathbf{hom}(Y, Z)$ yields

$$\theta^Y \circ \mathbf{hom}_{\mathcal{S}}(\mathcal{F}(p), d) = \mathbf{hom}_{\mathcal{G}}(p, \mathcal{U}(d)) \circ \theta^Z$$

Which can be confirmed by taking an element $\phi : \mathbf{hom}_{\mathcal{S}}(\mathcal{F}(Z), d)$ through the left side of the equation:

$$\begin{aligned}
&\theta^Y \circ \mathbf{hom}_{\mathcal{S}}(\mathcal{F}(p), d)(\phi) = \theta^Y(\phi \circ \mathcal{F}(p)) \\
&= \mathcal{U}(\phi \circ \mathcal{F}(p)) \circ \eta^Y = \mathcal{U}(\phi) \circ \mathcal{U}(\mathcal{F}(p)) \circ \eta^Y \\
&= \mathcal{U}(\phi) \circ \mathcal{U}(\mathcal{F}(p)) \circ \eta^Y = \mathcal{U}(\phi) \circ \theta^Y(\mathcal{F}(p)) \\
&= \mathcal{U}(\phi) \circ \theta^Y(\zeta^Y(\eta^Z \circ p)) = \mathcal{U}(\phi) \circ \eta^Z \circ p \\
&= \theta^Z(\phi) \circ p = \mathbf{hom}_{\mathcal{G}}(p, \mathcal{U}(d)) \circ \theta^Z(\phi)
\end{aligned}$$

Fixing the second parameter Y, the covariant square for $p : \mathbf{hom}(Y, Z)$ yields

$$\theta^Z \circ \mathbf{hom}_{\mathcal{S}}(\mathcal{F}(Y), p) = \mathbf{hom}_{\mathcal{Y}}(Y, \mathcal{U}(p))\theta^Y$$

Which can be confirmed by taking an element $\phi : \mathbf{hom}_{\mathcal{S}}(\mathcal{F}(Y), d)$ through the left side of the equation:

$$\begin{aligned}
&\theta^Z \circ \mathbf{hom}_{\mathcal{S}}(\mathcal{F}(Y), p)(\phi) = \theta^Z(p \circ \phi) \\
&= \mathcal{U}(p \circ \phi) \circ \eta^Y = \mathcal{U}(p) \circ \mathcal{U}(\phi) \circ \eta^Y \\
&= \mathcal{U}(p) \circ \theta^Y(\phi) = \mathbf{hom}_{\mathcal{S}}(Y, \mathcal{U}(p)) \circ \theta^Y(\phi)
\end{aligned}$$

The derivation of the unit and counit proceed as in Theorem IV.2 [39]. □

The importance of being able to derive the free functor from the universal morphism that defines the free structure in a category is that we are given a canonical means to lift morphisms between generators of free structures into homomorphisms between their respectively generated free structures. In other words, when an underlying alphabet of basic elements is chosen for generating a free structure, and there is a morphism m in $\mathcal{G}$ abstracting these elements or immersing them in a larger set of elements, there will be a corresponding morphism $f(m)$ in $\mathcal{S}$ that correspondingly abstracts or immerses the generated structures.

This is precisely what was done with the example of free monoids in defining the **map** function. It can be seen from the above constructions that our definition for **map** was really the morphism component of the free functor. We can thus take advantage of the convention we have already established, to a degree, and use the name **Mon** of the category to also name the derived free functor. Using this convention, we can state that $\mathbf{map}(f) = \mathbf{Mon}(f)$. Generalizing the free functor from our example of the case for monoids, in other cases the free functor acts like a **map** function over morphism, crawling along the structures of the free element and applying the morphism to the base elements.

Furthermore, the unit/counit pair, witnessing the adjunction $\mathcal{F} \dashv \mathcal{U}$, are important derivatives of the universal morphism, because of their role in defining hierarchical compositions in the free structures of $\mathcal{S}$. This generalizes our discussion of hierarchy in the case of free monoids. We can define the two composition of the free functor and forgetful functor as

$$\begin{aligned} &T_{\mathcal{S}} : \mathcal{G} \to \mathcal{G} \\ &T_{\mathcal{S}} \overset{\text{def}}{=} \mathcal{U}_{\mathcal{S}} \circ \mathcal{F} \\ &R_{\mathcal{S}} : \mathcal{S} \to \mathcal{S} \\ &R_{\mathcal{S}} \overset{\text{def}}{=} \mathcal{F} \circ \mathcal{U}_{\mathcal{S}} \end{aligned}$$

which are both endofunctors. Starting with $X \in \mathcal{G}$, one can construct the nth-level hierarchical free element through n applications of $T_{\mathcal{S}}$ followed by $\mathcal{F}$.

$$\mathcal{F}(X),\, \mathcal{F}(T_{\mathcal{S}}(X)),\, \mathcal{F}(\mathcal{F}(T_{\mathcal{S}}(X))),\, \dots,\, \mathcal{F}(T^n_{\mathcal{S}}(X)),\, \dots$$

For the nth level, as in the case of the hierarchical free monoids, the basic elements of the terms of the free element form the underlying structure of the $n-1$th level. The components of the unit η can be given the type

$$\eta : X \to T_{\mathcal{S}}(X)$$

and the interpretation of taking a set of elements one step up in this hierarchy. In the case of the free monoids, this step up was explicit, wrapping each element so as to maintain the distinguishability between the levels.

If we begin the series of hierarchical levels from the other side $\mathcal{S}$, starting with an arbitrary structure $M \in \mathcal{S}$, we can express the sequence of levels as follows.

$$M,\, R_{\mathcal{S}}(M),\, R_{\mathcal{S}}(R_{\mathcal{S}}(M)),\, \dots,\, R^n_{\mathcal{S}}(M),\, \dots$$

The components of the counit ϵ, can be given the type

$$\epsilon : R_{\mathcal{S}}(M) \to M$$

and analogously be interpreted as stepping down levels of this hierarchy. In the case of the free monoid, this was the role of the **reduce** function. Indeed we can state that

$$\epsilon_M = \mathbf{reduce}_M$$

as can be seen comparing the two definitions. Like the **reduce** function for free monoids, the counit is often a means of taking equivalence classes of terms in $R_{\mathcal{S}}(M)$, built up out of elements of M, and evaluating one of (any of) the term representations in M.

Finally, we will state *Adjoint Functor Theorem* in the form that we will use it, as a corollary of the more general form of the theorem, expressed in terms of limits rather than colimits, in [39].

Corollary 2. *Co-Adjoint Functor Theorem*
Given a small co-complete category $\mathcal{G}$, and a functor $\mathcal{F} : \mathcal{G} \to \mathcal{S}$, for some category $\mathcal{S}$, if $\mathcal{F}$ has a right adjoint, then it preserves all small colimits.

Proof. This follows from Theorem V.6.2 in [39], using duality. □

This theorem is relevant to free functors, since they have forgetful functors as their right adjoints. Provided it is shown that the category of underlying structures $\mathcal{G}$ has small colimits, it follows that these limits can be carried over to $\mathcal{S}$. In more intuitive terms, if one takes a collection of generating structures X_1, X_2, X_3, ..., and combining them into a generating structure X_* (using a colimit), $\mathcal{F}(X_*)$ is isomorphic to the structure produced by performing the same kind of combination to the collection $\mathcal{F}(X_1)$, $\mathcal{F}(X_2)$, $\mathcal{F}(X_3)$, ...

This is certainly true for monoids, since the underlying structures are sets, and sets are most certainly small co-complete. This will be relevant for us later, when dealing with OEGs in the form of free symmetric monoidal categories. Combining a collection of spaces of OEGs can be achieved through performing the same combination of the more simple spaces of their generators, then carrying the result back through the free functor.

Chapter 6

Free Monoidal Categories

Combining the concepts presented in Chapter 5, free elements and monoidal categories, we will construct free monoidal categories, the freely generated elements of the category of monoidal categories, along with all of the associated and derived structures. We will give this construction with details specifically regarding the *symmetric* variant. This variant, the free symmetric monoidal category, will serve as the mathematical backbone of our concurrent behavioral representation, OEGs. Rather than directly define OEGs as free symmetric monoidal categories, they will be defined more directly as a kind of enriched graph in Chapter 7. The result of Joyal and Street in [27] will connect this definition in Chapter 7 to the constructions we will give in this chapter.

Before generating free monoidal categories from underlying structures as we have demonstrated with monoids in Chapter 5, making use of a forgetful functor, it must first be established what these underlying structures are for monoidal categories. What basic information can a monoidal category be generated from? In the case of monoids, as we have seen, this basic information consists of a single set of generating elements. Whereas the underlying structures of monoids, and many other algebras, are indeed simply sets, monoidal categories instead should have both an underlying sets of basic objects as well as a set of basic morphisms. Moreover, to utilize the definition we wish to use for free structures, we must be able to define a forgetful functor that *forgets* the structure of monoidal categories, leaving behind the underlying objects and morphisms in the same form as the generators. To accomplish this, the idea of a monoidal schemes is taken from the work of Joyal and Street, who define free monoidal categories in [27]. Their monoidal schemes are precisely pairs containing a set of basic objects and basic morphisms.

Building on this concept, we define monoidal schemes here in a manner that fits our formal definition of a category and proceed to show how these structures form a category, $\mathcal{MS}$. The treatment in [27] of these structures does not do much with them beyond using them to generate free monoidal categories, but in the context of this thesis monoidal schemes will serve a more important, ontologically significant, role. They will be used to represent the so called “alphabets” of event types, called OESs, out of which the “words” of OEGs, which we will call *diagrams*, are constructed. In order to abstract and/or embed these “alphabets”, as a means of reasoning about different kinds of behavioral representation, it will be useful for us to formally define transformations between them, called *monoidal scheme functors*, that serve as the morphisms of their category.

Our construction of free monoidal categories from monoidal schemes will then proceed in a manner that bears similarity to that in [27], but more closely follows our account of the construction of free monoids in Chapter 5. The construction in [27] uses a different, but equivalent, definition of a universal morphism as the initial element of a category of functions from their *tensor schemes* into monoidal categories. The consequence of this approach is that there is not much elaboration on the schemes themselves. We will also take time during the construction to explicitly construct the algebraic language of terms that will arise from the operators and constants of monoidal categories and the relevant variant.

Drawing from Theorem 5, in Chapter 6, we will use the construction of free monoidal categories to further derive the free functors from $\mathcal{MS}$ to the category **MonCat** of monoidal categories with strict strong monoidal functors, left adjoint to the forgetful functor, $\mathcal{U}_{\mathbf{MonCat}}$. We will also derive the unit and counit that arise from this adjoint pair, and discuss the significance of these elements, which can be used as tools for working mathematically with OEGs.

We will then go on to explore important properties of both monoidal schemes and free monoidal categories. Specifically, we will show that the category $\mathcal{MS}$ is small co-complete, meaning that constrained combinations of monoidal schemes can be formed. Later this will be seen as a useful property for conjoining different kinds of behavioral representations, especially in heterogeneous systems. Using the dual of the *Adjoint Functor Theorem* of Freyd, it will be shown that small colimits are preserved through the free construction, and thus combining systems of representations can be reduced to combining their generating alphabets.

In the latter sections of this chapter, when it is important to distinguish between the specific variants of monoidal categories, either monoidal categories themselves or symmetric monoidal categories, we will give certain definitions and proofs with in a variant parametric form, using terms and notations such as *X-monoidal category* and **XMonCat**, where "X" can be omitted or replaced with "S". Where differences exist, they will be noted explicitly.

6.1 Monoidal Schemes

The generating elements of free monoidal categories, monoidal schemes, are defined as follows

Definition 16. *Monoidal scheme*
A monoidal scheme Σ *is defined*

$$\Sigma \overset{def}{=} (\mathbb{T}, \mathcal{A})$$

where

$$\begin{aligned}\mathbb{T} &: \mathcal{U}\\ \mathcal{A} &: \mathcal{I} \to \mathcal{U}\\ \mathcal{I} &\overset{def}{=} \mathbf{Mon}(\mathbb{T}) \times \mathbf{Mon}(\mathbb{T})\end{aligned}$$

under the constraint

$$\forall\, a,\, b : \mathcal{I}\, \cdot\, a \neq b \,\Rightarrow\, \mathcal{A}(a) \cap \mathcal{A}(b) = \emptyset$$

For a monoidal scheme:

- $\mathbb{T}$ *will be called the* primitive types
- $\mathcal{A}$ *will be called the* primitive blocks
- *the elements of* $\mathbf{Mon}(\mathbb{T})$ *will be called the* types
- $\mathcal{I}$ *will be called the interface*

The notation a_- *and* a_+*, for* $a \in \mathcal{I}$*, will be used to refer to the first and second components of* a*.*

This definition is based on that of [27], and similar to that of a graph (multigraph) used to generate a category, as in [39]. In the case of a graph, a collection of objects and primitive morphisms are given, which by themselves constitute a graph, without any notion of a composition or identity. Here, the monoidal scheme differs from a graph in that the first component $\mathbb{T}$ are themselves primitive objects used to generate the free monoid $\mathbf{Mon}(\mathbb{T})$ that will serve as the ultimate set of objects for the monoidal category. In other words, $\mathbf{Mon}(\mathbb{T})$, and not $\mathbb{T}$, parameterize the homsets. Consequently, the sets of primitive blocks in $\mathcal{A}$, which generate the homsets of the monoidal category are also indexed by $\mathbf{Mon}(\mathbb{T})$ – rather than $\mathbb{T}$ as they would be in a graph. The disjointness condition is a fairly obvious albeit necessary restriction, stating that each primitive block can only have one interface. This is a consequence of our choice of formalisms for categories themselves, built on a dependently typed total composition over homsets rather than a partial composition over a single set of morphisms.

One might posit a slightly different generator, based on a collection of types which is already an arbitrary monoid, making the construction more similar to that of a category from a graph. But the specific use of a free monoid $\mathbf{Mon}(\mathbb{T})$ is important because the elements of an arbitrary monoid cannot necessarily be decomposed into a unique product of primitive types. This decomposition is possible in $\mathbf{Mon}(\mathbb{T})$, because it is a free monoid, and is a feature we would like for the purposes of our behavioral representation. Specifically, we would like to be able to uniquely index the primitive components of the interface, and indexing is something we can do with a free monoid (as we discussed in Chapter 3).

As a convenience, the following function will be defined for an monoidal scheme to give the interface of a particular primitive block.

Definition 17. *For an monoidal scheme Σ, let the following be defined:*

- *the set of all blocks*

$$\mathcal{A}_{\cup} \overset{def}{=} \bigcup_{x \in \mathcal{I}} \mathcal{A}(x)$$

- *the declarative notation*

$$A : a \to b \overset{def}{=} A \in \mathcal{A}(a, b)$$

  as a means to indicate the interface of a block.

- *the function τ, giving the interface of a block*

$$\tau_\Sigma : \prod_{x : \mathcal{I}^\Sigma} \mathcal{A}^\Sigma(x) \to \mathcal{I}^\Sigma$$
$$\tau_\Sigma(A : \mathcal{A}^\Sigma(x)) \overset{def}{=} x$$

- *the function $\|-\|$, giving a pair of ordinals called a valence for each interface*

$$\|-\|^\Sigma : \mathcal{I}^\Sigma \to \mathbf{Ords} \times \mathbf{Ords}$$
$$\|x\|^\Sigma_{+/-} \overset{def}{=} \|x_{+/-}\|$$

  We will also further overload $\|-\|^\Sigma$ to mean $\|-\|^\Sigma \circ \tau$ when over blocks rather than interfaces.

As a convention, the superscript will be left out of the above functions, since the monoidal scheme can be inferred from the context. The notations τ_- and τ_+, and the like, will also be used to identify the left and right components, respectively, of the pair following the above convention for naming components of interfaces.

In light of the function τ, serving the purpose associating an interface with a block, another way of framing the definition of $\mathcal{A}$ in a monoidal scheme is that the entire collection of primitive blocks form a bundle over the space of interfaces $\mathcal{I}$. For each $x \in \mathcal{I}$, $\mathcal{A}(x)$ is the fiber over x. $\mathcal{A}$ is then the preimage of the projection τ, which is always disjoint because τ is a function.

The valence function $\|-\|$ gives a pair of ordinals to each type in the interface, indicating in the case of what we have defined so far the number of primitive type components in the type. Reflecting on what was discussed in Chapter 3, it is important that such a function can only be defined because $\mathbf{Mon}(\mathbb{T})$ is specifically a free monoid, and thus has a well-defined number of components in its canonical representation.

Maps can be defined between these monoidal schemes constituted of functions over each of their components in a manner consistent with the parameters of $\mathcal{A}$.

Definition 18. *Monoidal scheme functor*
A monoidal scheme functor F is defined

$$\begin{aligned} F &: (\mathbb{T}^{\Sigma}, \mathcal{A}^{\Sigma}) \to (\mathbb{T}^{\Sigma'}, \mathcal{A}^{\Sigma'}) \\ F &\overset{def}{=} (F_{\mathbb{T}}, F_{\mathcal{A}}) \end{aligned}$$

with the typing

$$\begin{aligned} F_{\mathbb{T}} &: \mathbb{T}^{\Sigma} \to \mathbb{T}^{\Sigma'} \\ F_{\mathcal{A}} &: \prod_{x\,:\,\mathcal{I}^{\Sigma}} \mathcal{A}^{\Sigma}(x) \to \mathcal{A}^{\Sigma'}(\mathbf{Mon}(F_{\mathbb{T}})(x)) \end{aligned}$$

We will use F to notate both components when it does not cause ambiguity.

(In the above we overload the notation $\mathbf{Mon}(F_{\mathbb{T}})$ to refer to the lifting of $F_{\mathbb{T}}$ acting pointwise over tuples). This definition is similar to that of a functor, except that no operators or constants are preserved, and that the parameters of $F_{\mathcal{A}}$ are in the free monoid of the types. The transformation over blocks can be thought of as constituted from individual function for each interface in $\mathcal{I}$. Another way to think of this is that the $F_{\mathbb{T}}$ component transform the interface structure, while $F_{\mathcal{A}}$ maps blocks from each interface in Σ into the corresponding interface in Σ'.

Because an monoidal scheme functor is defined in terms of functions over primitive types $\mathbb{T}$, rather than functions over all types $\mathbf{Mon}(\mathbb{T})$, the valence of each block is preserved. That is, for any monoidal scheme functor F

$$\|-\| \circ F = \|-\|$$

This property is important as it leaves structural (geometric) details about blocks invariant through monoidal scheme functors. A component primitive type cannot be broken into pieces, nor can two components be fused together.

Given the definition of these mappings are based entirely on functions, an identity can be defined for each monoidal scheme and a composition can be defined for any pair of monoidal scheme functors.

Definition 19.

- *For each monoidal scheme Σ,*

$$\begin{aligned} \mathbf{id}(\Sigma) &: \Sigma \to \Sigma \\ \mathbf{id}(\Sigma) &\overset{def}{=} (\mathbf{id}(\mathbb{T}), \mathbf{id}(\mathcal{A})) \end{aligned}$$

  where $\mathbf{id}(\mathbb{T})$ is the identity function over $\mathbb{T}$, and, for each $x \in \mathcal{I}$, $\mathbf{id}(\mathcal{A})(x)$ is the identity function over $\mathcal{A}(x)$.

- *For each pair of monoidal scheme functors*

$$F : \Sigma \to \Sigma'$$
$$G : \Sigma' \to \Sigma''$$

composition is defined

$$G \circ F : \Sigma \to \Sigma''$$
$$G \circ F \overset{def}{=} (G_{\mathbb{T}} \circ F_{\mathbb{T}}, (G \circ F)_{\mathcal{A}})$$

where

$$(G \circ F)_{\mathcal{A}}(x) \overset{def}{=} G_{\mathcal{A}}(F_{\mathbb{T}}(x)) \circ F_{\mathcal{A}}(x)$$

Using these two definitions, monoidal schemes form a category with monoidal scheme functors as its morphisms

Proposition 5. $\mathcal{MS}$
The set of monoidal schemes and monoidal scheme functors with the definitions identity and composition above form a category. Let this category be called $\mathcal{MS}$.

Proof. Since each monoidal scheme functor is constituted of a function and a set of functions parameterized over pairs of monoidal schemes, and composition and identity are defined by the definition for functions over each of these constituting functions, associative and identity laws can be derived from those over functions. □

This category has several useful properties, the identification and discussion of which we will defer till after we have completed the construction of free monoidal categories from the objects of this category.

6.2 Free Monoidal Categories

Having defined the category of monoidal schemes, we can now proceed in the construction of free monoidal categories. The first step in this process, is to define the forgetful functor mapping monoidal categories into monoidal schemes. By monoidal categories, we will specifically mean the category **XMonCat** of strict X-monoidal categories with strict strong *X-monoidal functors*.

Definition 20.
Let the forgetful functor for the category **XMonCat** *of (strict) X-monoidal categories with (strict strong) X-monoidal functors be defined*

$$\mathcal{U}_{\mathbf{XMonCat}} : \mathbf{XMonCat} \to \mathcal{MS}$$
$$\mathcal{U}_{\mathbf{XMonCat}}(\mathcal{O}, \mathbf{hom}, \ldots) = (\mathcal{O}, \mathbf{hom})$$
$$\mathcal{U}_{\mathbf{XMonCat}}(F) = F$$

where F is a X-monoidal functors.

This definition simply maps the objects of the category into a set of primitive types and the morphism into blocks, forming a monoidal scheme and forgetting the algebraic mechanisms.

Following the same procedure defined in Chapter 5 for constructing a free object, we must then define a universal morphism from each $\Sigma \in \mathcal{MS}$ into $\mathcal{U}_{\mathbf{XMonCat}}$. As was the case for free monoids, the first component of this universal morphism, f^{Σ}, the free element itself, will be defined as equivalence classes of terms in the term language of monoidal categories, and the associated variant. We will first construct this term language along the same lines as we did for the free monoid, then use this term language to complete the construction and derive the associated structures.

Monoidal Term Structures

The formal terms of X-monoidal categories, like those of monoids, are simply those formed from the operators and constants of X-monoidal categories. The construction will follow the same pattern as that of free monoids discussed in Chapter 5. We will refer to these collections of terms, constructed from particular monoidal scheme, as *X-monoidal term structures*. The terms will then be placed into equivalence classes over the axioms of the given monoidal category to form its morphism.

Although a similar inductive mechanism can be employed to construct the terms of X-monoidal categories from a monoidal scheme, as was used in constructing free monoids from a set, there is an important difference in this construction. The collection of terms for a particular monoidal scheme is no longer a single set. Instead, the collection $\mathbf{XMCTerms}^{\Sigma}$ is parameterized over pairs of elements of $\mathbf{Mon}(\mathbb{T}^{\Sigma})$. The induction process then mutually builds up these sets, while also constraining the construction of terms to fulfill what one might call the "typing constraints" of the language, specifically because the categorical composition is a dependent function of the interfaces of its operands.

Definition 21. *X-monoidal category Terms*
Given a monoidal scheme Σ, for each x, $y \in \mathbf{Mon}(\mathbb{T}^{\Sigma})$ there is set of X-monoidal category terms $\mathbf{XMCTerms}^{\Sigma}(x, y)$. These sets are mutually constructed inductively by the following inference rules.

$$\frac{A \in \mathcal{A}^{\Sigma}(x,\, y)}{A \in \mathbf{XMCTerms}^{\Sigma}(x,\, y)}$$

$$\frac{}{\underline{\mathbf{id}_x} \in \mathbf{XMCTerms}^{\Sigma}(x,\, x)}$$

$$\frac{}{\underline{\beta_{x,y}} \in \mathbf{XMCTerms}^{\Sigma}(x \otimes y,\, y \otimes x)}$$

$$\frac{f \in \mathbf{XMCTerms}^{\Sigma}(x,\, y) \qquad g \in \mathbf{XMCTerms}^{\Sigma}(y,\, z)}{g \,\underline{\circ}\, f \in \mathbf{XMCTerms}^{\Sigma}(x,\, z)}$$

$$\frac{f \in \mathbf{XMCTerms}^{\Sigma}(w,\, x) \qquad g \in \mathbf{XMCTerms}^{\Sigma}(y,\, z)}{f \,\underline{\otimes}\, g \in \mathbf{XMCTerms}^{\Sigma}(w \otimes y,\, x \otimes z)}$$

where the parameters $x,\, y,\, z,\, \dots$ of $\mathbf{XMCTerms}_{\Sigma}$ all vary over $\mathbf{Mon}(\mathbb{T}^{\Sigma})$, and there are a set of distinct constants $\underline{\mathbf{id}}(x)$ for each $x \in \mathbf{Mon}(\mathbb{T}^{\Sigma})$ and $\underline{\beta}(x, y)$ for each $x,\, y \in \mathbf{Mon}(\mathbb{T}^{\Sigma})$.

It is worth making the distinction that these rules depend on the elements of $\mathbf{Mon}(\mathbb{T}^{\Sigma})$ and not monoidal terms in $\mathbf{MonTerms}(\Sigma)$. If the latter were used to parameterized the collection of X-monoidal category terms there would be an additional complication of introducing the rewriting of these terms into the induction rules. This would pose a challenge for proof by structural induction because it would introduce inductive derivations that do not strictly increase the complexity of the terms. However, one does not encounter problems in establishing the identity of the words of free monoids in practice because the word problem on free monoids is decidable – in fact fairly trivial, since words in free monoid have a unique expansion into a sequence of their generating elements.

Complementing the X-monoidal category terms, we will define the corresponding linear contexts. As opposed to those of monoidal terms, the holes in these contexts are "typed" so that they can be filled with terms from a particular $\mathbf{XMCTerms}^{\Sigma}(x,\, y)$. We therefore must denote them as $\bigcirc_{x,y}$ with parameters from $\mathbf{Mon}(\mathbb{T}^{\Sigma})$.

Definition 22. *Linear X-monoidal category Contexts*
Given a monoidal scheme Σ, for each $x,\, y \in \mathbf{Mon}(\mathbb{T}^{\Sigma})$ there is set of linear X-monoidal category contexts $\mathbf{XMCContexts}^{\Sigma}_{u,v}(x,\, y)$. These are constructed inductively by the following inference rules

$$\frac{}{\bigcirc_{u,v} \in \mathbf{XMCContexts}^{\Sigma}_{u,v}(u,\, v)}$$

$$\frac{\gamma \in \mathbf{XMCContexts}^{\Sigma}_{u,v}(x,\, y) \qquad f \in \mathbf{XMCTerms}^{\Sigma}(y,\, z)}{f \,\overline{\circ}\, \gamma \in \mathbf{XMCContexts}^{\Sigma}_{u,v}(x,\, z)}$$

$$\frac{f \in \mathbf{XMCTerms}^{\Sigma}(x,\, y) \qquad \gamma \in \mathbf{XMCContexts}^{\Sigma}_{u,v}(y,\, z)}{\gamma \,\overline{\circ}\, f \in \mathbf{XMCContexts}^{\Sigma}_{u,v}(x,\, z)}$$

$$\frac{\gamma \in \mathbf{XMCContexts}^{\Sigma}_{u,v}(w,\, x) \qquad f \in \mathbf{XMCTerms}^{\Sigma}(y,\, z)}{\gamma \,\overline{\otimes}\, f \in \mathbf{XMCContexts}^{\Sigma}_{u,v}(w \otimes x,\, y \otimes z)}$$

$$\frac{f \in \mathbf{XMCTerms}^{\Sigma}(w,\, x) \qquad \gamma \in \mathbf{XMCContexts}^{\Sigma}_{u,v}(y,\, z)}{f \,\overline{\circ}\, \gamma \in \mathbf{XMCContexts}^{\Sigma}_{u,v}(w \otimes x,\, y \otimes z)}$$

Each element $\gamma \in \mathbf{XMCContexts}^{\Sigma}_{u,v}(x,\, y)$ is a function

$$\mathbf{XMCTerms}^{\Sigma}(u,\, v) \to \mathbf{XMCTerms}^{\Sigma}(x,\, y)$$

These are defined, recursively, for any $a,\, b \in \mathbf{MonTerms}(S)$

$$\begin{aligned}
\bigcirc_{u,v}(a) &\overset{def}{=} a \\
(\gamma \mathbin{\overline{\circ}} a)(b) &\overset{def}{=} \gamma(b) \mathbin{\underline{\circ}} a \\
(a \mathbin{\overline{\circ}} \gamma)(b) &\overset{def}{=} a \mathbin{\underline{\circ}} \gamma(b) \\
(\gamma \mathbin{\overline{\otimes}} a)(b) &\overset{def}{=} \gamma(b) \mathbin{\underline{\otimes}} a \\
(a \mathbin{\overline{\otimes}} \gamma)(b) &\overset{def}{=} a \mathbin{\underline{\otimes}} \gamma(b)
\end{aligned}$$

With both the terms and contexts, an equivalence relation can be defined, generated from the axioms of X-monoidal categories. Rather than explicitly writing out the inductive system completely, we will simply refer to the axioms for the corresponding variant of monoidal categories we gave in Definitions 7 and 9. The nuance in this definition again will be the presence of "typing"; in fact, it will be generated from a parameterized collection of equivalence relations.

Definition 23. *X-monoidal category Equivalence*
Given a monoidal scheme Σ, for each $x,\, y \in \mathbf{Mon}(\mathbb{T}^{\Sigma})$ there is an equivalence relation

$$\xLeftrightarrow{\mathbf{XMonCat}(x,\, y)} \subseteq \mathbf{XMCTerms}(x,\, y) \times \mathbf{XMCTerms}(x,\, y)$$

defined inductively as follows. For every axiom in $\mathbf{Axioms}_{\mathbf{XMonCat}}$ of the form $\mathbf{LHS} = \mathbf{RHS}$, let there be an inference rule

$$\frac{}{\mathbf{LHS} \xLeftrightarrow{\mathbf{XMonCat}(x,\, y)} \mathbf{RHS}}$$

where each operator and constant is replaced with its formal counterpart. For each $a,\, b \in \mathbf{XMCTerms}(u,\, v)$ and each $\gamma \in \mathbf{XMCContexts}^{\Sigma}_{u,v}(x,\, y)$, let there be an inference rule

$$\frac{a \xLeftrightarrow{\mathbf{XMonCat}(u,\, v)} b}{\gamma(a) \xLeftrightarrow{\mathbf{XMonCat}(x,\, y)} \gamma(b)}$$

For each $x,\, y \in \mathbf{Mon}(\mathbb{T}^{\Sigma})$, let the function

$$\langle - \rangle_{\mathbf{XMonCat}(x,\, y)} \ : \ \mathbf{XMCTerms}(x,\, y) \to \mathbf{XMCTerms}(x,\, y)/ \xLeftrightarrow{\mathbf{XMonCat}(x,\, y)}$$

be the canonical projection induced by $\mathbf{XMCTerms}(x,\, y)$, where

$$\mathbf{XMCTerms}(x,\, y)/ \xLeftrightarrow{\mathbf{XMonCat}(x,\, y)}$$

is the canonical quotient. Finally, the combined equivalence relation is defined

$$\xLeftrightarrow{\mathbf{XMonCat}} \overset{def}{=} \bigcup_{x,\,y \in \mathbf{Mon}(\mathbb{T}^{\Sigma})} \xLeftrightarrow{\mathbf{XMonCat}(x,\,y)}$$

Likewise, we define the parametric canonical projection

$$\langle - \rangle_{\mathbf{XMonCat}} : \mathbf{XMCTerms} \to \mathbf{XMCTerms} / \xLeftrightarrow{\mathbf{XMonCat}}$$
$$\langle - \rangle_{\mathbf{XMonCat}} \overset{def}{=} \sum_{x,\,y \in \mathbf{Mon}(\mathbb{T}^{\Sigma})} \langle - \rangle_{\mathbf{XMonCat}(x,\,y)}$$

where

$$\mathbf{XMCTerms} / \xLeftrightarrow{\mathbf{XMonCat}} \overset{def}{=} \bigcup_{x,\,y \in \mathbf{Mon}(\mathbb{T}^{\Sigma})} \mathbf{XMCTerms}(x,\,y) / \xLeftrightarrow{\mathbf{XMonCat}(x,\,y)}$$

It should be clear from this definition that the equivalence relations are disjoint. In other words, two X-monoidal category terms can only be equivalent if they, at least, have the same "type"; the rewritings preserve the "typing". Consequently, their union in $\xLeftrightarrow{\mathbf{XMonCat}}$ is a disjoint one, which is tantamount to asserting that the parameters can be left out of any statement. This also justifies the notation and use of $\mathbf{XMCTerms} / \xLeftrightarrow{\mathbf{XMonCat}}$. Either the parameters can be inferred from the terms, or if not, they are purely formal and apply to all such terms anyways.

Free Construction

Using the three concepts of a monoidal scheme, a *monoidal term structure* over a monoidal scheme, and the equivalence relation we defined over monoidal term structures, we can take the final step in constructing the free monoidal category generated by a monoidal scheme. We will then prove that what we construct is indeed *free* in **XMonCat**, the category of X-monoidal categories with (strict) X-monoidal functors. What will follow from Theorem 5 is that the free construction over an arbitrary OES induces a left adjoint to the forgetful functor, which maps not only monoidal schemes into X-monoidal categories, but more monoidal scheme functors into X-monoidal functors.

The construction is as follows.

Definition 24. Free X-monoidal category
Given a monoidal scheme Σ, the free X-monoidal category generated by Σ is defined

$$\mathbf{XMonCat}(\Sigma) \overset{def}{=} (\mathcal{O},\, \mathbf{hom},\, \circ\,,\, \mathbf{id},\, \otimes\,,\, \mathbb{1},\, \otimes_{hom}\,,\, \mathbb{1}_{hom},\, \underbrace{\beta}_{\mathbf{SymMonCat}}\,)$$

where

$$\begin{aligned}
&\mathcal{O} = \|\mathbf{Mon}(\mathbb{T}^{\Sigma})\| \\
&\mathbf{hom}(a,\, b) = \mathbf{XMCTerms}^{\Sigma}(a,\, b)/ \xLeftrightarrow{\mathbf{XMonCat}^{\Sigma}(a,\,b)} \\
&\mathbf{id}(a) = \langle \underline{\mathbf{id}_a} \rangle_{\mathbf{XMonCat}} \\
&\circ\ = \langle\, \underline{\circ}\, \rangle_{\mathbf{XMonCat}} \\
&\otimes\ =\ \otimes_{\mathbf{Mon}(\mathbb{T}^{\Sigma})} \\
&\mathbb{1} = \mathbb{1}_{\mathbf{Mon}(\mathbb{T}^{\Sigma})} \\
&\otimes_{hom}\ = \langle\, \underline{\otimes}\, \rangle_{\mathbf{XMonCat}} \\
&\mathbb{1}_{hom} = \mathbf{id}(\mathbb{1}) \\
&[\, \beta(a,\, b) = \langle \underline{\beta_{a,b}} \rangle_{\mathbf{XMonCat}} \quad {}_{\mathbf{SymMonCat}}\,]
\end{aligned}$$

As implied by the notation above, the β elements of the category are only in the symmetric monoidal category variant. We will describe this construction, recapitulating what we have indicated earlier. The set of objects for this X-monoidal category is the underlying set of the free monoid over $\mathbb{T}^{\Sigma}$, and the monoidal product and unit for the X-monoidal category are simply those of the free monoid. The homsets for each pair of objects are the quotient spaces of X-monoidal term structure terms over the corresponding equivalence classes, and the monoidal and categorical operators, as well as the family of trace operators, are liftings of the formal ones through the canonical projections. The identity and braiding constant morphisms are also brought through the projection into their corresponding equivalence classes. As stipulated by the axioms of monoidal categories the unit of monoidal composition over morphisms $\mathbb{1}_{hom}$ is set equal to the identity element for the unit object $\mathbb{1}$.

Although we have called this the free X-monoidal category, we must prove that it is a free object of the category **XMonCat**. What must specifically be shown is that the constructed object $\mathbf{XMonCat}(\Sigma)$ together with a morphism η^{Σ} form a universal morphism from Σ to the forgetful functor $\mathcal{U}_{\mathbf{XMonCat}}$. This can be summarized by the diagram

$$\begin{tikzcd}
\Sigma \arrow[r, "\eta^{\Sigma}"] \arrow[dr, "h = \theta^{\Sigma}(k)"'] & \mathcal{U}_{\mathbf{XMonCat}}(\mathbf{XMonCat}^{\Sigma}) \arrow[d, "\mathcal{U}_{\mathbf{XMonCat}}(k)"] & & \mathbf{XMonCat}^{\Sigma} \arrow[d, "k = \zeta^{\Sigma}(h)"] \\
 & \mathcal{U}_{\mathbf{XMonCat}}(S) & & S
\end{tikzcd}$$

in which ζ^{Σ} is the witness to the universal, which we will define.

Theorem 6.

Given a monoidal scheme Σ*, the triplet* $(\mathbf{XMonCat}(\Sigma),\, \eta^{\Sigma},\, \zeta^{\Sigma})$ *defined*

$$\begin{aligned}
&\eta^{\Sigma} \,:\, \Sigma \to \mathcal{U}_{\mathbf{XMonCat}}(\mathbf{XMonCat}(\Sigma))\\
&[\eta^{\Sigma}]_{\mathbb{T}} = \eta^{\mathbb{T}^{\Sigma}}\\
&[\eta^{\Sigma}]_{\mathcal{A}}(a,\, b)(A) = \langle A\rangle_{\mathbf{XMonCat}(a,\, b)}\\
&\zeta^{\Sigma} \,:\, \mathbf{hom}_{\mathcal{MS}}(\Sigma,\, \mathcal{U}_{\mathbf{XMonCat}}(Y)) \to \mathbf{hom}_{\mathbf{XMonCat}}(\mathbf{XMonCat}(\Sigma),\, Y)\\
&[\zeta^{\Sigma}(h)]_{\mathcal{O}} \overset{def}{=} \zeta^{\mathbb{T}^{\Sigma}}([h]_{\mathbb{T}})\\
&[\zeta^{\Sigma}(h)]_{\mathbf{hom}}(\langle f\rangle_{\mathbf{XMonCat}}) = W_h(f), \textit{ where}\\
&\quad W_h(f \mathbin{\underline{\circ}} g) = W_h(f) \circ W_h(g)\\
&\quad W_h(\underline{\mathbf{id}_x}) = \mathbf{id}(\zeta^{\Sigma}(h)(x))\\
&\quad W_h(f \mathbin{\underline{\otimes}} g) = W_h(f) \otimes W_h(g)\\
&\quad W_h(\underline{\beta_{x,y}}) = \beta(\zeta^{\Sigma}(h)(x),\, \zeta^{\Sigma}(h)(y))\\
&\quad W_h(A) = h(A)
\end{aligned}$$

witnesses a universal morphism.

Before proving this theorem, we will first comment on the construction. As in the case of the free monoid, η^{Σ} simply injects each basis element into an equivalence class. In this case, η^{Σ} is a monoidal scheme functors and has a component for primitive types and primitive blocks. The former is defined in terms of the universal morphism for free monoids defined in Chapter 5, while the latter uses the equivalence class $\langle - \rangle_{\mathbf{XMonCat}}$ over X-monoidal category terms.

The ζ^{Σ} component of the witness is a bijective transformation between particular monoidal scheme functors and X-monoidal functors. Given an monoidal scheme functors, the component of the image that maps between objects of X-monoidal categories is defined as well in terms of the universal morphism for free monoids. This component functions as is described in Section 5.2, mapping and reducing over the structure. The component that maps between morphism in X-monoidal categories is defined inductively in a manner similar to the corresponding definition for free monoids. The action on a morphism in $\mathbf{XMonCat}(\Sigma)$ can be thought of with similar intuitions. ζ^{Σ} takes a morphism and chooses a representative term from $\mathbf{XMCTerms}^{\Sigma}$, then crawls over this term replacing each formal operator with the corresponding operator in codomain X-monoidal categories, and applying the $\mathcal{A}$ component of the monoidal scheme functor to each primitive block at the leaves of the syntactic structure – in essence, a *map-reduce* over X-monoidal category diagrams.

We will now proceed with the proof in a manner similar to its counterpart for free monoids.

Proof. First, it can be seen from the definition of $\mathbf{XMCTerms}^{\Sigma}$ that the function W_h is well-defined via structural induction.

It must be then shown that the image of ζ^Σ is a X-monoidal functor. The preservation of the monoidal unit and product over objects holds as a consequence of $[\zeta^\Sigma(h)]_{\mathcal{O}}$ being defined as a monoid homorophism $\zeta^{\mathbf{Mon}(\mathbb{T})}$. For $[\zeta^\Sigma(h)]_{\mathbf{hom}}$, that the operators and constants over morphisms are all preserved can be seen from the recursive definition of W_h, along with the definition of constants as equivalence classes of the corresponding constant terms and the definition of operators being liftings through the canonical projection.

What remains in proving that the triplet $(\mathbf{XMonCat}(\Sigma),\, \eta^\Sigma,\, \zeta^\Sigma)$ is a universal morphism is that ζ^Σ is the inverse of the canonical θ^Σ, which, as we recall, is defined

$$\theta^\Sigma(k) \overset{\text{def}}{=} \mathcal{U}_{\mathbf{XMonCat}}(k) \circ \eta^\Sigma$$

We first show that $\theta^\Sigma \circ \zeta^\Sigma(h) = h$ for an monoidal scheme functor h. Expanding the definition

$$\theta^\Sigma \circ \zeta^\Sigma(h) = \mathcal{U}_{\mathbf{XMonCat}}(\zeta^\Sigma(h)) \circ \eta^\Sigma$$

This monoidal scheme functor can be applied to both a primitive type $x \in \mathbb{T}^\Sigma$ and to a block $\langle A\rangle_{\mathbf{XMonCat}} \in \mathcal{A}^\Sigma(a,\, b)$. Looking at the former

$$\begin{aligned}
&[\mathcal{U}_{\mathbf{XMonCat}}(\zeta^\Sigma(h)) \circ \eta^\Sigma]_{\mathbb{T}}(x)\\
&= [\mathcal{U}_{\mathbf{XMonCat}}(\zeta^\Sigma(h))]_{\mathbb{T}} \circ [\eta^\Sigma]_{\mathbb{T}}(x)\\
&= [\mathcal{U}_{\mathbf{XMonCat}}(\zeta^\Sigma(h))]_{\mathbb{T}}(\eta^{\mathbf{Mon}}(\mathbb{T}^\Sigma)(x))\\
&= [\zeta^\Sigma(h)]_{\mathcal{O}}(\eta^{\mathbf{Mon}}(\mathbb{T}^\Sigma)(x))\\
&= [\zeta^{\mathbf{Mon}}(\mathbb{T}^\Sigma)(h_{\mathbb{T}}) \circ \eta^{\mathbf{Mon}}(\mathbb{T}^\Sigma)](x)\\
&= h_{\mathbb{T}}(x)
\end{aligned}$$

where the last equality uses the property of the witnesses for free monoids. For the latter

$$\begin{aligned}
&[\mathcal{U}_{\mathbf{XMonCat}}(\zeta^\Sigma(h)) \circ \eta^\Sigma]_{\mathcal{A}}(A)\\
&= [\mathcal{U}_{\mathbf{XMonCat}}(\zeta^\Sigma(h))]_{\mathcal{A}}(\langle A\rangle_{\mathbf{XMonCat}})\\
&= [\zeta^\Sigma(h)]_{\mathbf{hom}}(\langle A\rangle_{\mathbf{XMonCat}})\\
&= W_h(A) = h_{\mathcal{A}}(x)
\end{aligned}$$

Hence, the identity is shown in this direction.

In the other direction, it must be shown that $\zeta^\Sigma \circ \theta^\Sigma(k) = k$ for a X-monoidal functor k. This X-monoidal functor can be applied to both an object $a \in \mathbf{XMonCat}(\Sigma)$ and a morphism $m \in \mathbf{XMonCat}(\Sigma)(a,\, b)$. Considering the former

$$\begin{aligned}
&[\theta^\Sigma(k)]_{\mathbb{T}}\\
&= [\mathcal{U}_{\mathbf{XMonCat}}(k) \circ \eta^\Sigma]_{\mathbb{T}}\\
&= [\mathcal{U}_{\mathbf{XMonCat}}(k)]_{\mathbb{T}} \circ \eta^{\mathbf{Mon}}(\mathbb{T}^\Sigma)\\
&= \mathcal{U}_{\mathbf{Mon}}(k_{\mathcal{O}}) \circ \eta^{\mathbf{Mon}}(\mathbb{T}^\Sigma)\\
&= \theta^{\mathbf{Mon}}(\mathbb{T}^\Sigma)(k_{\mathcal{O}})
\end{aligned}$$

then

$$\begin{aligned}
&[\zeta^{\Sigma} \circ \theta^{\Sigma}(k)]_{\mathcal{O}}(a) \\
&= \zeta^{\mathbf{Mon}}(\mathbb{T}^{\Sigma})([\theta^{\Sigma}(k)]_{\mathbb{T}})(a) \\
&= \zeta^{\mathbf{Mon}}(\mathbb{T}^{\Sigma})(\theta^{\mathbf{Mon}}(\mathbb{T}^{\Sigma})(k_{\mathcal{O}}))(a) \\
&= k_{\mathcal{O}}(a)
\end{aligned}$$

where the last equality, again, uses the property of the witnesses for free monoids. For the latter

$$[\zeta^{\Sigma} \circ \theta^{\Sigma}(k)]_{\mathbf{hom}}(\langle A\rangle_{\mathbf{XMonCat}}) = W_{\theta^{\Sigma}(k)}(A)$$

and we would like to show that

$$W_{\theta^{\Sigma}(k)}(A) = k_{\mathbf{hom}}(\langle A\rangle_{\mathbf{XMonCat}})$$

This can be shown by structural induction over the syntax of the representative A. The base cases are

$$\begin{aligned}
&W_{\theta^{\Sigma}(k)}(\underline{\mathbf{id}_x}) = \mathbf{id}(k_{\mathcal{O}}(x)) \\
&= k_{\mathbf{hom}}(\mathbf{id}(x)) = k_{\mathbf{hom}}(\langle\underline{\mathbf{id}}(x)\rangle_{\mathbf{XMonCat}})) \\
&W_{\theta^{\Sigma}(k)}(\underline{\beta_{x,y}}) = \beta(k_{\mathcal{O}}(x),\, k_{\mathcal{O}}(y)) \\
&= k_{\mathbf{hom}}(\beta(x,\, y)) = k_{\mathbf{hom}}(\underline{\langle\beta_{x,y}\rangle_{\mathbf{XMonCat}}}) \\
&W_{\theta^{\Sigma}(k)}(E) = \theta^{\Sigma}(k)(E) = [\theta^{\Sigma}(k)]_{\mathbf{hom}}(E) \\
&= [\mathcal{U}_{\mathbf{XMonCat}}(k) \circ \eta^{\Sigma}]_{\mathbf{hom}}(E) \\
&= [\mathcal{U}_{\mathbf{XMonCat}}(k)]_{\mathbb{T}}(\langle E\rangle_{\mathbf{XMonCat}}) \\
&= k_{\mathbf{hom}}(\langle E\rangle_{\mathbf{XMonCat}})
\end{aligned}$$

where $E \in \mathcal{A}^{\Sigma}$. The inductive cases all follow from expanding the definition of W on operators and using the preservation of the operators over $k_{\mathbf{hom}}$. For the $\circ$ case,

$$\begin{aligned}
&W_{\theta^{\Sigma}(k)}(f \mathbin{\underline{\circ}} g) = W_{\theta^{\Sigma}(k)}(f) \circ W_{\theta^{\Sigma}(k)}(g) \\
&= k_{\mathbf{hom}}(\langle f\rangle_{\mathbf{XMonCat}}) \circ k_{\mathbf{hom}}(\langle g\rangle_{\mathbf{XMonCat}}) \\
&= k_{\mathbf{hom}}(\langle f\rangle_{\mathbf{XMonCat}} \circ \langle g\rangle_{\mathbf{XMonCat}}) \\
&= k_{\mathbf{hom}}(\langle f \mathbin{\underline{\circ}} g\rangle_{\mathbf{XMonCat}})
\end{aligned}$$

which uses the inductive assumption in the second step. The cases for other operators follow in exactly the same fashion. This establishes the second identity, and thus verifies that the two functions are indeed inverses. □

It follows immediately that $\mathbf{XMonCat}(\Sigma)$ is the free object in $\mathbf{XMonCat}$ corresponding to Σ. Using Theorem 5, $\mathbf{XMonCat}(-)$ can be extended to a free functor that is left adjoint to $\mathcal{U}_{\mathbf{XMonCat}}$. With this adjunction, the unit and counits can also be derived. We state these derivations in a corollary.

Corollary 3.
There is a functor $\mathbf{XMonCat} : \mathcal{MS} \to \mathbf{XMonCat}$ *defined to map each monoidal scheme* Σ *to the corresponding free X-monoidal category and for each monoidal scheme functor* $h : \Sigma \to \Sigma'$

$$\mathbf{XMonCat}(h) \overset{def}{=} \zeta^{\Sigma}(\eta^{\Sigma'} \circ h)$$

such that

$$(\zeta, \theta) : \mathbf{XMonCat} \vdash \mathcal{U}_{\mathbf{XMonCat}}$$

where ζ *and* θ *have components* ζ^{Σ} *and* θ^{Σ}*, respectively.*

The unit and counit for this adjunction are defined

$$\eta_{\Sigma} \overset{def}{=} \eta^{\Sigma}$$
$$\epsilon_Y \overset{def}{=} \zeta^{\mathcal{U}_{\mathbf{XMonCat}}(Y)}(\mathbf{id}(\mathcal{U}_{\mathbf{XMonCat}}(Y)))$$

Proof. This all follows from Theorem 5. □

This set of derivations yield a number of valuable tools for working with monoidal schemes and free X-monoidal categories. As was the case for the free functor generated by free monoids, this free functor $\mathbf{XMonCat}(-)$ acts on a monoidal scheme functor h that performs some transformation – something like an abstraction or embedding, on a collection of generators. It lifts this monoidal scheme functor into a transformation $\mathbf{XMonCat}(h)$ over diagrams that crawls over structure of a diagram in one free X-monoidal category and replaces every primitive block at the leaf of the syntax with another from the original transform. The free functor is, in essence, a variant of the familiar **map** function, applying a transformation over a block diagram rather than a list, transforming each primitive block independently (a block-wise transformation). The structure of the diagram remains unchanged in this transformation, which follows from the fact we established earlier that monoidal scheme functors preserve the valence of primitive blocks, thereby retaining their connections to other blocks.

As is the case with the image of any functor, the image of $\mathbf{XMonCat}$, $\mathbf{XMonCat}(\mathcal{MS})$ gives us a sub-X-monoidal category consisting of only free X-monoidal categories and X-monoidal functors generated from monoidal scheme functors. To foreshadow the role of this in modeling behaviors, the image $\mathbf{XMonCat}(\mathcal{MS})$ provides the various languages that could be used for representing behaviors, each generated by a monoidal scheme carrying the primitives for the language. The process of abstracting or embedding these behavioral languages can be derived from abstracting or embedding their respective sets of primitives.

Similar to the case of the unit derived from free monoids in Chapter 5, the unit η_{Σ} has the typing

$$\eta : \prod_{\Sigma \in \mathcal{MS}} \Sigma \to \mathcal{U}_{\mathbf{XMonCat}} \circ \mathbf{XMonCat}(\Sigma)$$

For any monoidal scheme Σ, the unit is simply the injection of each primitive type a and primitive block A into the corresponding equivalence class $\langle a \rangle$ of single-letter words and $\langle A \rangle$ of single block diagrams, respectively. However, as was discussed in Chapter 5, because the functor $T = \mathcal{U}_{\mathbf{XMonCat}} \circ \mathbf{XMonCat}$ is an endofunctor over $\mathcal{MS}$, iterating it over a monoidal scheme Σ induces a hierarchy

$$\Sigma,\ T(\Sigma),\ T^2(\Sigma),\ T^3(\Sigma),\ T^4(\Sigma),\ \ldots$$

If we think of the first element $T(\Sigma)$ as the space of all diagrams generated from the primitive blocks in Σ, then $T^2(\Sigma)$ is the space of all diagrams generate from these diagrams; that is, the space of 2-level diagrams. $T^N(\Sigma)$ is thus the space of N-level diagrams, built up hierarchically.

This hierarchical structure is possible, in part, because each diagram in a monoidal category has an interface, just as each primitive block does. Consequently, one can abstract each diagram into a primitive block, which can then be instantiated in the construction of a 2-level diagram. This process of lifting an N-level diagram into a $N+1$-level diagram is precisely what the unit does over this hierarchy.

$$\Sigma \xrightarrow{\eta} T(\Sigma) \xrightarrow{\eta} T^2(\Sigma) \xrightarrow{\eta} T^3(\Sigma) \xrightarrow{\eta} T^4(\Sigma) \xrightarrow{\eta} \ldots$$

This mechanism allows free X-monoidal category to represent structures that have a non-trivial sense of hierarchy. As we had shown in the case of free monoids, which have the same kind of hierarchy, there is a distinction between a N-level diagram A and its image through the unit as a $N+1$-level diagram $\langle A \rangle$. That is,

$$A \neq \langle A \rangle \neq \langle\langle A \rangle\rangle \neq \langle\langle\langle A \rangle\rangle\rangle \neq \langle\langle\langle\langle A \rangle\rangle\rangle\rangle \neq \ldots$$

which makes it possible to draw hierarchical distinctions.

And yet this hierarchy can nevertheless be collapsed down to a 1-level diagram by way of the counit, which as we recall has the typing

$$\epsilon : \prod_{Y \in \mathbf{XMonCat}} \mathbf{XMonCat} \circ \mathcal{U}_{\mathbf{XMonCat}}(Y) \to Y$$

In contrast with the unit, the counit is parameterized over X-monoidal categories, and has as its domain the image of the functor $R = \mathbf{XMonCat} \circ \mathcal{U}_{\mathbf{XMonCat}}$, which is an endofunctor over X-monoidal categories. R therefore also induces a hierarchy, but one that starts with an arbitrary X-monoidal category Y rather than a monoidal scheme.

$$Y,\ R(Y),\ R^2(Y),\ R^3(Y),\ R^4(Y),\ \ldots$$

This Y can be any X-monoidal category, and not just a free X-monoidal category. However, if it is a free X-monoidal category, and thus $Y = \mathbf{XMonCat}(\Sigma)$, then this hierarchy simply the hierarchy we discussed earlier with an extra layer of free construction applied to each element in the series.

$$\mathbf{XMonCat}(\Sigma),\ \mathbf{XMonCat}\ \circ\ T(\Sigma),\ \mathbf{XMonCat}\ \circ\ T^2(\Sigma),\ \ldots$$

The Nth element here is the free X-monoidal category of $N+1$-level diagrams, whereas before each element was the underlying monoidal scheme of these structures (shifted over by one).

Although this is roughly the same hierarchy as before, with a phase shift to the free X-monoidal category, in contrast with the unit, the counit provides a means to take each $N+1$-level diagram and collapse it into a N-level diagram.

$$\mathbf{XMonCat}(\Sigma) \xleftarrow{\epsilon} \mathbf{XMonCat}\ \circ\ T(\Sigma) \xleftarrow{\epsilon} \mathbf{XMonCat}\ \circ\ T^2(\Sigma) \xleftarrow{\epsilon} \ldots$$

As in the case of the free monoid, this counit operation is, in essence, the **reduce** function, which simply takes the compositions of N-level diagrams, as $N+1$-level diagrams, and interprets them as constructing N-level diagrams. This fits the intuition that if one wanted to explicitly abolish the hierarchy of diagrams this could be done, although it is not done automatically, allowing the hierarchy to be retained when it serves a semantic purpose and collapsed in a well-defined way when the aim is to do so. The unit and counit therefore provide an well-defined mathematical means to construct hierarchy as well as a means to remove it.

6.3 Properties of Monoidal Schemes and Free Monoidal Categories

We will now look at some important properties of monoidal schemes and free monoidal categories. Several important category theoretical properties can be identified in the category of monoidal schemes $\mathcal{MS}$, which can be used to reason about the various sets of primitive elements that will generate monoidal categories. We will show that some of these properties will be carried over through the free functor to the image in the category of free monoidal categories, and characterize the relationships of abstraction or embedding between these.

The first of these significant properties is that $\mathcal{MS}$ has an initial element $\mathbf{init}^{\mathcal{MS}}$, which is simply the empty monoidal scheme.

Proposition 6.
Let the monoidal scheme $\mathbf{init}^{\mathcal{MS}}$ *be defined*

$$\mathbf{init}^{\mathcal{MS}} \overset{def}{=} (\emptyset,\ \mathcal{A}_\emptyset)$$
$$\textit{where}\ \mathcal{A}_\emptyset(\mathbb{1},\ \mathbb{1}) = \emptyset$$

and an monoidal scheme functor-valued function of monoidal schemes $init^{\mathcal{MS}}$ be defined

$$\begin{aligned}
&init^{\mathcal{MS}} : \prod_{\Sigma \in \mathcal{MS}} \mathbf{init}^{\mathcal{MS}} \to \Sigma \\
&[init^{\mathcal{MS}}_{\Sigma}]_{\mathbb{T}} = \emptyset_{\mathbb{T}^{\Sigma}} \\
&[init^{\mathcal{MS}}_{\Sigma}]_{\mathcal{A}}(\mathbb{1}, \mathbb{1}) = \emptyset_{\mathcal{A}^{\Sigma}(\mathbb{1},\, \mathbb{1})}
\end{aligned}$$

$\mathbf{init}^{\mathcal{MS}}$ *is the initial element of $\mathcal{MS}$ with $init^{\mathcal{MS}}$ as its witness.*

Proof. First we will verify that the above functions are well defined. Since $\mathbf{Mon}(\emptyset) = \{\mathbb{1}\}$ the function $\mathcal{A}_{\emptyset}$ is well defined above, as is $[init^{\mathcal{MS}}_{\Sigma}]_{\mathcal{A}}$. Moreover, all of the $\mathcal{A}_{\emptyset}$ sets are disjoint (since they are all empty). $\mathbf{init}^{\mathcal{MS}}$ is therefore a well defined monoidal scheme and $init^{\mathcal{MS}}_{\Sigma}$ a well defined monoidal scheme functor for any monoidal scheme Σ. $[init^{\mathcal{MS}}_{\Sigma}]_{\mathbb{T}}$ is clearly the unique function of the type $\emptyset \to \mathbb{T}^{\Sigma}$. For functions of the type

$$\prod_{x \in \mathcal{I}^{\mathbf{init}^{\mathcal{MS}}}} \mathcal{A}^{\mathbf{init}^{\mathcal{MS}}}(x) \to \mathcal{A}^{\Sigma}([init^{\mathcal{MS}}_{\Sigma}]_{\mathbb{T}}(x))$$

the unique choice of $[init^{\mathcal{MS}}_{\Sigma}]_{\mathbb{T}}$ reduces the type more specifically to

$$\mathcal{A}^{\mathbf{init}^{\mathcal{MS}}}(\mathbb{1},\, \mathbb{1}) \to \mathcal{A}^{\Sigma}(\mathbb{1},\, \mathbb{1}) \cong \emptyset \to \mathcal{A}^{\Sigma}(\mathbb{1},\, \mathbb{1})$$

for which the above definition of $[init^{\mathcal{MS}}_{\Sigma}]_{\mathcal{A}}$ is the unique witness. It follows that $init^{\mathcal{MS}}_{\Sigma}$ is unique, in addition to existing, for all Σ. □

This initial element will not, by itself, serve a very significant purpose in behavioral representations, since it generates trivial monoidal categories with only **id** channels and no blocks, which are all essentially empty. However, the initial element will have a technical significance in characterizing the properties of the category.

A more inherently significant distinguished element of $\mathcal{MS}$ is its terminal element $\mathbf{term}^{\mathcal{MS}}$, which has a single type $\star$ and a single block $\Box^{a}_{b}$ for every pair of ordinals a and b.

Proposition 7.
Let the monoidal scheme $\mathbf{term}^{\mathcal{MS}}$ be defined

$$\begin{aligned}
&\mathbf{term}^{\mathcal{MS}} \overset{def}{=} (\{\star\},\, \mathcal{A}_{\star}) \\
&\quad \textit{where } \mathcal{A}_{\star}(a,\, b) = \{\Box^{a}_{b}\}
\end{aligned}$$

where $\Box^{a}_{b}$ are a set of unique constants for each a and b, and an monoidal scheme functor-valued function of monoidal schemes $term^{\mathcal{MS}}$ be defined

$$\begin{aligned}
&term^{\mathcal{MS}} : \prod_{\Sigma \in \mathcal{MS}} \Sigma \to \mathbf{term}^{\mathcal{MS}} \\
&[term^{\mathcal{MS}}_{\Sigma}]_{\mathbb{T}} = \mathbf{Const}_{\mathbb{T}^{\Sigma}}(\star) \\
&[term^{\mathcal{MS}}_{\Sigma}]_{\mathcal{A}}(a,\, b) = \mathbf{Const}_{\mathcal{A}^{\Sigma}(a,\, b)}(\Box^{\mathbf{Ord}(a)}_{\mathbf{Ord}(b)})
\end{aligned}$$

$\mathbf{term}^{\mathcal{MS}}$ *is the terminal element of $\mathcal{MS}$ with $term^{\mathcal{MS}}$ as its witness.*

The proof of this is similar to that of the initial element.

Proof. It must be first stated that $\mathbf{Mon}(\{\star\})$ is isomorphic to $\mathbb{N}$, and thus the set $\mathbf{Mon}(\mathbb{T}^\Sigma)$ for $\mathbf{term}^{\mathcal{MS}}$ can be taken to be the ordinals in $\mathbb{N}$ (later this will be expanded into all countable ordinals). Consequently, the lifting of $\mathbf{Const}_{\mathbb{T}^\Sigma}(\star)$ into the free monoid is indeed $\mathbf{Ord}$. As in the proof of the initial element, the type $\mathbb{T}^\Sigma \to \{\star\}$ has the above definition for $[term^{\mathcal{MS}}_\Sigma]_\mathbb{T}$ as its unique element. And, the type

$$\prod_{x \in \mathcal{I}^\Sigma} \mathcal{A}^\Sigma(x) \to \mathcal{A}^{\mathbf{term}^{\mathcal{MS}}}([term^{\mathcal{MS}}_\Sigma]_\mathbb{T}(x))$$

with the unique transformation over types reduces to

$$\begin{aligned}&\prod_{(a,\,b) \in \mathcal{I}^\Sigma} \mathcal{A}^\Sigma(a,\,b) \to \mathcal{A}^{\mathbf{term}^{\mathcal{MS}}}(\mathbf{Ord}(a),\,\mathbf{Ord}(b))\\ \cong &\prod_{(a,\,b) \in \mathcal{I}^\Sigma} \mathcal{A}^\Sigma(a,\,b) \to \{\Box^a_b\}\end{aligned}$$

for which the above definition of $[term^{\mathcal{MS}}_\Sigma]_\mathcal{A}$ is the unique witness. It follows that $term^{\mathcal{MS}}_\Sigma$ is unique, in addition to existing, for all Σ. □

This terminal element is a purely structural generator for a monoidal category. In the graphical language, the channels are all untyped, and there is a single primitive block for every number of incoming and outgoing channels. This is the most abstracted representation of an monoidal scheme that maintains what it encodes topologically. The utility of this terminal monoidal scheme is that it generates a completely abstracted monoidal category that retains only topological information. In a sense, the transformations into this monoidal scheme (and as we will see later, the corresponding transformations between monoidal categories) forgets all labellings of blocks and types, serving as a kind of erasure.

This terminal element, and the mapping into it, can be generalized to contain only some structural blocks $\Box^a_b$ corresponding to the particular shapes present in some monoidal scheme Σ.

Definition 25.
Given a monoidal scheme Σ, the structural abstraction $\mathbf{str}^{\mathcal{MS}}(\Sigma)$ *is defined*

$$\begin{aligned}\mathbf{str}^{\mathcal{MS}}(\Sigma) &\overset{def}{=} (\{\star\},\,\mathcal{A}_\star)\\ \mathcal{A}_\star(n,\,m) &\overset{def}{=} \begin{cases}\Box^n_m & \{\exists\, x : \mathcal{I}^\Sigma;\ A : \mathcal{A}^\Sigma(x)\ \cdot\ \mathbf{Ord}(x) = (n,\,m)\}\\ \emptyset & \textit{otherwise}\end{cases}\end{aligned}$$

and there is a canonical projection into this monoidal scheme

$$\begin{aligned}
&str^{\mathcal{MS}} : \prod_{\Sigma \in \mathcal{MS}} \Sigma \to \mathbf{str}^{\mathcal{MS}}(\Sigma) \\
&[str^{\mathcal{MS}}_{\Sigma}]_{\mathbb{T}} = \mathbf{Const}_{\mathbb{T}^{\Sigma}}(\star) \\
&[str^{\mathcal{MS}}_{\Sigma}]_{\mathcal{A}}(a,\, b) = \mathbf{Const}_{\mathcal{A}^{\Sigma}(a,\, b)}(\Box^{\mathbf{Ord}(a)}_{\mathbf{Ord}(b)})
\end{aligned}$$

The definition of $str^{\mathcal{MS}}_{\Sigma}$ is essentially the same as $\mathbf{term}^{\mathcal{MS}}_{\Sigma}$, but restricted to $\mathbf{str}^{\mathcal{MS}}(\Sigma)$

The $\mathcal{MS}$ category also has mechanisms for combining monoidal schemes. Specifically, we will show that $\mathcal{MS}$ has all arbitrary coproducts.

Theorem 7.
Given an indexed set of monoidal schemes $(\Sigma_j \mid j \in J)$ for an indexing set J, let

$$\begin{aligned}
&\mathbb{T}_{\coprod} = \coprod_{j \in J} \mathbb{T}^{\Sigma_j} \\
&\kappa_j : \mathbb{T}^{\Sigma_j} \to \mathbb{T}_{\coprod} \\
&\overline{\kappa_j} = \Delta \circ \mathbf{Mon}(\kappa_j)
\end{aligned}$$

where we are using the standard coproduct over sets and κ_j are the corresponding injections on sets. $\overline{\kappa_j}$ is the lifting of these injections over the free monoid and pairs.

Let the coproduct of monoidal schemes be defined

$$\begin{aligned}
&\coprod_{j \in J} \Sigma_j \overset{def}{=} (\mathbb{T}_{\coprod},\, \mathcal{A}_{\coprod}) \\
&\textit{where} \quad \mathcal{A}_{\coprod}(x) = \begin{cases} \mathcal{A}^{\Sigma_n}(y) & \textit{if } y = \overline{\kappa_n}(x) \wedge n \in J \\ \emptyset & \textit{otherwise} \end{cases}
\end{aligned}$$

where the standard set coproduct and injections are used over the sets $\mathbb{T}^{\Sigma_j}$. Let the injections ι_j into the coproduct be defined

$$\begin{aligned}
&\iota_j : \Sigma_j \to \coprod_{j \in J} \Sigma_j \\
&[\iota_j]_{\mathbb{T}} \overset{def}{=} \iota_j \\
&[\iota_j]_{\mathcal{A}}(x) \overset{def}{=} \mathbf{id}_{\mathbf{Fun}}(\mathcal{A}^{\Sigma_j}(x))
\end{aligned}$$

Given additionally a monoidal scheme Σ and a indexed set of monoidal scheme functors into Σ

$$(F_j : \Sigma_j \to \Sigma \mid j \in J)$$

let the coproduct over morphisms (the witness) be defined

$$\coprod_{j\in J}\Sigma_j : \coprod_{j\in J}\Sigma_j \to \Sigma$$

$$[\coprod_{j\in J}F_j]_{\mathbb{T}} \overset{def}{=} \coprod_{j\in J}[F_j]_{\mathbb{T}}$$

$$[\coprod_{j\in J}F_j]_{\mathcal{A}}(x) \overset{def}{=} \begin{cases} [F_n]_{\mathcal{A}}(y), & \textit{if } y = \overline{\kappa_n}(x) \wedge n \in J \\ \emptyset_{\mathcal{A}^{\Sigma}(x)} & \textit{otherwise} \end{cases}$$

The above definition gives a well defined coproduct.

Before proving this proposition, it is worth describing the coproduct construction intuitively. Like a standard coproduct over sets, this coproduct combines a collection of monoidal schemes disjointly to form an monoidal schemes with all of the primitive types and blocks combined. The combination of primitive types $\mathbb{T}^{\Sigma_j}$ is just the standard coproduct over sets; a disjoint union that retains all of the elements distinguishably. The types in the $\mathbf{Mon}(\mathbb{T})$ component of the coproduct, however, may now be constructed from words of primitive types from any of the $\mathbb{T}^{\Sigma_j}$ sets. Consequently, the space of types in the coproduct is strictly larger than the disjoint union of those in the component monoidal schemes, since the coproduct now has mixed types.

Another way of describing this is as follows. The injections of primitive types from $\mathbb{T}^{\Sigma_j}$ is lifted into an injection of types from $\mathbf{Mon}(\mathbb{T})^{\Sigma_j}$ into the set of types in the coproduct. The union of images of all of these injections, however, constitutes only the "diagonal" elements of the coproduct types. For general cases (non-empty)

$$\bigcup_{j\in J}[\iota_j]_{\mathbb{T}}(\mathbf{Mon}(\mathbb{T})^{\Sigma_j}) \subset \mathbf{Mon}(\mathbb{T})^{\coprod_{j\in J}\Sigma_j}$$

This is a generalization of the more obvious fact that for non-empty sets

$$\mathbf{Mon}(A) \cup \mathbf{Mon}(B) \subset \mathbf{Mon}(A \cup B)$$

In the above definitions, therefore, some cases are predicated on an element $a \in \mathbf{Mon}(\mathbb{T})$ being in the image of some lifted ι_j, i.e. $a = \iota_j(c)$. This equivalently means that all of the components of a are from the same $\mathbb{T}^{\Sigma_j}$. We can call these *homogeneous* types for the moment. A consequence of this larger space of types in the coproduct is that there are $\mathcal{A}$ sets parameterized by pairs of elements in this larger space. In out definition, the $\mathcal{A}$ sets parameterized by homogeneous types, both from the same $\mathbf{Mon}(\mathbb{T})^{\Sigma_j}$, are set equal to the set from $\mathcal{A}^{\Sigma_j}$, while the rest of the $\mathcal{A}$ sets, parameterized by heterogeneous elements, are all defined to be empty.

In order to prove that this definition is correct, the witness (the coproduct of morphisms) must be verified to fulfill the universal property of coproducts, which can be stated as the natural isomorphism

$$\prod_{j\in J}\mathbf{hom}(\Sigma_j,\,\Sigma)\cong\mathbf{hom}(\coprod_{j\in J}\Sigma_j,\,\Sigma)$$

The mapping from left to right, given by the witness, can be denoted

$$\zeta_{\coprod}(f_j\mid j\in J)\overset{\text{def}}{=}\coprod_{j\in J}f_j$$

to fit the isomorphism. The mapping from right to left in this isomorphism is given by the function

$$\theta_{\coprod}(f)\overset{\text{def}}{=}(f\circ\iota_j\mid j\in J)$$

Proving that the coproduct fulfills this universal property amounts to showing that $\zeta_{\coprod}$ is the inverse of $\theta_{\coprod}$.

Proof. We will begin with showing that $\zeta_{\coprod}\circ\theta_{\coprod}$ is the identity. To do this we will look at its action on an element $h\in\mathbf{hom}(\coprod_{j\in J}\Sigma_j,\,\Sigma)$ Expanding the definition

$$\zeta_{\coprod}\circ\theta_{\coprod}(h)=\zeta_{\coprod}((h\circ\iota_j\mid j\in J))=\coprod_{j\in J}h\circ\iota_j$$

We must consider this monoidal scheme functor over its type and block components. For types

$$[\coprod_{j\in J}h\circ\iota_j]_{\mathbb{T}}=\coprod_{j\in J}[h\circ\iota_j]_{\mathbb{T}}=\coprod_{j\in J}[h]_{\mathbb{T}}\circ[\iota_j]_{\mathbb{T}}=\coprod_{j\in J}[h]_{\mathbb{T}}\circ\kappa_j=[h]_{\mathbb{T}}$$

For blocks, we first consider an interface x where $x=\overline{\kappa_n}(y)$ for some values n and y.

$$\begin{aligned}&[\coprod_{j\in J}h\circ\iota_j]_{\mathcal{A}}(x)=[h\circ\iota_j]_{\mathcal{A}}(y)\\&=[h]_{\mathcal{A}}(x)\circ[\iota_n]_{\mathcal{A}}(y)\\&=[h]_{\mathcal{A}}(x)\circ\mathbf{id}_{\mathbf{Fun}}(\mathcal{A}^{\Sigma_n})\\&=[h]_{\mathcal{A}}(x)\end{aligned}$$

For the case that no such y and n exist,

$$[\coprod_{j\in J}h\circ\iota_j]_{\mathcal{A}}(x)=\emptyset_{\mathcal{A}^{\Sigma}(x)}$$

which must be the case for any h since $\mathbb{T}_{\coprod}(x)=\emptyset$

In order to show the identity of the other direction $\theta_{\coprod} \circ \zeta_{\coprod}$, its action on an element $k : \coprod_{j\in J}(\Sigma_j \to \Sigma)$. Expanding the definition

$$\theta_{\coprod} \circ \zeta_{\coprod}(k) = \theta_{\coprod}(\coprod_{j\in J} k_j) = \left((\coprod_{j\in J} k_j) \circ \iota_n \mid n \in J \right)$$

Expanding the type and block components of each member of this tuple

$$[(\coprod_{j\in J} k_j) \circ \iota_n]_{\mathbb{T}} = [\coprod_{j\in J} k_j]_{\mathbb{T}} \circ \kappa_n = [\coprod_{j\in J} [k_j]_{\mathbb{T}}] \circ \kappa_n = [k_n]_{\mathbb{T}}$$

where the last step involved using the properties of coproducts over sets. For any x

$$\begin{aligned}
&[(\coprod_{j\in J} k_j) \circ \iota_n]_{\mathcal{A}}(x) \\
&= [\coprod_{j\in J} k_j]_{\mathcal{A}}(\Delta \circ \mathbf{Mon}([\iota_n]_{\mathbb{T}})(x)) \circ \kappa_n \\
&= [\coprod_{j\in J} k_j]_{\mathcal{A}}(\Delta \circ \mathbf{Mon}([\iota_n]_{\mathbb{T}})(x)) \circ \mathbf{id}_{\mathbf{Fun}}(\mathcal{A}^{\Sigma_n}) \\
&= [\coprod_{j\in J} k_j]_{\mathcal{A}}(\overline{\kappa_n}(x)) \circ \mathbf{id}_{\mathbf{Fun}}(\mathcal{A}^{\Sigma_n}) \\
&= [\coprod_{j\in J} k_j]_{\mathcal{A}}(\overline{\kappa_n}(x)) \\
&= [k_n]_{\mathcal{A}}(x)
\end{aligned}$$

The two functions $\zeta_{\coprod}$ and $\theta_{\coprod}$ are therfore inverses, and thus $\mathcal{MS}$ has all small coproducts. $\square$

We may be able to define products over monoidal schemes as well, but that is not of primary interest here. A more important construction is that of coequalizers on monoidal schemes. Coequalizers are the categorical generalization of concepts such as equivalence relations that conflate elements of objects together (when the objects are set-like). In the context of monoidal schemes, defining equivalence relations and coequalizers provides a means to conflate both types and blocks together in a consistent manner. In other words, two equivalences, one over primitive types $\approx_{\mathbb{T}}$ and one over blocks $\approx_{\mathcal{A}}$ can be defined so long as they fulfill a consistency constraint.

This consistency constraint on these equivalence relations can be understood in terms of the consequence of conflating only primitive types or only blocks, while leaving the other collection distinct. There is no barrier to conflating any set of types in a monoidal schemes, since they constitute a simple set, and thus $\approx_{\mathbb{T}}$ could be any equivalence relation. However, the ramification of doing so is that there is an induced equivalence relation $\approx_{\mathbf{Mon}(\mathbb{T})}$ on types, in which two types are equivalent if each primitive component is pointwise equivalent under the relation. That is, for any two types expressed in canonical form

$$a_1 \otimes \ldots \otimes a_N \approx_{\mathbf{Mon}(\mathbb{T})} b_1 \otimes \ldots \otimes b_M \overset{\text{def}}{=} N = M \wedge \forall_{k \leq N} a_k \approx_{\mathbb{T}} b_k$$

It follows immediately from this definition that ordinal length of types is always preserved under conflations, and ultimately, the coarsest possible partitioning is that generated by the mapping of all types to $\star$, as is done in the terminal mapping. This partitioning by ordinal length is, in a sense, the top in the partial order of these generated type conflations. It follows that members of $\mathcal{A}$ will then be conflated if their parameter types are equivalent under $\approx_{\mathbb{T}}$. Because these sets are all disjoint, conflated sets can simply be united without conflating any of the blocks within them.

If only blocks are conflated by a relation $\approx_{\mathcal{A}}$, while types remain distinct, it would not make sense to conflate any blocks unless they have the same interface. Moreover, because the sets in $\mathcal{A}$ are all disjoint, each set in $\mathcal{A}$ could be partitioned by an equivalence relation independently from any other. That is, we would necessarily get a disjoint union of relations $\approx_{\mathcal{A}(a,\, b)}$ for each $a,\, b \in \mathbf{Mon}(\mathbb{T})$. We can then see what it means for the two equivalence relations to be consistent, and that is that we can only conflate blocks if they end up having the same interface after the types have themselves been conflated.

We can now state a definition for an equivalence relation over monoidal schemes.

Definition 26. *Monoidal Scheme Equivalence Relations*
Given any monoidal schemes Σ, a monoidal schemes equivalence relation over Σ is defined as a pair of relations

$$E_\Sigma \overset{def}{=} (\approx_{\mathbb{T}},\, \approx_{\mathcal{A}})$$

where

$$\approx_{\mathbb{T}} \subseteq \mathbf{EqRel}(\mathbb{T}^\Sigma)$$
$$\approx_{\mathcal{A}} \subseteq \mathbf{EqRel}(\mathcal{A}^\Sigma_\cup)$$

subject to the constraint

$$A \approx_{\mathcal{A}} B \Rightarrow \tau_{+/-}(A) \approx_{\mathbb{T}} \tau_{+/-}(B)$$

The space of these relations is denotes $\mathbf{EqRel}_\Sigma$.

It is clear that this definition gives an equivalence relation over both types and blocks allowing both to be simultaneously partitioned. Consequently, a quotient monoidal scheme can be induced along with a partitioning monoidal scheme functor.

Definition 27. *Given a monoidal schemes Σ and a monoidal schemes equivalence relation E_Σ over Σ, the quotient monoidal scheme is defined*

$$\Sigma / E_\Sigma \overset{def}{=} (\mathbb{T}^\Sigma / \cong_{\mathbb{T}},\, \mathcal{A}^{E_\Sigma})$$

where for each $p,\, q \in \mathbb{T}^{\Sigma}/\cong_{\mathbb{T}}$

$$\mathcal{A}^{E_{\Sigma}}(p,\, q) \overset{def}{=} \left[\bigcup_{x \in p,\, y \in q} \mathcal{A}^{\Sigma}(x,\, y) \right] / \cong_{\mathcal{A}}$$

The canonical projections $\langle - \rangle_{\mathbb{T}}$ and $\langle - \rangle_{\mathcal{A}}$ map each primitive type and primitive block into their respective equivalence classes in $\mathbb{T}^{\Sigma}/\cong_{\mathbb{T}}$ and $\mathcal{A}^{E_{\Sigma}}$ respectively.

In this quotient construction, the set of primitive types of the quotient is just the standard quotient of the primitive types over the equivalence relation $\cong_{\mathbb{T}}$, which is just an ordinary equivalence relation. For the primitive blocks, we must define a set for each pair $(p,\, q)$ of words in $\mathbf{Mon}\mathbb{T}^{\Sigma}/\cong_{\mathbb{T}}$. In the definition, the sets in $\mathcal{A}^{\Sigma}$ for every pair $(x,\, y)$ of words in $\mathbf{Mon}(\mathbb{T})$ are combined together, then repartitioned by $\cong_{\mathcal{A}}$.

With this definition of quotients over monoidal schemes, it follows that the category of monoidal schemes has coequalizers, which can be defined in terms of quotients.

Proposition 8. *$\mathcal{MS}$ has all coequalizers.*

Proof. This follows from the fact that $\mathcal{MS}$ has quotients over equivalence relations. Coequalizers can be constructed using the above definition of quotients, by the standard method of constructing an equivalence relation from the images of any two monoidal scheme functors. □

Given that both arbitrary coproducts and coequalizers exist over $\mathcal{MS}$, it follows from Corollary V.2 in [39] that all small colimits exist in $\mathcal{MS}$.

Theorem 8. *$\mathcal{MS}$ colimits*
$\mathcal{MS}$ is co-complete.

Proof. It has been proven that $\mathcal{MS}$ has both arbitrary coproducts and coequalizers. Using the dual of Corollary V.2 [39], it follows that $\mathcal{MS}$ has all small colimits. □

Having established that all colimits exists, the co-Adjoint Functor Theorem can then be applied to the free functor $\mathbf{XMonCat}(-)$.

Corollary 4. $\mathbf{XMonCat}(-)$ *preserves all small colimits.*

Proof. This follows immediately from Corollary 2. □

The practical relevance of this property is that combinations of monoidal schemes, which can be structured as colimits, are carried through the free construction. Therefore combining the free X-monoidal categories can be achieved simply by combining the corresponding generators in the same fashion. One need only to combine the primitive types and blocks to get the combinations of languages of diagrams constructed from them.

Chapter 7

Ontological Event Graphs

We are now prepared to introduce our proposal for a behavioral representation fitting for the demands of HDDAs running on a Process Field platforms such as the IoT or the Swarm. This new representation, called an OEG, will address the particular concerns of composability, modularity, hierarchy, and comprehensiveness discussed in Chapter 2. It will be argued that an OEG is capable of denoting, as a mathematically well-constituted object, what happens when an application is executing on such a platform, consisting of all of the details necessary to identify the semantics – the precise meaning – of an application. It will also be seen how OEGs, in contrast with the other kinds of behavioral representations discussed in Chapter 4, improves upon the shortcomings of these representations for the purposes of this application domain.

We use the term *ontological* in the name of this representation to indicate that these graphs attempt to capture, not a particular perspective of the behavior of a concurrent system, such as the account of a sequential observer, discussed in [1] in the context of Mazurkiewicz Traces, but instead an underlying, perspectively invariant account of what happened in the concurrent system. That is, this representation aims at the ontological characterization, as opposed to the epistemological characterization, of distributed behavior. This is important because it can give the semantics of a concurrent process directly, without having to find it in the intersection of all perspectives. Instead, possible perspectives, epistemological accounts, may be derived from an OEG, while the OEG itself identifies the ground truth of the behavior.

Being *ontological*, in this sense, OEGs constitute a form a of *true concurrency*, like some of the other representations we have reviewed, such as Event Structures. However, OEGs are also modular, composable, and form a simple yet comprehensive algebra that can be used to reason with them. Having also developed, in Chapters 5 and 6, the tools to work with monoidal categories and specifically free monoidal categories, we will show that collections of OEGs form symmetric monoidal categories, and use the result of Joyal and Street in [27] to show that they are, more specifically, free symmetric monoidal categories. Consequently, the category theoretical tools for working with free symmetric monoidal categories can be used to reason about OEGs. In this sense, OEGs are an instance of a monoidal category representation of concurrent behavior in contrast with those instances of *generalized sequences*, as we discussed in Chapter 3. Thus, they have the compositional advantages of this approach absent in generalized sequences such as Event Structures, as we elaborated on in Chapter 4.

In short, as we have described them earlier, an OEG is essentially a kind of ported acyclic block diagram. Each block in the diagram has a sequence of input ports and a sequence of output ports, and the connections between blocks run from the output port of one block to the input port of another – in a manner that does not result in a cycle. What the blocks represent are behavioral events that occur in the execution of a concurrent process. The connections then represent an element of information or influence passed between events, produced as an effect of one and received as a dependency of another. These elements will be called *dependencies*.

In order to define a language of OEGs, we will first define the alphabet of this language, which we will call an OES. It will consist of the primitive event types and *dependency types* that will be used to build OEGs. Mathematically, these will turn out to be the monoidal scheme we defined in Chapter 6. We will discuss these event types and dependency types both conceptually and formally. In particular, OESs themselves form a category, as monoidal schemes, and can be related to one another as a means for abstraction, inclusion, combination, and so forth. We will give some practical examples of such languages that can be used to build OEGs.

Using the concept of an OES, we will then define OEGs formally, and define algebraic compositions over them, along with some important families of constant elements. It will be shown that with these compositions and constants, the OEGs built from a particular OES form a symmetric monoidal category. Drawing from the work of Joyal and Street in [27], we will then show that these symmetric monoidal categories of OEGs are more specifically isomorphic to the free symmetric monoidal categories that we defined in Chapter 6. We will show that a consequence of this is that any OEG can be constructed as a term in the language of symmetric monoidal categories constituted only of constants and atomic OEGs, containing single events.

7.1 An Event Ontology

The ontology for our representation can be built up from two basic concepts: *event types* and a *dependency types*. We will describe these concepts in both intuitive and graphical terms.

An *event type* is the atomic element of our representation of behavior. It characterizes a type of behavioral event in the execution of a concurrent process. An instance of this type, an *event*, happens once in the behavior, presumably at some time and in some place (to some observer, perhaps relativistic), or even some abstract region of both space and time. An event, in a concurrent system on a platform such as a Process Field, could be, for instance:

- the reception or sending of a message

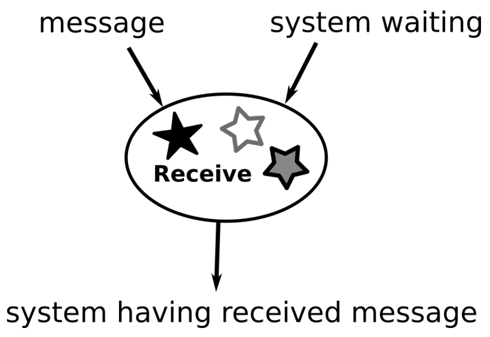


Figure 7.1: A message reception event.

- a change in the state of a component
- an execution of a procedure on a set of data
- a forking or joining of control flow
- an abstraction of a complex collection of events

In an event, there is a transformation that occurs to information and influence involved in its occurrence. Some data, a state, a message, control, a component, a physical entity, etc... contribute to a particular event, either as its cause or something it otherwise uses or depends on. We call these contributing pieces of information generically *dependencies*, and their types *dependency types*. An event not only depends on these dependencies, but also produces dependencies as a consequence; as its effect. These dependencies can then contribute to subsequent events, at later times and potentially in other places.

As an example of an event, let us consider the case of a message being received in a sequential process. The contributing, or *incoming*, dependencies are the message itself and the current state of the system receiving it – in this state it is waiting for the message. The reception occurs in the event, and as a consequence the single *outgoing* dependency is the new state of the system, having received the message. Here we have not specified whether it is a particular message and specific state of the system, or more abstracted forms of either, and indeed, both of these are possible. Our events and dependency will be as abstract or concrete as they need to be for a particular circumstance. We have rendered this event informally in Figure 7.1, however we will introduce a more refined diagrammatic language for representing these events that will be later complemented by their mathematical counterparts.

In general, an event type will have an ordered sequence of incoming dependency types and an ordered sequence of outgoing dependency types. This is depicted in Figure 7.2a in which an event type A is graphically represented as a block with incoming dependency types $i_1, \ldots, i_N$ and outgoing dependency types $o_1, \ldots, o_M$. Order is emphasized here since we will use the order to identify each of the dependencies of an event type (as opposed to naming them). The structure of surrounding dependency types of an event type constitutes its event interface, as is shown in Figure 7.2b with the event interface of A denoted $\tau(A)$.

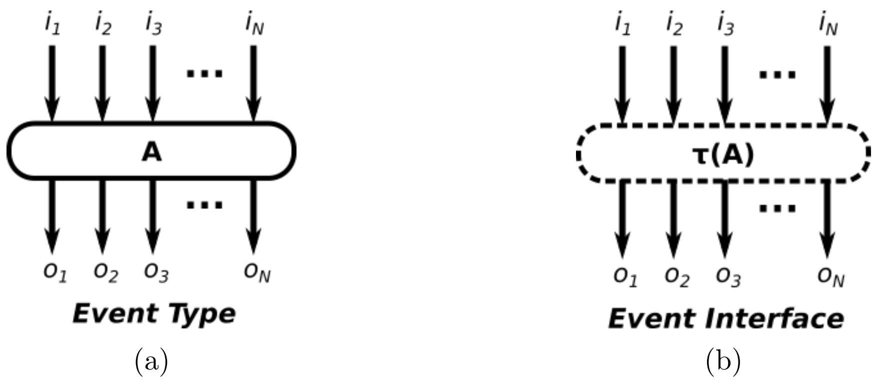


Figure 7.2: An event type and an event interface.

To clarify the difference between these two, multiple event types can have the same event interface. Another way of looking at this is that the event type itself can be thought of as an event interface further distinguished by additional identifying information.

The event interface can be further broken up into the incoming and outgoing event interfaces, each of which are sequences of dependency types. The indexed position of each dependency type around the incoming or outgoing event interface can therefore be referred to as a *dependency port*. We will index the dependency ports from left to right. As a matter of convenience we will sometimes name dependency ports in a pair along with their event type; $(A,\, k)$ for the kth incoming or outgoing dependency port of event type A. If we want to indicate whether it is incoming or outgoing, we will notate the pair with a $-$ or $+$ subscript, respectively. In this notation, we can say for A in the figure, that dependency port $(A,\, 3)_+$ has a dependency type o_3.

An event instantiates an event type in the context of a behavioral representation, specifically an OEG. For an event, as opposed to its event type, we will overload the same diagrammatic notation as that of its event type, since in any given context it should be clear which of these we are referring to. As a technical matter, the only thing distinguishing two separate instances of the same event type will be some underlying system of dummy indices we may use to name them in the mathematical formalism, but it will be never necessary, in practice to show these diagrammatically. For any given event e, we use the notation $\mathbf{C}(e)$ to refer to its event type.

For a particular event, in an OEG, we will refer to the corresponding dependency ports of its event type as the *dependency sites* of the event; they are the cites at which dependencies will be connected to the event. When constructing OEGs it will be useful to refer to particular dependency sites, and thus we will extend the notation $(e,\, k)_{+/-}$ to refer to the k-th incoming $-$ or outgoing $+$ dependency site for event e.

A couple examples of event types are depicted in Figure 7.3 using this diagrammatic language. The first example, Figure 7.3a, is the more formal diagrammatic counterpart to the message reception event types we described earlier. There are two incoming dependency types to this event type: the current state of the process s, waiting for the message, and the

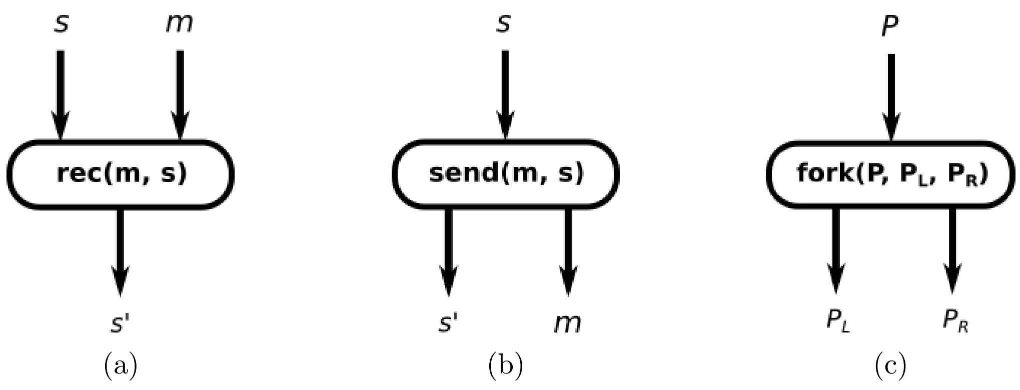


Figure 7.3: Three examples of event types.

message itself m. In other words, the event happening depends on these two things although they may be thought of very differently. It is also conceivable that the event type depends on more dependencies, such as blocking constraints, or a timer elapsing, which could be added to the incoming interface. The event type also has a single outgoing dependency type, which is the new state s' of the system subsequent to receiving the message. This state could be one in which the received message has been put into a particular register in the state representation, or one in which it has been added to some queue represented in the state.

Both states and the message dependency types for this event type could be either specific states and a specific message, or more abstract states and messages, depending on how abstract a representation of behavior is saught given the kind of reasoning one would like to use with it. In the former case, the event type is a very specific event, which could be described: “message m is received in state s resulting in a new state s'”. In the more abstract case, m and s could be symbolic, and the event type representative of a purely symbolic event, described simply: “a message is received”. One could imagine any number of intermediate abstractions between the two of these as well. In the figure, we have called this event type **rec**(m, s, s') to most freely cover any of these cases. Note that we might need to specify s' to cover nondeterministic cases, where m and s do not totally determine the event.

The second example, shown in Figure 7.3b, corresponding to the first, represents a message being sent. In this case, the event type has only one incoming event type for the current state s, in which the message m will be subsequently sent. The two outgoing event types are the new state s' of the process after the send and the message itself m. This event type will be called **send**(m, s, s').

In the third example, shown in Figure 7.3c, the event represented is the fork of a process P into two concurrent components P_L and P_R. The event type has as its only incoming dependency type the continuation of a process P (perhaps encoded in a process algebra), and has as its two outgoing dependency types the continuations P_L and P_R. Like the first example this kind of an event type could be abstracted in many ways. Here we similarly denote the event type concretely as **fork**(P, P_L, P_R).

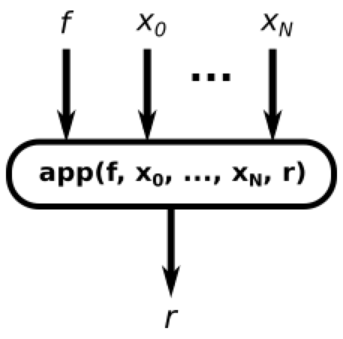


Figure 7.4: An example of an functional event type.

A fourth example, shown in Figure 7.4, the calling of a function f on a set of parameters that are all evaluated concurrently (as is done sometimes in the execution of functional programs). The event itself is the *application.* The leftmost incoming dependency type is the evaulated function f and the rest x_i are its evaluated parameters. The outgoing dependency type is the result r. We can denote this concretely as $\mathbf{app}(f,\, x_0,\, \ldots,\, x_N,\, r)$. While this codification of application is stateless, as in the case of λ-calculus, if we wished to encode the semantics of a language similar to LISP that applies with respect to an environment, this dependency could be concatenated to the incoming and outgoing dependency type.

It could be said that amongst these examples, with the exception of the completely symbolic version of the first, none are of a specific event type, but more precisely of parametric families of event types. $\mathbf{rec}(m,\, s,\, s')$ depends on its three parameters, and thus the space of all such event types might be generated from an even more basic set of primitive values. However, we presuppose such a process since this construction will generally be different for different kinds of event types.

In OEGs, events, which are instances of event types, are connected with dependencies, which, likewise, are instances of dependency types. These individual dependencies will be depicted as channels that connect to an outgoing dependency site on one end and an incoming dependency site on the other. In this manner, a block diagram is constructed of events and dependencies. In Figure 7.5 such a diagram is depicted. More than just characterizing the ordering of events, the OEG shows very explicitly the flow of information and influence between events in the behavior. In this flow of dependencies, it is also explicit what role each dependency plays in an event, because of the way that dependencies are connected to, not just the event, but specific dependency sites on the event. To be concrete, if one of these events was the reception of a message, one could tell from only the OEG itself which event the message came from, rather than having to interrogate the originating application that may have produced the behavior represented by it.

As can be seen in this figure, there are also dependency sites that are not attached to events, but instead form incoming or outgoing dependency sites of the whole OEG. In our notation for dependency sites, these are identified by replacing the event identifier with a $\star$, i.e. $(\star,\, 2)_-$ for the second incoming dependency site from the left. This concept is

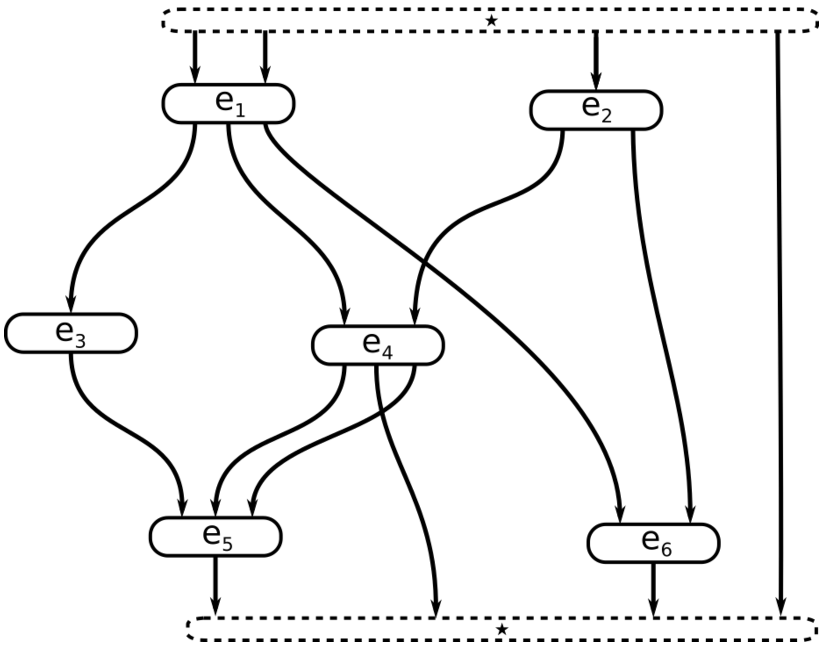


Figure 7.5: An example of an OEG.

illustrated in Figure 7.5 by the two bounding fixtures marked with a $\star$. Using these $\star$-dependency sites, the whole diagram is given a bounding event interface, and it can be made clear and explicit how preceding or following behaviors can influence or be influenced by the behavior represented in the diagram. On the right of the example OEG, we even illustrate the possibility of a dependency that runs from the incoming dependency site to an outgoing dependency site of the whole graph. This indicates a dependency that runs through the behavior, without interacting in any of its events. Although this may seem less interesting for the meaning of a specific OEG, it is included, in part, for formal completeness, and can even be used to make a non-trivial assertion about an OEG.

7.2 Ontological Event Schemes

A collection of dependency types and event types for a particular language of OEGs can be aggregated into a structure called an OES. Formally, this structure is captured by the monoidal scheme we have defined in Chapter 6. Here we provide this mathematical structure with a concrete interpretation in our event ontology.

Definition 28. Ontological Event Scheme *(OES)*
An OES Σ *is a monoidal scheme* $(\mathbb{T}, \mathcal{A})$*, where* $\mathbb{T}$ *is a set of dependency types, and* $\mathcal{A}$ *is a*

function

$$\mathcal{A} : \mathbf{Mon}(\mathbb{T}) \times \mathbf{Mon}(\mathbb{T}) \to \mathbf{Set}$$

mapping pairs $a, b \in \mathbf{Mon}(\mathbb{T})$ of words of dependency types into disjoint sets $\mathcal{A}(a, b)$ of event types.

This OES serves as an underlying language for OEGs defined in a particular context. What we called *primitive blocks* suggestively in Chapter 6 are here event types, and *primitive types* are dependency types.

An OES give a structure to the constituting events and dependencies of an OEGs that will govern the way it can be composed, and thus the OES can be seen as a vastly more powerful analog to an *alphabet of actions* often used label events in other representations of concurrent behavior. In comparison to these other representations, such as the ones we have discussed, the constraints embedded in OESs are fine grained and sophisticated, imposed by the typing $\mathbb{T}$ that will determine the connection between events. Whereas actions, by themselves, can only classify events, these event types can be used to determine how events can follow one another.

To make a specific comparison, the Reliance Alphabet[1] of Mazurkiewicz Traces contrasts with an OESs providing only a fixed set of relational constraints between its primitive events. This only determines whether events can, and must, follow one another or remain concurrent, not specifically how they can influence each other. More, the relation of Mazurkiewicz Trace must be imposed uniformly for every instance of an event with respect to all other events. This makes Reliance Alphabet far less flexible. It should be noted that while the elements of an OES impose constraints upon composition, unlike a Reliance Alphabet, they do not determine it, referring to the problem addressed earlier in Section 4.2 on Mazurkiewicz Traces. In Figure 4.1, we showed an example of two concurrent processes that, although different, look the same to sequential observers, and therefore cannot be distinguished using a Reliance Alphabet. One can see now that both of these diagrams are easily realized using an OES.

What follows from defining an OES as a monoidal scheme is that all of the mathematical structure developed around monoidal schemes in Chapter 6 can be carried over to OESs. Most significantly, OESs form a category $\mathcal{MS}$; therefore, in this context we can say that $\mathcal{MS}$ is, indeed, the category of OESs. A substantial benefit to having OESs in the form of the category $\mathcal{MS}$ is that the morphisms of this category provide a space of transformations between different OESs and a definition of composition under which this space is closed. We will discuss this further after presenting some examples.

The following additional structures and concepts can be defined for this formalism making many of the components of the event ontology that we have already defined informally more explicit.

Definition 29.
For each OES $\Sigma = (\mathbb{T}, \mathcal{A})$, the following are defined.

- *An event interface is a pair $(a,\, b) \in \mathcal{I}$, where the set of event interfaces for Σ is defined*

$$\mathcal{I} \overset{\text{def}}{=} \mathbf{Mon}(\mathbb{T}) \times \mathbf{Mon}(\mathbb{T})$$

  The first and second component will be called the incoming and outgoing components, and will be indexed for any $x \in \mathcal{I}$ by the notation x_- and x_+, respectively.

- *The entire set of event types in the OES is defined*

$$\mathcal{A}_\cup \overset{\text{def}}{=} \bigcup_{x \in \mathcal{I}} \mathcal{A}(x)$$

- *The function τ is defined*

$$\begin{aligned} &\tau \,:\, \mathcal{A}_\cup \to \mathcal{I} \\ &\tau(A \in \mathcal{A}(x)) \overset{\text{def}}{=} x \end{aligned}$$

  which is well-defined, since $\mathcal{A}(x)$ are all disjoint.

- *Incoming and outgoing dependency ports are defined as pairs $(A,\, n) \in \mathbf{P}_{+/-}$, where the sets of incoming dependency ports $\mathbf{P}_-$ and outgoing dependency ports $\mathbf{P}_+$ are defined*

$$\mathbf{P}_{+/-} \overset{\text{def}}{=} \{(A,\, n) \in \mathcal{A} \times \mathbb{N} \mid n \in \left|\tau_{+/-}(A)\right|\}$$

  The combined set of all dependency ports is defined

$$\mathbf{P} \overset{\text{def}}{=} \mathbf{P}_- \amalg \mathbf{P}_+$$

  where $(1,\, (A,\, n))$ and $(2,\, (A,\, n))$ are notated $(A,\, n,\, -)$ and $(A,\, n,\, +)$, respectively.

- *The functions $\kappa_{+/-}$ are defined*

$$\begin{aligned} &\kappa_{+/-} \,:\, \mathbf{P}_{+/-} \to \mathbb{T} \\ &\kappa_{+/-}(A,\, n) \overset{\text{def}}{=} \left[\tau_{+/-}(A)\right]_n \end{aligned}$$

  along with

$$\begin{aligned} &\kappa \,:\, \mathbf{P} \to \mathbb{T} \\ &\kappa(A,\, n,\, p) \overset{\text{def}}{=} \left[\tau_p(A)\right]_n \end{aligned}$$

One might notice that the information involved in grouping all of the event types by interface into the $\mathcal{A}$ sets could be reconstructed from $\mathcal{A}_\cup$ and τ, and indeed there is an equivalent definition of an OES of this form that may be useful in many practical circumstances.

Proposition 9. *An OES* $(\mathbb{T}, \mathcal{A})$ *can be defined uniquely by the triple* $(\mathbb{T}, \mathcal{A}_\cup, \tau)$*, where* $\mathcal{A}$ *is a set and* τ *has the type*

$$\tau : \mathcal{A}_\cup \to \mathbf{Mon}(\mathbb{T}) \times \mathbf{Mon}(\mathbb{T})$$

In this definition, $\mathcal{A}$ *is defined*

$$\mathcal{A}(a, b) \overset{def}{=} \{A \in \mathcal{A}_\cup \mid \tau(A) = (a, b)\}$$

Proof. Firstly, the definition of $\mathcal{A}$ from τ and $\mathcal{A}_\cup$ is the set inverse of τ, hence all sets $\mathcal{A}(a, b)$ are disjoint, thus this definition is of a well-defined OES. Using the above definitions for $\mathcal{A}_\cup$ and τ in terms of $\mathcal{A}$, the two transformations

$$a : (\mathcal{A}_\cup, \tau) \to \mathcal{A}$$
$$b : \mathcal{A} \to (\mathcal{A}_\cup, \tau)$$

can be shown to be inverses. First, $a \circ b$ is

$$(a \circ b)(\mathcal{A})(x) = \{A \in \bigcup_{x \in \mathcal{I}} \mid A \in \mathcal{A}(x)\} = \mathcal{A}(x)$$

since $\tau(A) = x$ whenever $A \in \mathcal{A}(x)$. Then the other direction, $b \circ a$ is for the $\mathcal{A}_\cup$ component

$$[(b \circ a)(\mathcal{A}_\cup, \mathbb{T})]_1 = \bigcup_{x \in \mathcal{I}} \{A \in \mathcal{A}_\cup \mid \mathbb{T}(A) = x\} = \mathcal{A}_\cup$$

since $\mathcal{I}$ is the entire codomain of τ. For the τ component,

$$[(b \circ a)(\mathcal{A}_\cup, \mathbb{T})]_2(A) = x \text{ if } A \in \{A \in \mathcal{A}_\cup \mid \mathbb{T}(A) = x\} = x \text{ if } \mathbb{T}(A) = x = \mathbb{T}(A)$$

Therefore the definitions are isomorphic. □

Example: Sequential Processes with Message Passing

As an example, we will define in formal detail an OES for representing the behavior of sequential processes that communicate through rendezvous message passing (a simple and pervasive paradigm of concurrency). This example will elaborate on the **send** and **rec** event types defined earlier, completing the set of parametric event types. We will start with a simple model of the individual sequential systems.

Let the states of any of the processes form a set of states S and the messages passed form a set M. Let there also be a set P of sequential procedures that modify the state. The semantics for each kind of operation in the sequential process then can be defined in terms of transformations over states. We assume the existence of three functions

$$proc : P \to S \to S$$
$$send : S \to S \times M$$
$$rec : M \to S \to S$$

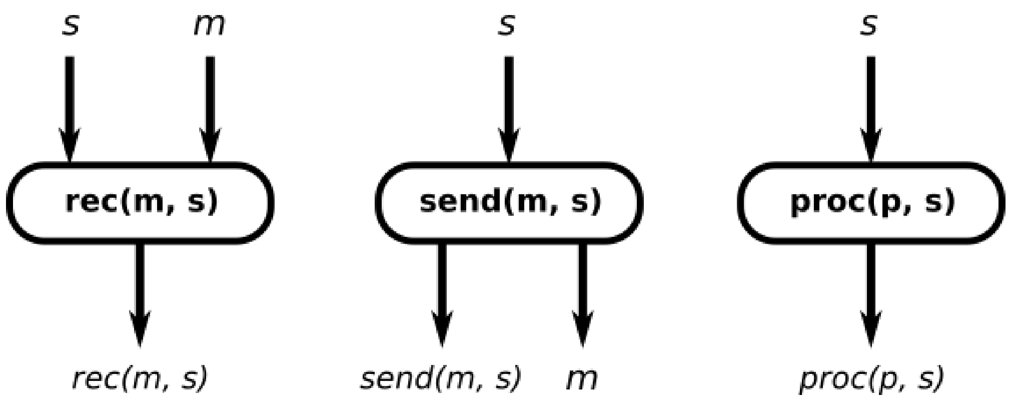


Figure 7.6: An OES for sequential processes with message passing.

And thus the model of each process is the collection of these three sets and three functions.

To make this more clear, one might imagine the kind of program that gives rise to the above semantics, particularly since it is more intuitive sometimes to give the semantics as a semantics for something. These sequential processes can be though of as a sequence of instructions executing on a processor, where S is the state of the processor. The two special instructions that send and receive are distinguished, whereas every subsequence of any other instructions can be aggregated into procedures in P. The histories of processes are then sequences of procedures, sends, and receives, which are then transformed into functions over states via the semantics.

We then use this model to define our dependency types and event types, forming a OES $\Sigma_{\mathbf{MP}}$. The dependency types $\mathbb{T}^{\Sigma_{\mathbf{MP}}}$ is defined

$$\mathbb{T}^{\Sigma_{\mathbf{MP}}} \overset{\text{def}}{=} S \cup M$$

meaning that each specific state and each specific message is its own dependency type, rather than the sets themselves, which serve, in contrast, to define the types of the semantic functions. This is a key distinction which might cause confusion if not explicitly emphasized. The event types can then be generated from model. Three families of event types are defined parameterized by the three sets S, M, and P. These generate $\mathcal{A}_{\cup}^{\Sigma_{\mathbf{MP}}}$ from the following derivations.

$$\frac{p \in P \qquad s \in S}{\mathbf{proc}(p,\, s) \in \mathcal{A}_{\cup}^{\Sigma_{\mathbf{MP}}}} \qquad \frac{m \in M \qquad s \in S}{\mathbf{rec}(m,\, s) \in \mathcal{A}_{\cup}^{\Sigma_{\mathbf{MP}}}} \qquad \frac{m \in M \qquad s \in S}{\mathbf{send}(m,\, s) \in \mathcal{A}_{\cup}^{\Sigma_{\mathbf{MP}}}}$$

These three families of event types are depicted graphically in Figure 7.6

We then define the function $\tau^{\Sigma_{\mathbf{MP}}}$ as follows

$$\begin{aligned}
\tau^{\Sigma_{\mathbf{MP}}}(\mathbf{proc}(p,\, s)) &= (s,\, \mathit{proc}(p)(s)) \\
\tau^{\Sigma_{\mathbf{MP}}}(\mathbf{rec}(m,\, s)) &= (s,\, \mathit{receive}(s,\, m) \otimes m) \\
\tau^{\Sigma_{\mathbf{MP}}}(\mathbf{send}(m,\, s)) &= (s \otimes m,\, \mathit{send}(s,\, m))
\end{aligned}$$

Here, in all three cases, the outgoing state dependency type does not have to be specified explicitly in the event type, since it is given deterministically by the corresponding semantic functions. The tuple $(\mathbb{T}^{\Sigma_{\mathbf{MP}}}, \mathcal{A}_{\mathrm{U}}^{\Sigma_{\mathbf{MP}}}, \tau^{\Sigma_{\mathbf{MP}}})$ defines the OES $\Sigma_{\mathbf{MP}}$.

Using $\Sigma_{\mathbf{MP}}$, some example graphs can be constructed by instantiating the event types as events and connecting these together with dependencies. Although we have not yet formally presented the definition of an OEG, enough has been described to understand the examples intuitively.

Figure 7.7 depicts the graph of a three-way handshake, like that of the TCP protocol. This protocol involves two processes, a client and a server. The client, in state s_0, ready to form a connection with a server in state r_0, sends message $m_{\mathbf{SYN}}$ to the server. The server receives this message and sends a message $m_{\mathbf{SYNACK}}$, in return, back to the client. After receiving the $m_{\mathbf{SYNACK}}$ message, the client sends an $m_{\mathbf{ACK}}$ message back to the server. The server receives this message. Between each message sending or receiving event, there is a **proc** event that accounts for the sequential steps performed between these message related events. This protocol leaves both the client and the server in connected states s_6 and r_6, respectively.

This is a simple example, but a lot can be said about it to illustrate how an OES can be used to form OEGs. An important structural detail is that the OEG represents a behavior that might be a fragment of a longer behavior. In contrast, some of the other representation of behavior such as Actor Event Diagrams and Event Structures do not have a clear means of representing a fragment that fits into a larger behavior. While Mazurkiewicz Traces do have a notion of composition, and thus a means to incorporate a fragment into a larger behavior, there is no means to impose a compositional constraint on a Mazurkiewicz Trace.

On the boundary of the example OEG, the compositional constraint is clear. The precondition for this behavior is a pair of state dependencies with the dependency types s_0 and r_0, the states of the client and the server. The postcondition dependency types, similarly, are a pair of states s_6 and r_5, for a client and server. This gives us a well defined notion of what kind of concurrent behavior can prepare a system to go through this protocol and what kinds of behaviors can follow it. This is depicted in Figure 7.8a.

More specifically, what can be seen from this example is that the two sequences of dependency types on the boundary of the OEG constitute, in essence, an event interface. In fact, the whole OEG could be transformed into an event type, as is shown in Figure 7.8b. Instances of it could be created and assembled into another OEG. This hierarchical property will be discussed later in terms of the adjoint functors used in the construction of the spaces of OEGs.

One simple characteristic worth noting is that the message passing dependencies along the right side cross the state dependencies. This is because it is important to preserve the ordering of the dependency ports, since this ordering has meaning. Strictly speaking, if the dependency ports were swapped, the event type would be different, having a different interfaces. While it might superficially seem reasonable to loosen this constraint, this canonical ordering of the dependency ports will be used to keep the composition operations on OEGs simple. On the other hand, as a matter of simplicity, various shorthands and mathematical

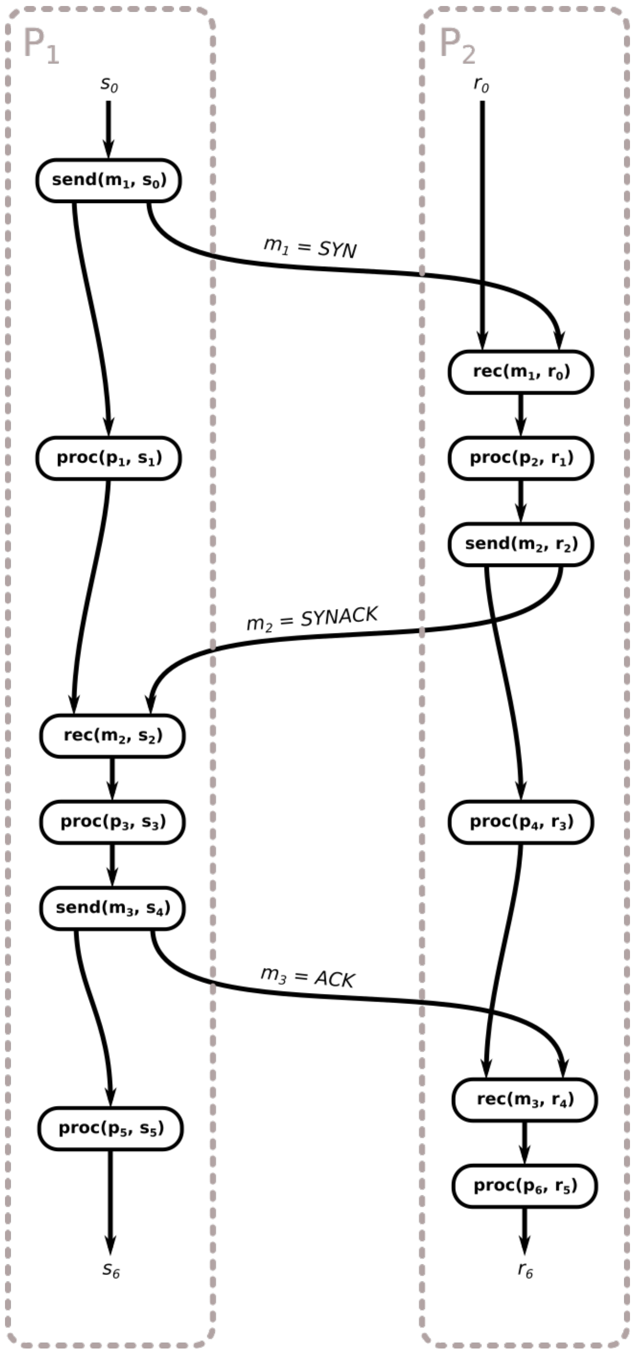
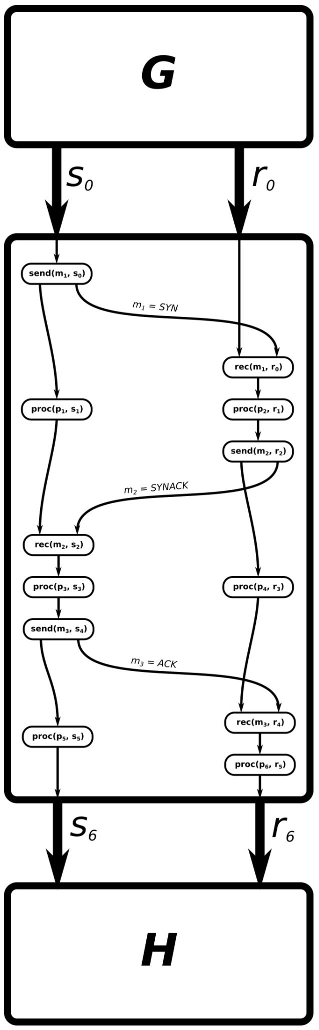
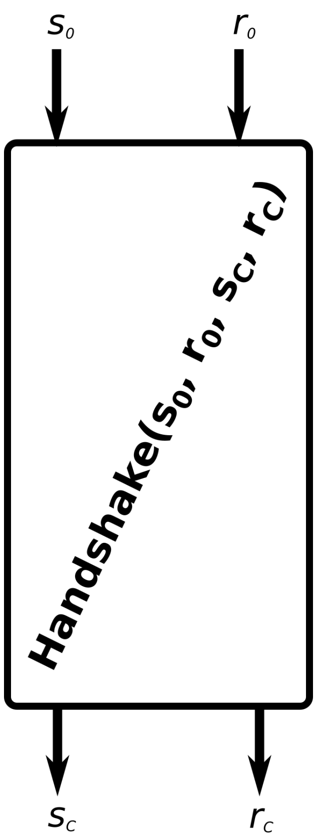


Figure 7.7: OEG for a three-way handshake protocol.



(a) A three-way handshake composed with other OEGs



(b) A three-way handshake abstracted hierarchically

Figure 7.8: Hierarchical abstraction of a three-way handshake.

macros could be explicitly introduced in a straightforward way.

Another notable characteristic one could point out about this example is looks like an Activation Diagram (AD), and has two clear columns of events that represent the client and server process. These are suggested by the two gray boxes in Figure 7.7, labeled P_1 and P_2. One can see the two chains of state dependencies running along both columns. However, there is no primitive concept of a *process* in the event ontology of OEGs; P_1 and P_2 do not illustrate a primitive feature of the representation. This is an asset to the representation because it is flexible enough to represent the same semantic notion of behavior on different platforms. In dynamic platforms like the IoT, the same behavior might be realized different ways through the dynamic creation of new concrete process and through code mobility.

Consider in Figures 7.9 and 7.10 two alternative renderings of the same OEGs, with the exact same topology as that of Figure 7.7. While the original presentation suggested two processes, the alternative in Figure 7.9 suggests three, shown by the dotted lines. If we were to define some abstract topological sense of a *process* in the OEG, in might be a column of events, but of course, this is not a topologically invariant notion, as is clear from the differences between the presentations show. Another definition of a process might be a dependency path through the OEG that goes from an incoming graph dependency site to an outgoing graph dependency site, which is a topological property. This might seem like a reasonable definition for $\Sigma_{\mathbf{MP}}$, where there are no events to create or join processes. However, in cases where these operations are possible, this definition would exclude child processes. Moreover, it is an ambiguous notion since different overlapping paths could be taken between dependency sites.

A definition could be more explicit with regards to dependency type, and the ontology of the system being modeled, and specify that a process is a path of dependencies that are "states" in some sense specific to the OEG. On the other hand, in platforms where code mobility and dynamic creation of processes are pervasive, let alone possible, the distinction between a "state" continued on one process and a "message" passed between processes can break down, no longer reflecting a fundamental dimension of the semantics of applications.

Consider the MoC introduced in ECMAScript interpreting runtime environments such as web browsers and NodeJS [24]. Procedures in these environments are often broken up into small atomic fragments, referred to colloquially as *callbacks*. These atomic fragments are processed in an event queue. When a fragment is dequeued and executed it can pass the continuation of its control asynchronously to another fragment that is deferred and ultimately placed in the event queue under a condition that may not be itself synchronous. A blocking read event, like that represented by **rec**, might concretely mean in a traditional multithreaded environment that a thread is suspended until the read occurs, then subsequently continued with the received message. In contrast, in a ECMAScript environment, a blocking read is often a function called with the continuation of control passed as a parameter. In this case, a **rec** event might concretely mean that when the call is made prior to the read event the incoming state dependency for the read is not simply the next operation in a thread, but rather more like a message sent to another process; in this case, the event queue. Therefore, the notion that paths of states are "processes" and other dependencies

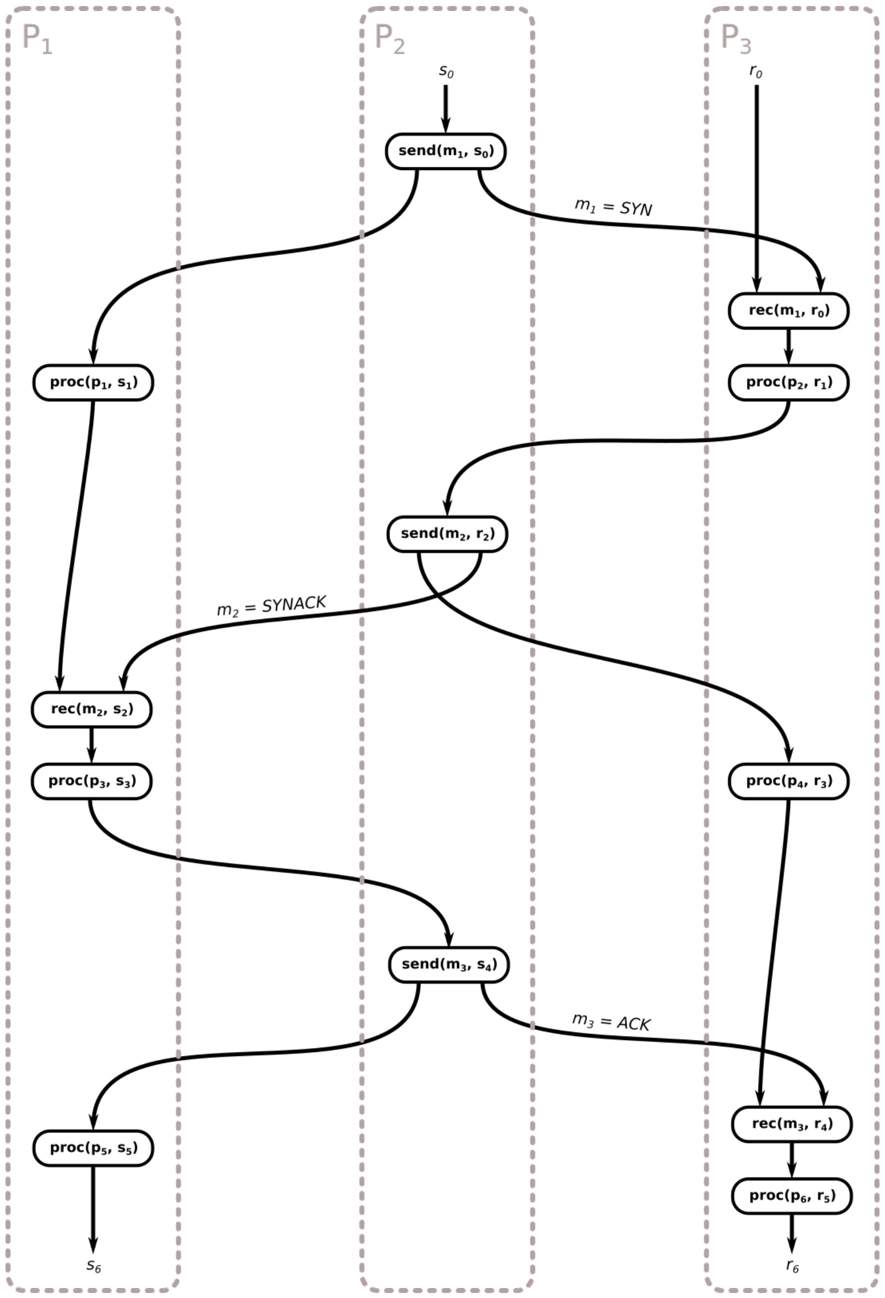


Figure 7.9: An alternate rendering of the OEG for a three-way handshake protocol.

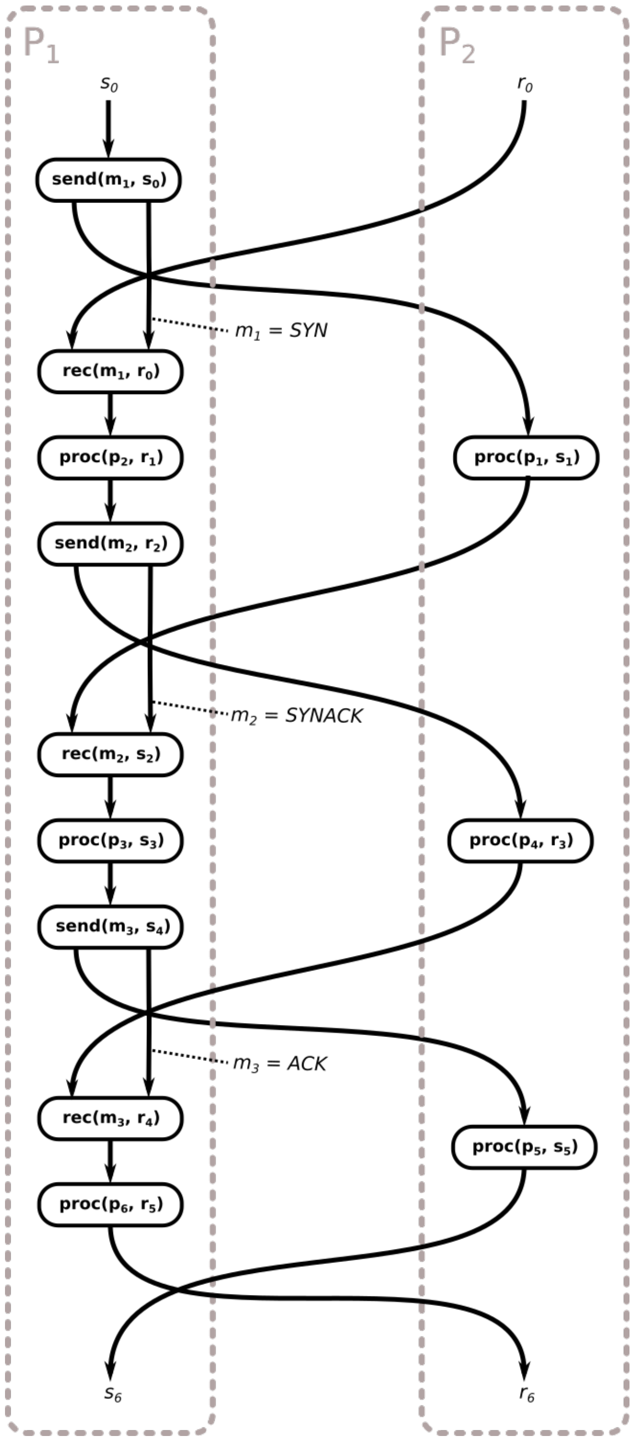


Figure 7.10: Another alternate rendering of the OEG for a three-way handshake protocol.

are “between processes” breaks down.

Allowing this distinction between states (or continuations) and messages to break down, the interpretation of the two alternative renderings of the OEG, particularly the rendering in Figure 7.10. In this rendering, the messages originally passed between “processes” are illustrated as though they are stationary, acting as the state in P_1. In contrast, the processing events subsequent to sends in this rendering appear deferred to a separate process. Again, this is just to illustrate the lack of this distinction in the OEG itself.

None of this is to say that any these notions we have discussed of a process within an OEG are not useful in some contexts. When mapping an application to a concrete system, the events of an OEG can be partitioned into processes in a manner similar to the scheduling of *Acyclic Precedence Graphs* (APGs), or other kinds of *Task Graphs*, found in DF *Models of Concurrent Computation* (MoCCs). The three variations of the same OEG in Figure 7.7, 7.9, and 7.10 can be interpreted as different schedulings of the same concurrent behavior. Some further comments will be made about this in the final comments in Chapter 9.

One could see easily how to extend $\Sigma_{\mathbf{MP}}$ in various ways to accommodate more kinds of event types that might occur in this kind of a system. Process initializations, forks, joins, and the like, could all be added to the OES using the same definitional procedures we have used. This OES could also be abstracted in many ways from the very concrete form we have defined it in.

For the purposes of many analyses, such as scheduling, which specific message and states may be immaterial, and thus a far more abstract OES $\Sigma_{\mathbf{AMP}}$ could be used in which the set of dependency types is defined

$$\mathbb{T}^{\Sigma_{\mathbf{AMP}}} \overset{\text{def}}{=} \{M,\, S\}$$

containing only one abstract message M and one abstract state S. The corresponding set of event types would then consist of three abstract events

$$\mathcal{A}_{\cup}^{\Sigma_{\mathbf{AMP}}} \overset{\text{def}}{=} \{\mathbf{proc},\, \mathbf{send},\, \mathbf{rec}\}$$

The typing function $\tau^{\Sigma_{\mathbf{AMP}}}$ can then be defined

$$\begin{aligned}\tau^{\Sigma_{\mathbf{AMP}}}(\mathbf{proc}) &= (S,\, S)\\ \tau^{\Sigma_{\mathbf{AMP}}}(\mathbf{send}) &= (S,\, S \otimes M)\\ \tau^{\Sigma_{\mathbf{AMP}}}(\mathbf{rec}) &= (S \otimes M,\, S)\end{aligned}$$

The OEG in Figure 7.7 could have just as well been assembled from $\Sigma_{\mathbf{AMP}}$. One arrives at this abstracted OEG by simply forgetting certain details about the typing along with the parameters of the event type. Given $\Sigma_{\mathbf{AMP}}$ is less constrained one could imagine a greater breadth of compositions arising from it, however it can be seen that this abstraction does not open the door for any topologies of connections that were not possible in $\Sigma_{\mathbf{MP}}$.

If one wanted to fully erase the typing distinctions from $\Sigma_{\mathbf{MP}}$, and $\Sigma_{\mathbf{AMP}}$, one could define a purely structural OES, $\Sigma_{\mathbf{MP-Str}}$. This OES would have a single type (untyped)

$$\mathbb{T}^{\Sigma_{\mathbf{MP-Str}}} \overset{\text{def}}{=} \{\star\}$$

and have three structural event types, defined

$$\mathcal{A}_{\cup}^{\Sigma_{\mathbf{MP-Str}}} \overset{\text{def}}{=} \{\mathbf{proc}, \mathbf{send}, \mathbf{rec}\}$$

where the notation for each simply indicates the number of incoming and outgoing dependency ports. The typing function would be

$$\tau^{\Sigma_{\mathbf{MP-Str}}}(1 \to 1) = (\star, \star)$$
$$\tau^{\Sigma_{\mathbf{MP-Str}}}(1 \to 2) = (\star, \star \otimes \star)$$
$$\tau^{\Sigma_{\mathbf{MP-Str}}}(2 \to 1) = (\star \otimes \star, \star)$$

Since a free monoid over a unit set $\{\star\}$ is isomorphic to the monoid of $\mathbb{N}$ with addition, we could notate the incoming and outgoing event interfaces with natural numbers, hence the notation for the event type. In this last case of $\Sigma_{\mathbf{MP-Str}}$, there are some graph topologies that can be constructed that cannot occur in $\Sigma_{\mathbf{MP}}$ or $\Sigma_{\mathbf{AMP}}$. From the perspective of Chapter 6, this last OES $\mathbb{T}^{\Sigma_{\mathbf{MP-Str}}}$ is a sub-OES of the terminal element in $\mathcal{MS}$, $\mathbf{term}^{\mathcal{MS}}$.

The Category of OESs

The most fundamental consequence of an OES being defined as an monoidal scheme is that the morphisms of the category $\mathcal{MS}$ provide a clear notion of a mapping between OESs. We will call these morphism *Ontological Event Scheme Morphisms* (OESMs) in the context of OESs. As morphisms in $\mathcal{MS}$, OESMs have the compositions and identities defined for monoidal schemes. In general, we can carry over all of the features of $\mathcal{MS}$ into OESs, giving them an interpretation in the event ontology.

An OESM $F : \Sigma \to \Sigma'$ is a mapping between two OESs, defined by a mapping $F_{\mathbb{T}}$ between the dependency types and a mapping $F_{\mathcal{A}}$ for each set of event types in Σ into a set of event types in Σ'. As is the case with most homomorphisms, both at the level of the dependency types and the event types, an OESM can either conflate these components, abstracting them, and/or immerse them in a larger space. Given that each OES represents the basic event types and dependency types of an application space, a OESM could immerse one such basic language of these elements into a larger one, extending the possible event types and dependency types possible. For instance, our $\Sigma_{\mathbf{MP}}$ could be included into a larger OES that contains additional event types such as process forking, joining, and termination. This extended version of $\Sigma_{\mathbf{MP}}$ is depicted in Figure 7.11 in which three additional families of events, **fork**, **join**, and **term** are included. These additional families of events have event interfaces with dependency types in S.

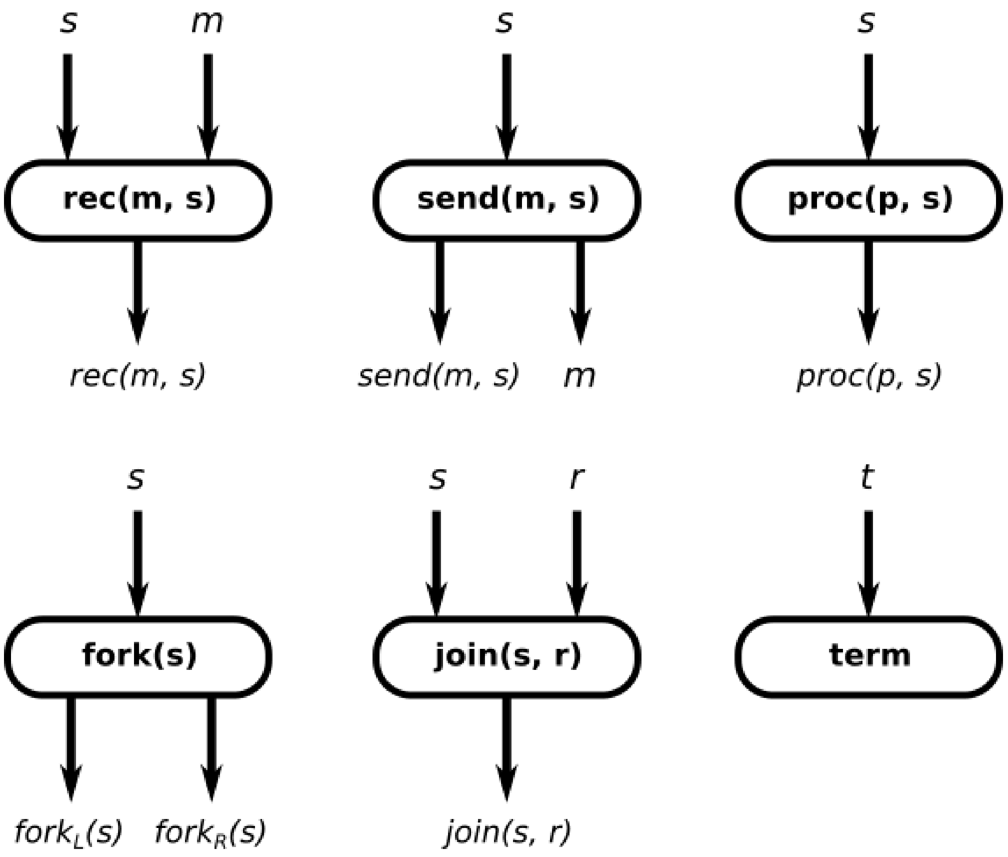


Figure 7.11: An extended version of the OES for processes with message passing.

While this kind of OESM is fairly simple and obvious, a perhaps more powerful use of OESMs is to express abstractions of OESs. Consider the two abstractions of $\Sigma_{\mathbf{MP}}$ we gave above. An OESM $\alpha_{\mathbf{MP}} : \Sigma_{\mathbf{MP}} \to \Sigma_{\mathbf{AMP}}$ can define this abstraction as follows.

$$[\alpha_{\mathbf{MP}}]_{\mathbb{T}}(s \in S) \overset{\text{def}}{=} S$$
$$[\alpha_{\mathbf{MP}}]_{\mathbb{T}}(m \in M) \overset{\text{def}}{=} M$$
$$[\alpha_{\mathbf{MP}}]_{\mathcal{A}}(\mathbf{proc}(p,\, s)) \overset{\text{def}}{=} \mathbf{proc}$$
$$[\alpha_{\mathbf{MP}}]_{\mathcal{A}}(\mathbf{rec}(m,\, s)) \overset{\text{def}}{=} \mathbf{rec}$$
$$[\alpha_{\mathbf{MP}}]_{\mathcal{A}}(\mathbf{send}(m,\, s)) \overset{\text{def}}{=} \mathbf{send}$$

Likewise, an OESM can be defined mapping $\Sigma_{\mathbf{AMP}}$ into $\Sigma_{\mathbf{MP-Str}}$. But this OESM is simply the structural monoidal scheme functor $str^{\mathcal{MS}}(\Sigma_{\mathbf{AMP}})$ from Chapter 6. These abstracting OESMs are important for reasoning about applications with OEGs because they provide a mechanism to develop an abstract interpretation of OEGs. Later we will see how OESMs are lifted to act on whole OEGs.

As was pointed out about monoidal scheme functors, an important property of OESM is that its mapping of event types preserves the valence of the interface. Although the dependency types my change, the number of incoming and outgoing dependency types remains the same. In principle, the coarsest possible abstraction of an OES is represented by the terminal element $\mathbf{term}^{\mathcal{MS}}$, which contains a collection of purely topological event types, one for each number of incoming and outgoing dependencies. OEGs built from $\mathbf{term}^{\mathcal{MS}}$ are only shapes of executions, which is nearly, but not quite the underlying space of a generalized sequence, a graph or partial order. What additionally must be forgotten is the ordering of the incoming and outgoing dependencies along with the incoming and outgoing dependencies of the whole OEG.

Given that we showed in Chapter 6 that $\mathcal{MS}$ is further a co-complete category, arbitrary small colimits can be taken of OESs. Although this covers quite a large territory, the most straightforward of these colimits are simple combinations of OESs. The extended $\Sigma_{\mathbf{MP}}$, depicted in Figure 7.11 could have been constructed by taking the coproduct of $\Sigma_{\mathbf{MP}}$ with an OES consisting of only the three additional elements. In addition to coproducts, which are disjoint, overlapping products of OESs can be taken by combining coproducts with equivalence classes.

7.3 The Formalism of Ontological Event Graphs

Having established the formal basis for OESs, and showing intuitively how one can construct OEGs from them, we can now give a sensible formal definition for an OEG constructed from the event types and dependency types in an OES. This definition will be given in the form of a directed acyclic graph in which the incoming and outgoing edges for each vertex will be ordered. In this graph the events will be represented by the vertices and the dependencies will be represented by the directed edges. There will also be edges with one or both edges not connected to a vertex, but rather ordered as incoming or outgoing for the graph as a whole (as we have discussed earlier). Consequently, this will be significantly more than just a labeled graph, akin to those underlying or constituting generalized sequences such as Event Structures, pomsets, or Actor Event Diagram. Having been given all of these enrichments, this kind of graph can be identified precisely as the *anchored progressive polarized oriented graph* discussed in [27], specifically in the section on free symmetric monoidal categories.

In this graph, each events will be labeled with an event types from an OESs. More than just labeling the events, in the sense that a generalized sequence is labeled, more, each event will be given an event interface, constraining the kinds of events that can connect with it via dependency, and even how they can be connected. In essence, this will amount to a formal graph-theoretic treatment of the acyclic ported block diagrams that informally describe OEGs. The advantage of this very particular set of features was indeed first established by Joyal and Street in [27] after being known informally for a while before. The finite form of this specific kind of graph was shown by Joyal and Street to be isomorphic to free symmetric monoidal categories.

A consideration that will play into the formal treatment of compositions over OEGs is that the set of events that will be used in defining a concrete representation of an OEGs (one that can be written down, where the events are given names) will only be significant geometrically. The names or identifiers given to specific events are only "dummy indices." Any renaming of the events along with a corresponding renaming in the structure, constituting an isomorphism between concrete OEGs, should be the same essential OEG. We will therefore define first a *concrete*-OEG, then define an isomorphism formally for this structure. Finally, we will take the quotient over the space of these *concrete*-OEGs to get our actual, geometrical OEGs. In most contexts, this whole affair might be elided, but this distinction will turn out to be important later in establishing the right kind of equivalences over these structures.

Definition 30. concrete-Ontological Event Graph *(OEG)*
For an OES $\Sigma = (\mathbb{T}, \mathcal{A})$*, a* concrete-*OEG over* Σ*,* $\mathbf{G}'$*, is a tuple*

$$\mathbf{G}' \overset{def}{=} (E,\, \mathbf{C},\, \mathcal{I},\, \gamma)_\Sigma$$

with the typing

$$\begin{aligned} E &: \mathbf{FinSet} \\ \mathbf{C} &: E \to \mathcal{A} \\ \mathcal{I} &: \mathcal{I}^\Sigma \\ \gamma &: \mathbf{c}_+ \times \mathbf{c}_- \to 2 \end{aligned}$$

where

- *E is a finite set of events.*
- *$\mathbf{C}$ is a function giving the event types for each event.*
- *$\mathcal{I}$ is the event interface of the OEG.*
- *γ, which we call the network of the OEG, is a total bijective acyclic binary relation between incoming and outgoing dependency sites, $\mathbf{c}_{+/-}$. The dependency sites are either the internal dependency ports of a specific event instance, defined*

  $$\mathbf{c}^\circ{}_{+/-} \overset{def}{=} \{(e,\, n) \in E \times \mathbb{N} \mid (\mathbf{C}(e),\, n) \in \mathbf{P}_{+/-}\}$$

  or a boundary dependency port referring to the event interface of the whole graph, defined

  $$\mathbf{c}^\partial{}_{+/-} \overset{def}{=} \{(\star,\, n) \in \bigstar \times \mathbb{N} \mid n \in \left|\mathcal{I}_{-/+}\right|\}$$

  The complete sets of incoming and outgoing dependency sites $\mathbf{c}_{+/-}$ are defined

  $$\mathbf{c}_{+/-} \overset{def}{=} \mathbf{c}^\circ{}_{+/-} \cup \mathbf{c}^\partial{}_{+/-}$$

and the combined sets of dependency sites $\mathbf{c}$ *are defined*

$$\begin{aligned}\mathbf{c}^{\circ} &\overset{def}{=} \mathbf{c}^{\circ}{}_{+} \amalg \mathbf{c}^{\circ}{}_{-}\\ \mathbf{c}^{\partial} &\overset{def}{=} \mathbf{c}^{\partial}{}_{+} \amalg \mathbf{c}^{\partial}{}_{-}\\ \mathbf{c} &\overset{def}{=} \mathbf{c}_{+} \amalg \mathbf{c}_{-}\end{aligned}$$

The definitions of $\kappa_{+/-}$ *and* κ *are extended to include dependency sites*

$$\begin{aligned}\kappa_{+/-}(e,\, n) &\overset{def}{=} \kappa_{+/-}(\mathbf{C}(e),\, n)\\ \kappa(e,\, n,\, p) &\overset{def}{=} \kappa(\mathbf{C}(e),\, n,\, p)\end{aligned}$$

under the constraint that

$$\forall\,(p,\, q) : \mathbf{c}_{+} \times \mathbf{c}_{-}\ \cdot\ \gamma(p,\, q)\ \Rightarrow\ \kappa_{-}(p) = \kappa_{+}(q) \tag{7.1}$$

In some respects, this definition resembles a labeled graph, which is no doubt abundantly familiar to many computer scientists. However, it differs in a couple important respects. The relation γ, rather than just connecting the events in E themselves, instead connects the dependency sites around each event. As we mentioned earlier in our presentation of the event ontology, these dependency sites are identified with a pair $(e,\, n)$ consisting of the event e with a port index n; whether it is an incoming or outgoing dependency site is often determined by context, but can be expressed explicitly as $(e,\, n)_{-}$ or $(e,\, n)_{+}$, respectively. In order to give an exact domain to γ we define the complete set of incoming and outgoing dependency sites for the OEG to be $\mathbf{c}^{\circ}{}_{-}$ and $\mathbf{c}^{\circ}{}_{+}$ respectively.

This structure also has an important feature that accounts for its composability, an event interface $\mathcal{I}$ that determines a sequence of incoming and outgoing dependencies for the whole OEG. The event interface consists of a collection of boundary dependency sites that are indexed by pairs of the form $(\star,\, n)$, where n is the index of the incoming or outgoing dependency port of the event interface. The collection of these incoming and outgoing boundary dependency sites are defined as $\mathbf{c}^{\partial}{}_{-}$ and $\mathbf{c}^{\partial}{}_{+}$, respectively. The collections of all dependency sites, the internal ones around each event along with the boundary dependency sites, are defined to be the sets $\mathbf{c}_{-}$ and $\mathbf{c}_{+}$.

For example, in the handshake example in Figure 7.7 the event interfaces for the graph is

$$\mathcal{I}_{\text{handshake}} = (s_0 \otimes r_0,\, s_6 \otimes r_6)$$

The incoming dependency sites for the graph are therefore $(\star,\, 0)$ and $(\star,\, 1)$, $\kappa_{-}(\star,\, 0) = s_0$, and so forth. Translating the illustrated graph into the above concrete formalism completely is straightforward and would involve choosing arbitrary identifiers to build the corresponding concrete set of events E.

With the complete set of incoming dependency sites $\mathbf{c}_-$ and outgoing dependency sites $\mathbf{c}_+$ of an OEG, the network γ of the OEG can be defined as a bijection from the outgoing dependency sites to the incoming dependency sites. In other words, each dependency site must be connected to one and only one other of the opposite polarity, either around another event or in the event interface of the OEG. More, the single axiom 7.1 further constrains γ such that each pair of connected incoming and outgoing dependency sites must share the same dependency type, as determined by the corresponding event interfaces. In essence, γ must "type check". This property is clearly witnessed in Figure 7.7.

An equivalence $\cong_{\mathbf{OEG}}$ can now be defined between *concrete*-OEGs.

Definition 31. *For any G' and H', both* concrete-*OEGs of the same OES Σ, let the equivalence of* concrete-*OEGs be defined $G' \cong_{\mathbf{OEG}} H'$ iff there exists a bijection*

$$\mathfrak{h} : E^{G'} \to E^{H'}$$

for which the following hold

$$\begin{aligned}
&\mathbf{C}^{G'} = \mathbf{C}^{H'} \circ \mathfrak{h}\\
&\mathcal{I}^{G'} = \mathcal{I}^{H'}\\
&\gamma^{G'}((e_1, n), (e_2, m)) \Leftrightarrow \gamma^{H'}((\mathfrak{h}_\star(e_1), n), (\mathfrak{h}_\star(e_2), m))
\end{aligned}$$

where $\mathfrak{h}_\star = \mathfrak{h} + \{\star \mapsto \star\}$ and the last two equivalences quantify over elements in the corresponding dependency sites.

This definition corresponds precisely to the intuition of simply changing the names of events, while changing the other structures correspondingly to accommodate this change.

We can now define some relevant additional structures.

Definition 32. *Let the following be defined:*

- *Let* $\mathrm{OEG_{concrete}}^{\Sigma}$ *be the set of all* concrete-*OEGs over an OES Σ.*
- *Let* OEG^{Σ} *be defined*

$$\mathrm{OEG}^{\Sigma} \overset{def}{=} \mathrm{OEG_{concrete}}^{\Sigma} / \cong_{\mathbf{OEG}}$$

- *Let an OEG over an OES Σ be a member of* OEG^{Σ}.
- *For any OEG $G \in \mathrm{OEG}^{\Sigma}$, let $G' \in G$ be a* concrete *representation of G in* $\mathrm{OEG_{concrete}}^{\Sigma}$.
- *For any OES Σ and an event interface $\mathcal{I}$, let the set of OEGs over OES with $\mathcal{I}$ as its event interface be notated* $\mathrm{OEG}^{\Sigma}(\mathcal{I})$ *(or* $\mathrm{OEG}^{\Sigma}(\mathcal{I}_-, \mathcal{I}_+)$*). Let the corresponding sets be defined for* concrete-*OEGs as* $\mathrm{OEG_{concrete}}^{\Sigma}(\mathcal{I})$ *(and* $\mathrm{OEG_{concrete}}^{\Sigma}(\mathcal{I}_-, \mathcal{I}_+)$*).*

In the last item, the collections $\mathrm{OEG}^{\Sigma}(\mathcal{I})$ aggregated by their event interface are analogous to the corresponding collections $\mathcal{A}(\mathcal{I})$ of event types in an OES. Given that each space OEG^{Σ} is a quotient over $\cong_{\mathbf{OEG}}$-isomorphic *concrete*-OEGs, there is a canonical projection

$$\langle - \rangle \, : \, \mathrm{OEG}_{\mathrm{concrete}}{}^{\Sigma} \rightarrow \mathrm{OEG}^{\Sigma}$$

such that $\langle G' \rangle$ is the OEG associated with the *concrete* G'.

Returning to the message passing example again, we formulated the OES $\Sigma_{\mathbf{MP}}$, which we used to construct the handshake example. Thus using the above notations we would state that handshake $\in \mathrm{OEG}_{\mathrm{concrete}}{}^{\Sigma}(s_0 \otimes r_0, s_6 \otimes r_6)$, which determines how this OEG can be composed with behaviors leading into it or following it. Although we have discussed this earlier, in this section we will see how this constraint on composition manifests formally.

Composition of OEGs

We now have a means to take an OES and build OEGs directly by defining the events and the network of their connections. However, we would like to also be able to build up OEGs from other OEGs using composition operations, so that they can be assembled in a modular fashion. Two of the most important desiderata of our representation are indeed that it is modular and compositional. In order to define these compositions, and an accompanying set of important primitive OEGs, we will first define some interstitial operators that will be used in the definitions, as well as to simplify the process of reasoning about the validity and properties of these composition operators. Most of these operators will specifically be used for handling the network γ components of *concrete*-OEGs.

More than just the presentation of a series of useful constructions, the formidable technical task that must be initiated in this section is to close the gap between the concrete and abstract presentations of OEGs, so that the former can be left in the background and handled transparently when needed. Even more formidable will be demonstrating that these operators fulfill a set of axioms over OEGs that establish collections of OEGs as forming symmetric monoidal categories. As a matter of fact, the latter task depends on the former, because many of the details in the gap between concrete and abstract OEGs would prevent them from forming a strict symmetric monoidal category representation, forcing us to instead use a non-strict one, where some equivalences are reduced to natural isomorphisms. If the latter sounds more expedient, the prices paid would be in these details inevitably emerging somewhere else in the formalism, particularly in the establishment of naturality conditions on the isomorphisms.

When presenting each operator or constant, we will briefly discuss its ontological role in modeling behavior. We will then show that with these operators and constants, sets of OEGs form symmetric monoidal categories.

Network Operators and Other Ancillary Structures

The following operators will be defined to work on predicates like that of γ in a concrete OEG. It will be assumed that, although γ is constrained in its domain to only the dependency sites in a particular OEG, any γ can be implicitly extended to a larger domain upon which its value on all other inputs are false. More specifically, we will implicitly immerse these predicates on dependency sites into a space in which the indices of dependency ports can vary over integers. In practice this will make the reasoning easier.[1] As opposed to the networks that are specifically part of OEGs, immersing these networks into a larger space will mean that the members of this space will no longer necessarily be total (they certainly will not if they are finite), but these networks will nevertheless remain bijective.

Definition 33. *Network Operations*
Let γ and ϕ be bijective (partial) relations, networks, of the type

$$\gamma,\, \phi,\, \dots \,:\, ((E + \bigstar) \times \mathbb{Z}) \times ((E + \bigstar) \times \mathbb{Z}) \to 2$$

The following operations will be defined over these networks.

- *Given any injective function $i\,:\,E \to D$ for some set D, let the following be defined*

  $$\begin{aligned}&\mathbb{L}[i](\gamma)((e_1,\, n),\, (e_2,\, m)) \overset{def}{=}\\ &\quad \exists\, f_1,\, f_2 \cdot e_1 = i_\star(f_1) \,\wedge\, e_2 = i_\star(f_2) \,\wedge\, \gamma((f_1,\, n),\, (f_2,\, m))\end{aligned}$$

  which lifts γ into a larger space, potentially unconstrained by a set of clear dependency sites.

- *For any number $a \in$ **Ords**, let the following be defined*

  $$(x,\, n) \uparrow a \overset{def}{=} \begin{cases} x = \star & (\star,\, n - a) \\ (x,\, n) & \textit{otherwise} \end{cases}$$

  This operator shifts a boundary dependency site back by a and leaves internal dependency sites unchanged.

- *For any pair of numbers $a,\, b \in$ **Ords**, let the following be defined*

  $$\gamma \uparrow (a,\, b)(x,\, y) \overset{def}{=} \gamma(x \uparrow a,\, y \uparrow a)$$

  which applies a shift from the pair to the incoming and outgoing dependency sites.

[1] On the other hand, if one wanted to be even more precise about this, and important step would be ensuring the support of network matches the interface. This detail is one we will generally be left as implicit here.

- *Let the following be defined*

$$\begin{aligned} \delta_-(p,\,(e,\,m)) &\overset{def}{=} e \neq \star & \qquad \nabla_+(p,\,(e,\,m)) &\overset{def}{=} e = \star \\ \delta_+((e,\,n),\,q) &\overset{def}{=} e \neq \star & \nabla_-((e,\,n),\,q) &\overset{def}{=} e = \star \\ [\gamma]_{+/-} &\overset{def}{=} \gamma \wedge \delta_{+/-} & (\!|\gamma|\!)_{+/-} &\overset{def}{=} \gamma \wedge \nabla_{+/-} \\ [\gamma]_\circ &\overset{def}{=} \gamma \wedge \delta_- \wedge \delta_+ & (\!|\gamma|\!)_{\|} &\overset{def}{=} \gamma \wedge \nabla_- \wedge \nabla_+ \end{aligned}$$

- *Let the following be defined*

$$(\gamma \frown \phi)(p,\,q) \overset{def}{=} \exists\, k : \mathbb{Z} \cdot \gamma(p,\,(\star,\,k)) \wedge \phi((\star,\,k),\,q)$$

  which constructs a relation connecting the outgoing dependency sites of γ with the incoming ones of ϕ, in order. In general, this operation assumes a common domain, hence the two relations will often have to be lifted with the $\mathbb{L}[-](-)$ operator into the appropriately common domain.

- *For any $N \in \mathbb{N}$, let the following family of networks be defined*

$$\gamma^N((e_1,\,n_1),\,(e_2,\,n_2)) \overset{def}{=} n_1 = n_2 \wedge e_1 = e_2 = \star \wedge 0 \leq n_1 < N$$

- *For any $N,\,M \in \mathbb{N}$, let the following family of networks be defined*

$$\gamma^{(N,\,M)} \overset{def}{=} \gamma^N \uparrow (0,\,M) \vee \gamma^M \uparrow (N,\,0)$$

These operators have a number of properties that will server greatly to simplify proofs in which they are involved.

Proposition 10.
The following hold given a set of relations

$$\gamma,\,\phi,\,\ldots : ((E + \bigstar) \times \mathbb{Z}) \times ((E + \bigstar) \times \mathbb{Z}) \to 2$$

1.

$$\begin{aligned} \mathbb{L}[i](\gamma \vee \phi) &= \mathbb{L}[i](\gamma) \vee \mathbb{L}[i](\phi) \\ \mathbb{L}[i](\gamma \wedge \phi) &= \mathbb{L}[i](\gamma) \wedge \mathbb{L}[i](\phi) \end{aligned}$$

2.

$$\mathbb{L}[j](\mathbb{L}[i](\gamma)) = \mathbb{L}[j \circ i](\gamma)$$

3.

$$\gamma \uparrow (a,\, b) \uparrow (c,\, d) = \gamma \uparrow (a + c,\, b + d)$$

4.

$$\mathbb{L}[i](\gamma) \uparrow x = \mathbb{L}[i](\gamma \uparrow x)$$

5.

$$\begin{aligned}
&[[\gamma]_{+/-}]_{+/-} = [\gamma]_{+/-}\\
&[\gamma]_{\circ} = [[\gamma]_{+}]_{-} = [[\gamma]_{-}]_{+}\\
&[\gamma]_{\circ} = [[\gamma]_{+/-}]_{\circ} = [[\gamma]_{\circ}]_{+/-}\\
&(\!|(\!|\gamma|\!)_{+/-}|\!)_{+/-} = (\!|\gamma|\!)_{+/-}\\
&(\!|\gamma|\!)_{||} = (\!|(\!|\gamma|\!)_{+}|\!)_{-} = (\!|(\!|\gamma|\!)_{-}|\!)_{+}\\
&(\!|\gamma|\!)_{||} = (\!|(\!|\gamma|\!)_{+/-}|\!)_{||} = (\!|(\!|\gamma|\!)_{||}|\!)_{+/-}\\
&[(\!|\gamma|\!)_{+/-}]_{+/-} = (\!|[\gamma]_{+/-}|\!)_{+/-}\\
&[(\!|\gamma|\!)_{-/+}]_{+/-} = (\!|[\gamma]_{-/+}|\!)_{+/-} = \mathit{false}
\end{aligned}$$

6.

$$\begin{aligned}
&[\gamma \wedge \phi]_{+/-} = [\gamma]_{+/-} \wedge [\phi]_{+/-}\\
&[\gamma \vee \phi]_{+/-} = [\gamma]_{+/-} \vee [\phi]_{+/-}\\
&(\!|\gamma \wedge \phi|\!)_{+/-} = (\!|\gamma|\!)_{+/-} \wedge (\!|\phi|\!)_{+/-}\\
&(\!|\gamma \vee \phi|\!)_{+/-} = (\!|\gamma|\!)_{+/-} \vee (\!|\phi|\!)_{+/-}\\
&(\!|\gamma|\!)_{+/-} \vee [\gamma]_{-/+} = \gamma\\
&(\!|\gamma|\!)_{+/-} \wedge [\phi]_{-/+} = \mathit{false}\\
&(\!|\gamma|\!)_{+/-} \wedge [\phi]_{+/-} = (\!|[\gamma \wedge \phi]_{+/-}|\!)_{+/-} = [(\!|\gamma \wedge \phi|\!)_{+/-}]_{+/-}
\end{aligned}$$

7.

$$\begin{aligned}
&[\mathbb{L}[i](\gamma)]_{+/-} = \mathbb{L}[i]([\gamma]_{+/-})\\
&(\!|\mathbb{L}[i](\gamma)|\!)_{+/-} = \mathbb{L}[i]((\!|\gamma|\!)_{+/-})
\end{aligned}$$

8.

$$(\gamma \frown \phi) \frown \psi = \gamma \frown (\phi \frown \psi)$$

9.

$$\mathbb{L}[i](\gamma \frown \phi) = \mathbb{L}[i](\gamma) \frown \mathbb{L}[i](\phi)$$

10.

$$\begin{aligned}
&[\gamma \frown \phi]_+ = [\gamma]_+ \frown \phi \\
&[\gamma \frown \phi]_- = \gamma \frown [\phi]_- \\
&\gamma \frown [\phi]_+ = \mathit{false} \\
&[\gamma]_- \frown \phi = \mathit{false} \\
&(\!|\gamma \frown \phi|\!)_+ = \gamma \frown (\!|\phi|\!)_+ \\
&(\!|\gamma \frown \phi|\!)_- = (\!|\gamma|\!)_- \frown \phi \\
&(\!|\gamma \frown \phi|\!)_{||} = (\!|\gamma|\!)_- \frown (\!|\phi|\!)_+ \\
&(\!|\gamma|\!)_+ \frown \phi = \gamma \frown \phi = \gamma \frown (\!|\phi|\!)_-
\end{aligned}$$

11.

$$\begin{aligned}
(\gamma \vee \psi) \frown \phi = (\gamma \frown \phi) \vee (\psi \frown \phi) \\
\gamma \frown (\phi \vee \psi) = (\gamma \frown \phi) \vee (\gamma \frown \psi)
\end{aligned}$$

12.

$$\begin{aligned}
&\gamma^{(N,\,M)} \frown \gamma^{(M,\,N)} = \gamma^{N+M} \\
&[\gamma^{N}]_{+/-} = \mathit{false} \\
&[\gamma^{(N,\,M)}]_{+/-} = \mathit{false}
\end{aligned}$$

Proof.

1. Expanding the definition for the LHS of the first identity the RHS comes directly from logic identities

$$\begin{aligned}
&\mathbb{L}[i](\gamma \vee \phi)((e_1,\, n),\, (e_2,\, m)) \\
&\ \Leftrightarrow\ \exists\, f_1,\, f_2 \cdot e_1 = i_\star(f_1) \wedge e_2 = i_\star(f_2) \\
&\qquad \wedge (\gamma((f_1,\, n),\, (f_2,\, m)) \vee \phi((f_1,\, n),\, (f_2,\, m))) \\
&\ \Leftrightarrow\ \exists\, f_1,\, f_2 \cdot (e_1 = i_\star(f_1) \wedge e_2 = i_\star(f_2) \wedge \gamma((f_1,\, n),\, (f_2,\, m))) \\
&\qquad \vee (e_1 = i_\star(f_1) \wedge e_2 = i_\star(f_2) \wedge \phi((f_1,\, n),\, (f_2,\, m))) \\
&\ \Leftrightarrow\ \exists\, f_1,\, f_2 \cdot (e_1 = i_\star(f_1) \wedge e_2 = i_\star(f_2) \wedge \gamma((f_1,\, n),\, (f_2,\, m))) \\
&\qquad \vee \exists\, f_1,\, f_2 \cdot (e_1 = i_\star(f_1) \wedge e_2 = i_\star(f_2) \wedge \phi((f_1,\, n),\, (f_2,\, m))) \\
&\ \Leftrightarrow\ \mathbb{L}[i](\gamma)((e_1,\, n),\, (e_2,\, m)) \vee \mathbb{L}[i](\phi)((e_1,\, n),\, (e_2,\, m))
\end{aligned}$$

For the second identity, the first expansion step of the RHS give

$$\begin{aligned}&\mathbb{L}[i](\gamma \wedge \phi)((e_1, n), (e_2, m))\\ &\ \Leftrightarrow \exists f_1, f_2 \cdot e_1 = i_\star(f_1) \wedge e_2 = i_\star(f_2)\\ &\quad \wedge \gamma((f_1, n), (f_2, m)) \wedge \phi((f_1, n), (f_2, m))\end{aligned}$$

The existential can be distributed here on account of the fact that i is injective, therefore two separate sets of witnesses must be identical. Using this distribution the RHS can be derived.

$$\begin{aligned}&\Leftrightarrow \exists f_1, f_2 \cdot (e_1 = i_\star(f_1) \wedge e_2 = i_\star(f_2) \wedge \gamma((f_1, n), (f_2, m)))\\ &\quad \wedge \exists f_1, f_2 \cdot (e_1 = i_\star(f_1) \wedge e_2 = i_\star(f_2) \wedge \phi((f_1, n), (f_2, m)))\\ &\Leftrightarrow \mathbb{L}[i](\gamma)((e_1, n), (e_2, m)) \wedge \mathbb{L}[i](\phi)((e_1, n), (e_2, m))\end{aligned}$$

2. Expanding the LHS, this identity can be derived directly

$$\begin{aligned}&\mathbb{L}[j](\mathbb{L}[i](\gamma))((e_1, n), (e_2, m))\\ &\ \Leftrightarrow \exists f_1, f_2 \cdot e_1 = j_\star(f_1) \wedge e_2 = j_\star(f_2)\\ &\quad \wedge \exists h_1, h_2 \cdot f_1 = i_\star(h_1) \wedge f_2 = i_\star(h_2) \wedge \gamma((h_1, n), (h_2, m))\\ &\ \Leftrightarrow \exists h_1, h_2 \cdot \gamma((h_1, n), (h_2, m))\\ &\quad \wedge (\exists f_1, f_2 \cdot e_1 = j_\star(f_1) \wedge e_2 = j_\star(f_2) \wedge f_1 = i_\star(h_1) \wedge f_2 = i_\star(h_2))\\ &\ \Leftrightarrow \exists h_1, h_2 \cdot e_1 = (j \circ i)_\star(h_1) \wedge e_2 = (j \circ i)_\star(h_2) \wedge \gamma((h_1, n), (h_2, m))\\ &\ \Leftrightarrow \mathbb{L}[j \circ i](\gamma)((e_1, n), (e_2, m))\end{aligned}$$

3. First, it can be shown that

$$(e, n) \uparrow a \uparrow b = (e, n) \uparrow a + b$$

considering the case first that $e \neq \star$

$$(e, n) \uparrow a \uparrow b = (e, n) \uparrow b = (e, n) = (e, n) \uparrow a + b$$

then $e = \star$

$$(\star, n) \uparrow a \uparrow b = (\star, n - a) \uparrow b = (\star, n - a - b) = (\star, n) \uparrow a + b$$

This can then be shown through expanding the definition and simple derivation

$$\begin{aligned}&(\gamma \uparrow (a, b) \uparrow (c, d))(x, y)\\ &= (\gamma \uparrow (c, d))(x \uparrow a, y \uparrow b)\\ &= \gamma(x \uparrow a \uparrow c, y \uparrow b \uparrow d)\\ &= \gamma(x \uparrow a + c, y \uparrow b + d)\\ &= (\gamma \uparrow (a + c, b + d)(x, y)\end{aligned}$$

4. Expanding the LHS

$$\begin{aligned}&(\mathbb{L}[i](\gamma \uparrow (a,\, b)))((e_1,\, n),\, (e_2,\, m))\\ &\Leftrightarrow \exists\, f_1,\, f_2 \cdot e_1 = i_\star(f_1) \,\wedge\, e_2 = i_\star(f_2) \,\wedge\, \gamma \uparrow (a,\, b)((f_1,\, n),\, (f_2,\, m))\\ &\Leftrightarrow \exists\, f_1,\, f_2 \cdot e_1 = i_\star(f_1) \,\wedge\, e_2 = i_\star(f_2) \,\wedge\, \gamma((f_1,\, n) \uparrow a,\, (f_2,\, m) \uparrow b)\end{aligned}$$

Expanding the RHS

$$\begin{aligned}&(\mathbb{L}[i](\gamma) \uparrow (a,\, b))((e_1,\, n),\, (e_2,\, m))\\ &\Leftrightarrow \mathbb{L}[i](\gamma)((e_1,\, n) \uparrow a,\, (e_2,\, m) \uparrow b)\end{aligned}$$

For each of the dependency sites, $(e_k,\, n_k) \uparrow a = (e_k,\, r)$, where r depends on whether e_k is $\star$ or not. Nevertheless, the corresponding dependency sites in the expansion is $(f_k,\, r)$. Since $i_\star$ preserves $\star$, and if e_k is not $\star$ then $r = n_k$, it follows that the corresponding dependency sites is $(f_k,\, n_k) \uparrow a$. Therefore, the above RHS further expands to

$$\Leftrightarrow \exists\, f_1,\, f_2 \cdot e_1 = i_\star(f_1) \,\wedge\, e_2 = i_\star(f_2) \,\wedge\, \gamma((f_1,\, n) \uparrow a,\, (f_2,\, m) \uparrow b)$$

completing the proof.

5. Verifying these identities can be done directly from the definitions.

6. Expanding the terms by definition, the identities can be derived straightforwardly

$$\begin{aligned}&[\gamma \,\wedge\, \phi]_{+/-}\\ &= \gamma \,\wedge\, \phi \,\wedge\, \delta_{+/-} = \gamma \,\wedge\, \delta_{+/-} \,\wedge\, \phi \,\wedge\, \delta_{+/-} = [\gamma]_{+/-} \,\wedge\, [\phi]_{+/-}\\ &[\gamma \,\vee\, \phi]_{+/-}\\ &= (\gamma \,\vee\, \phi) \,\wedge\, \delta_{+/-} = (\gamma \,\wedge\, \delta_{+/-}) \,\vee\, (\phi \,\wedge\, \delta_{+/-}) = [\gamma]_{+/-} \,\vee\, [\phi]_{+/-}\end{aligned}$$

7. Expanding the definition of the RHS

$$\begin{aligned}&\mathbb{L}[i]([\gamma]_{+/-})((e_1,\, n),\, (e_2,\, m))\\ &\Leftrightarrow \exists\, f_1,\, f_2 \cdot e_1 = i_\star(f_1) \,\wedge\, e_2 = i_\star(f_2) \,\wedge\, [\gamma]_{+/-}((e_1,\, n),\, (e_2,\, m))\\ &\Leftrightarrow \exists\, f_1,\, f_2 \cdot e_1 = i_\star(f_1) \,\wedge\, e_2 = i_\star(f_2)\\ &\quad \wedge\, \gamma((f_1,\, n),\, (f_2,\, m)) \,\wedge\, \delta_{+/-}((f_1,\, n),\, (f_2,\, m))\end{aligned}$$

Since $\delta_{+/-}$ only depends on f_1 or f_2 being $\star$, and this property is preserved by $i_\star$, it follows that

$$\begin{aligned}&\Leftrightarrow \exists\, f_1,\, f_2 \cdot e_1 = i_\star(f_1) \,\wedge\, e_2 = i_\star(f_2)\\ &\quad \wedge\, \gamma((f_1,\, n),\, (f_2,\, m)) \,\wedge\, \delta_{+/-}((e_1,\, n),\, (e_2,\, m))\\ &\Leftrightarrow \mathbb{L}[i](\gamma)((e_1,\, n),\, (e_2,\, m)) \,\wedge\, \delta_{+/-}((e_1,\, n),\, (e_2,\, m))\\ &\Leftrightarrow [\mathbb{L}[i](\gamma)]_{+/-}((e_1,\, n),\, (e_2,\, m))\end{aligned}$$

8. This follows from directly from expanding the definitions, using the associativity of conjunction, and extending the quantifiers.

9. Expanding the definition of the LHS

$$\begin{aligned}
&\mathbb{L}[i](\gamma \frown \phi)((e_1,\, n),\, (e_2,\, m)) \\
&\quad \Leftrightarrow\ \exists\, f_1,\, f_2 \cdot e_1 = i_\star(f_1)\ \wedge\ e_2 = i_\star(f_2) \\
&\qquad \wedge\ \exists\, k \cdot \gamma((f_1,\, n),\, (\star,\, k))\ \wedge\ \phi((\star,\, k),\, (f_2,\, m)) \\
&\quad \Leftrightarrow\ \ \wedge \exists\, k \cdot\ (\exists\, f_1 \cdot e_1 = i_\star(f_1)\ \wedge\ \gamma((f_1,\, n),\, (\star,\, k))) \\
&\qquad \wedge\ (\exists\, f_2 \cdot e_2 = i_\star(f_2)\ \wedge\ \phi((\star,\, k),\, (f_2,\, m))) \\
&\quad \Leftrightarrow\ \exists\, k \cdot\ (\exists\, f_1, f_2 \cdot e_1 = i_\star(f_1)\ \wedge\ \star = i_\star(f_2)\ \wedge\ \gamma((f_1,\, n),\, (f_2,\, k))) \\
&\qquad (\exists\, f_1, f_2 \cdot \star = i_\star(f_1)\ \wedge\ e_2 = i_\star(f_2)\ \wedge\ \phi((f_1,\, k),\, (f_2,\, m))) \\
&\quad \Leftrightarrow\ \exists\, k \cdot \mathbb{L}[i](\gamma)((e_1,\, n),\, (\star,\, k))\ \wedge\ \mathbb{L}[i](\phi)((\star,\, k),\, (e_2,\, m)) \\
&\quad \Leftrightarrow\ (\mathbb{L}[i](\gamma) \frown \mathbb{L}[i](\phi))\, ((e_1,\, n),\, (e_2,\, m))
\end{aligned}$$

using the fact that for any function x, $[_\star x]^{-1}(\star) = \star$.

10. These properties all follow fairly directly from definitions.

11. These properties follow from expanding the definitions, distributing the disjunctions over the conjunctions, and distributing the existential quantifications over the disjunctions.

12. The first formula is conceptually easy to prove, but very tedious. The second two formulae follow immediately from the networks γ^N and $\gamma^{(N,\,M)}$ having no events.

$\square$

With the aid of these definitions and properties two kinds of composition can be defined over OEGs. In the case of both compositions, the approach we will take in defining them will be to combine the sets of their events disjointly, combine their interfaces, and connect their networks making use of these above mechanisms.

While the former two steps in forming compositions are comparatively simpler than the last, there is another complication that must be addressed before proceeding to the definitions of compositions. In many treatments of formalisms in which set-like objects are combined disjointly, it is conventional to presuppose, or at least assume, that their underlying sets are already disjoint – that the elements of the structures being composed always already have disjoint names. In the interest of modularity, however, it is worth showing emphasizing that we do not need to keep track of a supply of names alongside the compositions to avoid collision. This is particularly important when building up OEGs out of single-*event* OEGs. For any two sets of events, E_1 and E_2, it suffices to combine them with a proper coproduct, $E_1 \amalg E_2$, which prevents name collision by marking the elements of the right and left operands so that they are distinguished. We will use this in our constructions.

But there is a small price to pay for this convenience. The coproduct of two sets E_1 and E_2 no longer commute, and the product of three sets, the previous two with an additional E_3, is no longer associative. This is well-known but it might be important to remind readers of this. Consider $e_1 \in E_1$, $e_2 \in E_2$, and $e_3 \in E_3$. In the first case, one gets

$$\begin{aligned}&\iota_L^{E_1 \amalg E_2}(e_1),\ \iota_R^{E_1 \amalg E_2}(e_1) \in E_1 \amalg E_2\\&\iota_R^{E_1 \amalg E_2}(e_1),\ \iota_L^{E_1 \amalg E_2}(e_1) \in E_2 \amalg E_1\end{aligned}$$

where ι_L and ι_R are the left and right injections into the coproduct.[2] This is subtle, but the problem with associativity is more obvious.

$$\begin{aligned}&\iota_L^{(E_1 \amalg E_2) \amalg E_3}(\iota_L^{E_1 \amalg E_2}(e_1)),\ \iota_L^{(E_1 \amalg E_2) \amalg E_3}(\iota_R^{E_1 \amalg E_2}(e_2)),\ \iota_R^{(E_1 \amalg E_2) \amalg E_3}(e_3) \in (E_1 \amalg E_2) \amalg E_3\\&\iota_L^{E_1 \amalg (E_2 \amalg E_3)}(e_1),\ \iota_R^{E_1 \amalg (E_2 \amalg E_3)}(\iota_L^{E_2 \amalg E_2}(e_2)),\ \iota_R^{E_1 \amalg (E_2 \amalg E_3)}(\iota_R^{E_2 \amalg E_2}(e_3)) \in (E_1 \amalg E_2) \amalg E_3\end{aligned}$$

However, it is clear that there is are canonical isomorphisms

$$\begin{aligned}&E_1 \amalg E_2 \to E_2 \amalg E_1\\&(E_1 \amalg E_2) \amalg E_3 \to E_1 \amalg (E_2 \amalg E_3)\end{aligned}$$

as well as

$$\begin{aligned}&E_1 \amalg \emptyset \to E_1\\&\emptyset \amalg E_1 \to E_1\end{aligned}$$

for all sets. Taking the quotient over these isomorphism, gives a commutative monoid over equivalence classes of sets, and indeed, there is always a way to use compositions of coproducts of the left and right injection functions to construct these isomorphism. Incidentally, this is a superlative example of the *coherence* property we discussed in Chapter 6. In fact, coproducts of sets with coproducts of injections between these sets form a non-strict symmetric monoidal category.

This is worth mentioning for clarity, although for a computer scientist, this detail could be relegated to the status of an implementation consideration. If a multiset were implemented with pairs, associativity and commutativity would not be syntactic equivalence, but rather something that would have to be checked by traversing each structure or normalizing them. In any case, we will not hide this dimension entirely, but make liberal use of the fact that we can always construct and appropriate isomorphisms between these structures. Given our definition of equivalence over OEGs is witnessed by an isomorphism over the sets of events, we can state a useful lemma regarding the lifting of these witnesses through coproducts, which we will later use to show that our operators are well-defined.

[2]They simply wrap their operand with a marker L or R so that they can be distinguished in the coproduct set

Lemma 5. *Let there be two pairs of sets* (E^G, E^H) *and* $(E^{G'}, E^{H'})$*, along with two corresponding pairs of functions*

$$\begin{aligned} \mathbf{C}^G &: E^G \to X \\ \mathbf{C}^H &: E^H \to X \\ \mathbf{C}^{G'} &: E^{G'} \to X \\ \mathbf{C}^{G'} &: E^{H'} \to X \end{aligned}$$

mapping each set into some set X*. If two bijections exist*

$$\begin{aligned} \mathfrak{g} &: E^G \to E^{G'} \\ \mathfrak{h} &: E^H \to E^{H'} \end{aligned}$$

such that

$$\begin{aligned} \mathbf{C}^G &= \mathbf{C}^{G'} \circ \mathfrak{g} \\ \mathbf{C}^H &= \mathbf{C}^{H'} \circ \mathfrak{h} \end{aligned}$$

then the coproduct of the bijections

$$\mathfrak{g} \amalg \mathfrak{h} : \mathbf{C}^G \amalg \mathbf{C}^H \to \mathbf{C}^{G'} \amalg \mathbf{C}^{H'}$$

is also a bijection, and

$$\mathbf{C}^G \triangledown \mathbf{C}^H = \mathbf{C}^{G'} \triangledown \mathbf{C}^{H'} \circ \mathfrak{g} \amalg \mathfrak{h}$$

Proof. The coproduct $E^G \amalg E^H$ is a disjoint union of values of the form $\iota_L^{E^G \amalg E^H}(e_G)$ and $\iota_R^{E^G \amalg E^H}(e_H)$, where $e_G \in E^G$ and $e_H \in E^H$. By definition, $\mathfrak{g} \amalg \mathfrak{h}$ over each of these two subsets is

$$\begin{aligned} \mathfrak{g} \amalg \mathfrak{h}(\iota_L^{E^G \amalg E^H}(e_G)) &= \iota_L^{E^{G'} \amalg E^{H'}}(\mathfrak{g}(e_G)) \\ \mathfrak{g} \amalg \mathfrak{h}(\iota_L^{E^G \amalg E^H}(e_H)) &= \iota_R^{E^{G'} \amalg E^{H'}}(\mathfrak{h}(e_H)) \end{aligned}$$

Since all of the functions on the RHS are injective over their domains, images of $\iota_L^{E^{G'} \amalg E^{H'}}$ and $\iota_R^{E^{G'} \amalg E^{H'}}$ are disjoint, $\mathfrak{g} \amalg \mathfrak{h}$ is injective. More, because the union of the images of $\iota_L^{E^{G'} \amalg E^{H'}}$ and $\iota_R^{E^{G'} \amalg E^{H'}}$ equals $E^{G'} \amalg E^{H'}$ and both $\mathfrak{g}$ and $\mathfrak{h}$ are surjective, $\mathfrak{g} \amalg \mathfrak{h}$ is surjective. Thus, $\mathfrak{g} \amalg \mathfrak{h}$ is bijective. The second assertion, follows directly from the definition of $\triangledown$. □

Parallel Composition

The first composition simply places any two OEGs together in parallel without connecting their networks at all.

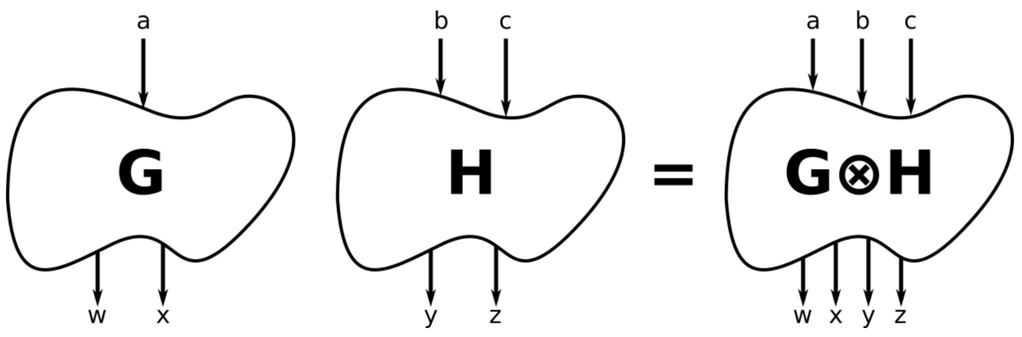


Figure 7.12: The parallel composition of G and H.

Definition 34. *OEG Parallel Composition*
For any two OEGs $\mathbf{G}$ *and* $\mathbf{H}$*, such that*

$$\mathbf{G} = \langle E^{\mathbf{G}},\, \mathbf{C}^{\mathbf{G}},\, \mathcal{I}^{\mathbf{G}},\, \gamma^{\mathbf{G}} \rangle$$
$$\mathbf{H} = \langle E^{\mathbf{H}},\, \mathbf{C}^{\mathbf{H}},\, \mathcal{I}^{\mathbf{H}},\, \gamma^{\mathbf{H}} \rangle$$

(any two concrete representatives are chosen) the product $\mathbf{G} \otimes \mathbf{H}$ *is defined*

$$\mathbf{G} \otimes \mathbf{H} \overset{def}{=} \langle E^{\mathbf{G}} \amalg E^{\mathbf{H}},\, \mathbf{C}^{\mathbf{G}} \triangledown \mathbf{C}^{\mathbf{H}},\, \mathcal{I}^{\mathbf{G}\otimes\mathbf{H}},\, \gamma^{\mathbf{G}\otimes\mathbf{H}} \rangle$$

where

$$\mathcal{I}^{\mathbf{G}\otimes\mathbf{H}} = (\mathcal{I}^{\mathbf{G}}_{-} \otimes \mathcal{I}^{\mathbf{H}}_{-},\, \mathcal{I}^{\mathbf{G}}_{+} \otimes \mathcal{I}^{\mathbf{H}}_{+})$$
$$\gamma^{\mathbf{G}\otimes\mathbf{H}} = \mathbb{L}[\iota_L^{\mathbf{G}\amalg\mathbf{H}}](\gamma^{\mathbf{G}}) \wedge \mathbb{L}[\iota_R^{\mathbf{G}\amalg\mathbf{H}}](\gamma^{\mathbf{H}}) \uparrow \|\mathcal{I}^{G}\|$$

The above definition relies on the selection of arbitrary members the equivalence classes of the operands, thus it must be shown that this choice does not effect the outcome of the composition.

Proposition 11. *For any two OEGs* $\mathbf{G}$ *and* $\mathbf{H}$*,* $\mathbf{G} \otimes \mathbf{H}$ *is a well-defined function.*

Proof. Suppose alternate representatives are chosen for $\mathbf{G}$ and $\mathbf{H}$

$$\mathbf{G} = \langle E^{\mathbf{G}'},\, \mathbf{C}^{\mathbf{G}'},\, \mathcal{I}^{\mathbf{G}'},\, \phi^{\mathbf{G}'} \rangle$$
$$\mathbf{H} = \langle E^{\mathbf{H}'},\, \mathbf{C}^{\mathbf{H}'},\, \mathcal{I}^{\mathbf{H}'},\, \phi^{\mathbf{H}'} \rangle$$

Since these alternates are equivalent, there exists witnesses

$$\mathfrak{g} : E^{\mathbf{G}} \to E^{\mathbf{G}'}$$
$$\mathfrak{h} : E^{\mathbf{H}} \to E^{\mathbf{H}'}$$

to the equivalences. What must be shown is that

$$\begin{aligned}&(E^{\mathbf{G}} \amalg E^{\mathbf{H}},\, \mathbf{C}^{\mathbf{G}} \triangledown \mathbf{C}^{\mathbf{H}},\, \mathcal{I}^{\mathbf{G}\otimes\mathbf{H}},\, \gamma^{\mathbf{G}\otimes\mathbf{H}})\\ &\quad \cong_{\mathbf{OEG}} (E^{\mathbf{G}'} \amalg E^{\mathbf{H}'},\, \mathbf{C}^{\mathbf{G}'} \triangledown \mathbf{C}^{\mathbf{H}'},\, \mathcal{I}^{\mathbf{G}'\otimes\mathbf{H}'},\, \gamma^{\mathbf{G}'\otimes\mathbf{H}'})\end{aligned}$$

It follows from Lemma 5 that $\mathfrak{g} \amalg \mathfrak{h}$ is a bijection

$$E^{\mathbf{G}} \amalg E^{\mathbf{H}} \to E^{\mathbf{G}'} \amalg E^{\mathbf{H}'}$$

and that

$$\mathbf{C}^{\mathbf{G}} \triangledown \mathbf{C}^{\mathbf{H}} = E^{\mathbf{G}} \amalg E^{\mathbf{H}} \circ \mathfrak{g} \amalg \mathfrak{h}$$

By definition, $\mathcal{I}^{\mathbf{G}} = \mathcal{I}^{\mathbf{G}'}$ and $\mathcal{I}^{\mathbf{H}} = \mathcal{I}^{\mathbf{H}'}$, therefore $\mathcal{I}^{\mathbf{G}\otimes\mathbf{H}} = \mathcal{I}^{\mathbf{G}'\otimes\mathbf{H}'}$

To show the equivalence over the γ components, with $\mathfrak{g} \amalg \mathfrak{h}$ as the witness, first we show

$$\begin{aligned}&\mathbb{L}[\iota_L^{\mathbf{G}\amalg\mathbf{H}}](\gamma^{\mathbf{G}}((e_1,\, n),\, (e_2,\, m)))\\ &\quad \Leftrightarrow \mathbb{L}[\iota_L^{\mathbf{G}'\amalg\mathbf{H}'}](\gamma^{\mathbf{G}'}(([\mathfrak{g} \amalg \mathfrak{h}]_\star(e_1),\, n),\, ([\mathfrak{g} \amalg \mathfrak{h}]_\star(e_2),\, m)))\end{aligned}$$

by expanding the RHS

$$\begin{aligned}&\mathbb{L}[\iota_L^{\mathbf{G}'\amalg\mathbf{H}'}](\gamma^{\mathbf{G}'}(([\mathfrak{g} \amalg \mathfrak{h}]_\star(e_1),\, n),\, ([\mathfrak{g} \amalg \mathfrak{h}]_\star(e_2),\, m)))\\ &\quad \Leftrightarrow \exists\, f_1,\, f_2 \cdot [\mathfrak{g} \amalg \mathfrak{h}]_\star(e_1) = [\iota_L^{\mathbf{G}'\amalg\mathbf{H}'}]_\star(f_1) \,\wedge\, [\mathfrak{g} \amalg \mathfrak{h}]_\star(e_2) = [\iota_L^{\mathbf{G}'\amalg\mathbf{H}'}]_\star(f_2)\\ &\qquad \wedge\, \gamma^{\mathbf{G}'}((f_1,\, n),\, (f_2,\, m))\end{aligned}$$

Since $\mathfrak{g}$ is a bijection, substitutions can be made $f_k = \mathfrak{g}(r_k)_\star$ for $k \in \{1,\, 2\}$.

$$\begin{aligned}&\Leftrightarrow \exists\, r_1,\, r_2 \cdot [\mathfrak{g} \amalg \mathfrak{h}]_\star(e_1) = [\iota_L^{\mathbf{G}'\amalg\mathbf{H}'} \circ \mathfrak{g}]_\star(r_1)\\ &\quad \wedge\, [\mathfrak{g} \amalg \mathfrak{h}]_\star(e_2) = [\iota_L^{\mathbf{G}'\amalg\mathbf{H}'} \circ \mathfrak{g}]_\star(r_2) \,\wedge\, \gamma^{\mathbf{G}'}((\mathfrak{g}_\star(r_1),\, n),\, (\mathfrak{g}_\star(r_2),\, m))\end{aligned}$$

applying the equivalence witnessed by $\mathfrak{g}$

$$\begin{aligned}&\Leftrightarrow \exists\, r_1,\, r_2 \cdot [\mathfrak{g} \amalg \mathfrak{h}]_\star(e_1) = \iota_L^{\mathbf{G}'\amalg\mathbf{H}'} \circ \mathfrak{g}_\star(r_1)\\ &\quad \wedge\, [\mathfrak{g} \amalg \mathfrak{h}]_\star(e_2) = \iota_L^{\mathbf{G}'\amalg\mathbf{H}'} \circ \mathfrak{g}_\star(r_2) \,\wedge\, \gamma^{\mathbf{G}}((r_1,\, n),\, (r_2,\, m))\end{aligned}$$

Recalling the identity for coproducts that

$$\mathfrak{g} \amalg \mathfrak{h} \circ \iota_L^{\mathbf{G}\amalg\mathbf{H}} = \iota_L^{\mathbf{G}'\amalg\mathbf{H}'} \circ \mathfrak{g}$$

Applying this identity to terms

$$[\mathfrak{g} \amalg \mathfrak{h}]_\star(e_k) = [\iota_L^{\mathbf{G}'\amalg\mathbf{H}'} \circ \mathfrak{g}]_\star(r_k) = [\mathfrak{g} \amalg \mathfrak{h} \circ \iota_L^{\mathbf{G}\amalg\mathbf{H}}]_\star(r_k)$$

hence, using the fact that $\mathfrak{g} \amalg \mathfrak{h}$ is a bijection

$$e_k = [\iota_L^{\mathbf{G}\amalg\mathbf{H}}]_\star(r_k)$$

Substituting this in

$$\begin{aligned}
&\Leftrightarrow \exists\, r_1,\, r_2 \cdot e_1 = [\iota_L^{\mathbf{G}\amalg\mathbf{H}}]_\star(r_1) \,\wedge\, e_2 = [\iota_L^{\mathbf{G}\amalg\mathbf{H}}]_\star(r_2) \,\wedge\, \gamma^{\mathbf{G}}((r_1,\, n),\, (r_2,\, m)) \\
&\Leftrightarrow \mathbb{L}[\iota_L^{\mathbf{G}\amalg\mathbf{H}}](\gamma^{\mathbf{G}}((e_1,\, n),\, (e_2,\, m)))
\end{aligned}$$

Using the same reasoning with the right injection

$$\begin{aligned}
&\mathbb{L}[\iota_R^{\mathbf{G}\amalg\mathbf{H}}](\gamma^{\mathbf{H}}((e_1,\, n),\, (e_2,\, m))) \\
&\quad \Leftrightarrow \mathbb{L}[\iota_R^{\mathbf{G}'\amalg\mathbf{H}'}](\gamma^{\mathbf{H}'}(([\mathfrak{g} \amalg \mathfrak{h}]_\star(e_1),\, n),\, ([\mathfrak{g} \amalg \mathfrak{h}]_\star(e_2),\, m)))
\end{aligned}$$

Since $\mathcal{I}^{\mathbf{G}} = \mathcal{I}^{\mathbf{G}'}$, it follows that

$$\begin{aligned}
&\mathbb{L}[\iota_R^{\mathbf{G}\amalg\mathbf{H}}](\gamma^{\mathbf{H}}((e_1,\, n),\, (e_2,\, m))) \uparrow \|\mathcal{I}^{\mathbf{G}}\| \\
&\quad \Leftrightarrow \mathbb{L}[\iota_R^{\mathbf{G}'\amalg\mathbf{H}'}](\gamma^{\mathbf{H}'}(([\mathfrak{g} \amalg \mathfrak{h}]_\star(e_1),\, n),\, ([\mathfrak{g} \amalg \mathfrak{h}]_\star(e_2),\, m))) \uparrow \|\mathcal{I}^{\mathbf{G}'}\|
\end{aligned}$$

Putting these together, it follows that

$$\gamma^{\mathbf{G}\otimes\mathbf{H}}((e_1,\, n),\, (e_2,\, m)) \,\Leftrightarrow\, \gamma^{\mathbf{G}'\otimes\mathbf{H}'}(([\mathfrak{g} \amalg \mathfrak{h}]_\star(e_1),\, n),\, ([\mathfrak{g} \amalg \mathfrak{h}]_\star(e_2),\, m))$$

completing the proof. □

This operation is depicted in Figure 7.12, in which one can see graphically the simple nature of the composition, in spite of the complications that go into formulating it. Ontologically, the meaning of this operation is the non-interacting conjunction of concurrent behaviors. Although powerful and necessary, it is quite unremarkable, resembling similar operators that are easily defined over graphs or partial orders. The only distinction is that here the already ordered event interface of the OEGs are combined in an ordered fashion.

It follows very obviously that this product, in contrast to similar concurrent or parallel products is not commutative. The order matters in a way that is reflected in the interface. On the other hand, the intrinsic meaning of $G \otimes H$ and $H \otimes G$ differ only in how they compose with other OEGs in larger composites. As we will see later, using *braidings* we can explicitly commute these OEGs, but this explicitness maintains a deliberate ordering of one sort or another on the incoming and outgoing dependencies.

It should also be intuitively clear that this composition is associative, but this does require a proof in order to handle the details of concrete presentations.

Proposition 12. *Given any three OEGs over a OES Σ, G, H, $K \in \mathrm{OEG}(\Sigma)$*

$$(G \otimes H) \otimes K = G \otimes (H \otimes K)$$

Proof. Consider the constructions of the RHS and LHS

$$\begin{aligned}
&(G \otimes H) \otimes K = \\
&\langle (E^G \amalg E^H) \amalg E^K,\ (\mathbf{C}^G \triangledown \mathbf{C}^H) \triangledown \mathbf{C}^K,\ \mathcal{I}^{(G \otimes H) \otimes K},\ \gamma^{(G \otimes H) \otimes K} \rangle \\
&G \otimes (H \otimes K) = \\
&\langle E^G \amalg (E^H \amalg E^K),\ \mathbf{C}^G \triangledown (\mathbf{C}^H \triangledown \mathbf{C}^K),\ \mathcal{I}^{G \otimes (H \otimes K)},\ \gamma^{G \otimes (H \otimes K)} \rangle
\end{aligned}$$

Coproducts are associative up to isomorphism, hence the mapping

$$(E^G \amalg E^H) \amalg E^K \to E^G \amalg (E^H \amalg E^K)$$

has a bijective witness

$$\alpha_R \stackrel{\text{def}}{=} (\iota_L^{G\amalg(H\amalg K)} \triangledown \iota_R^{G\amalg(H\amalg K)} \circ \iota_L^{H\amalg K}) \triangledown \iota_R^{G\amalg(H\amalg K)} \circ \iota_R^{H\amalg K}$$

which can be taken as the witness for the equivalence of concrete presentations. Its inverse is

$$\alpha_L \stackrel{\text{def}}{=} \iota_L^{(G\amalg H)\amalg K} \circ \iota_L^{G\amalg H} \triangledown (\iota_L^{(G\amalg H)\amalg K} \circ \iota_R^{G\amalg H} \triangledown \iota_R^{(G\amalg H)\amalg K})$$

It is clear from cases on injections that

$$(\mathbf{C}^G \triangledown \mathbf{C}^H) \triangledown \mathbf{C}^K = \mathbf{C}^G \triangledown (\mathbf{C}^H \triangledown \mathbf{C}^K) \circ \alpha_R$$

It is also clear that

$$\mathcal{I}^{(G \otimes H) \otimes K} = \mathcal{I}^{G \otimes (H \otimes K)}$$

since these are constructed monoidially, and the monoids are associative.

What remains to be shown is that

$$\gamma^{(G \otimes H) \otimes K}((e_1, n), (e_2, m)) \Leftrightarrow \gamma^{G \otimes (H \otimes K)}((\alpha_R(e_1), n), (\alpha_R(e_2), m))$$

Expanding the definition for the LHS and applying properties from Proposition 10

$$\begin{aligned}
&\gamma^{(G \otimes H) \otimes K} \\
&= \mathbb{L}[\iota_L^{(G\amalg H)\amalg K}](\gamma^{G \otimes H}) \wedge \mathbb{L}[\iota_R^{(G\amalg H)\amalg K}](\gamma^K) \uparrow \|\mathcal{I}^{G \otimes H}\| \\
&= \mathbb{L}[\iota_L^{(G\amalg H)\amalg K}](\mathbb{L}[\iota_L^{G\amalg H}](\gamma^G) \wedge \mathbb{L}[\iota_R^{G\amalg H}](\gamma^H) \uparrow \|\mathcal{I}^G\|) \\
&\quad \wedge \mathbb{L}[\iota_R^{(G\amalg H)\amalg K}](\gamma^K) \uparrow \|\mathcal{I}^{G \otimes H}\| \\
&= \mathbb{L}[\iota_L^{(G\amalg H)\amalg K} \circ \iota_L^{G\amalg H}](\gamma^G) \wedge \mathbb{L}[\iota_L^{(G\amalg H)\amalg K} \circ \iota_R^{G\amalg H}](\gamma^H) \uparrow \|\mathcal{I}^G\| \\
&\quad \wedge \mathbb{L}[\iota_R^{(G\amalg H)\amalg K}](\gamma^K) \uparrow \|\mathcal{I}^{G \otimes H}\|
\end{aligned}$$

Expanding the definition for the RHS similarly

$$\begin{aligned}
&\gamma^{G \otimes (H \otimes K)} \\
&= \mathbb{L}[\iota_L^{G \amalg (H \amalg K)}](\gamma^G) \wedge \mathbb{L}[\iota_R^{G \amalg (H \amalg K)}](\gamma^{H \otimes K}) \uparrow \|\mathcal{I}^G\| \\
&= \mathbb{L}[\iota_L^{G \amalg (H \amalg K)}](\gamma^G) \\
&\quad \wedge \mathbb{L}[\iota_R^{G \amalg (H \amalg K)}](\mathbb{L}[\iota_L^{H \amalg K}](\gamma^H) \wedge \mathbb{L}[\iota_R^{H \amalg K}](\gamma^K) \uparrow \|\mathcal{I}^H\|) \uparrow \|\mathcal{I}^G\| \\
&= \mathbb{L}[\iota_L^{G \amalg (H \amalg K)}](\gamma^G) \wedge \mathbb{L}[\iota_R^{G \amalg (H \amalg K)} \circ \iota_L^{H \amalg K}](\gamma^H) \uparrow \|\mathcal{I}^G\| \\
&\quad \wedge \mathbb{L}[\iota_R^{G \amalg (H \amalg K)} \circ \iota_R^{H \amalg K}](\gamma^K) \uparrow \|\mathcal{I}^H\| + \|\mathcal{I}^G\| \\
&= \mathbb{L}[\iota_L^{G \amalg (H \amalg K)}](\gamma^G) \wedge \mathbb{L}[\iota_R^{G \amalg (H \amalg K)} \circ \iota_L^{H \amalg K}](\gamma^H) \uparrow \|\mathcal{I}^G\| \\
&\quad \wedge \mathbb{L}[\iota_R^{G \amalg (H \amalg K)} \circ \iota_R^{H \amalg K}](\gamma^K) \uparrow \|\mathcal{I}^{G \otimes H}\|
\end{aligned}$$

The two reductions can be summarized as follows

$$\begin{aligned}
\gamma^{(G \otimes H) \otimes K} &= \mathbb{L}[x^G](\gamma^G) \wedge \mathbb{L}[x^H](\gamma^H) \uparrow U \wedge \mathbb{L}[x^K](\gamma^K) \uparrow V \\
\gamma^{G \otimes (H \otimes K)} &= \mathbb{L}[y^G](\gamma^G) \wedge \mathbb{L}[y^H](\gamma^H) \uparrow U \wedge \mathbb{L}[y^K](\gamma^K) \uparrow V
\end{aligned}$$

where

$$\begin{aligned}
x^G &= \iota_L^{(G \amalg H) \amalg K} \circ \iota_L^{G \amalg H} \\
x^H &= \iota_L^{(G \amalg H) \amalg K} \circ \iota_R^{G \amalg H} \\
x^K &= \iota_R^{(G \amalg H) \amalg K} \\
y^G &= \iota_L^{G \amalg (H \amalg K)} \\
y^H &= \iota_R^{G \amalg (H \amalg K)} \circ \iota_L^{H \amalg K} \\
y^K &= \iota_R^{G \amalg (H \amalg K)} \circ \iota_R^{H \amalg K}
\end{aligned}$$

What must be proven is that for each $X \in \{G, H, K\}$ is that

$$\mathbb{L}[x^X](\gamma^X)((e_1, n), (e_2, n)) \Leftrightarrow \mathbb{L}[y^X](\gamma^X)(([\alpha_R]_\star(e_1), n), ([\alpha_R]_\star(e_2), n))$$

Expanding the RHS

$$\begin{aligned}
&\mathbb{L}[y^X](\gamma^X)(([\alpha_R]_\star(e_1), n), ([\alpha_R]_\star(e_2), n)) \\
&\quad \Leftrightarrow \exists f_1, f_2 \cdot [\alpha_R]_\star(e_1) = [y^X]_\star(f_1) \wedge [\alpha_R]_\star(e_2) = [y^X]_\star(f_2) \\
&\quad \wedge \gamma^X((f_1, n), (f_2, m))
\end{aligned}$$

For each formula

$$[\alpha_R]_\star(e_k) = [y^X]_\star(f_k)$$

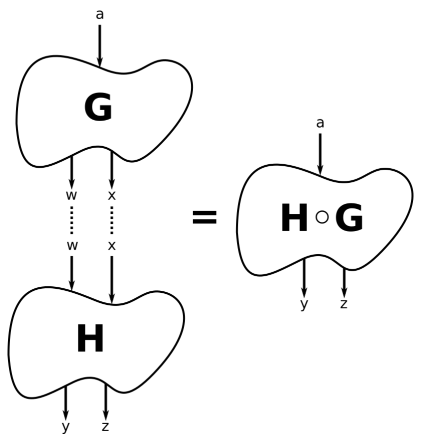


Figure 7.13: The sequential composition of G and H.

if the inverse α_L is applied to both sides, lifted with the unit, one gets

$$\begin{aligned}&[\alpha_L \circ \alpha_R]_\star(e_k)\\ &= e_k = [\alpha_L \circ y^X]_\star(f_k)\end{aligned}$$

Expanding the LHS

$$\begin{aligned}&\mathbb{L}[y^X](\gamma^X)((e_1, n), (e_2, n))\\ &\Leftrightarrow \exists\, f_1, f_2 \cdot e_1 = [x^X]_\star(f_1) \wedge e_2 = [x^X]_\star(f_2)\\ &\wedge \gamma^X((f_1, n), (f_2, m))\end{aligned}$$

Therefore, the equivalences can be proven by showing that

$$\alpha_L \circ y^X = x^X$$

Each of these can be verified. This completes the proof. □

Sequential Composition

The second kind of composition is the more substantial one, key to the expressive power of OEGs. It takes two OEGs and composes them sequentially under the condition that the *outgoing event interface* of one is equal to the *incoming event interface* of the other.

Definition 35. *OEG Sequential Composition*
For any two OEGs $\mathbf{G}$ *and* $\mathbf{H}$*, such that*

$$\begin{aligned}\mathbf{G} &= \langle E^{\mathbf{G}}, \mathbf{C}^{\mathbf{G}}, \mathcal{I}^{\mathbf{G}}, \gamma^{\mathbf{G}}\rangle\\ \mathbf{H} &= \langle E^{\mathbf{H}}, \mathbf{C}^{\mathbf{H}}, \mathcal{I}^{\mathbf{H}}, \gamma^{\mathbf{H}}\rangle\end{aligned}$$

and $\mathcal{I}_+^{\mathbf{G}} = \mathcal{I}_-^{\mathbf{H}}$ *the composition* $\mathbf{H} \circ \mathbf{G}$ *is defined*

$$\mathbf{H} \circ \mathbf{G} = \langle E^{\mathbf{G}} \amalg E^{\mathbf{H}},\, \mathbf{C}^{\mathbf{G}} \triangledown \mathbf{C}^{\mathbf{H}},\, (\mathcal{I}_-^{\mathbf{G}}, \mathcal{I}_+^{\mathbf{H}}),\, \gamma^{\mathbf{H} \circ \mathbf{G}} \rangle$$

where

$$\gamma^{\mathbf{H} \circ \mathbf{G}} \overset{def}{=} [\mathbb{L}[\iota_L^{\mathbf{G} \amalg \mathbf{H}}](\gamma^{\mathbf{G}})]_- \vee [\mathbb{L}[\iota_R^{\mathbf{G} \amalg \mathbf{H}}](\gamma^{\mathbf{H}})]_+ \vee \mathbb{L}[\iota_L^{\mathbf{G} \amalg \mathbf{H}}](\gamma^{\mathbf{G}}) \frown \mathbb{L}[\iota_R^{\mathbf{G} \amalg \mathbf{H}}](\gamma^{\mathbf{H}})$$

This operation is depicted in Figure 7.13. Ontologically, this operation continues one behavioral fragment with another upon which it can entirely depend. The definition utilizes the linking operator presented earlier $\gamma \frown \phi$ to take each outgoing dependency site on the boundary of G and connect it to each incoming dependency site on the boundary of H. This is intuitively what is depicted in Figure 7.13, and consequently, the construction of the γ element reflects this, containing three conjoined terms, respectively representing the connections of G excluding those to its outgoing boundary, the connections of H excluding those to its incoming boundary, and the link stitching the two OEGs together.

The remarkable nature of this composition can be seen in comparison to other kinds of sequential compositions that appear in other behavioral representations, particularly those presented in Chapter 4. The specific ordering of incoming and outgoing dependencies for each OEG provides a canonical means to join two OEGs, and thus two successive behaviors together. Because it is established by the event interfaces of OEGs which dependencies are joined, no additional constraints on concurrency are imposed on the composition, as they were in the case of interleaved traces. Unlike Mazurkiewicz Traces, and their more general counterparts, the rules for maintaining concurrency in the composition are not derived from a general schema, such as a Reliance Alphabet. Instead, what becomes sequential and what remains concurrent is determined by the event interfaces of the OEGs themselves.

As was the case with parallel composition, this operator must be proven to be a well-defined function; in this case, over the operands meeting the condition. Again, the simplicity of the intuitive geometric description is betrayed in the intricacies of the formalism. On the other hand, some of the steps have already taken in the previous proof for the parallel composition.

Proposition 13. *For any two OEGs* $\mathbf{G}$ *and* $\mathbf{H}$*, such that* $\mathcal{I}_+^{\mathbf{G}} = \mathcal{I}_-^{\mathbf{H}}$*,* $\mathbf{H} \circ \mathbf{G}$ *is a well-defined function.*

Proof. As in the previous proof for parallel composition, suppose alternate representatives are chosen for $\mathbf{G}$ and $\mathbf{H}$

$$\mathbf{G} = \langle E^{\mathbf{G}'},\, \mathbf{C}^{\mathbf{G}'},\, \mathcal{I}^{\mathbf{G}'},\, \phi^{\mathbf{G}'} \rangle$$
$$\mathbf{H} = \langle E^{\mathbf{H}'},\, \mathbf{C}^{\mathbf{H}'},\, \mathcal{I}^{\mathbf{H}'},\, \phi^{\mathbf{H}'} \rangle$$

with witnesses

$$\mathfrak{g} : E^{\mathbf{G}} \to E^{\mathbf{G}'}$$
$$\mathfrak{h} : E^{\mathbf{H}} \to E^{\mathbf{H}'}$$

From the previous proof it follows similarly that $\mathfrak{g} \amalg \mathfrak{h}$ serves as a provisional witness for the equivalence we are trying to show. Moreover, from the previous proof the conditions for the first two components are met similarly by this witness.

As well, since the $\mathcal{I}$ components are equivalent for both representatives, the corresponding component for the product is also equivalent.

What remains is to show equivalence over the γ components of the products. From the previous proof we have that

$$\begin{aligned}
&\mathbb{L}[\iota_L^{\mathbf{G} \amalg \mathbf{H}}](\gamma^{\mathbf{G}}((e_1, n), (e_2, m))) \\
&\quad \Leftrightarrow \mathbb{L}[\iota_L^{\mathbf{G}' \amalg \mathbf{H}'}](\gamma^{\mathbf{G}'}(([\mathfrak{g} \amalg \mathfrak{h}]_\star(e_1), n), ([\mathfrak{g} \amalg \mathfrak{h}]_\star(e_2), m))) \\
&\mathbb{L}[\iota_R^{\mathbf{G} \amalg \mathbf{H}}](\gamma^{\mathbf{H}}((e_1, n), (e_2, m))) \\
&\quad \Leftrightarrow \mathbb{L}[\iota_R^{\mathbf{G}' \amalg \mathbf{H}'}](\gamma^{\mathbf{H}'}(([\mathfrak{g} \amalg \mathfrak{h}]_\star(e_1), n), ([\mathfrak{g} \amalg \mathfrak{h}]_\star(e_2), m)))
\end{aligned}$$

Furthermore, we can show that if for some bijection $\mathfrak{g}$

$$\gamma((e_1, n), (e_2, m)) \Leftrightarrow \gamma'((\mathfrak{g}_\star(e_1), n), (\mathfrak{g}_\star(e_2), m))$$

then

$$[\gamma((e_1, n), (e_2, m))]_{+/-} \Leftrightarrow [\gamma'((\mathfrak{g}_\star(e_1), n), (\mathfrak{g}_\star(e_2), m)))]_{+/-}$$

by taking the assumption and conjoining $\delta_{+/-}$ to both sides.

$$\begin{aligned}
&\gamma((e_1, n), (e_2, m)) \wedge \delta_{+/-}((e_1, n), (e_2, m)) \\
&\quad \Leftrightarrow \gamma'((\mathfrak{g}_\star(e_1), n), (\mathfrak{g}_\star(e_2), m)) \wedge \delta_{+/-}((e_1, n), (e_2, m))
\end{aligned}$$

Since $\mathfrak{g}_\star$ preserves $\star$ bijectively, and $\delta_{+/-}$ only depends on whether or not a dependency site is $\star$

$$\Leftrightarrow \gamma'((\mathfrak{g}_\star(e_1), n), (\mathfrak{g}_\star(e_2), m)) \wedge \delta_{+/-}((\mathfrak{g}_\star(e_1), n), (\mathfrak{g}_\star(e_2), m))$$

proving that $[-]_{+/-}$ preserves the relations between γ terms.

Putting these two facts together covers the equivalence relationship for the first two disjuncts of the definition for $\gamma^{\mathbf{H} \circ \mathbf{G}}$. The condition for the third conjoint can be proven as follows.

Assume for some bijection $\mathfrak{g}$

$$\begin{aligned}
\gamma((e_1, n), (e_2, m)) &\Leftrightarrow \gamma'((\mathfrak{g}_\star(e_1), n), (\mathfrak{g}_\star(e_2), m)) \\
\phi((e_1, n), (e_2, m)) &\Leftrightarrow \phi'((\mathfrak{g}_\star(e_1), n), (\mathfrak{g}_\star(e_2), m))
\end{aligned}$$

what needs to be proven is that

$$(\gamma \frown \phi)((e_1, n), (e_2, m)) \Leftrightarrow (\gamma' \frown \phi)((\mathfrak{g}_\star(e_1), n), (\mathfrak{g}_\star(e_2), m))$$

Expanding the RHS

$$\begin{aligned}&(\gamma' \frown \phi)((\mathfrak{g}_\star(e_1),\, n),\, (\mathfrak{g}_\star(e_2),\, m))\\ &\ \Leftrightarrow\ \exists\, k : \mathbb{N} \cdot \gamma'((\mathfrak{g}_\star(e_1),\, n),\, (\star,\, k)) \,\wedge\, \phi'((\star,\, k),\, (\mathfrak{g}_\star(e_1),\, n))\end{aligned}$$

Using again the fact that $\mathfrak{g}_\star$ preserves $\star$

$$\Leftrightarrow\ \exists\, k : \mathbb{N} \cdot \gamma'((\mathfrak{g}_\star(e_1),\, n),\, (\mathfrak{g}_\star(\star),\, k)) \,\wedge\, \phi'((\mathfrak{g}_\star(\star),\, k),\, (\mathfrak{g}_\star(e_1),\, n))$$

Using the assumptions

$$\begin{aligned}&\Leftrightarrow\ \exists\, k : \mathbb{N} \cdot \gamma((e_1,\, n),\, (\star,\, k)) \,\wedge\, \phi((\star,\, k),\, (e_1,\, n))\\ &\Leftrightarrow\ (\gamma \frown \phi)((e_1,\, n),\, (e_2,\, m))\end{aligned}$$

Combining all of these together, the equivalence on $\gamma^{\mathbf{H} \circ \mathbf{G}}$ is proven, and the proof is complete. ☐

Along the same lines, it can be shown that this operation is also associative.

Proposition 14. *Given any three OEGs over a OES* Σ*,* $G,\, H,\, K \in \mathrm{OEG}(\Sigma)$*, such that* $\mathcal{I}^G_+ = \mathcal{I}^H_-$ *and* $\mathcal{I}^H_+ = \mathcal{I}^K_-$*,*

$$K \,\circ\, (H \,\circ\, G) = (K \,\circ\, H) \,\circ\, G$$

Proof. Proceeding in the same fashion as the proof of associativity for parallel composition, consider the constructions of the RHS and LHS

$$\begin{aligned}&K \,\circ\, (H \,\circ\, G) =\\ &\langle (E^G \amalg E^H) \amalg E^K,\, (\mathbf{C}^G \triangledown \mathbf{C}^H) \triangledown \mathbf{C}^K,\, (\mathcal{I}^G_-,\, \mathcal{I}^K_+),\, \gamma^{K \circ (H \circ G)} \rangle\\ &(K \,\circ\, H) \,\circ\, G =\\ &\langle E^G \amalg (E^H \amalg E^K),\, \mathbf{C}^G \triangledown (\mathbf{C}^H \triangledown \mathbf{C}^K),\, (\mathcal{I}^G_-,\, \mathcal{I}^K_+),\, \gamma^{(K \circ H) \circ G} \rangle\end{aligned}$$

Following the proof for the associativity for parallel composition, the same witness

$$\alpha_R \overset{\text{def}}{=} (\iota_L^{G\amalg(H\amalg K)} \triangledown \iota_R^{G\amalg(H\amalg K)} \circ \iota_L^{H\amalg K}) \triangledown \iota_R^{G\amalg(H\amalg K)} \circ \iota_R^{H\amalg K}$$

with the inverse

$$\alpha_L \overset{\text{def}}{=} \iota_L^{(G\amalg H)\amalg K} \circ \iota_L^{G\amalg H} \triangledown (\iota_L^{(G\amalg H)\amalg K} \circ \iota_R^{G\amalg H} \triangledown \iota_R^{(G\amalg H)\amalg K})$$

can be used to meet the condition for the first two components of the concrete presentation. The third element is the same in both presentations. This leaves only the γ elements, which must be shown to be consistent via the α_R witness.

$$\begin{aligned}&\gamma^{K \circ (H \circ G)}((e_1,\, n),\, (e_2,\, m))\\ &\ \Leftrightarrow\ \gamma^{(K \circ H) \circ G}((\alpha_{R\star}(e_1),\, n),\, (\alpha_{R\star}(e_2),\, m))\end{aligned}$$

First, we will expand the operators on both sides and use identities from Proposition 10 to perform simplifications. Starting with the LHS

$$\begin{aligned}\gamma^{K \circ (H \circ G)} \overset{\text{def}}{=} & [\mathbb{L}[\iota_L^{(G \amalg H) \amalg K}](\gamma^{H \circ G})]_- \vee [\mathbb{L}[\iota_R^{(G \amalg H) \amalg K}](\gamma^K)]_+ \\ & \vee \mathbb{L}[\iota_L^{(G \amalg H) \amalg K}](\gamma^{H \circ G}) \frown \mathbb{L}[\iota_R^{(G \amalg H) \amalg K}](\gamma^K)\end{aligned}$$

To handle the complications of the notation, we will deal with each of the three disjuncts separately

$$\begin{aligned}& [\mathbb{L}[\iota_L^{(G \amalg H) \amalg K}](\gamma^{H \circ G})]_- \\ & = [\mathbb{L}[\iota_L^{(G \amalg H) \amalg K}]([\mathbb{L}[\iota_L^{G \amalg H}](\gamma^G)]_- \vee [\mathbb{L}[\iota_R^{G \amalg H}](\gamma^H)]_+ \\ & \quad \vee \mathbb{L}[\iota_L^{G \amalg H}](\gamma^G) \frown \mathbb{L}[\iota_R^{G \amalg H}](\gamma^H))]_- \\ & = [\mathbb{L}[\iota_L^{(G \amalg H) \amalg K}]([\mathbb{L}[\iota_L^{G \amalg H}](\gamma^G)]_-)]_- \\ & \quad \vee [\mathbb{L}[\iota_L^{(G \amalg H) \amalg K}]([\mathbb{L}[\iota_R^{G \amalg H}](\gamma^H)]_+)]_- \\ & \quad \vee [\mathbb{L}[\iota_L^{(G \amalg H) \amalg K}](\mathbb{L}[\iota_L^{G \amalg H}](\gamma^G) \frown \mathbb{L}[\iota_R^{G \amalg H}](\gamma^H))]_- \\ & = [\mathbb{L}[\iota_L^{(G \amalg H) \amalg K} \circ \iota_L^{G \amalg H}](\gamma^G)]_- \\ & \quad \vee [\mathbb{L}[\iota_L^{(G \amalg H) \amalg K} \circ \iota_R^{G \amalg H}](\gamma^H)]_\circ \\ & \quad \vee \mathbb{L}[\iota_L^{(G \amalg H) \amalg K} \circ \iota_L^{G \amalg H}](\gamma^G) \frown [\mathbb{L}[\iota_L^{(G \amalg H) \amalg K} \circ \iota_R^{G \amalg H}](\gamma^H)]_- \\ & = [\mathbb{L}[x^G](\gamma^G)]_- \vee [\mathbb{L}[x^H](\gamma^H)]_\circ \vee \mathbb{L}[x^G](\gamma^G) \frown [\mathbb{L}[x^H](\gamma^H)]_-\end{aligned}$$

In the last step, the names used for the injections in the previous proof of associativity were substituted in. The second disjunct is already in a simplified form, however we will make the same substitution.

$$[\mathbb{L}[\iota_R^{(G \amalg H) \amalg K}](\gamma^K)]_+ = [\mathbb{L}[x^K](\gamma^K)]_+$$

Finally, the third disjunct will be expanded.

$$\begin{aligned}& \mathbb{L}[\iota_L^{(G \amalg H) \amalg K}](\gamma^{H \circ G}) \frown \mathbb{L}[\iota_R^{(G \amalg H) \amalg K}](\gamma^K) \\ & = \mathbb{L}[\iota_L^{(G \amalg H) \amalg K}]([\mathbb{L}[\iota_L^{G \amalg H}](\gamma^G)]_- \vee [\mathbb{L}[\iota_R^{G \amalg H}](\gamma^H)]_+ \\ & \quad \vee \mathbb{L}[\iota_L^{G \amalg H}](\gamma^G) \frown \mathbb{L}[\iota_R^{G \amalg H}](\gamma^H)) \frown \mathbb{L}[\iota_R^{(G \amalg H) \amalg K}](\gamma^K) \\ & = ([\mathbb{L}[\iota_L^{(G \amalg H) \amalg K} \circ \iota_L^{G \amalg H}](\gamma^G)]_- \vee [\mathbb{L}[\iota_L^{(G \amalg H) \amalg K} \circ \iota_R^{G \amalg H}](\gamma^H)]_+ \\ & \quad \vee \mathbb{L}[\iota_L^{(G \amalg H) \amalg K} \circ \iota_L^{G \amalg H}](\gamma^G) \frown \mathbb{L}[\iota_L^{(G \amalg H) \amalg K} \circ \iota_R^{G \amalg H}](\gamma^H)) \\ & \quad \frown \mathbb{L}[\iota_R^{(G \amalg H) \amalg K}](\gamma^K) \\ & = ([\mathbb{L}[\iota_L^{(G \amalg H) \amalg K} \circ \iota_L^{G \amalg H}](\gamma^G)]_- \frown \mathbb{L}[\iota_R^{(G \amalg H) \amalg K}](\gamma^K)) \\ & \quad \vee ([\mathbb{L}[\iota_L^{(G \amalg H) \amalg K} \circ \iota_R^{G \amalg H}](\gamma^H)]_+ \frown \mathbb{L}[\iota_R^{(G \amalg H) \amalg K}](\gamma^K)) \\ & \quad \vee ((\mathbb{L}[\iota_L^{(G \amalg H) \amalg K} \circ \iota_L^{G \amalg H}](\gamma^G) \frown \mathbb{L}[\iota_L^{(G \amalg H) \amalg K} \circ \iota_R^{G \amalg H}](\gamma^H)) \frown \mathbb{L}[\iota_R^{(G \amalg H) \amalg K}](\gamma^K)) \\ & = ([\mathbb{L}[x^H](\gamma^H)]_+ \frown \mathbb{L}[x^K](\gamma^K)) \vee (\mathbb{L}[x^G](\gamma^G) \frown \mathbb{L}[x^H](\gamma^H) \frown \mathbb{L}[x^K](\gamma^K))\end{aligned}$$

Putting these together

$$\begin{aligned}
&\gamma^{K\circ(H\circ G)} \overset{\text{def}}{=} \\
&\quad [\mathbb{L}[x^G](\gamma^G)]_- \vee [\mathbb{L}[x^H](\gamma^H)]_\circ \vee [\mathbb{L}[x^K](\gamma^K)]_+ \\
&\quad \mathbb{L}[x^G](\gamma^G) \frown [\mathbb{L}[x^H](\gamma^H)]_- \vee ([\mathbb{L}[x^H](\gamma^H)]_+ \frown \mathbb{L}[x^K](\gamma^K)) \\
&\quad \vee (\mathbb{L}[x^G](\gamma^G) \frown \mathbb{L}[x^H](\gamma^H) \frown \mathbb{L}[x^K](\gamma^K))
\end{aligned}$$

[A note at this stage: the above should make intuitive sense. The first three terms are the internal connections of each operand, the subsequent two are the direct links between the neighboring pairs, and the final term consists of the connections that go directly from G to K passing through H]

Following the same procedure the RHS

$$\begin{aligned}
&\gamma^{(K\circ H)\circ G} \overset{\text{def}}{=} [\mathbb{L}[\iota_L^{G\amalg(H\amalg K)}](\gamma^G)]_- \vee [\mathbb{L}[\iota_R^{G\amalg(H\amalg K)}](\gamma^{K\circ H})]_+ \\
&\quad \vee \mathbb{L}[\iota_L^{(G\amalg H)\amalg K}](\gamma^G) \frown \mathbb{L}[\iota_R^{(G\amalg H)\amalg K}](\gamma^{K\circ H})
\end{aligned}$$

The first disjunct here is the reduced one

$$[\mathbb{L}[\iota_L^{G\amalg(H\amalg K)}](\gamma^G)]_- = [\mathbb{L}[y^G](\gamma^G)]_-$$

The second requires an expansion

$$\begin{aligned}
&[\mathbb{L}[\iota_R^{G\amalg(H\amalg K)}](\gamma^{K\circ H})]_+ \\
&= [\mathbb{L}[\iota_R^{G\amalg(H\amalg K)}]([\mathbb{L}[\iota_L^{H\amalg K}](\gamma^H)]_- \vee [\mathbb{L}[\iota_R^{H\amalg K}](\gamma^K)]_+ \\
&\quad \vee \mathbb{L}[\iota_L^{H\amalg K}](\gamma^H) \frown \mathbb{L}[\iota_R^{H\amalg K}](\gamma^K))]_+ \\
&= [\mathbb{L}[\iota_L^{G\amalg(H\amalg K)}]([\mathbb{L}[\iota_L^{H\amalg K}](\gamma^H)]_-)]_+ \\
&\quad \vee [\mathbb{L}[\iota_L^{G\amalg(H\amalg K)}]([\mathbb{L}[\iota_R^{H\amalg K}](\gamma^K)]_+)]_+ \\
&\quad \vee [\mathbb{L}[\iota_L^{G\amalg(H\amalg K)}](\mathbb{L}[\iota_L^{H\amalg K}](\gamma^H) \frown \mathbb{L}[\iota_R^{H\amalg K}](\gamma^K))]_+ \\
&= [\mathbb{L}[\iota_L^{G\amalg(H\amalg K)} \circ \iota_L^{H\amalg K}](\gamma^H)]_\circ \\
&\quad \vee [\mathbb{L}[\iota_L^{G\amalg(H\amalg K)} \circ \iota_R^{H\amalg K}](\gamma^K)]_+ \\
&\quad \vee [\mathbb{L}[\iota_L^{G\amalg(H\amalg K)} \circ \iota_L^{H\amalg K}](\gamma^H)]_+ \frown \mathbb{L}[\iota_L^{(G\amalg H)\amalg K} \circ \iota_R^{G\amalg H}](\gamma^H) \\
&= [\mathbb{L}[y^H](\gamma^H)]_\circ \vee [\mathbb{L}[y^K](\gamma^K)]_+ \vee [\mathbb{L}[y^H](\gamma^H)]_+ \frown \mathbb{L}[y^K](\gamma^K)
\end{aligned}$$

The third is as follows

$$
\begin{aligned}
&\mathbb{L}[\iota_L^{G\amalg(H\amalg K)}](\gamma^G) \frown \mathbb{L}[\iota_R^{G\amalg(H\amalg K)}](\gamma^{K\circ H})\\
&= \mathbb{L}[\iota_L^{G\amalg(H\amalg K)}](\gamma^G) \frown \mathbb{L}[\iota_R^{G\amalg(H\amalg K)}]([\mathbb{L}[\iota_L^{H\amalg K}](\gamma^H)]_- \vee [\mathbb{L}[\iota_R^{H\amalg K}](\gamma^K)]_+\\
&\quad \vee \mathbb{L}[\iota_L^{H\amalg K}](\gamma^H) \frown \mathbb{L}[\iota_R^{H\amalg K}](\gamma^K))\\
&= \mathbb{L}[\iota_L^{G\amalg(H\amalg K)}](\gamma^G) \frown\\
&\quad ([\mathbb{L}[\iota_R^{G\amalg(H\amalg K)} \circ \iota_L^{H\amalg K}](\gamma^H)]_- \vee [\mathbb{L}[\iota_R^{G\amalg(H\amalg K)} \circ \iota_R^{H\amalg K}](\gamma^K)]_+\\
&\quad \vee \mathbb{L}[\iota_R^{G\amalg(H\amalg K)} \circ \iota_L^{H\amalg K}](\gamma^H) \frown \mathbb{L}[\iota_R^{G\amalg(H\amalg K)} \circ \iota_R^{H\amalg K}](\gamma^K))\\
&= (\mathbb{L}[\iota_L^{G\amalg(H\amalg K)}](\gamma^G) \frown [\mathbb{L}[\iota_R^{G\amalg(H\amalg K)} \circ \iota_L^{H\amalg K}](\gamma^H)]_-)\\
&\quad \vee (\mathbb{L}[\iota_L^{G\amalg(H\amalg K)}](\gamma^G) \frown [\mathbb{L}[\iota_R^{G\amalg(H\amalg K)} \circ \iota_R^{H\amalg K}](\gamma^K)]_+)\\
&\quad \vee (\mathbb{L}[\iota_L^{G\amalg(H\amalg K)}](\gamma^G) \frown (\mathbb{L}[\iota_R^{G\amalg(H\amalg K)} \circ \iota_L^{H\amalg K}](\gamma^H) \frown \mathbb{L}[\iota_R^{G\amalg(H\amalg K)} \circ \iota_R^{H\amalg K}](\gamma^K)))\\
&= (\mathbb{L}[y^G](\gamma^G) \frown [\mathbb{L}[y^H](\gamma^H)]_-) \vee (\mathbb{L}[y^G](\gamma^G) \frown \mathbb{L}[y^H](\gamma^H) \frown \mathbb{L}[y^K](\gamma^K))
\end{aligned}
$$

Putting these together

$$
\begin{aligned}
&\gamma^{K\circ(H\circ G)} \overset{\text{def}}{=}\\
&\quad [\mathbb{L}[y^G](\gamma^G)]_- \vee [\mathbb{L}[y^H](\gamma^H)]_\circ \vee [\mathbb{L}[y^K](\gamma^K)]_+\\
&\quad (\mathbb{L}[y^G](\gamma^G) \frown [\mathbb{L}[y^H](\gamma^H)]_-) \vee [\mathbb{L}[y^H](\gamma^H)]_+ \frown \mathbb{L}[y^K](\gamma^K)\\
&\quad \vee \vee(\mathbb{L}[y^G](\gamma^G) \frown \mathbb{L}[y^H](\gamma^H) \frown \mathbb{L}[y^K](\gamma^K))
\end{aligned}
$$

we end up with the same structure but with x^X replaced by y^X in the liftings. It follows from the previous proof of associativity that the witness α_R can be used to show each disjoint in $\gamma^{(K\circ H)\circ G}$ is compatible with the corresponding disjoint in $\gamma^{K\circ(H\circ G)}$. This suffices to complete the proof. □

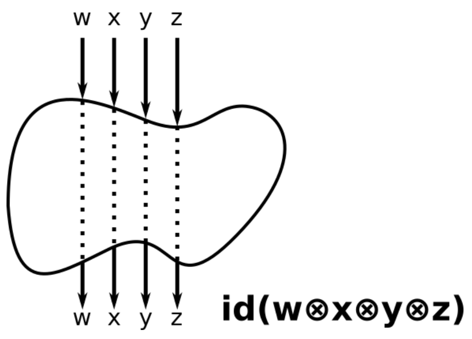
Constants

Because sequential composition only allows the composition of two OEGs with an exactly matching outgoing event interface and incoming event interface, additional primitive structural elements are necessary to compose OEGs in a more general fashion. *Identity* OEGs consists only of dependency edges, and are used to allow edges to bypass OEGs. This is achieved through composing OEGs in parallel with identities on either side.

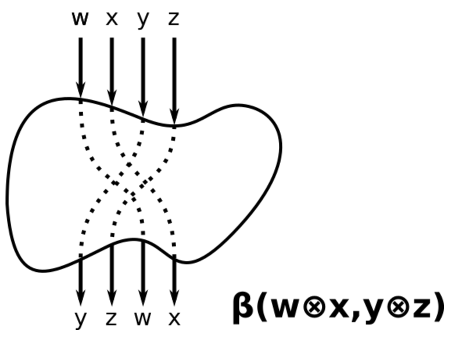
Definition 36. *OEG Identities*
For any OES Σ and any $a \in \mathbf{Mon}(\mathbb{T}^\Sigma)$, the corresponding identity OEGs is

$$\mathbf{id}_\Sigma(a) \overset{\text{def}}{=} \langle \emptyset, \mathbf{C}_\emptyset, (a, a), \gamma^{\|a\|} \rangle$$

Indeed, these identity OEGs are the identity elements of sequential composition.



(a) Identity OEGs.



(b) Braiding OEGs.

Figure 7.14: Constant OEGs.

Proposition 15. *For any OEG G over an OES Σ,*

$$G = \mathbf{id}_\Sigma(\mathcal{I}_+^G) \circ G = G \circ \mathbf{id}_\Sigma(\mathcal{I}_-^G)$$

Proof. Starting with the first identity, given a concrete presentation

$$G = \langle E^G, \mathbf{C}^G, \mathcal{I}^G, \gamma^G \rangle$$

the construction for the product is

$$\mathbf{id}_\Sigma(\mathcal{I}_+^G) \circ G = \langle G \amalg \emptyset, \mathbf{C}^G \triangledown \mathbf{C}_\emptyset, (\mathcal{I}_-^G, \mathcal{I}_+^G), \gamma^{\mathbf{id}_\Sigma(\mathcal{I}_+^G) \circ G} \rangle$$

There is a canonical bijection

$$\alpha_\emptyset : G \to G \amalg \emptyset$$

the inverse of which $\alpha_\emptyset^{-1}$ will serve as our witness to equivalence.

Clearly,

$$\mathbf{C}^G = \mathbf{C}^G \triangledown \mathbf{C}_\emptyset \circ \alpha_\emptyset^{-1}$$

and $\mathcal{I}^G = (\mathcal{I}_-^G, \mathcal{I}_+^G)$.

Expanding the definition for the γ element

$$\gamma^{\mathbf{id}_\Sigma(\mathcal{I}_+^G) \circ G} = [\mathbb{L}[\iota_L^{G \amalg \emptyset}](\gamma^G)]_- \vee [\mathbb{L}[\iota_R^{G \amalg \emptyset}](\gamma_\Sigma^{\mathbf{id}}(\mathcal{I}_+^G))]_+ \\ \vee \mathbb{L}[\iota_L^{G \amalg \emptyset}](\gamma^G) \frown \mathbb{L}[\iota_R^{G \amalg \emptyset}](\gamma_\Sigma^{\mathbf{id}}(\mathcal{I}_+^G))$$

The term $[\gamma_\Sigma^{\mathbf{id}}(\mathcal{I}_+^G)]_{+/-}$ = false from the definition, since every pair in the relation is to a boundary dependency site, hence the expansion can be reduced to

$$= [\mathbb{L}[\iota_L^{G \amalg \emptyset}](\gamma^G)]_- \vee \mathbb{L}[\iota_L^{G \amalg \emptyset}](\gamma^G) \frown \mathbb{L}[\iota_R^{G \amalg \emptyset}](\gamma_\Sigma^{\mathbf{id}}(\mathcal{I}_+^G))$$

Expanding the lifted $\gamma_\Sigma^{\mathbf{id}}(\mathcal{I}_+^G)$ term

$$\begin{aligned}&\mathbb{L}[\iota_R^{G \amalg \emptyset}](\gamma_\Sigma^{\mathbf{id}}(\mathcal{I}_+^G)) \Leftrightarrow \exists\, f_1,\, f_2 \cdot \\ &\quad e_1 = {\iota_R^{G \amalg \emptyset}}_\star(f_1) \wedge e_2 = {\iota_R^{G \amalg \emptyset}}_\star(f_2) \wedge n = m \wedge f_1 = \star \wedge f_2 = \star\end{aligned}$$

Because the liftings of the injections preserve units, and units always exist, $e_k = \star$ and the existential can be eliminated, leaving

$$\Leftrightarrow n = m \wedge e_1 = \star \wedge e_2 = \star$$

Substituting this into the expression for $\gamma_\Sigma^{\mathbf{id}}(\mathcal{I}_+^G)$, one gets

$$\begin{aligned}&\gamma_\Sigma^{\mathbf{id}}(\mathcal{I}_+^G)((e_1,\, n),\, (e_2,\, m)) \Leftrightarrow [\mathbb{L}[\iota_L^{G \amalg \emptyset}](\gamma^G)]_-((e_1,\, n),\, (e_2,\, m)) \\ &\quad \vee \exists\, k : \mathbb{N} \cdot \mathbb{L}[\iota_L^{G \amalg \emptyset}](\gamma^G)((e_1,\, n),\, (\star,\, k)) \wedge (k = m \wedge \star = \star \wedge e_2 = \star) \\ &\Leftrightarrow [\mathbb{L}[\iota_L^{G \amalg \emptyset}](\gamma^G)]_-((e_1,\, n),\, (e_2,\, m)) \vee (\mathbb{L}[\iota_L^{G \amalg \emptyset}](\gamma^G)((e_1,\, n),\, (e_2,\, m)) \wedge e_2 = \star) \\ &\Leftrightarrow (\mathbb{L}[\iota_L^{G \amalg \emptyset}](\gamma^G)((e_1,\, n),\, (e_2,\, m)) \wedge e_2 \neq \star) \vee (\mathbb{L}[\iota_L^{G \amalg \emptyset}](\gamma^G)((e_1,\, n),\, (e_2,\, m)) \wedge e_2 = \star) \\ &\Leftrightarrow \mathbb{L}[\iota_L^{G \amalg \emptyset}](\gamma^G)((e_1,\, n),\, (e_2,\, m))\end{aligned}$$

Using the same result from the previous proofs, it follows that

$$\gamma^G((e_1,\, n),\, (e_2,\, m)) \Leftrightarrow \mathbb{L}[\iota_L^{G \amalg \emptyset}](\gamma^G)(({\alpha_\emptyset^{-1}}_\star(e_1),\, n),\, (({\alpha_\emptyset^{-1}}_\star(e_2),\, m))$$

The second identity is proven in precisely the same fashion with some of the details reversed in order. □

Braiding elements are also added to explicitly reorder dependency sites, taking a word of dependency types a and exchanging them with a word b.

Definition 37. *OEG Braidings*
For any OES Σ and any $a,\, b \in \mathbf{Mon}(\mathbb{T}^\Sigma)$, the braiding OEGs are

$$\beta_\Sigma(a,\, b) \overset{def}{=} \langle \emptyset,\, \mathbf{C}_\emptyset,\, (a \otimes b,\, b \otimes a),\, \gamma^{(\|a\|,\, \|b\|)} \rangle$$

It should be the case that performing this operation is reversible, that one can exchange the order of a and b, then return to the original order by exchanging b and a subsequently.

Proposition 16. *For any OES Σ and any pair of words $a,\, b \in \mathbf{Mon}(\mathbb{T}^\Sigma)$*

$$\beta_\Sigma(b,\, a) \circ \beta_\Sigma(a,\, b) = \mathbf{id}_\Sigma(a \otimes b)$$

Proof. Expanding the two terms in the LHS

$$\begin{aligned}\beta_\Sigma(a,\, b) &= \langle \emptyset,\, \mathbf{C}_\emptyset,\, (a \otimes b,\, b \otimes a),\, \gamma^{(\|a\|,\, \|b\|)} \rangle \\ \beta_\Sigma(b,\, a) &= \langle \emptyset,\, \mathbf{C}_\emptyset,\, (b \otimes a,\, a \otimes b),\, \gamma^{(\|b\|,\, \|a\|)} \rangle\end{aligned}$$

the composition of which is

$$\beta_\Sigma(b,\, a) \circ \beta_\Sigma(a,\, b) = \langle \emptyset \amalg \emptyset,\, \mathbf{C}_\emptyset \triangledown \mathbf{C}_\emptyset,\, (a \otimes b,\, b \otimes a),\, \gamma' \rangle$$

where

$$\gamma' = [\gamma^{(\|a\|,\, \|b\|)}]_- \vee [\gamma^{(\|b\|,\, \|a\|)}]_+ \vee \gamma^{(\|a\|,\, \|b\|)} \frown \gamma^{(\|b\|,\, \|a\|)}$$

Using the properties of 10, this relation can be reduced as follows

$$\begin{aligned} &= \gamma^{(\|a\|,\, \|b\|)} \frown \gamma^{(\|b\|,\, \|a\|)} \\ &= \gamma^{\|a\|+\|b\|} \end{aligned}$$

Expanding the RHS

$$\mathbf{id}_\Sigma(a,\, b) = \langle \emptyset,\, \mathbf{C}_\emptyset,\, (a \otimes b,\, a \otimes b),\, \gamma^{\|a\|+\|b\|} \rangle$$

Since the empty sets of events are isomorphic and the **C** components are consistent, the event interfaces of the LHS and RHS are equivalent, and as we have shown the networks are equivalent, the identity holds. □

Finally, as a matter of algebraic completeness, the unit OEG can be defined as the completely empty one. Since it is empty, the unit has the monoidal unit $\mathbb{1}$ as its incoming and outgoing event interface. Since this OEG is essentially the same for all OESs, we can notate it as $\mathbb{1}$.

Definition 38. *The Unit OEG*
The unit OEG $\mathbb{1}$ is defined

$$\mathbb{1} \overset{def}{=} \langle \emptyset,\, \mathbf{C}_\emptyset,\, (\mathbb{1},\, \mathbb{1}),\, \mathit{false} \rangle$$

As one might expect, this unit is specifically the unit of parallel composition.

Proposition 17. *For any OEG G (over any OES),*

$$G \otimes \mathbb{1} = \mathbb{1} \otimes G = G$$

Proof. Starting with the first identity, given a concrete presentation

$$G = \langle E^G,\, \mathbf{C}^G,\, \mathcal{I}^G,\, \gamma^G \rangle$$

the construction for the product is

$$\mathbb{1} \otimes G = \langle G \amalg \emptyset,\, \mathbf{C}^G \triangledown \mathbf{C}_\emptyset,\, (\mathbb{1} \otimes \mathcal{I}^G_-,\, \mathbb{1} \otimes \mathcal{I}^G_+),\, \gamma^G \vee \text{false} \uparrow \mathcal{I}^G \rangle$$

Since

$$\text{false} \uparrow \mathcal{I}^G \;\Leftrightarrow\; \text{false}$$

it follows that

$$\mathbb{1} \otimes G = \langle G \amalg \emptyset,\, \mathbf{C}^G \triangledown \mathbf{C}_\emptyset,\, (\mathcal{I}^G_-,\, \mathcal{I}^G_+),\, \gamma^G \rangle$$

From the previous proof, it follows that this presentation is equivalent to G. □

It is also the case that this unit is equivalent to the identity for the monoidal unit of event interfaces.

Proposition 18. *For any OES Σ, $\mathbb{1} = \mathbf{id}_\Sigma(\mathbb{1}_{\mathbf{Mon}(\mathbb{T}^\Sigma)})$.*

Proof. This follows immediately from expanding the definition of the RHS. □

Furthermore, the identity for a monoidal product of event interface is the parallel product of the identities of each event interface.

Proposition 19. *For any OES Σ, and any pair a, $b \in \mathbf{Mon}(\mathbb{T}^\Sigma)$*

$$\mathbf{id}_\Sigma(a \otimes b) = \mathbf{id}_\Sigma(a) \otimes \mathbf{id}_\Sigma(b)$$

Proof. Expanding the RHS of the identity with the definition of parallel composition

$$\mathbf{id}_\Sigma(a) \otimes \mathbf{id}_\Sigma(b) = \langle \emptyset \amalg \emptyset,\, \mathbf{C}_\emptyset \amalg \mathbf{C}_\emptyset,\, (a \otimes b,\, a \otimes b),\, \gamma^{\|a\|} \vee \gamma^{\|b\|} \uparrow (\|a\|,\, \|a\|) \rangle$$

Using identities on networks

$$\gamma^{\|a\|} \vee \gamma^{\|b\|} \uparrow (\|a\|,\, \|a\|) = \gamma^{\|a\| + \|b\|} = \gamma^{\|a \otimes b\|}$$

Given the expansion of the LHS is

$$\mathbf{id}_\Sigma(a \otimes b) = \langle \emptyset \amalg \emptyset,\, \mathbf{C}_\emptyset \amalg \mathbf{C}_\emptyset,\, (a \otimes b,\, a \otimes b),\, \gamma^{\|a\| \otimes \|b\|} \rangle$$

the two presentations are clearly isomorphic. □

The conjunction of these last two identities amounts to a lifting of the monoid $\mathbf{Mon}(\mathbb{T}^\Sigma)$ into a monoid of identity OEGs, parameterized by $\mathbf{Mon}(\mathbb{T}^\Sigma)$.

Category of OEGs

As one might anticipate from what has been discussed in Chapter 5, this language of operators and constants forms the algebraic structure of a symmetric monoidal category for the set of OEGs built from a particular OES. Indeed, it can be proven that this is the case, completing the set of axioms we have already established for this algebra with the remaining necessary to establish that OEGs form a symmetric monoidal category.

Theorem 9. *OEG symmetric monoidal category*
Given a particular OES Σ *the tuple*

$$\mathcal{OEG}(\Sigma) \overset{def}{=} \big(\mathbf{Mon}(\mathbb{T}),\, \mathrm{OEG}^{\Sigma},\, \circ\,,\, \mathbf{id}_{\Sigma},\, \otimes\,,\, \mathbb{1},\, \beta_{\Sigma}\big)$$

defines a (strict) symmetric monoidal category, where $\circ$ *and* $\otimes$ *are the sequential and parallel compositions of OEGs and* $\mathbb{1} = \mathbf{id}_{\Sigma}(\mathbb{1}_{\mathbf{Mon}(\mathbb{T})})$. *The identities and braidings are defined as above. The set* OEG^{Σ} *is, here, broken up into homsets* $\mathrm{OEG}^{\Sigma}(a,\, b)$ *for each interface* $(a,\, b) \in \mathcal{I}^{\Sigma}$.

Proof. Most of the axioms for symmetric monoidal categories have been proven. The remaining axioms can be proven in a similar fashion to those above, choosing concrete representations of operands, expanding the definitions of operations, and making use of the properties of networks. □

In this category, $\mathcal{OEG}(\Sigma)$, the objects of the category are sequences of dependency types and the OEGs themselves are the morphisms. This theorem reflects the similar result from Joyal and Street [27], with regards to topological graphs embedded into a space.

That OEGs of a particular OES form a symmetric monoidal category implies that OEGs can be built up with compositions into larger OEGs within this category. Which would mean that we could begin with a collection of OEGs and take their closure under these operations to generate a symmetric monoidal categories containing all of their compositions. However, for the case of finite OEGs, we would also like this construction process to have a clear foundation of generating elements of the simplest form. Specifically, we would like to show that the generating OEGs are the ones containing only a single event of an event type in the OES. This way, any OEG could be constructed from atomic, single event OEGs, identity and braiding elements, and the two basic compositions. This is what we will derive from the work of Joyal and Street [27].

But first we will look at a couple relevant special kinds of OEGs that will serve an important purpose in this construction, and which are important in their own right.

Special OEGs

Relevant to our aim to build up OEGs from primitive elements are two special classes of OEGs, the first of which are *atomic* OEGs, which lift each of the event types of a particular OES into an OEG containing a single event of that event type.

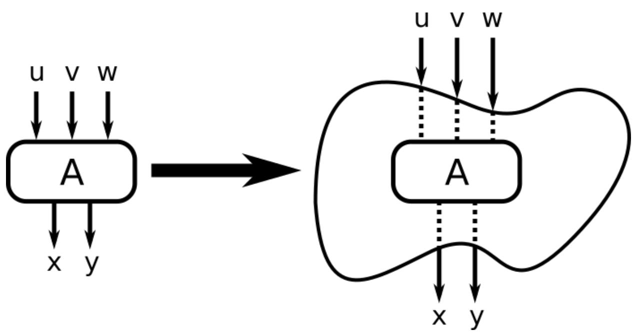


Figure 7.15: An atomic OEG

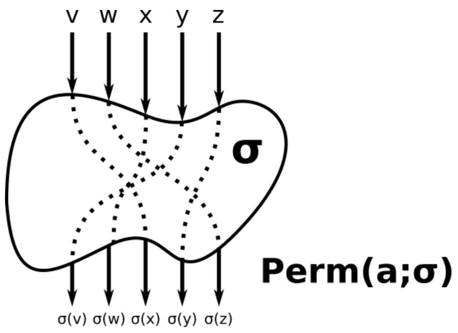


Figure 7.16: A permutation OEG.

Definition 39. *Atomic OEGs*
Given an OES Σ *and an event type* $A \in \mathcal{A}(a, b)$ *for any* $a, b \in \mathbf{Mon}(\mathbb{T})$*, the atomic OEG is defined*

$$\langle\langle A\rangle\rangle \overset{def}{=} \langle\{A\}, \{A \mapsto A\}, (a, b), \gamma^e\rangle$$

where

$$\gamma^e((x, n), (y, m)) \overset{def}{=} n = m \wedge \\ ((x = \star \wedge y = e \wedge 0 \leq n < \|e\|_{\_}) \vee (x = e \wedge y = \star \wedge 0 \leq n < \|e\|_{\_}))$$

In order to distinguish the instance from the type, the event type e is wrapped suggestively in brackets, $\langle\langle e\rangle\rangle$, indicating a lifting similar to that of equivalence class lifting of primitive blocks by η into the free symmetric monoidal categories. These two liftings from the same domain, an OES, will ultimately be tied together, though it is important not to prematurely conflate them.

The other special class of OEGs are those that simply permute a collection of incoming and outgoing dependencies. For these OEGs the incoming and outgoing event interfaces are, as words of $\mathbf{Mon}(\mathbb{T})$ in the OES, permutations of each others primitive types. We will represent permutations as bijective functions σ, ρ, ... of a particular natural number $\|\sigma\|$, ... For any given $a \in \mathbf{Mon}(\mathbb{T})$ and permutation σ such that $\|\sigma\| = \|a\|$, we will denote the permutation of elements in a by σ as a_σ. A permutation OEG $\mathbf{Perm}(a;\, \sigma)$ is defined using precisely these two pieces of information.

Definition 40. *Permutation OEGs Given a OES* $(\mathbb{T},\, \mathcal{A})$*, a word* $a \in \mathbf{Mon}(\mathbb{T})$*, and a permutation* σ*, such that* $\|a\| = \|\sigma\|$*, a permutation OEG is defined*

$$\mathbf{Perm}(a;\, \sigma) \overset{def}{=} \langle \emptyset,\, \mathbf{C}_\emptyset,\, (a,\, a_\sigma),\, \gamma^\sigma \rangle$$

where

$$\gamma^\sigma((e_1,\, n_1),\, (e_2,\, n_2)) \overset{def}{=} e_1 = e_2 = \star \wedge n_2 = \sigma(n_1) \wedge 0 \leq n_1 < \|\sigma\|$$

Another way of describing these OEGs is by the fact that they have no events. In fact, any OEG without events is a permutation.

Proposition 20. *An OEG G has no events if and only if it is a permutation OEG. Specifically,*

$$G = \mathbf{Perm}(\mathcal{I}^G_-;\, \sigma)$$

for some σ. We will notate this permutation

$$\sigma = \mathbf{ParsePerm}(G)$$

Proof. Given a G without events,

$$G = \langle \emptyset,\, \mathbf{C}_\emptyset,\, \mathcal{I},\, \gamma \rangle$$

since γ is bijective, the permutation can be defined

$$\sigma(n) = m, \text{ where } \gamma((\star,\, n),\, (\star,\, m))$$

Because γ must connect each primitive type in $\mathcal{I}_-$ to $\mathcal{I}_+$, $\mathcal{I}_+ = \mathcal{I}_{-_\sigma}$. □

It is a consequence of the result from Joyal and Street that we will discuss in the following section that these permutation OEGs can always be constructed completely from identities and braidings. Doing this in practice, however, is tedious, and thus these permutations can be treated as an alternative basis for reordering incoming and outgoing dependencies so that they match under sequential composition in the desired way. Another way of looking at the space of permutations is that they constitute the sub-free symmetric monoidal category of any free symmetric monoidal category generated entirely from constants. Permutations are therefore closed under compositions with each other.

From OEGs to Free Symmetric Monoidal Categories

We can now move on to what is potentially the most important property of OEGs. Thus far, we have established that the collection of OEGs over an OES Σ form a symmetric monoidal category, $\mathcal{OEG}(\Sigma)$. Using the aforementioned result from [27], we can go further and say that any $\mathcal{OEG}(\Sigma)$ is specifically a free symmetric monoidal categories. Their specific result regards the symmetric monoidal category, $\mathbb{F}_S(\Sigma)$, the objects of which are the words in $\mathbf{Mon}(\mathbb{T}^{\Sigma})$ and the morphisms of which are of isomorphism classes of anchored progressive polarized diagrams built up from the elements of $\mathcal{A}^{\Sigma}_{\cup}$. As we have mentioned, aside from minor details, our construction for OEGs are anchored progressive polarized diagrams.

To be more specific about the differences, given a concrete OEG

$$(E,\, \mathbf{C},\, \mathcal{I},\, \gamma)_{\Sigma}$$

the topological components $(E,\, \gamma)$ form an anchored progressive polarized graph, with E as its vertices and two vertices e_1 and e_2 having a directed edge when, for any pair $n,\, m \in \mathbb{N}$, $\gamma((e_1,\, n),\, (e_2,\, m))$. To clarify the complete qualifier *anchored progressive polarized*, progressive is synonymous with acyclic (notwithstanding the $\star$ vertex, if one interprets it as such), which γ is by definition; polarized means that the incoming and outgoing edges to each vertex in E, which γ does with the additional dependency port indices n and m; and, anchored means that the incoming and outgoing edges of the graph are ordered, which is also accomplished with the pairs in γ involving the $\star$ value.

The remaining two elements of the OEG, $(\mathbf{C},\, \mathcal{I})$, constitute what is referred to in [27] as a *valuation* over an anchored progressive polarized graph, which is a mapping from the vertices and edges of the graph into the primitive blocks and primitive types of Σ. $\mathbf{C}$ very directly gives the latter mapping, while the former can be constructed from γ, by referring to the event interfaces attached to the vertices on either side of an edge, or to the elements in $\mathcal{I}$. The combination of the graph and the valuation is what is called the *anchored progressive polarized diagram.*

Furthermore, again with minor differences in details, our definitions for the operators and constants of $\mathcal{OEG}(\Sigma)$ match those of $\mathbb{F}_S(\Sigma)$, although we go into much greater details in constructing them. In essence, our $\mathcal{OEG}(\Sigma)$ is equivalent to their $\mathbb{F}_S(\Sigma)$, and therefore we are justified in stating their result as follows.

Theorem 10. *Joyal and Street [27] For any OES Σ, $\mathcal{OEG}(\Sigma)$ is a free symmetric monoidal category.*

Since free elements are unique up to isomorphism, it follows immediately from this that for all monoidal schemes Σ

$$\mathcal{OEG}(\Sigma) \cong \mathbf{SymMonCat}(\Sigma)$$

The important ramification of this isomorphism, connecting the two constructions for $\mathcal{OEG}(\Sigma)$ given in Chapter 7 and $\mathbf{SymMonCat}(\Sigma)$ given in Chapter 6, is that every OEG built from elements of Σ can also be constructed as a term in the language of symmetric monoidal categories involving only constants and atomic OEG. This is the case because every morphism in $\mathbf{SymMonCat}(\Sigma)$ was constructed inductively using only constants and elements in the image of η^{Σ}. Since η^{Σ} is isomorphic to the atomic OEG lifting of $\mathcal{A}^{\Sigma}$, which we have been notating as $\langle\langle - \rangle\rangle$, the image of η^{Σ} are isomorphic to the atomic OEGs.

Another way of phrasing this important corollary is that the language of symmetric monoidal categories with a monoidal schemes is complete in its ability to define OEGs. It was most certainly not a given that in constructing such a language that this entire space would be covered. If we had originally widened the definition of OEGs to accommodate infinite numbers of events this certainly would not have been the case.

Chapter 8

OEG Semantics for Process Calculi

In order to use any representation of behavior as a basis for working with substantial applications, it must be possible to construct representations precisely from the specification of the applications as programs. This is done through giving a set of programs in a particular language a formal semantics that maps each program in the language to an appropriate representation of what happens when the program executes. The most conventional method of constructing this kind of semantics is to give the language what is often called an *operational semantics*. This kind of semantics emphasizes the operational steps involved in the execution of a program, rather than the denotational value the computation computes. Therefore, this kind of semantics is more a means to produce a record of behavior than a formulaic reduction of the program to a function.

The most notable form that operational semantics takes is the *structural operational semantics* of Plotkin [54], which conceives of an operational step in the execution of a program as a change in the form of a program and its accompanying data. In the simplest cases, this amounts to a syntactic rewriting system for the programming language, constructed via a set of inductive rules. In this case, the state of the system is represented by the continuation of the program, which gets further evaluated with each rewriting. The rewritings constitute a transition system, and the result is that a program operationally forms a family of sequential *traces*.

$$P_1 \longrightarrow P_2 \longrightarrow P_3 \longrightarrow P_4 \longrightarrow \dots$$

In essence, the resulting representation of behavior is a sequence of states or programs

$$P_1,\ P_2,\ P_3,\ P_4,\ \dots$$

Concurrent programming languages or abstract calculi for concurrent systems are often defined with such an operational semantics that constructs sequential traces as their behavioral representations. An example of an abstract calculi for concurrent systems is the π-calculus of Milner, Parrow, and Walker [48, 47], which is thought of by many as the λ-calculus of dynamic concurrent systems. In spite of being the superlative calculus of concurrency, nevertheless, the formal presentation of this calculus was originally, and is still conventionally, given as a structural operational semantics. Each term in the calculus produces a set of possible reductions or interactions that define the behavior of execution judged from the perspective of a sequential observer; or for that matter, the possible sequential steps involved in simulating the concurrent process on a sequential computer.

As is common in concurrent calculi, two programs P and Q can be placed into a concurrent composition $P \talloblong Q$, which is intuitively conceived of as a program in which each of the two programs can proceed to execute asynchronously alongside one another. If there exists an operational form of progress that P can make to a new program P', that is there exists a transition $P \longrightarrow P'$ in the structural operational semantics, as well as a transition $Q \longrightarrow Q'$ indicating progress Q can make concurrently, one would expect the semantics to reflect this notion of concurrent yet asynchronous progress directly in the behavioral representation of $P \talloblong Q$.

However, the idiosyncratic method of expressing the operational semantics of $P \talloblong Q$ is characterized in the following two induction rules

$$\frac{P \longrightarrow P'}{P \talloblong Q \longrightarrow P' \talloblong Q} \qquad \text{and} \qquad \frac{Q \longrightarrow Q'}{P \talloblong Q \longrightarrow P \talloblong Q'}$$

These state, in essence, that there is a nondeterministic choice between $P \longrightarrow P'$ happening first or $Q \longrightarrow Q'$ happening first. Subsequently, the other of the two programs could take a step in execution, or just as well, the same program that took the first step could take an additional one while the other remains unchanged. In this fashion, the concurrent execution of P and Q is represented as the nondeterministic interleaving of the steps in their respective sequential execution. Supposing each of P and Q can take several independent steps, many different interleavings exist.

$$\begin{array}{l}
P_0 \talloblong Q_0 \longrightarrow P_1 \talloblong Q_0 \longrightarrow P_1 \talloblong Q_1 \longrightarrow \ldots \\
P_0 \talloblong Q_0 \longrightarrow P_0 \talloblong Q_1 \longrightarrow P_1 \talloblong Q_1 \longrightarrow \ldots \\
P_0 \talloblong Q_0 \longrightarrow P_1 \talloblong Q_0 \longrightarrow P_2 \talloblong Q_0 \longrightarrow \ldots \\
P_0 \talloblong Q_0 \longrightarrow P_0 \talloblong Q_1 \longrightarrow P \talloblong Q_2 \longrightarrow \ldots \\
\ldots \quad \text{and so forth}
\end{array}$$

This convention of using interleaving semantics for concurrent process calculi has existed at least since the predecessor of π-calculus, CCS, was devised much earlier by Milner [46].

In contrast with this convention of giving interleaved semantics to concurrent process calculi, the aim of this thesis has been to propose a true concurrent behavioral representation based on monoidal categories, the OEG, and argue its merits for what we called HDDAs running on platforms that we have termed Process Field. This argument has contrasted OEGs with both interleaved sequential representations, as well as representations based on generalized sequences such as Event Structures and pomsets. It would only make sense to demonstrate how OEGs can be used to give a modular, compositional, true concurrent semantics to process calculi such as the π-calculus.

In this chapter, we will give an overview of the π-calculus and an account of its conventional structural operational semantics. With a couple important modifications, we will then proceed to give π-calculus an OEG semantics.

8.1 The π-*calculus*

The π-calculus is a process calculus aimed at reasoning about concurrent processes, particularly in the context of *mobility*, where the topology of communication channels between concurrent process can evolve over the course of its execution. This kind of evolution arises when modeling a system in which the concurrently operating components may be physically moving, thus changing the topology of their communication. However, this calculus has been shown to be extremely general in the breadth of what it can model, from data structures, protocols, and object oriented programming models [47] to business processes [37]. It is nevertheless basic and elegant enough to serve as a foundational language for studying concurrent processes.

This calculus was first developed in [48], by Milner, Parrow, and Walker, elaborating upon the earlier concurrent formalism, CCS, devised by Milner. Many variants have also been devised, such as the *asynchronous* π-calculus of Boudol [9]. The relationship between a number of these variants is given in detail in [51]. Whereas the fundamental atomic action of λ-calculus is *application*, in which one subterm is applied to another, the fundamental operation of π-calculus is message passing, in which one subterm passes a message to another. In this respect, the π-calculus resembles the Actor model

Just as λ-calculus has been instrumental in developing the semantics of functional programming languages and models, likewise π-calculus plays an instrumental role in developing the semantics of concurrent programming models. Consequently, defining the semantics of this calculus in terms of OEGs not only exemplifies the use of OEGs, but further, it should give a sense of how one could give OEG semantics to many other concurrent languages, particularly those inspired by π-calculus. At the least, one should be able to give several useful variants of π-calculus OEG semantics following the schema of what will be shown here.

As we have stated, the convention for defining the semantics of π-calculus is to give it a structural operational semantics. More specifically, two variants of these kinds of semantics are usually developed for a calculus, both detailed in [47]. One of these semantics is a *reduction semantics*, the transitions of which rewrite the terms of the calculus, reducing them. In this case, the term in the calculus is given a complete semantics as a closed system, containing all of its concurrent components, and the transition is the reduction of the whole system. The second semantics is the Labeled Transition System *(LTS) semantics*, which can give semantics to an open fragment of a system that may either send or receive a message as a transition action.

Nevertheless, in both of these semantics, the representations of behavior are interleaved sequential ones. A reduction trace can therefore happen multiple ways, reflected in the nondeterministic choices in the induction rules of the reduction relation. Concurrency, therefore, is reflected, not in each behavior, but rather in the possibility of different behaviors nondeterministically produced in the derivations. Alternatively, one can consider the behavior proper to be the collection of all such sequential behaviors. This kind of semantics brings up the concerns raised in Chapter 4. Yet, the mathematical conventions for dealing with observational or behavioral process equivalence via *bisimulation* rest on the presumption of this kind of semantics.

We will begin presenting π-calculus with the conventional semantics and subsequently show how these semantics can be modified to produce OEGs rather than traces.

Syntax

The terms of π-calculus represent systems of concurrent processes in a particular state described by the term. The syntax of these terms is given by induction over three syntactic sets $\mathcal{G}_\pi$, representing process guards, $\mathcal{N}_\pi$ representing choices between processes, and $\mathcal{T}_\pi$ representing the proper process terms. The syntax involves the use of a set of variables $\mathcal{V}_\pi$ that have the dual role of representing both variables in the traditional sense, that are bound to particular values, and the names of channels through which messages are passed between concurrent components of a process.

Definition 41. *π-calculus Syntax*
Given a set of variables $\mathcal{V}_\pi$, the set of π-calculus guards $\mathcal{G}_\pi$, terms $\mathcal{T}_\pi$, and normal forms $\mathcal{N}_\pi$ are defined inductively as follows

$$\frac{}{\tau \in \mathcal{G}_\pi} \qquad \frac{v \in \mathcal{V}_\pi \qquad \mathbf{x} \in \mathbf{Mon}(\mathcal{V}_\pi)}{\bar{v}\langle \mathbf{x} \rangle \in \mathcal{G}_\pi} \qquad \frac{v \in \mathcal{V}_\pi \qquad \mathbf{x} \in \mathbf{Mon}(\mathcal{V}_\pi)}{v(\mathbf{x}) \in \mathcal{G}_\pi}$$

$$\frac{N \in \mathcal{N}_\pi}{N \in \mathcal{T}_\pi} \qquad \frac{v \in \mathcal{V}_\pi \qquad P \in \mathcal{T}_\pi}{\{v\} \bullet P \in \mathcal{T}_\pi} \qquad \frac{P, Q \in \mathcal{T}_\pi}{P \,[\!]\, Q \in \mathcal{T}_\pi} \qquad \frac{P \in \mathcal{T}_\pi}{P! \in \mathcal{T}_\pi}$$

$$\frac{}{\emptyset \in \mathcal{N}_\pi} \qquad \frac{\pi \in \mathcal{G}_\pi \qquad P \in \mathcal{T}_\pi}{\pi \bullet P \in \mathcal{N}_\pi} \qquad \frac{P, Q \in \mathcal{N}_\pi}{P + Q \in \mathcal{N}_\pi}$$

The two primary connectives that structure these terms are $[\!]$ and !. The former combines two processes in parallel, forming a concurrent combination. In this concurrent combination $P \,[\!]\, Q$, each process can execute independent of the other. That is if P can go through a transition that does not depend on Q it can occur without changing Q. The processes can interact with each other. For instance, P could send a message to Q, causing both processes to synchronously transition. We will call the operands of a product its *concurrent components*.

The replication operator !, applied to a process $P!$ indicates that the process P can be replicated any number of times in parallel to react concurrently by itself or with other terms. In a sense, $P!$ can be thought of as an infinite $[\!]$-product of P's, although this raises some important absurdities that will be discussed later.

Putting aside the syntactic case of $\{v\}_\bullet P$ for a moment, when a process P is not a product or repetition of some other process, it is a *reactant* in $\mathcal{N}_\pi$, and thus a $+$-product (sum) of processes guarded by members of $\mathcal{G}_\pi$

$$\pi_{1\bullet} P_1 + \pi_{2\bullet} P_2 + \ldots + \pi_{N\bullet} P_N$$

These summands represent mutually exclusive choices for a process to reaction corresponding to the guard π_k of each summand P_k. The possible reactions, represented by each kind of guard, are an internal reaction τ, a send $\bar{v}\langle\mathbf{x}\rangle$ of the sequence of values $\mathbf{x}$ along the channel named v, and a receive $v(\mathbf{x})$ of a sequence of values on channel v, which are then bound to variables $\mathbf{x}$ in the guarded process.

The empty sum, with no summands, captured by the term $\emptyset$, represents a terminating process. In the case where $\emptyset$ is guarded $\pi_\bullet \emptyset$, we will syntactically notate this as π, leaving out the $\emptyset$.

More specifically, if there is a reactant $\pi_\bullet P$ in a sum of reactants

$$\pi_\bullet P + N$$

and the reaction that π represents occurs, the process P replaces the whole sum as its continuation. In the specific case of reception, the received names are also substituted in P. For a particular reactant, τ may always happen, whereas a send or receive can only happen if the corresponding operation is occurring somewhere else, either in another concurrent component or in the environment (depending on whether the semantics is formulated in a closed or open fashion). This kind of semantics is often referred to as *rendezvous*, because the send and receive operations must rendezvous with each other synchronously. In contrast, dataflow models of computation often involve channels that can be sent to independent of whether a receive can synchronously occur with it.

In order to emphasize the intuitive meaning of a reaction (which will later be formalized), we consider the following examples.

Example 1. *Consider the following term.*

$$\bar{a}\langle b\rangle \parallel b(x) \parallel a(y)_\bullet \bar{y}\langle c\rangle + \tau_\bullet \bar{b}\langle d\rangle$$

In this example, there are three concurrent terms, the second of which involves two possible behavioral choices. There are possible reactions for this term. One involves the second reactant in the third term guarded by a τ, which can always react. If this reaction is taken the sum of reactants is reduced to $\bar{b}\langle d\rangle$, and the remaining term is

$$\bar{a}\langle b\rangle \parallel b(x) \parallel \bar{b}\langle d\rangle$$

Now that a send and receive guarded reactants for channel b exist in parallel with each other, the reaction of sending d along channel b can occur. Both guards are eliminated in this reaction, leaving terminating processes.

$$\bar{a}\langle b\rangle \parallel \emptyset \parallel \emptyset$$

Since these two latter components are terminated they can be thrown out, leaving

$$\bar{a}\langle b\rangle$$

The other reaction that could have taken place from the original term would be between the send and receive guarded reactants for channel a in the first and third components, respectively. The result of this reaction is that name b is sent on channel a and bound to variable y in the term guarded by the receive. Specifically,

$$\emptyset \,[]\, b(x) \,[]\, \bar{y}\langle c\rangle\{y \mapsto b\}$$

which reduces under substitution to

$$\emptyset \,[]\, b(x) \,[]\, \bar{b}\langle c\rangle$$

This creates the condition for the new third term to react with the second, resulting in

$$\emptyset \,[]\, \emptyset \,[]\, \emptyset$$

which, after removing terminating process is simply the empty process.

As one can see from this example, the reactions occurring for a term can be nondeterministic and lead to non-confluent contracta. This example simply demonstrates the semantics of reactions and illustrates some of the features of the calculus. The next example will be more interesting for our purposes.

Example 2. *Consider the following term.*

$$\bar{a}\langle b\rangle \bullet c(x) \,[]\, a(y) \bullet \bar{y}\langle d\rangle \,[]\, \bar{e}\langle f\rangle \bullet b(z) \,[]\, e(w)$$

In this example, one may notice immediately that the first two and second two components both have a potential reaction on channels a and e, respectively. If the reaction over channel a is taken, the result is

$$c(x) \,[]\, \bar{b}\langle d\rangle \,[]\, \bar{e}\langle f\rangle \bullet b(z) \,[]\, e(w)$$

This leaves the other reaction on channel e to take place.

$$c(x) \,[]\, \bar{b}\langle d\rangle \,[]\, b(z)$$

Finally, the reaction on channel b can occur. The final result is

$$c(x)$$

If the first reaction that happened was on channel e rather than e, the subsequent state would have been

$$\bar{a}\langle b\rangle \bullet c(x) \,[]\, a(y) \bullet \bar{y}\langle d\rangle \,[]\, b(z)$$

But after taking the reaction on channel a, the final two states would be the same.

What is illustrated in this example is a different kind of nondeterminism arising from the arbitrary order in which two concurrent reactions occurred. Unsurprisingly, in either order the result was the same; that is, the reactions commutated, forming a confluence. One could easily imagine loosening the hint of sequentiality in the intuitive notion of reduction and supposing that the two reductions both happened "concurrently", in the manner that one might imagine concurrency, where there is no notion of the orderings of the reductions at all. Rather, they both simply happen independent of each other.

When one pursues the conventional semantics for π-calculus and represents these reactions as transitions forming traces, the concurrency of reactions appearing in examples like the above are witnessed in the multiplicity of traces produced by the semantics, taking every nondeterministic choice. Here we encounter the problem considered in 4 with traces that we wish to overcome in a OEG semantics. Indeed, we will see later how the above example can be given a behavior that is indeed more intuitive. However, before pursuing this interpretation, we will proceed to complete the account of the more conventional semantics.

A final syntactic feature of π-calculus, the name scoping operator $\{x\}\bullet P$, which binds the name x, used either as a channel, or as a variables, into the scope of P. As in the case of λ-calculus, there are hence both free and bound names in each term. The scoping operator and receive both serve as binders. Appropriately, the set of free variables can be defined inductively for terms.

Definition 42. *The function* $\mathbf{FV}_\pi : \mathcal{T}_\pi \to \mathcal{V}_\pi$ *is defined inductively as follows*

$$\begin{aligned}
&\mathbf{FV}_\pi(P \mathbin{[\!]} Q) \overset{def}{=} \mathbf{FV}_\pi(P) \cup \mathbf{FV}_\pi(Q)\\
&\mathbf{FV}_\pi(P!) \overset{def}{=} \mathbf{FV}_\pi(P)\\
&\mathbf{FV}_\pi(\{x\}\bullet P) \overset{def}{=} \mathbf{FV}_\pi(P) \setminus \{x\}\\
&\mathbf{FV}_\pi(N + M) \overset{def}{=} \mathbf{FV}_\pi(N) \cup \mathbf{FV}_\pi(M)\\
&\mathbf{FV}_\pi(\emptyset) \overset{def}{=} \emptyset\\
&\mathbf{FV}_\pi(\tau\bullet P) \overset{def}{=} \mathbf{FV}_\pi(P)\\
&\mathbf{FV}_\pi(\bar{v}\langle\mathbf{x}\rangle\bullet P) \overset{def}{=} \mathbf{FV}_\pi(P) \cup \{v, \mathbf{x}\}\\
&\mathbf{FV}_\pi(v(\mathbf{x})\bullet P) \overset{def}{=} (\mathbf{FV}_\pi(P) \setminus \{\mathbf{x}\}) \cup \{v\}
\end{aligned}$$

As with many languages with bound variables, the standard notion of *α-conversion* conflates terms where a bound name has been changed to any other non-conflicting name. Equivalence under this conversion is denoted by $\cong_\alpha$. Syntactic equivalence between terms can therefore be generally weakened to this equivalence under which $\mathcal{T}_\pi$ becomes $\mathcal{T}_\pi/\cong_\alpha$.

Expanding on α-equivalence, a further notion of *structural congruence* is defined over $\mathcal{T}_\pi$ equivocating terms for intuitively clear reasons.

Definition 43. *π-calculus Structural Congruence*
The equivalence relation of structural congruence $\cong$ is defined over $\mathcal{T}_\pi$ as $\cong_\alpha$ along with the following axioms.

For $\llbracket$	$P \llbracket (Q \llbracket R) \cong (P \llbracket Q) \llbracket R$ $\emptyset \llbracket P \cong P \llbracket \emptyset \cong P$ $P \llbracket Q \cong Q \llbracket P$
For !	$P! \cong P \llbracket P!$
For +	$N + (M + K) \cong (N + M) + K$ $\emptyset + N \cong N + \emptyset \cong N$ $N + M \cong M + N$
For $\{\bullet\}$	$\{a\}_\bullet \{b\}_\bullet P \cong \{b\}_\bullet \{a\}_\bullet P$ $\{a\}_\bullet \emptyset \cong \emptyset$ $\{a\}_\bullet (P \llbracket Q) \cong \{a\}_\bullet P \llbracket Q$, *where* $a \notin \mathbf{FV}_\pi(Q)$

with $\cong$ incorporating the equivalence of α-conversion on bound variables.

The essence of the rules for both $\llbracket$ and + is that they both form commutative monoids over their terms with $\emptyset$ as the unit. Like many such identities, these express, in some sense, a superfluousness in the presentation of the syntax itself. Particularly, associativity and commutativity in theses cases indicate that the operators construct collections rather than proper sequences, let alone trees.

But in contrast, the rule for ! expresses something arguably more semantic. On one hand, it suggests that the term $P!$ is a compact presentation of a countably infinite product of Ps, as mentioned earlier. Yet in texts written about π-calculus, the notion of *replication* implies that copies are being produced (perhaps as-needed), indicating a kind of active event. This latter interpretation is the one we will intuitively lean on in constructing an OEG semantics.

The set of congruences involving the scoping operator are important for establishing the semantics, but are less of an important focal point for understanding the behavior that arises out of these. They are, in a much more obscure way, also, perhaps, a matter of syntactic superfluousness.[1]

Semantics

The formal semantics of π-calculus is typically given in two forms, as mentioned earlier. The first form, determining the reduction of closed terms, is called the *reduction semantics.* This semantics defines one rewrite relation $\longrightarrow$ over $\mathcal{T}_\pi$ that corresponds to one of the two kinds of reactions: internal τ reaction on one reactant, or the sending of names along a channel between two concurrent reactants guarded with $\bar{a}\langle\mathbf{b}\rangle$ and $a(\mathbf{x})$.

[1]What I am implying here is not that scoping is at all unnecessary, but rather that it is necessary because of how names function in concrete syntax.

Definition 44. *Reduction Semantics*
The operational semantics of evaluation for π-calculus is given by the rewrite relation $\longrightarrow$: $\mathcal{T}_\pi \to \mathcal{T}_\pi$ defined inductively by the following derivations

$$\frac{}{\tau_{\bullet} P + N \longrightarrow P}\ \textit{(Internal)}$$

$$\frac{}{(\bar{v}\langle \mathbf{x}\rangle_{\bullet} P + N) \mathbin{[\!]} (v(\mathbf{y})_{\bullet} Q + M) \longrightarrow P \mathbin{[\!]} Q[\mathbf{y} \mapsto \mathbf{x}]}\ \textit{(Communicate)}$$

$$\frac{P \longrightarrow P'}{P \mathbin{[\!]} Q \longrightarrow P' \mathbin{[\!]} Q}\ \textit{(Parallel)} \qquad \frac{P \longrightarrow P'}{\{v\}_{\bullet} P \longrightarrow \{v\}_{\bullet} P'}\ \textit{(Scoping)}$$

$$\frac{Q \cong P \qquad P \longrightarrow P' \qquad P' \cong Q'}{Q \longrightarrow Q'}\ \textit{(Structural)}$$

The two axioms of the inductive definition, *Internal* and *Communicate*, formalize the two basic cases of reactions in the calculus, while the other three inductive rules determine how and where the axioms can be applied. The *Parallel* rule conveys an important semantic idea that was illustrated in Example 2, that concurrent components of the term can react independently, concurrently. This rule is also an important basis for a revision in the later OEG semantics. In the form given here, it opens the door for the induction of a nondeterministic set of possible transitions that do not reflect an actual, essential or ontological, divergence in executions. Instead, the divergence is between different sequential epistemological records of the same execution. The *Scoping* rule is less illustrative and simply allows transitions to happen within scoped variable contexts.

The final *Structural* rule inductively closes the rewrite transitions to span over all structurally congruent terms. The congruences in the premise of the rule are suggestive, implying to some degree the idea that in order for the reaction $Q \longrightarrow Q'$ to take place, the Q process must be "converted" into P, and subsequent to the reduction step the P' must be converted into Q'. However, for many congruences, as discussed earlier, there is no ontological process of conversion when the congruence is, for instance, an association over one of the operators. Instead, the conversion is purely syntactic. On the other hand, the issue of the ontology of replication may give one pause about the congruence $P! \cong P \mathbin{[\!]} P!$. This could indeed be interpreted as a kind of ontological event representing the actual event of replicating the state of a process.

The reduction semantics, as they have been given, and as is generally the case for structural operational semantics, specifically defines the relational set $\longrightarrow$ of atomic steps. This, of course, defines a *transition system* (which is essentially a graph). The behaviors themselves, corresponding to the executions of the systems the terms represent, are then the paths through this transition system – the chains of reductions beginning at some term.

$$P_0 \longrightarrow P_1 \longrightarrow P_2 \longrightarrow P_3 \longrightarrow P_4 \longrightarrow \ldots$$

Thereby, one can extract a sequential behavioral representation from this semantics, and for other semantics for other calculi in a similar fashion. In contrast with the sequential representations of behavior we discussed in Chapter 3, these reduction sequences are defined primarily by the sequence of terms, which here are states, rather than a sequence of events. The events associated with the reduction of a term in π-calculus can be most easily approximated to be the $\longrightarrow$ transitions themselves.

But unlike the event representations we have discussed throughout this thesis, the reductions themselves are bare, only denoting that a transition has happened. To be proper labeled events, the transitions would have to be labeled with something. The term before or after the reaction would not be enough to fully distinguish transitions, since, in general, a term can have multiple reductions, and certainly could be the result of multiple reductions. On the other hand, the pair of terms would suffice to pin down the event itself for most intents and purposes, and certainly contains as much information as the sequence of terms – we assume for the above rules that any ambiguity in which reduction used must be a trivial one. With this in mind, our change to the semantics will both be from an interleaved sequential representation to one rooted in free symmetric monoidal categories, as well as from a state-based representation to an event-based one.

8.2 OEG Semantics

In order to use OEGs as a representation of the behaviors of π-calculus executions, the first step is to identify which dependency types and event types can be used to represent these behaviors. That is, the OES must be defined for π-calculus. Once an OES is defined, one can consider the language of relevant OEGs to be those that can be assembled from this OES. Rules can then be developed for building these OEGs for a given π-calculus terms. There are multiple ways of approaching this task and many variations could serve as the basis for the construction of the OES. However, we will focus on constructing an OEG semantics that is as close to the *reduction semantics* as possible.

OEG Reduction Semantics

Before pinning down the dependency types of the OES, it is worth enumerating the possible kinds of events. The key event that appears in the reduction semantics is the reduction itself. There are two kinds of reductions: one that involves only one reactant, guarded by a τ, and one that involves two reactants, one guarded by a send and the other by a receive. These two events are shown in the reduction rules **internal** and **communicate**. In addition, following the interpretation that replication is an active process, replication constitutes a third kind of event. These three kinds of events are depicted informally in Figure 8.1, with a sketch of their incoming and outgoing dependencies.

Considering what goes into each of the first two of these events, a candidate for the set of dependency types would be the subset of $\mathcal{N}_\pi$ consisting of guarded processes $\mathcal{G}_\pi \bullet P$: the

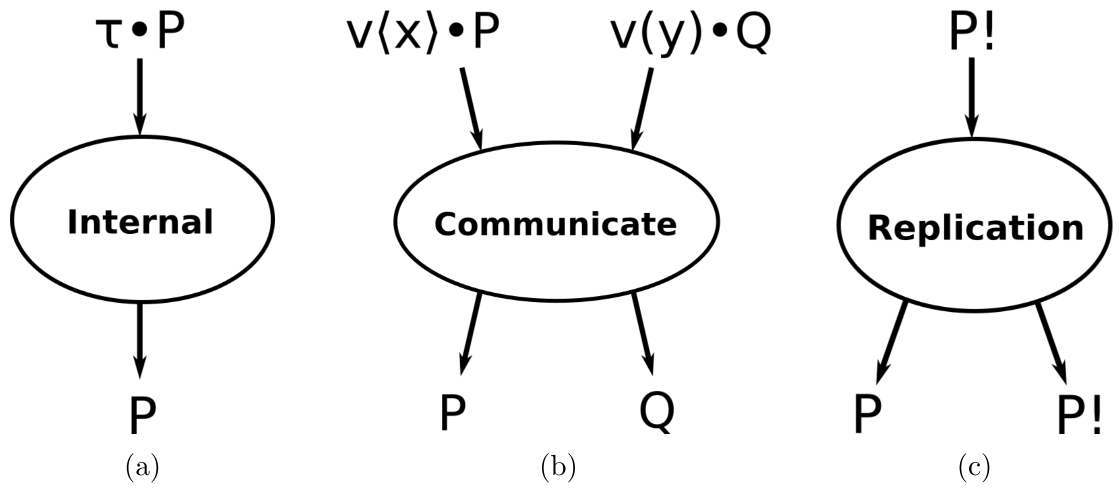


Figure 8.1: The three primary reduction events of π-calculus

monomials of $\mathcal{N}_\pi$. However, looking at the semantics, it can be seen that adding additional monomials to the ones reacting in the event would still be consistent with what is happening in the event. An internal action, for instance, from $\tau\bullet P$ to P could eliminate other possible actions N, and thus be an event with $\tau\bullet P + N$ as its incoming dependency. In a nondeterministic case, we could consider the occurrence of the event to represent a choice being made for a term $N_1 + \ldots + N_n \in \mathcal{N}_\pi$ about which of the monomials, N_k, participates in the event. We could therefore consider all of $\mathcal{N}_\pi$ to be potential incoming dependency types. Looking at the third fundamental event, **replication**, we must also consider terms of the form $P!$ to be potential incoming dependency types.

While the choice of $\mathcal{N}_\pi$ along with $P!$ seem reasonable, the outgoing terms of our provisional events must be much broader than these two constrained classes, encompassing the whole scope of π-calculus terms. This would raise a problem with our original assumptions, however, since none of the events would seem to act on $P \mathbin{[]} Q$ as an incoming dependency. We would have to posit an additional event to split this into its components, yet such an event would not actually represent a substantial transformation in the constitution of the term.

Nevertheless, we can use these considerations to restructure our provisional events in a way that makes them consistent with all of our intuitions, particularly if we also incorporate the fundamental notion of concurrency in the calculus. If our dependencies were π-calculus terms, two dependencies P and Q, each involved in independent concurrent events, would be represented in the language of OEGs as the product $P \otimes Q$. We therefore have a good reason to try and conflate the product of π-calculus $[]$ with the monoidal $\otimes$-product of dependency types so that monoidal products of dependencies correspond to $[]$ products of terms.

This observation would suggest that concurrent components of some sort (each term in a $[]$-product) should each be dependency type, and if we decompose these $[]$ products as much as possible, the atomic concurrent components will be either terms in $\mathcal{N}_\pi$ or terms of the form $P!$ for some P. In order to formally justify this intuition about decomposing terms we can appeal to the *standard form* of a π-calculus term.

$$\{\mathbf{x}\}_\bullet (M_1 \mathbin{[\!]} \dots \mathbin{[\!]} M_n \mathbin{[\!]} Q_1! \mathbin{[\!]} \dots \mathbin{[\!]} Q_n!) \tag{8.1}$$

where $M_i \in \mathcal{N}_\pi$ and each Q_i is itself a term in this standard form. Milner shows in [47] that any term $P \in \mathcal{T}_\pi$ is structurally congruent to some term of this form. The proof of this proposition is quite straightforward, and can be seen by simply considering the syntactic possibilities. Supposing the top level of the syntax is a $[\!]$-product, every scope binding can be pulled outside of the product using α-conversion. This would leave a product decomposed entirely into either $P!$ terms or terms in $\mathcal{N}_\pi$. This process could then be carried out recursively down the syntax tree to put the term inside of a repetition, or in front of a guard, all in this same standard form.

This property ensures that any term in $\mathcal{T}_\pi$ can be decomposed into irreducible concurrent components that are precisely from the two classes, $\mathcal{N}_\pi$ and $P!$, that we considered earlier to be candidates for incoming dependency types. We can therefore use this fact to break up the outgoing dependency types of our three provisional events to get three proper families of event types. Depicted in Figure 8.2, the outgoing dependency types, rather than being arbitrary π-calculus terms, are instead broken up into their concurrent components. In Figure 8.2, we have glossed over the outer scoping around each $[\!]$ product of outgoing terms, however, we will return to this subject later.

The picture we have created of a π-calculus term in standard form represented in the interfaces of a OEG representing the behavior of reduction makes intuitive sense if we consider the outermost product layer of a term to be the active concurrent state of the system, while deeper layers represent latent continuations. Each time a guard is removed from a term an inner layer can become part of the active concurrent state. To illustrate this intuition, consider a term

$$\tau_\bullet (A \mathbin{[\!]} \tau_\bullet B) \mathbin{[\!]} a(y)_\bullet (C + D) \mathbin{[\!]} \bar{a}\langle z\rangle_\bullet E$$

with three concurrent components at the outermost layer. The leftmost component has two latent components that become part of the outermost layer when an **internal** reduction happens.

$$\longrightarrow A \mathbin{[\!]} \tau_\bullet B \mathbin{[\!]} x(y)_\bullet (C + D) \mathbin{[\!]} \bar{a}\langle z\rangle_\bullet E$$

Before this transition, the inner term $\tau_\bullet B$ cannot reduce, instead remaining latent.[2]

$$\begin{gathered}\tau_\bullet (A \mathbin{[\!]} \tau_\bullet B) \mathbin{[\!]} a(y)_\bullet (C + D) \mathbin{[\!]} \bar{a}\langle z\rangle_\bullet E \\ \not\longrightarrow \tau_\bullet (A \mathbin{[\!]} B) \mathbin{[\!]} a(y)_\bullet (C + D) \mathbin{[\!]} \bar{a}\langle z\rangle_\bullet E\end{gathered}$$

[2]We could conceive of an unorthodox semantics in which inner terms can reduce, but this is rarely considered. At the least, there is surely no tunneling of communication between layers, although this would be a fascinating departure.

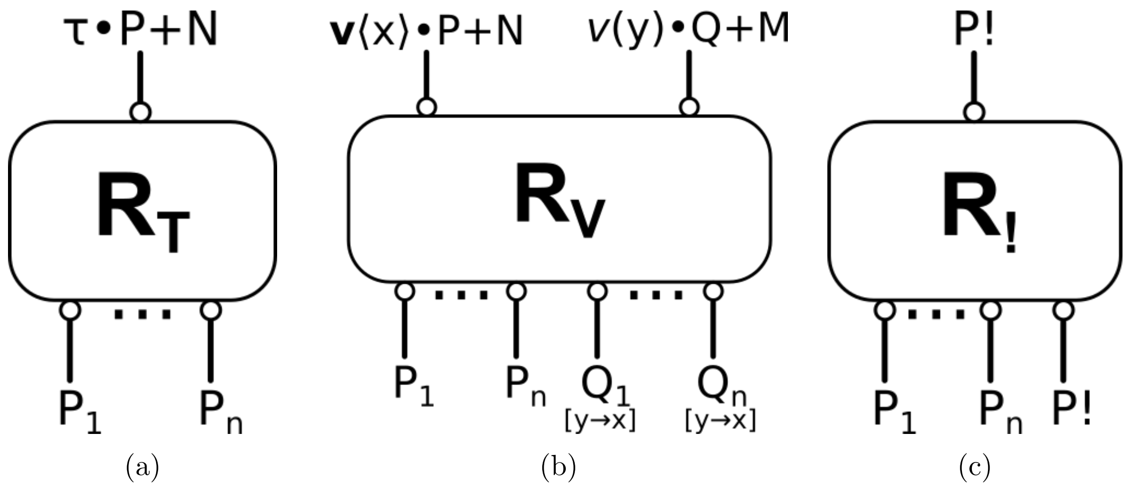


Figure 8.2: The three families of event types forming the OES for π-calculus

But after this transition, this term is now part of the outer layer and can reduce, further exposing B to the outer layer.

$$\longrightarrow A \parallel B \parallel x(y)_\bullet (C + D) \parallel \bar{a}\langle z\rangle_\bullet E$$

Hence, it is only the outer layer of concurrent components that is actively involved in the *behavior* of reduction, and must appear in an OEG representing this reduction. Replication is the same in principle. Because behavior happens on the outermost layer of the term, there is no reason to replicate on an inner layer. Replication can thus be treated as a behavioral event happening exclusively on the outermost layer.

This idea is similar to that of *weak-head normal form* reduction in λ-calculus. Inner layers of elements, guarded by λ-abstractions, do not get reduced, but instead stay passive until they are exposed by the removal of the guard (which is $\lambda x_\bullet$ in this case in this case) following an application. As an event, these reductions remove guards and expose the next layer of internal components to the outermost layer where they can interact.

We can therefore define the dependency types of an OES Σ_π as the set of ever kind of term that will appear as a concurrent component the outermost layer of a term in standard form.

Definition 45. *Let* $\mathbb{T}_\pi \overset{def}{=} \mathcal{N}_\pi \cup \{P! \mid P \in \mathcal{T}_\pi\}$.

Consequently, a general element of $\mathcal{T}_\pi$, in the standard form 8.1, if we put the name scoping aside for the moment, becomes a member of the free monoid of $\mathbb{T}_\pi$, $\mathbf{Mon}(\mathbb{T}_\pi)$ with $\parallel$ as the monoidal product and $\emptyset$ as the unit. In this OES, each π-calculus terms will be the incoming or outgoing event interface of both event types and whole OEGs representing the reduction of one whole term into another as a concurrent behavior.

This notion can be made more exact, and we can define a monoid over $\parallel$ products of terms in $\mathbb{T}_\pi$ as follows.

Definition 46. *Let $\mathcal{M}_\pi$ be the monoid defined*

$$\mathcal{M}_\pi = (\|\mathbf{Mon}(\mathbb{T}_\pi)\| / \cong,\ [],\ \emptyset)$$

where $\mathbf{Mon}(\mathbb{T}_\pi)/\cong$ are equivalence classes of $[]$ products of terms in $\mathbb{T}_\pi$ over structural congruence.

Using the standard form, as we defined it above, any term in $\mathcal{T}_\pi$ can, via structural congruence, be put into the form $\{\mathbf{x}\}_\bullet P$, where $\langle P \rangle \in \mathcal{M}_\pi$ and $\mathbf{x}$ is some, potentially empty, set of names. In other words, $\mathcal{M}_\pi$ covers $\mathcal{T}_\pi$, modulo structural congruence, up to the scoping of names.

The above definition of $\mathcal{M}_\pi$ as the free monoid that naturally emerges from our choice for $\mathbb{T}_\pi$ leaves a gap that we will fill in, but only after a couple outstanding considerations are addressed. It is important to note that in the proof that 8.1 is the standard form for any π-calculus term, congruence $P! \cong P \,[]\, P!$ is not used. Given that we are positing replication as a proper event, and will be representing it as a event type, we can remove it from the axioms for structural congruence. Given the language of symmetric monoidal categories has a family of braiding constants, which explicitly commute terms

$$\beta(P, Q) : P \,[]\, Q \to Q \,[]\, P$$

the axiom of commutativity could also be taken out of structural congruence. The effect of removing this axiom on the standard form would only be that the order of terms in the $[]$ product would be changed, and therefore the replication terms could not always be placed after the terms in $\mathcal{N}_\pi$, but this is of little semantic consequence. Here a trade-off is made between the explicit need to commute terms, potentially so that they can interact, and having the ability to sequentially compose two reductions in a well-defined fashion with the order of concurrent terms made explicit.

Another reason to omit these two axioms from structural congruence is that the concurrent composition monoid $\mathcal{M}_\pi$ becomes isomorphic with $\mathbf{Mon}(\mathbb{T}_\pi)$, since the only remaining axioms over $[]$ are associativity and identity. Consequently, we can state that two words $a, b \in \mathbf{Mon}(\mathbb{T}_\pi)$ are equal if and only if they satisfy out modified form of structural congruence.

The new, shorter set of axioms, outlining a weaker form of structural congruence is as follows.

Definition 47. *Modified π-calculus Structural Congruence*
The modified equivalence relation of structural congruence $\cong$ is defined over $\mathcal{T}_\pi$ as $\cong_\alpha$ along with the following axioms.

For $[\!]$	$P \mathbin{[\!]} (Q \mathbin{[\!]} R) \cong (P \mathbin{[\!]} Q) \mathbin{[\!]} R$ $\emptyset \mathbin{[\!]} P \cong P \mathbin{[\!]} \emptyset \cong P$
For $+$	$N + (M + K) \cong (N + M) + K$ $\emptyset + N \cong N + \emptyset \cong N$ $N + M \cong M + N$
For $\{\bullet\}$	$\{a\}_\bullet \{b\}_\bullet P \cong \{b\}_\bullet \{a\}_\bullet P$ $\{a\}_\bullet \emptyset \cong \emptyset$ $\{a\}_\bullet (P \mathbin{[\!]} Q) \cong \{a\}_\bullet P \mathbin{[\!]} Q$, *where* $a \notin \mathbf{FV}_\pi(Q)$ $\{a\}_\bullet (P \mathbin{[\!]} Q) \cong P \mathbin{[\!]} \{a\}_\bullet Q$, *where* $a \notin \mathbf{FV}_\pi(P)$

with $\cong$ *incorporating the equivalence of* α*-conversion on bound variables.*

From this point forward $\cong$ will be of this modified form, while $\cong_o$ will represent the original congruence. Furthermore, from this point forward, we will make an important additional simplification to ease subsequent definitions. The sets of terms $\mathcal{T}_\pi$, $\mathbb{T}_\pi$, $\mathcal{N}_\pi$, etc... will all be projected onto their quotient over α-conversions. That is, we will consider $\{\mathbf{x}\}_\bullet P$ and $\{\mathbf{y}\}_\bullet P[x \mapsto y]$ to be the same terms in $\mathcal{T}_\pi$. This expedites matters greatly, and should not be very controversial to most who study programming languages. Another small detail that needed to be changed in the new definition of structural congruence is that the last law, regarding scope extension, must now be given for both the left and right-hand sides of a $[\!]$-product. Without commutativity, one is no longer derivable from the other, as was previously the case.

Using this new structural congruence and our (α-invariant) definition of $\mathbb{T}_\pi$, we can define a modified standard form.

Definition 48. *A* π*-calculus term* $P \in \mathcal{T}_\pi$ *is in standard form,* $P \in \mathcal{T}_\pi^\sharp$*, iff*

$$P = \{\mathbf{x}\}_\bullet (P_1 \mathbin{[\!]} \ldots \mathbin{[\!]} P_N), \text{ where } P_k \in \mathbb{T}_\pi$$

We can then prove, using our new structural congruence rules that any term is structurally congruent to one in this standard form. No less, there is a deterministic procedure to normalize a term into standard form, and thus normalization to standard form can be defined as a function.

Proposition 21. *There exists a function*

$$\sqsupset \,:\, \mathcal{T}_\pi \to \mathcal{T}_\pi^\sharp$$

defined recursively

$$\begin{aligned}
&\sqsupset(N \in \mathbb{T}_\pi) = N \\
&\sqsupset(\{\mathbf{x}\}_\bullet P) = \{\mathbf{x}\}_\bullet \sqsupset(P) \\
&\sqsupset(P \mathbin{[\!]} Q) = u(\sqsupset(P), \sqsupset(Q))
\end{aligned}$$

where

$$
\begin{aligned}
&u : \mathcal{T}_\pi^\sharp \times \mathcal{T}_\pi^\sharp \to \mathcal{T}_\pi^\sharp \\
&u(\{\mathbf{x}\}\bullet P,\, \{\mathbf{y}\}\bullet Q) = \{\mathbf{w}, \mathbf{v}\}\bullet (P[\mathbf{x} \mapsto \mathbf{w}] \mathbin{[\,]} Q[\mathbf{y} \mapsto \mathbf{v}]) \\
&\quad \textit{with} \quad \mathbf{w},\, \mathbf{v} \notin \mathbf{FV}_\pi(P) \cup \mathbf{FV}_\pi(Q)
\end{aligned}
$$

Proof. Addressing the function u first, it must be shown that the type of the function is correct. Since every parameter to the function is of the form $\{\mathbf{x}\}\bullet P$ by definition, the P and Q in the definition are both in $\mathcal{M}_\pi$. Consequently, $P \mathbin{[\,]} Q \in \mathcal{M}_\pi$ as well, and thus the RHS of the definition is in $\mathcal{T}_\pi^\sharp$.

Next, we will show that

$$\{\mathbf{x}\}\bullet P \mathbin{[\,]} \{\mathbf{y}\}\bullet Q \cong u(\{\mathbf{x}\}\bullet P,\, \{\mathbf{y}\}\bullet Q)$$

By α-conversion, for a set of names $\mathbf{w}$ and $\mathbf{v}$

$$\{\mathbf{x}\}\bullet P \mathbin{[\,]} \{\mathbf{y}\}\bullet Q \cong \{\mathbf{w}\}\bullet P[x \mapsto w] \mathbin{[\,]} \{\mathbf{v}\}\bullet Q[y \mapsto v]$$

If these names have been chosen such that they are not in $\mathbf{FV}_\pi(P) \cup \mathbf{FV}_\pi(Q)$, then applying the structural congruence laws for scope extension

$$
\begin{aligned}
&\{\mathbf{w}\}\bullet P[x \mapsto w] \mathbin{[\,]} \{\mathbf{v}\}\bullet Q[y \mapsto v] \\
&= \{\mathbf{w}\}\bullet (P[x \mapsto w] \mathbin{[\,]} \{\mathbf{v}\}\bullet Q[y \mapsto v]) \\
&= \{\mathbf{w}\}\bullet \{\mathbf{v}\}\bullet (P[x \mapsto w] \mathbin{[\,]} Q[y \mapsto v]) \\
&= \{\mathbf{w}, \mathbf{v}\}\bullet (P[x \mapsto w] \mathbin{[\,]} Q[y \mapsto v])
\end{aligned}
$$

verifying the congruence through u.

We must then establish that $\beth$ is well-defined for the given typing. This is the case because u is a well-defined function into $\mathcal{T}_\pi^\sharp$ and $\mathbb{T}_\pi \subseteq \mathcal{T}_\pi^\sharp$. The rest follows from structural induction over the syntax of terms in $\mathcal{T}_\pi$. In the base case,

$$\beth(N) = N \in \mathbb{T}_\pi \subseteq \mathcal{T}_\pi^\sharp$$

In the second case, if $\beth(P) \in \mathcal{T}_\pi^\sharp$, which is the induction hypothesis, then $\{\mathbf{x}\}\bullet P \in \mathcal{T}_\pi^\sharp$ as well. The third case follows from the above proof of the typing for the function u.

Finally, it must be shown that $\beth(P) \cong P$. This also follows from structural induction. In the base case, this congruence is trivially true. In the second case, this congruence follows from the induction hypothesis and substitution. In the final case, this congruence follows from the above property of u along with the induction hypothesis and substitution.

$$\beth(P \mathbin{[\,]} Q) = u(B(P),\, B(Q)) \cong B(P) \mathbin{[\,]} B(Q) \cong P \mathbin{[\,]} Q$$

$\square$

Given this modified structural congruence, we are left with the final question of how to properly address the scoping gap between $\mathcal{M}_\pi$ and $\mathcal{T}_\pi$. Here we must ask an important semantic question: with respect to the reduction semantics, what is the effect of the outermost scoping operator on the reduction of a term in standard form? We can restrict the question to the case of the outermost scoping, since this fully accounts for the gap. From the semantic **scoping** rule involving it, it can be seen that, behaviorally, a term has all of the same reductions with or without the outermost scoping. The scoping operator can therefore be seen (at least in the case of the reduction semantics) as of syntactic importance primarily to the inner layers of a term.

We can therefore state that when studying reduction of a term in $\mathcal{T}_\pi$, we can look at its projection to $\mathcal{M}_\pi$ achieved by putting the term in standard form and stripping off the outer layers of scoping.

Definition 49. *Let the scoping projection be defined*

$$\begin{aligned}&\gimel : \mathcal{T}_\pi \to \mathcal{M}_\pi\\ &\gimel(P) = Q,\ \textit{where}\ \{\mathbf{x}\}_{\bullet} Q = \beth(P),\ Q \in \mathcal{M}_\pi\end{aligned}$$

This definition is well-defined as a consequence of the above definition of standard form, because a deterministic procedure was given for normalizing a term to standard form.

This projection $\gimel$ operation reduces any whole π-calculus term into a member of $\mathcal{M}_\pi$, and thus a free monoidal product of dependency types in $\mathbb{T}_\pi$, the interfaces of our reduction OEGs. But we must consider how this form is maintained through reduction, which exposes inner layers of $\mathcal{T}_\pi$ to the product of $\mathbb{T}_\pi$. This exposure of inner layers must be normalized such that the whole term is again in a member of $\mathcal{M}_\pi$, stripped of outer scoping.

It is almost a give that the above description will be difficult to interpret, so we will give a clarifying example to illustrate the point. Consider the following term in $\mathcal{M}_\pi$.

$$P_1 \mathbin{[\,]} P_2 \mathbin{[\,]} \tau_{\bullet} \{x\}_{\bullet} (Q_1 \mathbin{[\,]} Q_2) \mathbin{[\,]} P_3$$

We will first exercise some of the descriptive language we have used liberally in couple preceding paragraphs. This term consists of four concurrent components: P_1, P_2, P_3, and $\tau_{\bullet} \{x\}_{\bullet} (Q_1 \mathbin{[\,]} Q_2)$. The last of these components is clearly in $\mathbb{T}_\pi$, and we presume the other three to be as well. These four terms constitute the outer layer of the reduction process, and the three possible reduction events could occur to any of these four, or any pair, under the conditions imposed by the guards or replication operators. The term that has been written out explicitly, $\tau_{\bullet} \{x\}_{\bullet} (Q_1 \mathbin{[\,]} Q_2)$, can be reduced using the **internal** rule. This would expose the inner layer $\{x\}_{\bullet} (Q_1 \mathbin{[\,]} Q_2)$. The concurrent components in this inner layer, Q_1 and Q_2, cannot react until they are part of the outer layer of concurrent components.

If this **internal** reaction occurs via the standard reduction semantics, the result would be the term

$$P_1 \mathbin{[\,]} P_2 \mathbin{[\,]} \{x\}_{\bullet} (Q_1 \mathbin{[\,]} Q_2) \mathbin{[\,]} P_3$$

which, of course, is no longer in standard form. The inner layer is exposed, but it is still bounded by a scoping of the variable x. Nevertheless, because of α conversion, the only barrier to simply shaking off the scoping is a potential collision of x with the free variables in other three P_k terms. If a supply of new names existed for the outermost layer, distinct from the free variables of the whole term, one could simply draw one of these names y and substitute for x, resulting in the term

$$P_1 \mathbin{[\!]} P_2 \mathbin{[\!]} \{y\}_{\bullet} (Q_1[x \mapsto y] \mathbin{[\!]} Q_2[x \mapsto y]) \mathbin{[\!]} P_3$$

Then, given y is disjoint from the free variables in all the other terms, scope extension can pull the scope to the outside, returning the term to standard form.

$$\{y\}_{\bullet} (P_1 \mathbin{[\!]} P_2 \mathbin{[\!]} Q_1[x \mapsto y] \mathbin{[\!]} Q_2[x \mapsto y] \mathbin{[\!]} P_3)$$

Finally, the outer scoping can be again stripped off, yielding a term in $\mathcal{M}_\pi$.

$$P_1 \mathbin{[\!]} P_2 \mathbin{[\!]} Q_1[x \mapsto y] \mathbin{[\!]} Q_2[x \mapsto y] \mathbin{[\!]} P_3$$

The Q_k concurrent components are then exposed completely to the outer layer of the term, and can participate in reductions.

What can be taken from this example is that if we are provided with a supply of unused names, each kind of reaction can incorporate the above form of substitution to its outgoing dependencies as part of its action exposing an inner layer of concurrent components. To be specific, the **internal** event type does the following, given a set of new names $\mathcal{V}$:

1. takes in a term of the form $\tau_{\bullet} P$ as an incoming dependency
2. reduces the term to P
3. normalizes P to standard form $\{\mathbf{x}\}_{\bullet} (Q_1 \mathbin{[\!]} \ldots \mathbin{[\!]} Q_N)$
4. substitutes new names $\mathbf{y} \in \mathcal{V}$ for $\mathbf{x}$ resulting in the term

$$\{\mathbf{y}\}_{\bullet} (Q_1[\mathbf{x} \mapsto \mathbf{y}] \mathbin{[\!]} \ldots \mathbin{[\!]} Q_N[\mathbf{x} \mapsto \mathbf{y}])$$

5. strips the outer scoping, since it is safe to do so, producing a term

$$Q_1[\mathbf{x} \mapsto \mathbf{y}] \mathbin{[\!]} \ldots \mathbin{[\!]} Q_N[\mathbf{x} \mapsto \mathbf{y}]$$

   in $\mathcal{M}_\pi$
6. yields each of $Q_k[\mathbf{x} \mapsto \mathbf{y}]$ as an outgoing dependency

These same steps can be adapted to the other two event type cases similarly, thereby dealing with the issue of scoping. In order to notationally simplify these extra steps, the following operator will be defined.

Definition 50. *Given a list of variables $\mathcal{V}$ and $P \in \mathcal{T}_\pi$, let the following operation be defined*

$$\Xi^{\mathcal{V}}_{\mathbf{x}}(P) = P[\mathbf{x} \mapsto \mathbf{y}]$$

where $\mathbf{y}$ *are* $\|\mathbf{x}\|$ *variables drawn from* $\mathcal{V}$*.*

Although this operation should suffice to convey the basic idea precisely, a further technicality elided in it is that the given list of variables must be returned having had the used variables removed. The reader should be able to augment the above definition, and the uses of it below to account for this without much more than additional notational machinery.

Using this set of dependency types, and this new definition of structural congruence, we can proceed to formally define the event types of the OES inductively.

Definition 51. *π-calculus OES*
*Let a family of **internal** event types, $\mathcal{R}_\tau(-;\, -,\, -)$, be defined*

$$\frac{\begin{array}{cc} N,\, M,\, P_1,\, \ldots,\, P_n \in \mathcal{N}_\pi & \mathcal{V} \textit{ is a list of new variables} \\ N \cong \tau_{\bullet} \{\mathbf{x}\}_{\bullet} (P_1 \,[\!]\, \ldots \,[\!]\, P_n) + M & P \cong \Xi^{\mathcal{V}}_{\mathbf{x}}(P_1) \otimes \ldots \otimes \Xi^{\mathcal{V}}_{\mathbf{x}}(P_n) \end{array}}{\mathcal{R}_\tau(\mathcal{V};\, N,\, P) \,:\, N \to P}$$

*Let a family of **communication** event types, $\mathcal{R}_a(-;\, -,\, -,\, -,\, -)$, be defined*

$$\frac{\begin{array}{cc} N,\, M,\, P_1,\, \ldots,\, P_n, Q_1,\, \ldots,\, Q_m \in \mathcal{N}_\pi & \mathcal{V} \textit{ is a list of new variables} \\ N \cong \bar{a}\langle \mathbf{b} \rangle_{\bullet} \{\mathbf{x}\}_{\bullet} (P_1 \,[\!]\, \ldots \,[\!]\, P_n) + M & L \cong a(\mathbf{y})_{\bullet} \{\mathbf{z}\}_{\bullet} (Q_1 \,[\!]\, \ldots \,[\!]\, Q_m) + K \\ P \cong \Xi^{\mathcal{V}}_{\mathbf{x}}(P_1) \otimes \ldots \otimes \Xi^{\mathcal{V}}_{\mathbf{x}}(P_n) & Q \cong \Xi^{\mathcal{V}}_{\mathbf{z}}(Q_1) \otimes \ldots \otimes \Xi^{\mathcal{V}}_{\mathbf{z}}(Q_n) \end{array}}{\mathcal{R}_a(\mathcal{V};\, N,\, P,\, L,\, Q) \,:\, N \otimes L \to P \otimes Q[\mathbf{y} \mapsto \mathbf{b}]}$$

*Let a family of **replication** event types, $\mathcal{R}_!(-;\, -)$, be defined*

$$\frac{\begin{array}{cc} P_1,\, \ldots,\, P_n \in \mathcal{N}_\pi & \mathcal{V} \textit{ is a list of new variables} \\ P \cong \{\mathbf{x}\}_{\bullet} (P_1 \,[\!]\, \ldots \,[\!]\, P_n) & P' \cong \Xi^{\mathcal{V}}_{\mathbf{x}}(P_1) \otimes \ldots \otimes \Xi^{\mathcal{V}}_{\mathbf{x}}(P_n) \end{array}}{\mathcal{R}_!(\mathcal{V};\, P') \,:\, P! \to P' \otimes P!}$$

Let $\mathcal{A}_\pi$ consist of all $\mathcal{R}_\tau$, $\mathcal{R}_a$, and $\mathcal{R}_!$ event types, over all parameters (sorted by event interface).

When using these notations, $\mathcal{R}$, $\mathcal{R}_\tau$, and $\mathcal{R}_!$, we will sometimes suppress their parameters for brevity.

Having defined the dependency types and event types, the OES can be defined as follows.

Definition 52. *π-calculus OES*
Let the OES of π-calculus be defined $\Sigma_\pi \overset{def}{=} (\mathbb{T}_\pi,\, \mathcal{A}_\pi)$.

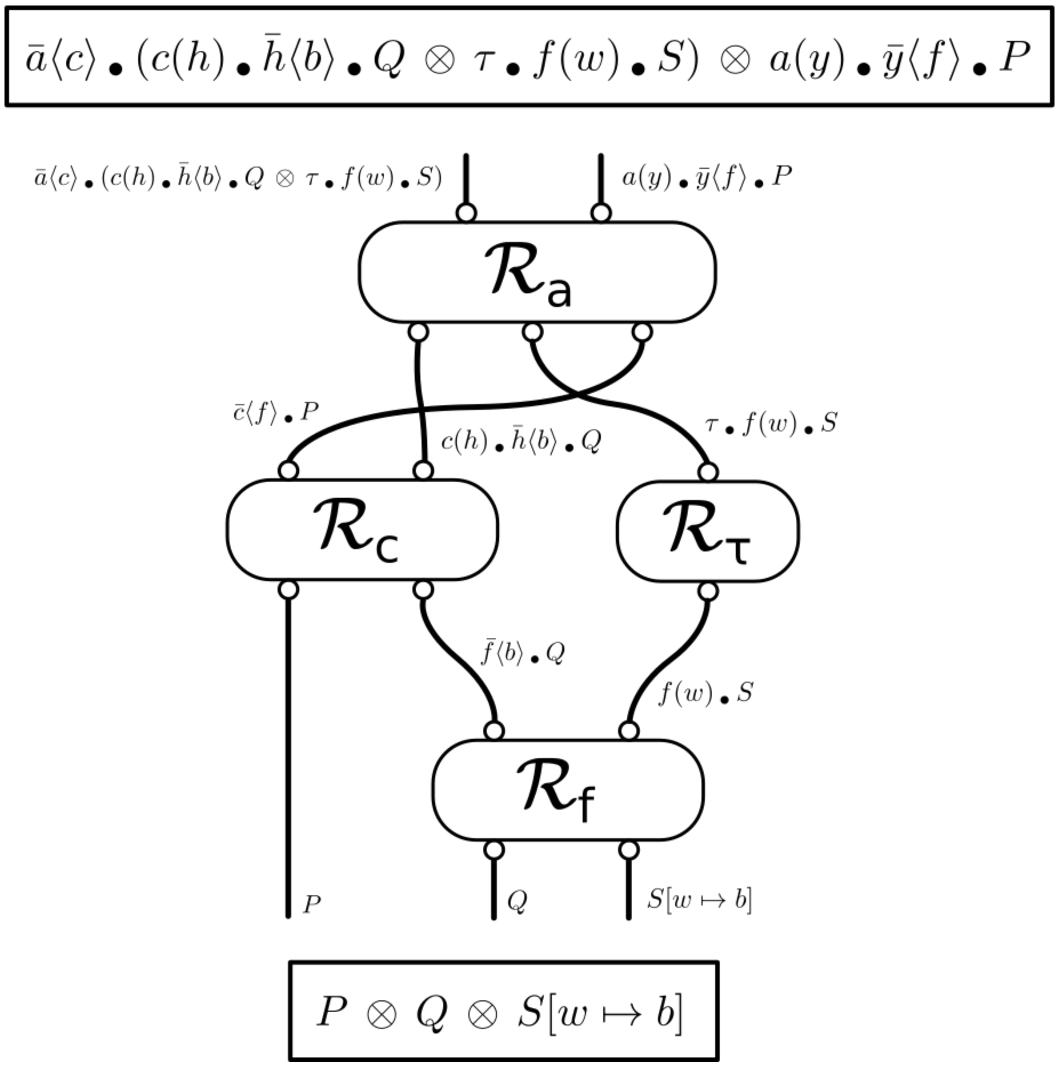

$\bar{a}\langle c\rangle \bullet (c(h) \bullet \bar{h}\langle b\rangle \bullet Q \otimes \tau \bullet f(w) \bullet S) \otimes a(y) \bullet \bar{y}\langle f\rangle \bullet P$
$\bar{a}\langle c\rangle \bullet (c(h) \bullet \bar{h}\langle b\rangle \bullet Q \otimes \tau \bullet f(w) \bullet S)$
$a(y) \bullet \bar{y}\langle f\rangle \bullet P$
$\mathcal{R}_a$
$\bar{c}\langle f\rangle \bullet P$
$c(h) \bullet \bar{h}\langle b\rangle \bullet Q$
$\tau \bullet f(w) \bullet S$
$\mathcal{R}_c$
$\mathcal{R}_\tau$
$\bar{f}\langle b\rangle \bullet Q$
$f(w) \bullet S$
$\mathcal{R}_f$
P
Q
$S[w \mapsto b]$
$P \otimes Q \otimes S[w \mapsto b]$

Figure 8.3: An example of an OEG representing the reduction of a term.

From this OES, the free symmetric monoidal category of OEGs, $\mathcal{OEG}(\Sigma_\pi)$, is determined. This is ultimately what we intend to use to represent the behavior of π-calculus terms under reduction. To summarize the structure of this free symmetric monoidal category, each object is identified with an element of $\mathcal{M}_\pi$, which, as we reasoned earlier, are the standard forms of $\mathcal{T}_\pi$ without the outer scoping (and with fixed component orderings), while each morphisms between any two objects represents a reduction from one term in $\mathcal{M}_\pi$ to another in $\mathcal{M}_\pi$. The category $\mathcal{OEG}(\Sigma_\pi)$ gives the semantics of π-calculus insomuch as for any term $t \in \mathcal{T}_\pi$ the possible reductions are represented by the morphisms that have $\daleth(t)$ as their domain.

To see how all of this fits together, we will look in detail at a straightforward example.

Consider the term

$$\bar{a}\langle c\rangle \bullet (c(h) \bullet \bar{h}\langle b\rangle \bullet Q \mathbin{[\,]} \tau \bullet f(w) \bullet S) \mathbin{[\,]} a(y) \bullet \bar{y}\langle f\rangle \bullet P$$

Figure 8.3 depicts a reduction of this term as an OEG. The term itself is the incoming event interface of the OEG, while the outgoing event interface is the reduced term

$$P \mathbin{[\,]} Q \mathbin{[\,]} S[w \mapsto b]$$

As is shown in the figure, the dependency edges internal to the OEG can all be identified with terms from $\mathcal{N}_\pi$ that constitute the concurrent components of the intermediate steps of the reduction. These terms connect together four events: three communication reductions, $\mathcal{R}_a$, $\mathcal{R}_c$, $\mathcal{R}_f$, and one internal reduction, $\mathcal{R}_\tau$ – to simplify syntax, for purposes of discussion, we will suppress the parameters of each term as they can be inferred from the context. This OEG can also be presented as term in algebraic language of symmetric monoidal category as

$$(\mathbf{id} \otimes \mathcal{R}_f) \circ (\mathcal{R}_c \otimes \mathcal{R}_\tau) \circ \mathcal{P}_{(3\,1\,2)} \circ \mathcal{R}_a$$

where $\mathcal{P}_{(3\,1\,2)}$ is the permutation term constructed of β and **id** terms. Note that we also have conflated the notation for events and event types such that $\langle\langle A\rangle\rangle$, the atomic OEG containing an instance of event type A, can just be written as A. Of course, there are many possible equivalent symmetric monoidal category terms, and the one we chose was simply a particularly compact one that matches the concrete rendering of the OEG.

To hold this in contrast with the conventional operational semantics, the two possible confluent reduction sequences that can be carried out are:

$$\begin{aligned}
&\bar{a}\langle c\rangle \bullet (c(h) \bullet \bar{h}\langle b\rangle \bullet Q \mathbin{[\,]} \tau \bullet f(w) \bullet S) \mathbin{[\,]} a(y) \bullet \bar{y}\langle f\rangle \bullet P\\
&\quad\longrightarrow c(h) \bullet \bar{h}\langle b\rangle \bullet Q \mathbin{[\,]} \tau \bullet f(w) \bullet S \mathbin{[\,]} \bar{c}\langle f\rangle \bullet P\\
&\quad\longrightarrow^{\beta} \bar{c}\langle f\rangle \bullet P \mathbin{[\,]} c(h) \bullet \bar{h}\langle b\rangle \bullet Q \mathbin{[\,]} \tau \bullet f(w) \bullet S\\
&\quad\longrightarrow P \mathbin{[\,]} \bar{f}\langle b\rangle \bullet Q \mathbin{[\,]} \tau \bullet f(w) \bullet S\\
&\quad\longrightarrow P \mathbin{[\,]} \bar{f}\langle b\rangle \bullet Q \mathbin{[\,]} f(w) \bullet S\\
&\quad\longrightarrow P \mathbin{[\,]} Q \mathbin{[\,]} S[w \mapsto b]
\end{aligned}$$

and

$$\begin{aligned}
&\bar{a}\langle c\rangle \bullet (c(h) \bullet \bar{h}\langle b\rangle \bullet Q \mathbin{[\,]} \tau \bullet f(w) \bullet S) \mathbin{[\,]} a(y) \bullet \bar{y}\langle f\rangle \bullet P\\
&\quad\longrightarrow c(h) \bullet \bar{h}\langle b\rangle \bullet Q \mathbin{[\,]} \tau \bullet f(w) \bullet S \mathbin{[\,]} \bar{c}\langle f\rangle \bullet P\\
&\quad\longrightarrow^{\beta} \bar{c}\langle f\rangle \bullet P \mathbin{[\,]} c(h) \bullet \bar{h}\langle b\rangle \bullet Q \mathbin{[\,]} \tau \bullet f(w) \bullet S\\
&\quad\longrightarrow \bar{c}\langle f\rangle \bullet P \mathbin{[\,]} c(h) \bullet \bar{h}\langle b\rangle \bullet Q \mathbin{[\,]} f(w) \bullet S\\
&\quad\longrightarrow P \mathbin{[\,]} \bar{f}\langle b\rangle \bullet Q \mathbin{[\,]} f(w) \bullet S\\
&\quad\longrightarrow P \mathbin{[\,]} Q \mathbin{[\,]} S[w \mapsto b]
\end{aligned}$$

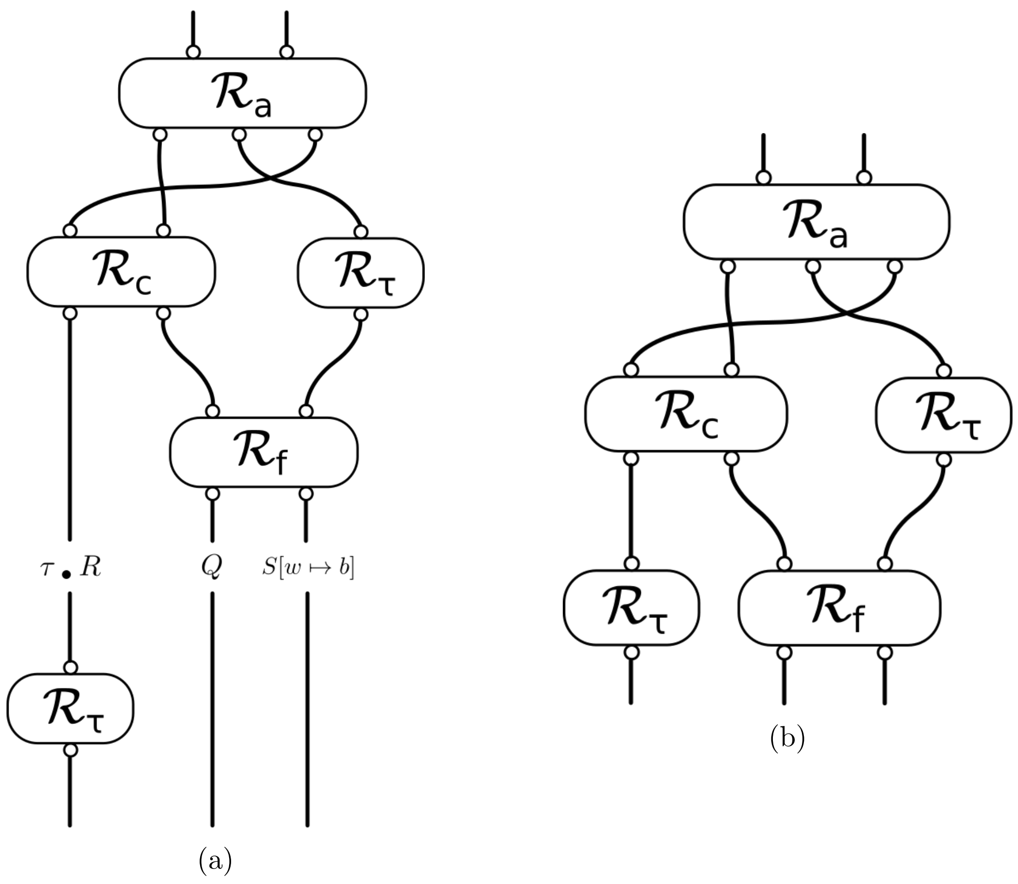


Figure 8.4: A composition (a) between two OEGs and the resulting OEG (b).

In both of these reductions, the first step corresponds to the $\mathcal{R}_a$ event in Figure 8.3. In the second step of both, we make commutation of concurrent components explicit with a reduction marked with β. In the figure, this corresponds to the crossing dependencies. The third and fourth steps in the two reductions, however, are different. These correspond to the $\mathcal{R}_c$ reduction of the first two concurrent components in the third term, and to the $\mathcal{R}_\tau$ reduction of the third concurrent component. In the OEG, these two events appear as truly concurrent. The final step is the same in both reductions and corresponds to the $\mathcal{R}_f$ event in the figure.

Intuitively, this OEG indicates something much closer to what we expect the reduction to actually represent. The concrete processes represented as the reductions of the pair of terms $c(h) \bullet \bar{h}\langle b\rangle \bullet Q$ and $\bar{c}\langle f\rangle \bullet P$, and the term $\tau \bullet f(w) \bullet S$, are causally independent processes. One may start before the other, or they may start at the same time. Moreover, if they occur concurrently, both taking a duration of time, they do not necessarily begin and end at the same time; they may overlap only partially. The interleaving interpretation, contrastingly, presupposes a kind of atomicity and synchrony.

A more important point about the OEG framing of the reduction that goes beyond intuition can be seen in the way is that it composes. The term P, emerging from the $\mathcal{R}_c$ reduction, could potentially participate in a subsequent reduction without waiting for the $\mathcal{R}_f$ event to occur. The structure of the OEG makes this potential clear. For instance, if $P \cong \tau_{\bullet} R$, a separate reduction from $P \mathbin{[\!]} Q \mathbin{[\!]} S[w \mapsto b]$ to $R \mathbin{[\!]} Q \mathbin{[\!]} S[w \mapsto b]$ could be constructed as a OEG and composed in sequence with the reduction shown in Figure 8.3. This composition is depicted in Figure 8.4a while the result shown in Figure 8.4b emphasizes that the lowermost occurrence of the $\mathcal{R}_\tau$ can occur concurrently with the $\mathcal{R}_f$ event.

Connecting the OEG Semantics to Structural Semantics

With $\mathcal{M}_\pi \cong \mathbf{Mon}(\mathcal{N}_\pi)$ as our objects, the morphisms of $\mathcal{OEG}(\Sigma_\pi)$ are, indeed, the reduction OEGs that account for the behavioral semantics of π-calculus. Even if the category of OEGs provides an interesting semantics, the question remains of how this semantics relates to the original semantics 44 we recounted above. We have already made two important alterations to the original reduction semantics (the way they most often appear). Specifically, we have moved replication from being part of structural congruence to being an explicit reduction (even though it is an expansion to be exact). We have also made commutation explicit to ultimately match our language of OEGs.

Applying these changes first, we can formulate a modified reduction semantics as a structural operational semantics.

Definition 53. *Modified Reduction Semantics*
The operational semantics of evaluation for π-calculus is given by the rewrite relation $\longrightarrow : \mathcal{T}_\pi \to \mathcal{T}_\pi$ defined inductively from the following derivations

$$\frac{}{\tau_{\bullet} P + N \longrightarrow P}\ \textit{(Internal)}$$

$$\frac{}{(\bar{v}\langle \mathbf{x}\rangle_{\bullet} P + N) \mathbin{[\!]} (v(\mathbf{y})_{\bullet} Q + M) \longrightarrow P \mathbin{[\!]} Q[\mathbf{y} \mapsto \mathbf{x}]}\ \textit{(Communicate)}$$

$$\frac{}{P! \longrightarrow P \mathbin{[\!]} P!}\ \textit{(Replicate)}$$

$$\frac{}{P \mathbin{[\!]} Q \longrightarrow Q \mathbin{[\!]} P}\ \textit{(Commute)}$$

$$\frac{P \longrightarrow P'}{P \mathbin{[\!]} Q \longrightarrow P' \mathbin{[\!]} Q}\ \textit{(Parallel)} \qquad \frac{P \longrightarrow P'}{\{v\}_{\bullet} P \longrightarrow \{v\}_{\bullet} P'}\ \textit{(Scoping)}$$

$$\frac{Q \cong P \qquad P \longrightarrow P' \qquad P' \cong Q'}{Q \longrightarrow Q'}\ \textit{(Structural)}$$

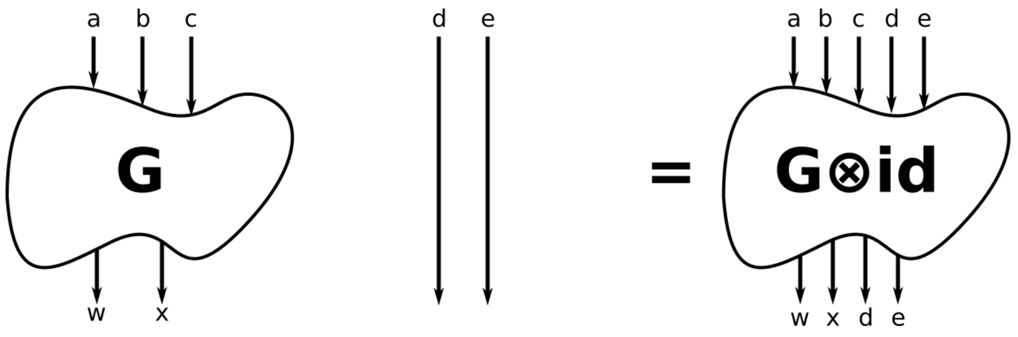


Figure 8.5: An OEG interpretation of the **parallel** rule.

Here we have added the two new rules **Replicate** and **Commute**. The first three rules now correspond very directly to the three families of event types in Σ_π. Furthermore, the fourth, **Commute** rule corresponds to a braiding OEG. There is a clear connection between the reduction rules and the events and constants of the OEG-semantics. Reflecting on the comment made earlier regarding structural operational semantics as being rooted in states rather than events, it was argued that to move to an event-oriented representation, one might somehow label or otherwise qualify the transitions. In the case of the above modified reduction semantics, we can identify with the first four rules with the OEGs $\mathcal{R}$, $\mathcal{R}_\tau$, $\mathcal{R}_!$, and β, respectively, connecting the two representations of reduction events.

The remaining three rules, **Parallel**, **Scope**, and **Structural**, all derive reductions from other reductions. In these cases, assume that we start with a reduction $P \longrightarrow P'$, and that, like the other four cases, we have a OEG G that corresponds to the reduction event. The **Scope** and **Structural** rules would yield a reduction that could be identified with the same OEG. In the first case, the scoping mechanisms have incorporated into the reduction using the α-conversion technique we discussed earlier. In the second case, structural congruence identifies terms that would already correspond to the same OEG.

This leaves the **Parallel** rule, which marks the most significant difference between the two kinds of semantics. In operational semantics, as we discussed earlier, this rule allows a single concurrent component (consisting of one or two reactants) of a product reduce by itself while the rest of the components remain unchanged. Although the reduction rule is written for the left component in the product, reducing the component on the right can be achieved similarly via the **Commute** rule. If we assume that the reduction of one component P in a product $P \mathbin{[\!]} Q$ can be reduced to P' in a manner captured by the OEG G, then the OEG representing the entire reduction, where Q remains unchanged, would be the OEG depicted in Figure 8.5, consisting of a parallel composition of G and an identity element for all of the dependencies that remain unchanged. We can write this term as $G \otimes \mathbf{id}(Q)$, and therefore this OEG can be used to represent events that takes place during this reduction.

Using these OEG fragments, attached to each reduction transition, the operational semantics rules can be augmented to produce sequences of OEGs from sequences of transitions.

Definition 54. *OEG-labeled Reduction Semantics*

The operational semantics of evaluation for π-calculus is given by the rewrite relation $\longrightarrow : \mathcal{T}_\pi \to \mathcal{T}_\pi$ defined inductively from the following derivations

$$\frac{}{\tau \bullet P + N \xrightarrow{\mathcal{R}_\tau} P}\ \textit{(Internal)}$$

$$\frac{}{(\bar{v}\langle \mathbf{x} \rangle \bullet P + N) \mathbin{[\!]} (v(\mathbf{y}) \bullet Q + M) \xrightarrow{\mathcal{R}_v} P \mathbin{[\!]} Q[\mathbf{y} \mapsto \mathbf{x}]}\ \textit{(Communicate)}$$

$$\frac{}{P! \xrightarrow{\mathcal{R}_!} P \mathbin{[\!]} P!}\ \textit{(Replicate)}$$

$$\frac{}{P \mathbin{[\!]} Q \xrightarrow{\beta(P,Q)} Q \mathbin{[\!]} P}\ \textit{(Commute)}$$

$$\frac{P \xrightarrow{G} P'}{P \mathbin{[\!]} Q \xrightarrow{G [\!] \mathbf{id}(Q)} P' \mathbin{[\!]} Q}\ \textit{(Parallel)} \qquad \frac{P \xrightarrow{G} P'}{\{v\} \bullet P \xrightarrow{G} \{v\} \bullet P'}\ \textit{(Scoping)}$$

$$\frac{Q \cong P \qquad P \xrightarrow{G} P' \qquad P' \cong Q'}{Q \xrightarrow{G} Q'}\ \textit{(Structural)}$$

Applying these semantics to a term P, one gets a sequence of reductions,

$$P \xrightarrow{G_1} P_1 \xrightarrow{G_2} P_2 \xrightarrow{G_3} \ldots \xrightarrow{G_N} P_N$$

resulting in a term P_N. Composing the OEGs in the labels of each reduction then gives an OEG

$$G = G_N \circ \ldots \circ G_2 \circ G_1$$

representing the reduction. This composition is always well-defined because the terms (modulo the outer scoping) represent the event interface of the OEGs. That is, in the above sequence

$$\xrightarrow{G_n} \in \mathbf{hom}_{\mathcal{OEG}(\Sigma_\pi)}(P_{n-1},\, P_n)$$

Nevertheless, this construction, to be complete, must account for the issue of constructing instances of event types with the right set of new variables $\mathcal{V}$ to correctly parameterize each event types in this chain of compositions. Because the construction is a chain, there is no problem in inserting this additional detail along the composition. One could either generate a corresponding sequence of disjoint lists of variables, perhaps the mathematically easiest means of achieving this end, or perhaps more modularly, each OEG G_k in the chain of compositions could be wrapped in a monad, carrying the remaining elements of a single list along the chain and removing elements as they are needed.[3]

[3]This latter technique should bear some familiarity to Haskell programmers.

It is worth reminding the reader that while this construction is sequential, the value it produces, the OEG itself, is nonetheless a true concurrent representation of reduction behavior. As a matter of fact, while the constructing sequences of compositions recapitulate the structure of the existing operational semantics, the result of the composition correctly conflates every such sequence that represents the same essential behavior when viewed concurrently.

Take for instance the term we considered earlier in great detail, depicted in Figure 8.3. If we take the two confluent reduction sequences worked out for the term and use the above OEG-labeled semantics to convert them into OEG terms, we get the terms

$$(\mathbb{1} \otimes \mathcal{R}_f) \circ (\mathbb{1} \otimes \mathcal{R}_\tau) \circ (\mathcal{R}_c \otimes \mathbb{1}) \circ \beta \circ \mathcal{R}_a$$

and

$$(\mathbb{1} \otimes \mathcal{R}_f) \circ (\mathcal{R}_c \otimes \mathbb{1}) \circ (\mathbb{1} \otimes \mathcal{R}_\tau) \circ \beta \circ \mathcal{R}_a$$

respectively. Under the axioms of symmetric monoidal categories, these two terms are equivalent, both amounting to the same OEG shown in the figure. Of the many equivalent ways this OEG could be described by a term, perhaps the one that most intuitively conveys the notion of concurrency is neither of the above two, but instead the term

$$(\mathbb{1} \otimes \mathcal{R}_f) \circ (\mathcal{R}_c \otimes \mathcal{R}_\tau) \circ \beta \circ \mathcal{R}_a$$

which places the two concurrent events in a parallel composition.

The key to the OEG representation of this reduction is that all of these term presentations, some representing the sequential interleavings and some involving parallel compositions, have the same underlying OEG as their value. The OEG therefore captures something more essential, more ontological, about the reduction as a distributed computational behavior.

Chapter 9

Conclusions and Further Work

This thesis provides the potential beginning for a new approach to reasoning about distributed behavior in computational systems, rooted in a new representation of behavior that has monoidal category theory at its mathematical foundation. The motivation has been to identify and develop a representation of behavior capable of meeting the challenges of modeling what we have termed HDDAs, applications that run on burgeoning platforms such as the IoT and the Swarm. We have characterized the nature of these kinds of platforms, which we have termed Process Fields, and illustrated how they diverge from more well-established models of concurrent computation rooted in static networks of sequential components. Instead, Process Fields begin to dissolve the distinction between messages passed and process states. We have argued that this increase in complexity necessitates a behavioral representation that is modular, composable, true concurrent, and event-based. Such a model would constitute a basis for reasoning about distributed behavior that would open the door to developing formal semantics and formal verification and analysis methods for HDDAs running on Process Fields.

We explored the possibilities of how distributed behavior could be represented from two directions. First, we started with representations of sequential behavior, looking at them in all of their mathematical nuances, then showed two distinct paths of generalization that could be taken from sequential to concurrent computation, *generalized sequences* and *free monoidal categories*, giving theoretical arguments for preferring the latter over the former. Next, we explored existing representations of concurrent behavior, showing that they have focused on the generalized sequence branch of concurrent generalization rather than that of free monoidal categories, and consequently did not provide us with the measure of general composability and modularity needed for our demands. This all suggested to instead pursue free monoidal categories as a representation of behavior.

In order to prepare the reader (and ourselves) for the exploration of free monoidal categories as a mathematical foundation for behavioral representation, we devoted two chapters to defining the tools we needed from category theory to construct such a representation. First, we reviewed in detail the two constituting conceptual components of free monoidal categories: monoidal categories and free constructions. The latter subject involved a review of the connection between universal properties in categories and adjoint functors, which provided us with the needed mathematical machinery for applying the adjoint functor theorem. Subsequently, we gave a detailed construction of the monoidal schemes of Joyal and Street, defined in [27], and elaborated on this construction to show that monoidal schemes form a co-complete category, possessing both arbitrary coproducts and coequalizers. Using monoidal schemes as generating structures, we gave a detailed construction for generating free monoidal categories along with the specific variant of free symmetric monoidal categories, proving what was needed to both establish this construction and to show, using adjoint functor theorem, that all our construction carried over all small colimits from monoidal schemes to monoidal categories.

A proposal was then given for a new representation of distributed behavior we called an OEG. We gave an intuitive informal account of these structures and how they could be used to represent behavior. OEGs are, in essence, acyclic ported block diagrams in which each block represents an event and each connection represents a dependency. The collection of event types, the primitive blocks out of which we can construct languages of these diagrams, constitute what we called an OES, the alphabet of our representation. Examples were given of OESs along with OEGs that could be built using them. Throughout this exposition, we showed how OEGs fulfill our demands for modularity and composability, and contrast with the other models we described previously.

We then proceeded to give a comprehensive formal definition of OESs and OEGs, defining parallel and sequential operations over them, as well as families of constant elements and special classes of OEGs. It was shown that the OEGs generated from a specific OES, along with the compositions and constants, formed a symmetric monoidal categories, providing an algebra to these representations. Using the results of Joyal and Street in [27], we showed that the symmetric monoidal category defined by the OEGs generated from a particular OES is in fact a free symmetric monoidal category. Consequently, the results regarding free symmetric monoidal categories that we described in our discussion of free monoidal categories apply directly to OEGs. Specifically, the co-completeness of the category of OESs provides a means to abstract, embed, combine, and constrain OESs, while adjoint functor theorem permits us to carry these operations all over to the corresponding spaces of OEGs.

Finally, we gave a detailed account of how OEGs can be used to defined a behavioral semantics for the π-calculus. This involved a brief introduction to π-calculus and a recapitulation of the conventional means of defining its reduction behavior via structural operational semantics. We show how to use this conventional account to reason out a set of basic behavioral event types that form an OES for term reduction. This involves some small modifications to the original semantics, most notably making replication into an explicit behavioral event. We then show how an OEG-based semantics arise from this OES that captures the essence of the original semantics in a true concurrent yet composable and modular fashion.

What we hope to have accomplished at this point is to have given the reader a convincing and thorough case for using OEGs as the basis for the formal study of distributed behavior. Nevertheless, what could be accounted for in the scope of this thesis has encompassed only the beginning of the potential research that can be done around OEGs, or perhaps further elaborations on these kinds of behavioral representations. There are many directions for further work that we will suggest, some of which have already been explored, albeit not to the completion needed to include them in this thesis. We will give a brief account of some of these areas of further work.

9.1 More Semantics

In Chapter 8, we gave an OEG-based semantics for reduction in the π-calculus in great detail as a demonstration of how OEGs could be used to define the semantics of concurrent programming languages. We also gave a simpler example in Chapter 7 of how to use OEGs to define the semantics of message passing sequential processes. These examples should hopefully provide enough conceptual intuition to see how a great many other programming languages, abstract calculi, and MoC could be given OEG-based semantics. Preliminary work has already been done exploring how to do this for variants of the π-calculus, as well as for DF models.

Variants of π-calculus

The most obvious extensions of π-calculus that could be given OEG semantics are the *asynchronous* variants, such as that of Boudol devised in [9]. In Milner's original version, which we have focused on, the primary communication reduction involves the rendezvous between a sending and receiving term in $\mathcal{N}_\pi$. As we recall,

$$\bar{x}\langle a\rangle \bullet P \mathbin{[\!]} x(y) \bullet Q \longrightarrow P \mathbin{[\!]} Q[y \mapsto a]$$

where we have elided choices for the moment. This reduction happens synchronously to the two incoming terms, or in other words, the send and the receive are both blocking operations. In some cases such as DF, it is more desirable to have only reads blocking in the fundamental semantics of the calculus. Therefore, in Boudol's variant, this reduction is broken up into separate send and receive reductions. The former of these is no longer a reduction proper, but a *dissociation* of a message $\bar{x}\langle a\rangle$ from a term $\bar{x}\langle a\rangle \bullet P$ freeing P to further reduce before the message is received (this *dissociation* is built upon a chemical metaphor in Boudol's paper). As a matter of fact, the $\bar{x}\langle a\rangle$ no longer a guard, but instead a proper term.

The receive reduction for an asynchronous π-calculus then is a simpler reduction

$$\bar{x}\langle a\rangle \mathbin{[\!]} x(y) \bullet Q \longrightarrow Q[y \mapsto a]$$

It is not hard to see how one would begin to modify the OEG semantics we have given to codifying this variant. The event type for communication depicted in Figure 8.2b, that has two incoming dependency types and two corresponding groups of outgoing dependency types would have to be modified to produce only the concurrent components of the receiver, since the sending term no longer has any. It would then remain to be determined whether the dissociation of messages, in essence sending them, would be an explicit event type in the OES, or implicit somehow in the structure of dependencies.

Along similar lines, perhaps travelling in the opposite direction, one could also conceive of a variant of π-calculus in which there is a reception of several simultaneous sends at once. In this variant, a receive term might look like

$$[x_1, \ldots, x_N](y_1, \ldots, y_N) \bullet Q$$

where each of x_k is a name at which the term must receive a message and y_k are the variables to which the corresponding message would be bound (a better syntax might of course be proposed). The corresponding reduction rule for this kind of a receive construct might look like the following.

$$\begin{aligned}&\bar{x_1}\langle a_1\rangle \bullet P_1 \mathbin{[\!]} \ldots \mathbin{[\!]} \bar{x_N}\langle a_N\rangle \bullet P_N \mathbin{[\!]} [x_1, \ldots, x_N](y_1, \ldots, y_N) \bullet Q\\ &\quad\longrightarrow P_1 \mathbin{[\!]} \ldots \mathbin{[\!]} P_N \mathbin{[\!]} Q[y_1 \mapsto a_1, \ldots, y_N \mapsto a_N]\end{aligned}$$

This might appear superficially to be a superfluous enhancement since in some cases it would be equivalent (for all intents and purposes) to replace this construct with a sequence of a succession of receive guards in an arbitrary order. But if one considers the effect on the sender, the above rule only permits any given sender to advance beyond its guard only when all senders reduce together. This constrains the possible reductions.

Constructing the OEG semantics for this simultaneous variant would make the above point regarding its semantic difference abundantly clear. The event type in Figure 8.2b would be modified to potentially take in many sending dependency types along with the receiving one, and produce as outgoing dependency types the components of all of them.

It might normally warrant some involved mathematical constructions to prove that these variants are genuinely different, particularly using structural operational semantics, since the comparison would be over sequences of reduction steps. But with OEG semantics one can easily see, given the event type variations we have proposed for these two variants, their the languages of OEGs are manifestly different topologically. The three different OES generate three different (non-isomorphic) free symmetric monoidal categories. Comparing and relating these semantics becomes a much more topological endeavor, and perhaps a far more intuitive one as a consequence. Sketching out these different kinds of OEGs with different kinds of event types reveals a tremendous amount of insight about the kinds of concurrency, asynchrony, and synchrony achieved in each case.

Other Models of Computation and Languages

Developing OEG-based semantics for other Models of Computation and for practical programming languages can proceed similarly to the way in which we showed it could be done for the π-calculus. The basic event types can be sketched out, the dependency types involved can be determined, and an OES can be constructed with them. Moreover, many of the constructs developed as event types will likely be topologically similar in nature to those we have already discussed in the context of π-calculus. In the case of DF models, the event types will likely be similar to those we discussed in the context asynchronous variants of π-calculus, involving separate send and receive event types. In the case of synchronous models, such as the SR or DE Models of Computation, defined in [14] and [11] respectively, event types accomplishing synchronization will likely follow along the same lines as those we discussed in latter variant of π-calculus, which can receive simultaneously.

In the context of CPS, there is little stopping one from complementing the event types involved in the computational model with event types modeling physical events, the occurrences of which are semantically entangled in causal relationships with the computational ones. The analogy we pursued in Chapter 2 between particle collisions in Quantum Field Theory and the dynamics of our Process Fields could be taken as an indication that OEG semantics can accommodate physical phenomena.

9.2 Mathematical Developments

A good deal of mathematical foundation is laid out in this thesis, particularly that of free symmetric monoidal categories which serve as the formal underpinning of OEGs. Yet, this work only scratches the surface of a more comprehensive mathematical study of OEGs. There are a couple particularly important questions that demand further investigation.

Infinite OEGs

A dimension inevitable to the study of OEGs as a representation of behavior, and alluded to in several places, is how to deal with infinite behaviors. As anyone who is familiar with this kind of formal work knows all too well, dealing with the infinite presents many challenges and complications. It would have been simple enough to permit the set of events in an OEG to be infinite, a change about which parallel composition would remain indifferent. But then it would raise a question about what the outgoing event interface would be for a OEG that goes on infinitely. How would sequential composition work? Would we only sequentially compose over the outgoing dependencies finitely deep in the OEG? Would it make sense to compose two infinitely deep OEGs?

Even in these simple questions we gloss over an important distinction that might be made between an infinitely deep OEG, with infinite chains of events, and an infinitely wide OEG, with infinite sets of concurrent events. We could partially, if not entirely, avoid the latter. Using Kőnig's lemma[1], we can be sure that a finitely deep OEG will only be infinitely wide if we either permit the event interfaces of event types to be infinite or have a parallel product of infinitely many disconnected components. But even then, at infinite depth there can still be infinitely many concurrent events and no less uncountably many of them. This many not be a problem however, and this may even serve as useful in certain situations.

[1]I had hoped to get to use this ubiquitous lemma at least once.

It may even be useful to have infinite event interface, which could remedy some of the issues with infinite depth. An example of this might be an open DF OEG which would typically have an infinite sequence of incoming and outgoing dependency. Seeing how to represent, let alone compose, these infinite interfaces cannot but strike one as challenging. However, some encouraging progress has already been made in this direction.

Even if the problem of infinite OEGs were worked out around the formal treatment given in Chapter 7, the more daunting task may be to work out the corresponding structures on the side of monoidal category theory so that spaces of OEGs that include infinite members remain connected to monoidal categories. If the right modifications could be made to the operators, spaces of OEGs could remain symmetric monoidal categories, but they would certainly no longer be free symmetric monoidal categories in the fashion developed in this thesis. Algebraic structures that include infinite elements often can no longer be free. Take a free monoid $\mathbf{Mon}A$ generated from an alphabet A. If we add in the infinite words to the underlying set, forming a set often notated A^{**} in computer science, the monoid can no longer be free, since the infinite elements cannot be generated from finite applications of the operators and constants to the alphabet.

The way to deal with this issue might follow from some suggestions we made in Chapter 3. Because free monoids have a natural prefix ordering on them, we can limit-complete (co-limit in categorical terms) the space in order to include the infinite elements, giving us a kind of free κ-CPO-monoid. Likewise, for a free monoidal category, we can show that the morphisms have a similar prefix ordering that could be used to generate limit elements along iterative applications of sequential composition (categorical composition). The limit elements would have to have well-defined event interface, meaning that along infinite chains of prefixes, the outgoing event interface would have to converge to something unique; themselves forming a chain with a limit.

Working out all of these details would involve considerable work on the purely mathematical side, unless such structures have already been investigated in category theory. To the best knowledge of the author, no such structures appear to have been defined in the literature.

Recursive Equations

A very related area that remains to be investigated further is that of recursive equations in the algebraic language of OEGs, which is of course that of free symmetric monoidal categories. In defining an OEG-based semantics of programming languages with explicit forms of recursion it would only make sense to interpret the recursive constructions in, for instance, a functional programming language, with recursive OEG equations. But of course, these equations would only be meaningful characterizations of semantics if it could be determined whether solutions could model them. This would at least depend on an understanding of infinite OEGs since these would be the natural candidates for solutions to these recursive equations. Analogous to the study of regular languages of strings in Kleene Algebras, it might be relevant to explore the possibility of regular kinds of OEGs, that involve periodic repetitions of sub-diagrams of events, as the space of solutions to certain kinds of recursive formulae.

9.3 Logic and Formal Verification

One of the original questions that lead to the developments in this thesis is one that remains unanswered by it. Yet, part of the motivation of pursuing the course of development herein was to lay a foundation that made it possible to even begin to answer this question. The question is of what kind of logic should be used in the specification of HDDAs in Process Fields. In the introduction, we suggested that to answer such a question would be difficult without having a sense of which structures, in our case behavioral representations, would serve to model the formulae of such a logic.

The problem might have been approached in the opposite direction, first making informal statements about HDDAs and formalizing them until the question could be raised of how to interpret them precisely. But this would still require an intuition of what about we are making these statements, and without a clear sense of the ontology of the space of computation that we are describing, it would be hard to determine which kind of informal statements would be meaningful.

Instead, we argued that having a behavioral representation first foremost would provide a common currency to both construction of languages with precise semantics and logics with precise models, serving the mediator between these two domains. We now have proposed such a common currency as the behavioral representation of OEGs, and thus, insomuch as one is convince that this is the representation we are looking for, we can address the question about logics and specification languages by asking more specifically: what formal language describes relevant properties of OEGs?

We have stated a number of times, in contrast to the logic we may be looking for, that LTL has been the convention for describing properties of sequential (albeit perhaps interleaved) traces [55]. If we move from interleaved traces to true concurrency, and more specifically, to OEGs as a representation of behavior, what is the correspondent to LTL? Although this question is yet to be answered, work has already been done to approach an answer from the perspective of OEGs.

Towards a Logic for OEGs

What are relevant features of OEGs that can be made formal? For starters, OEGs provide a lot of structure about which one might make a statement, in contrast to Event Structures and other generalized sequences. In excess of Event Structures, OEGs express information about precisely which role each event plays in the direct influence of another. Statements can be therefore made regarding both the events as well as the dependencies that constitute an OEG.

A simple and obvious logic that might be appropriate for specification, but is likely too broad for the purposes of verification, would simply be a first-order logic over the theory of symmetric monoidal categories. This logic has as its atomic propositions symmetric monoidal category equations, i.e.

$$\beta \circ (G \otimes F) = H \circ (K \otimes \beta \otimes \mathbf{id})$$

This would cover event-oriented questions. To capture questions regarding dependencies, the logic should be extended to also include statements about the incoming and outgoing event interfaces of OEGs, which are simply assertions about the inclusion of morphisms in symmetric monoidal category into their respective homsets, i.e.

$$G \in \mathbf{hom}(a,\, b)$$

What follows from this, is that we should also be able to state equations in monoid formed by the objects of the symmetric monoidal category, i.e.

$$a \otimes b = c \otimes b \otimes b \otimes e$$

Putting all of these kinds of formulae together with finite conjunctions, finite disjunctions, negation, and quantification over both OEG and dependency variables would most certainly constitute a powerful language for defining properties (collections) of OEGs.

Bibliography

[1] I. J. Aalbersberg. Theory of traces. *Theor. Comput. Sci.*, 60(1):1–82, September 1988.

[2] S. Abramsky. What are the fundamental structures of concurrency? we still don't know! In *Electronic Notes in Theoretical Computer Science, 162*, pages 37–41, 2006.

[3] Samson Abramsky. Two puzzles about computation. *arXiv preprint arXiv:1403.4880*, 2014.

[4] Gul Abdulnabi Agha. *Actors: a model of concurrent computation in distributed systems.* PhD thesis, 1985.

[5] Steve Awodey. *Category theory.* Oxford University Press, 2010.

[6] Henry Baker and Carl Hewitt. Laws for communicating parallel processes. Technical report, MIT Artificial Intelligence Laboratory, 1977.

[7] Carlos Baquero and Nuno Preguiça. Why logical clocks are easy. *Commun. ACM*, 59(4):43–47, March 2016.

[8] Michael Blackstock and Rodger Lea. Toward a distributed data flow platform for the web of things (distributed node-red). In *Proceedings of the 5th International Workshop on Web of Things*, pages 34–39. ACM, 2014.

[9] Gérard Boudol. Asynchrony and the Pi-calculus. Research Report RR-1702, INRIA, 1992.

[10] J Dean Brock and William B Ackerman. Scenarios: A model of non-determinate computation. In *Formalization of programming concepts*, pages 252–259. Springer, 1981.

[11] Adam Cataldo, Edward Lee, Xiaojun Liu, Eleftherios Matsikoudis, and Haiyang Zheng. A constructive fixed-point theorem and the feedback semantics of timed systems. In *Workshop on Discrete Event Systems*, July 10-12 2006.

[12] William Douglas Clinger. *Foundations of actor semantics.* PhD thesis, 1981.

[13] Jack B. Dennis. First version of a data flow procedure language. In B. Robinet, editor, *Programming Symposium*, volume 19 of *Lecture Notes in Computer Science*, pages 362–376. Springer Berlin Heidelberg, 1974.

[14] Stephen A. Edwards. *The Specification and Execution of Heterogeneous Synchronous Reactive Systems*. PhD thesis, EECS Department, University of California, Berkeley, 1997.

[15] J. Eker, J.W. Janneck, E.A. Lee, Jie Liu, Xiaojun Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Yuhong Xiong. Taming heterogeneity - the ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, Jan 2003.

[16] Colin J Fridge. Timestamps in message-passing systems. In *Proceedings of the 11th Australian Computer Science Conference*, pages 56–66, 1988.

[17] Dan R Ghica. Function interface models for hardware compilation. In *Formal Methods and Models for Codesign (MEMOCODE), 2011 9th IEEE/ACM International Conference on*, pages 131–142. IEEE, 2011.

[18] Carl Hewitt. Actor model of computation: scalable robust information systems. *arXiv preprint arXiv:1008.1459*, 2010.

[19] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence*, pages 235–245. Morgan Kaufmann Publishers Inc., 1973.

[20] Thomas Hildebrandt, Prakash Panangaden, and Glynn Winskel. A relational model of non-deterministic dataflow. In *International Conference on Concurrency Theory*, pages 613–628. Springer, 1998.

[21] Charles Antony Richard Hoare. Communicating sequential processes. In *The origin of concurrent programming*, pages 413–443. Springer, 1978.

[22] Charles Antony Richard Hoare et al. *Communicating sequential processes*, volume 178. 1985.

[23] Paul Hudak and Steve Anderson. Pomset interpretations of parallel functional programs. In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 234–256. Springer Berlin Heidelberg, 1987.

[24] Joyent Inc. Nodejs, 2016. [Online; accessed 27-September-2016].

[25] Thomas Jech. *Set theory*. Springer Science & Business Media, 2013.

[26] Bengt Jonsson. A fully abstract trace model for dataflow networks. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 155–165. ACM, 1989.

[27] André Joyal and Ross Street. The geometry of tensor calculus, i. *Advances in Mathematics*, 88(1):55–112, 1991.

[28] André Joyal and Ross Street. Braided tensor categories. *Advances in Mathematics*, 102(1):20–78, 1993.

[29] André Joyal, Ross Street, and Dominic Verity. Traced monoidal categories. In *Mathematical Proceedings of the Cambridge Philosophical Society*, volume 119, pages 447–468. Cambridge Univ Press, 1996.

[30] Gilles Kahn and David Macqueen. Coroutines and Networks of Parallel Processes. Research report, 1976.

[31] Joost N Kok. A fully abstract semantics for data flow nets. In *International Conference on Parallel Architectures and Languages Europe*, pages 351–368. Springer, 1987.

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

[33] Elizabeth Latronico, Edward Lee, Marten Lohstroh, Chris Shaver, Armin Wasicek, Matthew Weber, et al. A vision of swarmlets. *Internet Computing, IEEE*, 19(2):20–28, 2015.

[34] Edward A Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.

[35] Edward A Lee, Jan Rabaey, Björn Hartmann, John Kubiatowicz, Kris Pister, Alberto Sangiovanni-Vincentelli, Sanjit A Seshia, John Wawrzynek, David Wessel, Tajana Simunic Rosing, et al. The swarm at the edge of the cloud. *IEEE Design & Test*, 31(3):8–20, 2014.

[36] Marten Lohstroh and Chris Shaver. The universal information identifier, February 2014.

[37] Roberto Lucchi and Manuel Mazzara. A pi-calculus based semantics for ws-bpel. *The Journal of Logic and Algebraic Programming*, 70(1):96–118, 2007.

[38] Nancy A Lynch and Eugene W Stark. A proof of the kahn principle for input/output automata. *Information and Computation*, 82(1):81–92, 1989.

[39] Saunders Mac Lane. *Categories for the working mathematician*, volume 5. Springer Science & Business Media, 1978.

[40] Saunders MacLane. Natural associativity and commutativity. *Rice Institute Pamphlet-Rice University Studies*, 49(4), 1963.

[41] Eleftherios Matsikoudis and Edward A. Lee. The fixed-point theory of strictly causal functions. *Theoretical Computer Science*, 574:39 – 77, 2015.

[42] Friedemann Mattern. Virtual time and global states of distributed systems. *Parallel and Distributed Algorithms*, 1(23):215–226, 1989.

[43] Antoni Mazurkiewicz. Concurrent program schemes and their interpretations. *DAIMI Report Series*, 6(78), 1977.

[44] Antoni Mazurkiewicz. Traces, histories, graphs: Instances of a process monoid. In *Mathematical Foundations of Computer Science 1984*, pages 115–133. Springer, 1984.

[45] Antoni Mazurkiewicz. Trace theory. In *Petri nets: applications and relationships to other models of concurrency*, pages 278–324. Springer, 1986.

[46] Robin Milner. *A calculus of communicating systems.* 1980.

[47] Robin Milner. *Communicating and mobile systems: the pi calculus.* Cambridge university press, 1999.

[48] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, i. *Information and Computation*, 100(1):1 – 40, 1992.

[49] N. Mor, B. Zhang, J. Kolb, D. S. Chan, N. Goyal, N. Sun, K. Lutz, E. Allman, J. Wawrzynek, E. A. Lee, and J. Kubiatowicz. Toward a global data infrastructure. *IEEE Internet Computing*, 20(3):54–62, May 2016.

[50] Mogens Neilsen, Gordon Plotkin, and Glynn Winskel. *Petri nets, event structures and domains.* University of Edinburgh, 1979.

[51] Catuscia Palamidessi. Comparing the expressive power of the synchronous and asynchronous *pi*-calculi. *Mathematical Structures in Computer Science*, 13(05):685–719, 2003.

[52] David Park. On the semantics of fair parallelism. In *Abstract Software Specifications*, pages 504–526. Springer, 1980.

[53] Benjamin C Pierce. *Basic category theory for computer scientists.* MIT press, 1991.

[54] GD Plotkin. A structural approach to operational semantics. 1981.

[55] Amir Pnueli. The temporal logic of programs, Oct 1977.

[56] Vaughan Pratt. Modeling concurrency with partial orders. *International Journal of Parallel Programming*, 15(1):33–71, 1986.

[57] Claudius Ptolemaeus, editor. *System Design, Modeling, and Simulation using Ptolemy II.* Ptolemy.org, 2014.

[58] Vladimiro Sassone, Mogens Nielsen, and Glynn Winskel. A classification of models for concurrency. In *CONCUR'93*, pages 82–96. Springer, 1993.

[59] Vladimiro Sassone, Mogens Nielsen, and Glynn Winskel. Deterministic behavioural models for concurrency. In *International Symposium on Mathematical Foundations of Computer Science*, pages 682–692. Springer, 1993.

[60] Dana Scott. *Outline of a mathematical theory of computation.* Oxford University Computing Laboratory, Programming Research Group, 1970.

[61] Dana S Scott and Christopher Strachey. *Toward a mathematical semantics for computer languages*, volume 1. Oxford University Computing Laboratory, Programming Research Group, 1971.

[62] P. Selinger. *A Survey of Graphical Languages for Monoidal Categories*, pages 289–355. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[63] Chris Shaver, Marten Lohstroh, and Matt Weber. Semantics of the swarm, October 2014. Poster presented at the 2014 TerraSwarm Annual Meeting.

[64] Carolyn L Talcott. Composable semantic models for actor theories. *Higher-Order and Symbolic Computation*, 11(3):281–343, 1998.

[65] IBM Emerging Technologies. Node-red, 2016. [Online; accessed 27-September-2016].

[66] Stavros Tripakis, Christos Stergiou, Chris Shaver, and Edward A Lee. A modular formal semantics for ptolemy. *Mathematical Structures in Computer Science*, 23(04):834–881, 2013.

[67] Steven Weinberg. *The quantum theory of fields*, volume 1. Cambridge university press, 1996.

[68] Glynn Winskel. *Events in Computation.* PhD thesis, The University of Edinburgh, 1980.

[69] Glynn Winskel and Mogens Nielsen. Models for concurrency. *DAIMI Report Series*, 22(463), 1993.