mrb: Formalizing Concurrency, Distribution, and Mobility

Formalizing Concurrency, Distribution, and Mobility

Carlos A. Varela's *Programming Distributed Computing Systems: A Foundational Approach* is a slim but impressive volume which takes a methodical approach to exploring the landscape of distributed computation, with very rich results.[1] A major premise of the book, that *an awareness and understanding of the various models of concurrent computation is necessary for reasoning about various methods of distributed computation* inspired this post, which will cover the major models used.

A computational model is a formal mathematical model that is used as a representation of computation, the most prominent example being the the (lambda) calculus, developed by Church and Kleene in the 1930s. Varela's coverage of the models of computation is carried out with precision, and is based on the idea that, at an even higher level than the discussion of distributed computing:

> "Theoretical models of computation are critical to be able to reason about properties of software, including the programming languages used to develop the software. Examples of software properties include expressive power, correctness, performance and reliability."

Nothing revolutionary perhaps, but no less meaningful. Considering that each of the five models introduced could alone occupy several years if not a lifetime of research, this post aims to give an overview, and to provide some insight into the relationship that these models have to each other. Many of the most

practical lessons one can take away from theoretical computer science result from comparing and contrasting computational models, and that practice is at the heart of this book.

Covered in the text are the **(lambda) calculus**, the **(pi) calculus**, the **actor model**, the **join calculus**, and the **ambient calculus**. Most distributed computing work uses ideas taken from the calculus or the actor model, though others are becoming more prominent. For me the impressive aspect of Varela's work is how through rigorous comparisons and painstaking research, he shows the direct relationships between the computational models and the semantics of the programming languages which implement them. The direct encoding of the concept of concurrency into the post calculus models is an abstract but powerful concept that Varela communicates quite effectively.

The Calculus The earliest and best known model is notable for its ability to concisely express sequential computation. While the calculus does not explicitly encode concurrency, it is the basis for all other models (even appearing "inside" the actor model) and understanding sequential computation is crucial for understanding concurrent computation.

The calculus (like all of the other models presented) is Turing-complete, that is "any computable function can be expressed and evaluated using the calculus." The usefulness of a formal model like the calculus is that its simplicity and high level of abstraction make it very easy to reason about. This quality makes it well suited for studying and formalizing even complex programming language concepts.

A direct connection with programming languages is made in covering the various evaluation orders that have been explored for the calculus. Evaluation order refers to the rules for *reducing* or "solving" calculus expressions. This concept maps cleanly onto the semantics of modern functional programming languages. *Normal-order* is the simplest in many ways and is the "best-behaved" with respect to termination guarantees, but most programming languages use applicative order, for efficiency reasons. The Haskell programming language is well known for implementing a *by-need* or *lazy* evaluation order where partially computed functions are passed around to represent the results of not-yet-needed work. The following trivial example shows how values are passed in to a function and evaluated:

```
( x.x*2 2)
2*2
4
```

The value of 2 is captured by the free variable x in a substitution. The function `x*2` becomes `2*2` which becomes `4`. This should start to give you a feel for the semantics of the calculus, which are simple but very powerful, but by no

means represents what is possible. If the calculus is new to you, seek its beauty elsewhere.

In addition to evaluation order, other familiar concepts to functional programmers are covered included Currying, higher-order programming, sequencing, and recursion. Finally some details on how numbers and booleans can be represented in the calculus are discussed, which is a fascinating bit of detail that many professional programmers might not be familiar with. *Combinators used to express numbers* and *true and false as a curried function* is not day-to-day programming, but can provide some deep insights into the foundations of the systems we become intimately familiar with through use.

While it is amazing how deeply the model tracks with a host of mathematical concepts, as the author proceeds to introduce other models, the depth of the fundamental hook of *explicitly representing multiple processes*, which is missing from this model, becomes clear.

The Calculus The Calculus is a formal model for specification, analysis and verification of systems composed of communicating concurrent processes, developed by Robin Milner et al in 1992, as a continuation of Milner's earlier work in specifying a process calculus known as the calculus of communicating systems (CCS). In the calculus, *concurrent computation is modeled as processes communicating over shared channels.* The operational semantics of the calculus is presented, which explains how a well-defined set of inference rules determine the evaluation order and scoping rules of a calculus expression.

The calculus syntax includes operations for reading and writing values on channels, in addition to operations for declaring, composing, comparing, and executing processes. Take the following basic example of a calculus interaction:

```
a(x).P | āb.Q.
P{b/x} | Q.
```

The `|` operator denotes *concurrent composition*. Process `P` and process `Q` are communicating over channel `a`. The first line represents the initial state, and the second line represents the state post-transition. The `Q` process will write the value `b` over the channel `a`, written here as `ā` because it is being written to. The value `b` has been communicated from `Q` to `P` over `a`. Being able to model concurrent processes in practice make expressing complex systems easier and formal reasoning about them possible.

The model relies on the concept of *strong bisimilarity* to provide a notion of *process equivalence*. In using modern programming languages, we sometimes take for granted our ability to compare two abstract entities and ask the computation at hand if they are in some way equivalent. This idea drills all the way down to the model level, where those decisions have to be made, based on the rules of reduction, possible inputs, etc. In other words, it is mathematically

possible to prove, even in the full calculus, that two processes are functionally equivalent.

We also see various examples modeled in the calculus that very viscerally demonstrate how concepts common to a practitioner (a "get server" and a "set server" are modeled, along with a mutex and an implementation of the dining philosophers problem) relate to the expressiveness of the underlying computational model. Varela says "The semantics of a concurrency model explains precisely the behavior of its elements." We see written "in math" what is normally written in a scripting language to make similar points. Varela fearlessly tackles this challenge, providing these examples not only in each computational model discussed, but also in programming languages which rely on these models later in the book.

The key difference between the calculus and its predecessors in CCS and C.A.R. Hoare's communicating sequential processes (CSP) is its explicit ability to send channel names between processes, allowing dynamic topologies, a property known as *mobility*. The remaining models in the book also have this property, which was inspired by the actor model. Since the calculus can explicitly model changing topologies, it takes a very strong step in the right direction of *unifying the practice with the underlying computational model.*

The Actor Model Of the models covered in this book, the actor model is probably the best known besides the calculus. Embodied in programming libraries and languages like Erlang, the actor model defines *actors* as the primary components of computation, where shared-nothing memory and asynchronous communication are assumed. Varela's snappy definition says that:

> "Actors are first-class history-sensitive entities with an explicit identity used for communication."

In practice, actors are somewhat complex when combined in large numbers. The rules on a micro level, however, are simple. Actors can perform one or more of the following actions upon receiving an asynchronous message:

1. send a message to an *acquaintance,* an actor whose identity is known to the actor *a*,
2. create a new actor *a'*, with a given *behavior b*, or
3. become *ready* to receive a new message with new *behavior b*

This behavior allows the actor model to be an *open* system, so called because new components can be added by creating new actors dynamically, and because an actor can change behaviors dynamically. This means that a few actors can create a large graph of actors, and that the messages between them can include the names of actors, making very complex systems possible.

The actor model assumes *fairness* in a way that the other models do not - message delivery is guaranteed, and thus message processing is guaranteed under the correct conditions. It is worth noting that while assuming fairness makes reasoning simple, it potentially makes implementation challenging and expensive.

The formalized version of the actor model presented in this book is a more modern semantics that is sensitive to the realities of the implementations of actors in programming languages. The computational expense of creating new actors impacted the decision for which primitives to include, favoring a more efficient approach. Varela has this to say on the influence of the actor model on programming language theory:

> "The actor model has influenced the development of several programming languages. Sussman and Steele developed Scheme in 1975 in an attempt to understand the actor model first conceived by Carl Hewitt (Sussman and Steele, 1998). Early actor languages developed at MIT include PLASMA (Hewitt, 1975) and Act1 (Lieberman, 1981). More recent actor languages include Acor, Rosette, ABCL, Erlang, E, and SALSA."

Actors stand outside the other process calculi presented in many ways - seemingly spontaneously conceived and radically different in formation then the others, actors seem more tangible, can be explained simply, and model real world problems well. But because they are "higher level" and guarantee fairness, but do not explicitly encode notions of location, for instance, research continues toward designing models that approach being isomorphic with production distributed systems.

The Join Calculus At this point in the book it becomes clear that we are searching for a model which allows for explicit reasoning about *distribution* and *mobility*. As straightforward as the translation from the actor model to a programming language is, given my background, the join calculus was equally abstract and bizarre to encounter. Varela has a concise definition, which is nicely fleshed out throughout the chapter that covers it:

> "Concurrent computation is modeled in the join calculus as a reflective chemical abstract machine, where heating and cooling of molecules and atoms occurs according to *reaction rules* that can evolve over time."

Molecules are processes running in parallel, which can combine and evolve into other processes over time according to the *reaction rules*. This sounds similar to how in the actor model, actors can change behaviors. In the join calculus, distribution and mobility are represented with a combination of molecules

and *reaction sites* that model localities and correspond to rules with *statically scoped names*. Communication is constrained to reaction sites which are lexically scoped definitions. The join calculus, like the calculus, but unlike the actor model, does not address fairness.

The join calculus is the first model I encountered that represents systems using the idea that *conditional actions in a distributed system can be modeled as conditional reactions occurring at explicitly visited sites.* Isolating the logic of a system to the sites where they occur is definitely a step in the right direction. To show a very simple example, as shown below: reactions are denoted `D` or `J` `P` where `J` denotes a *join pattern* that molecules must match for a reaction to take place, and `P` denotes the resulting molecules from the reaction.

```
D = ready<printer> | job<file>   printer<file>
D   ready<laser> | job<f1>
D   laser<f1>.
```

The initial solution shows that if the pattern `ready<printer> | job<file>` is matched, the reaction `printer<file>` will take place, ostensibly sending the data from `file` to the `printer`. The second line shows `ready<laser>` and `job<f1>` visiting the reaction site, thus activating the condition above. The final line shows `laser<f1>`, the data from `f1` is sent to the `printer` called `laser`. This kind of program can become arbitrarily elaborate, but this simple program shows a novel approach. The challenge remains of how to represent the rules in a distributed system in as concise, declarative, and central manner. There is a later chapter in *Varela* that is devoted to programming in *JOCaml*, an OCaml variant that uses the join calculus model for concurrency.

Processes in the join calculus can be compared for equivalence based on the idea that "Two programs are considered to be *equivalent* if they behave the same when placed in any observing contexts." The join calculus is extremely complex and the semantics are hard to grasp. While I still struggle with them, I have taken away the idea that the join calculus provides at least the power of the calculus with some apparent gains in expressiveness. Additionally since the join calculus explicitly reasons about *distribution* and *mobility,* the challenges of addressing locality, for example, in distributed systems becomes possible where it is now very challenging.

The Ambient Calculus Besides having an awesome name, the ambient calculus is a fascinating formulation of a very difficult type of system to model - one where *mobility* and *hierarchies of process* are explicit. From a paper by the authors of the model:

> "The ambient calculus is a process calculus whose basic abstraction, the ambient, represents mobile, nested, computational environments,

> with local communications. Ambients can represent the standard components of distributed systems, such as nodes, channels, messages, and mobile code. They can also represent situations where entire active computational environments are moved, as happens with mobile computing devices, and with multi-threaded mobile agents." [2]

Varela's treatment of the ambient calculus conveys the novelty of the expressive power of the model, illustrating how *ambients,* which contain calculations, are reduced from complex expressions to calculated outcomes. Notions of scope, memory sharing and fairness are covered, as well as how the semantics of the model relates to the others presented. The following example is given to introduce the expressive powers and aims of the model:

```
m[p[out m.in n.<M>]] | n[open p.(x).Q].
m[] | n[open p.(x).Q] | p[in n.<M>].
m[] | n[open p.(x).Q | p[<M>]].
m[] | n(x).Q | <M>].
m[] | n[Q{M/x}].
```

These five lines represent an initial state, three transitions, and a final state for a simple interaction where a network packet `p` moves between two machines on a network represented by two processes `m` and `n`. The keywords `out`, `open`, and `in` should stand out, as they are used to represent the notion of ambients representing mobile data in a system. `<M>` is data ready to be written and `(x).Q` is data to be read. To step through:

1. Process `m` is an ambient which contains one other ambient, `p`. Ambient `n` contains some instructions pertaining to ambient `p`.
2. Ambient `p` moves outside of ambient `m`.
3. Ambient `p` moves inside ambient `n`.
4. The `open` expression inside ambient `n` is now activated: an ambient named `p` is on the same level and a message is ready to be sent.
5. `p` has been opened and destroyed, and the processes can now interact, the data is read, and instances of `x` are replaced with `M`.

While a bit confusing out of context, the above should illustrate how far we have come in being capable of modeling in a formal sense what we have had to deal with on a *library level* for so long. Concurrency, distribution, and mobility being baked into our formal models can have potentially huge payoff. As Varela says:

> "The key difference between the calculus and the ambient calculus is the latter's ability to model hierarchical process boundaries, movement of *active* ambients across those boundaries, and the restricted interprocess communication to inside an ambient boundary."

Unique to the ambient model are the concepts of *entry, exit,* and *open,* which are the operations the model uses to specify the movement of ambients. The notation is illustrated and although even basic examples take time to get through, it is worth it to see how a correct program concerning the communication of distributed processes might look, however tiny.

In fact the idea that the calculus is capable of explicitly modeling mobility and hierarchies makes me believe that some day we will have verified distributed programs written in higher level languages whose semantics are modeled after advanced models of computation such as the ambient or join calculi.

Ontological Commitments, Expressive Power, and Semantics One of the highlights of the book is the chapter that acts as a bridge between the description of the various models and the example implementations. After laying out the models, we see how to compare them, which prepares us well for how they will be implemented in a programming language and used in various examples.

The **ontological commitments** of each model consist of two primary decisions - **synchronous vs. asynchronous communication** and **shared vs. distributed memory**. The actor model stands out amongst all of the others: it alone assumes an *asynchronous* approach to communication and a *distributed* approach to modeling memory. If you've wondered about the depth of Hewitt's work on the actor model but haven't been convinced, this chapter will likely give you pause - it is truly a radical, simple, powerful abstraction.

The section about the **expressive power** of each model is dense but fascinating. Each model is shown to be *Turing complete,* and each is expressed in terms of either the or the calculus, which is mind-bending and something that I surely do not fully grasp the depth of. You might, in which case I recommend this book highly.

Finally the **semantics** of each model is discussed, with the following properties being highlighted:

- Can it model *unbounded concurrency*? According to Varela:

  > "One of the key differences between earlier models of concurrency – such as Petri nets, calculus of communicating systems (CCS), or communicating sequential processes (CSP) – and more recent ones is the latter's ability to model unbounded concurrency."

In this context, *unbounded concurrency* relates to the expressive power of the model.

- *Composability* refers to the ability to combine simpler systems into more complex ones. In order to be composable, a model must be modular.

Modularity relates to how reasoning that applies to the parts of a system hold when the system becomes more complex through composition. Each of the modern models of concurrency supports composition through an interface that allows the model to consider processes interacting with each other.

- *Fairness* is "a semantic property that ensures that valid (infinite) computation paths do not consistently ignore one of the possible ways in which a computation may evolve." The only model that explicitly supports fairness is the actor model. The others are designed in a way which hopefully mitigates this necessity.

The discussion of the final two semantic properties, *distribution* and *mobility*, drive home some of the central points this book is trying to make.

> "Distributed computing is inherently concurrent. However, distributed aspects go far beyond concurrency. Of particular importance from a modeling perspective is the capability to reason about the location (and potential co-location) of concurrent computations, the heterogeneous cost of interaction, the security aspects of interaction across multiple locations, and the potential for partial failures."

Each of the models presented allow us to reason about distribution to varying degrees. The calculus has no explicit representation of locations, although it can implement a variety of (potentially expensive) abstractions to represent mobility. The actor model similarly does not provide a semantic distinction between processes on the same machine and processes distributed across a network.

The formal aspects of this chapter were very helpful in my attempts to understand the subtleties and value of each model, and the categories above are valuable when evaluating other models.

Conclusion The five computational models presented as the basis for understanding concurrent computation in Varela's *Programming Distributed Computing Systems* present an array of approaches for representing computation in a distributed system. What is made explicit in this text is that computational models exist which explicitly model qualities of distributed computation. Programming languages based on these computational models will inherently be more powerful and potentially more expressive (if they are implemented properly) because they do not have to consider or implement qualities which their models do not account for.

The last part of the book covers programming languages which implement these various models. While far from perfect, these languages and their summation provides evidence that there is purposeful work to be done if we are to achieve

a natural system of designing programs destined to run on a distributed platform. Layers and layers of software currently represent what could be shed if assumptions were baked into the design of systems from the outset. Distributed systems practitioners could benefit from researching the various process calculi and their predecessors, to appreciate the deep connection between their work and the theoretical underpinnings which sometimes seem so distant.

Works Cited *All quotes unless otherwise noted from Varela*

[1] Varela, Carlos A. *Programming Distributed Computing Systems: A Foundational Approach* MIT Press, hardcover, ISBN 978-0-262-01898-2, 2013

[2] Luca Cardelli, Giorgio Ghelli, and Andrew D. Gordon. 2002. Types for the ambient calculus. Inf. Comput. 177, 2 (September 2002), 160-194.

Michael Robert Bernstein
@mrb_bk
github.com/mrb
michaelrbernstein@gmail.com