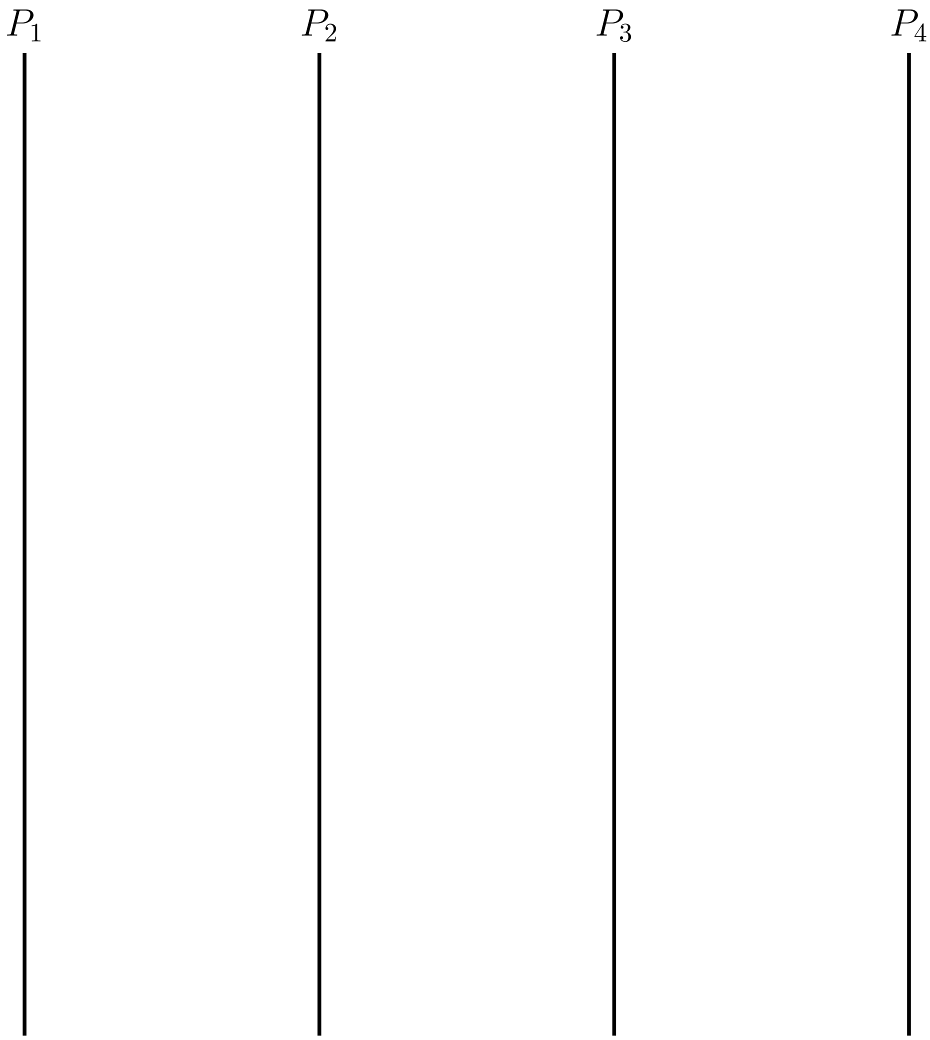
CS3110 Spring 2017 Lecture 22: Distributed Computing with Functional Processes continued

Robert Constable

Topics

1. *Message sequence diagrams* and event structures.
2. A General Process Model (GPM) for asynchronous distributed computing.
3. Coding 2/3 consensus as a functional process.
4. A Logic of Events for reasoning about protocols.

1 Message sequence diagrams for consensus – ($3f + 1$ case)



2 General Process Model (GPM)

Reasoning about protocols requires a precise yet realistic mathematical model with evaluation rules as we have for OCaml programs. Adding rules for sending and receiving messages is more complex than the evaluation rules used for core OCaml because rules about the *communication infrastructure* are also needed. At one extreme is communication over the internet and at the other, communication in a data center.

The communication layer introduces a level of *uncertainty*. This includes unpredictability, but not necessarily *randomness* in the sense of probability theory because there might not be a known distribution function that characterizes the unpredicatable behavior. The formal models of communication are limited in various ways, leading to a plethora of formal models for concurrency and communication such as Petri nets (one of the oldest), communicating sequential processes (CSP, Hoare [4]), Calculus of Communicating Systems [10] (CCS, Milner), IO Automata (Lynch [8]), Message Automata ([1, 3]), and several more [9].

Here we use a very simple method called a General Process Model (GPM [2]) tailored to treat asynchronous protocols in a functional style.

2.1 Overview

We will not study this model in detail, although it is precisely defined and presented in various publications mentioned below. Here the basic ideas are presented informally. The goal is to be more detailed about the notion of an *asynchronous system* and the idea of a *functional process.* The details are presented in the article *Generating Event Logics with Higher-Order Processes as Realizers* [2], and they have been implemented in conjunction with the Nuprl proof assistant *Developing Correctly Replicated Databases Using Formal Tools* [15].

A system consists of *components.* A component has a *location*, an address for sending it messages. It has a *internal* part which is the computation process. It also has an *external part* used for communication. Communication is done with *messages.* These can hold data and processes. A distributed system computes in an *environment* which provides communication among components. In the next section we present more details of this model. This will not be a topic that we cover on the final exam nor do we expect that PS6 implementations will use all of these concepts. We sketch the ideas to show that the concept of functional processes makes sense and is useful in practice.

2.2 Basic Concepts of the General Process Model

The following section is taken from [2]. It provides an overview of our model of distributed computation and the concepts we use to reason about distributed functional processes. We have used concepts from Computational Type Theory (CTT) to define the italicized terms completely precisely. The details of this approach could be incorporated into a course at this level once some of the simpler ideas have found their way into lower level courses.

A *system* consists of a set of *components*. Each component has a *location*, an *internal* part, and an *external* part. Locations are just abstract identifiers. There may be more than one component with the same location.

The internal part of a component is a *process*—its program and internal (hidden) state. The external part of a component is its interface with the rest of the system. In these notes this interface will be a list of *messages*, containing either *data* or a process, each labeled with the location of its recipient. The "higher order" ability to send a message containing a process allows a system to grow by "forking" or "bootstrapping" new components. (The external part can also be used to model the shared memory accessible to components at the same location, but will not be discussed in this paper.)

A system computes in steps as follows. In each step, the *environment* may *choose* and remove a message from the external part of a component. If components exist at the location to which the message is addressed, each of them receives the message as input and computes a pair consisting of a process, which becomes the next internal part of the component, and a list of messages, which is appended to the current external part of the component. If the chosen message is addressed to a location that is not yet in the system, then a *boot process* creates a new component at that location. The boot process to be used is supplied as a system parameter.

An unbounded sequence of steps, starting from a given system and using a given boot-process, is a *run* of that system. From a run of a system we derive an abstraction of its behavior by focusing on the *events* in the run. The events are the pairs, $\langle x, n\rangle$, of a location and a step (a "point in space-time") at which location x gets an input message at step n (i.e."information is transferred"). Every event has a location, and there is a natural *causal-ordering* on the set of events, the ordering first considered by Lamport [7]. This allows us to define an *event-ordering*, a structure, in which the causal ordering $<$ is transitive relation on E that is well-founded, and locally-finite (each event has only finitely many predecessors). Also, the events at a given location are totally ordered by $<$. The information, $info(e)$, associated with event e is the message input to $loc(e)$ when the event occurred.

2.3 Leader Election in a Ring

We have found that requirements for distributed systems can be expressed as (higher-order) logical propositions about event-orderings. To illustrate this we present a more thorough account of *leader election* in a group of *processes arranged in a ring* which we covered in the previous lecture.

Each participating component will be a member of one or more groups and each group has a name. A message $\langle G, j\rangle$ from the environment to component i informs it that it is in group G and has neighbor j in group G. We assume that, by the time the protocol begins, each such group is a ring, that is, the graph of the relation $j = neighbor(G, i)$ is a simple cycle. When any component in a group G receive a message $\langle [start], G\rangle$ it starts the leader election protocol whose goal is to choose one member of group G to be the leader and inform every member of G of the leader's location (presumably as the first step in a more complex protocol). To make this easy we also assume that each component at location i has a unique identifier $uid(i)$ that is a number—so that the uid's can be ordered.

The simple protocol is this: every component that receives a start message proposes itself by sending, to its neighbor, its uid in a message with header *propose*. Every component that receives a proposal with a uid, p, different from its own uid, u, proposes the maximum, $\max(u, p)$ to its neighbor. A component i that receives its own uid in a proposal is the leader and so sends a message with its location, i, and header *leader*. Every component other than the leader that receives a leader message forwards the message to its neighbor.

We describe protocols like this by classifying the events in the protocol. The events in this protocol are the start events, the propose events and the leader events. The components can recognize events in each of these classes (in this example they all have distinctive headers) and they can associate information with each event (e.g. the group G, the proposed uid, the location of the leader). Events in some classes cause events with related information content in other classes.

In general, an *event class* X is function on events in an event ordering that partitions the events into two sets, $E(X)$ and $E - E(X)$, and assigns a value $X(e)$ to events $e \in E(X)$. In our example, let us suppose that the list xs contains the locations of all the components that are participating in the protocol and might be members of the groups. An event e that is the receipt of a start message $\langle [start], G\rangle$ at a location $i \in xs$ is a member of an event class *Start*, with value $Start(e) = G$. Such classes, defined by a list of locations and a particular message header, are the *basic event classes*. Likewise, we may define basic classes *Propose* and *Leader* with values of the form $Propose(e) = \langle G, p\rangle$ and $Leader(e) = \langle G, x\rangle$. When an event in any of these basic classes occurs, the receiving component, at location $i \in xs$, will be able to associate additional pieces of information with the event, such as its $uid(i)$, or its location i, or $neighbor(G, i)$ from the most recent message from the environment. For example, we define event class $Start^+$ to have the same events as class *Start* but assign values given by

$$\begin{array}{rcl} Start^+(e) & = & \langle G, uid(i), j\rangle \\ \textbf{where} & & i = loc(e),\ j = neighbor(G, i) \end{array}$$

Similarly, we define $Propose^+$, and $Leader^+$ by

$$\begin{array}{rcl} Propose^+(e) & = & \langle G, p, i, uid(i), j\rangle \\ Leader^+(e) & = & \langle G, x, i, j\rangle \end{array}$$

To describe the leader election protocol in terms of these event classes, we declare that every event e with $Start^{+}(e) = \langle G, uid, j\rangle$ causes an event e' with location j and value $Propose(e') = \langle G, uid\rangle$. Every event e with $Propose^{+}(e) = \langle G, p, i, uid, j\rangle$ for which $p \neq uid$ causes an event e' with location j and value $Propose(e') = \langle G, \max(p, uid)\rangle$. Every event e with $Propose^{+}(e) = \langle G, p, i, uid, j\rangle$ for which $p = uid$ causes an event e' with location j and value $Leader(e') = \langle G, i\rangle$. Every event e with $Leader^{+}(e) = \langle G, x, i, j\rangle$ for which $x \neq i$ causes an event e' with location j and value $Leader(e') = \langle G, x\rangle$.

Clearly, these constraints (and the assumption that group G forms a ring) imply that after a *Start* event, the member $\max \in G$ with the maximum $uid_{\max}$ must eventually propose $uid_{\max}$ and this will be proposed by all members of the group, until component max receives its own $uid_{\max}$. It will then cause a *Leader*-event with value $\langle G, \max\rangle$ at its neighbor and this will be forwarded around the ring, so every member of the group is informed of the location max. The formal proof of these statements is easily constructed using standard logical methods. (If we want to be sure that all *Leader*-events for G have the same value, then we also need constraints that say that *Propose* and *Leader* events are caused *only* by the above rules.)

Programmable classes

Each event class in this example is *programmable.* A class X is programmable if at each location l there is a process that can recognize X-events at l and compute their values using only information received at l.

We describe distributed computation by defining programmable event classes and specifying their interactions in term of *propagation rules* and *propagation constraints.*

Propagation rules and constraints If A and B are event classes, the *propagation rule* $A \overset{f}{\Rightarrow} B@g$ is a proposition about event orderings saying that for every A-event with value v, there is a B-event, with value $f(v)$, causally after it, at each location $x \in g(v)$. We require that distinct A-events cause distinct B-events. Formally,

$$\begin{aligned}
&\forall x\colon Loc.\ \exists p\colon \{e\colon E(A) | x \in g(A(e))\} \\
&\qquad\qquad\qquad \rightarrow \{e'\colon E(B) | loc(e') = x\}. \\
&\qquad injection(p)\ \wedge \\
&\qquad \forall e\colon E(A).\ e < p(e)\ \wedge\ B(p(e)) = f(A(e))
\end{aligned}$$

where injection(p) asserts that that the function p is one-to-one.

The *propagation constraint* $A \overset{f}{\Leftarrow} B@g$ is the same proposition, but with $injection(p)$ replaced by $surjection(p)$. This says that every B-event "comes from" an appropriate A-event.

We can express our leader election protocol as a conjunction of propagation rules and constraints. For instance, two of the propagation rules are:

$$\begin{aligned}
&Start^{+} \overset{f}{\Rightarrow} Propose@g,\ \textbf{where} \\
&\quad f(\langle G, uid, j\rangle) = \langle G, uid\rangle, \quad g(\langle G, uid, j\rangle) = [j] \\
&Leader^{+} \overset{f}{\Rightarrow} Leader@g,\ \textbf{where} \\
&\quad f(\langle G, x, i, j\rangle) = \langle G, x\rangle \\
&\quad g(\langle G, x, i, j\rangle) = \text{if } x = i \text{ then } nil \text{ else } [j]
\end{aligned}$$

If ψ is a proposition about event orderings, we say that a system *realizes* ψ if the event-ordering of any run of the system satisfies ψ. We extend the "proofs-as-programs" paradigm to "proofs-as-processes" for distributed computing by making constructive proofs that requirements are *realizable.* For compositional reasoning, it is desirable to create, when possible, a *strong realizer* of requirement ψ—a system that realizes ψ in any context. Formally, system S is a strong realizer of ψ if the event-ordering of any run of a system S' such that $S \subseteq S'$, satisfies ψ. If S_1 is a strong realizer of ψ_1 and S_2 is a strong realizer of ψ_2, then $S_1 \cup S_2$ is a strong realizer of $\psi_1 \wedge\ \psi_2$.

One of the main advantages of this approach is that we can automatically extract strong realizers (programs) for propagation rules like those used in the leader election example in the previous lecture. Basic event classes are programmable, and the set of programmable event classes is closed under a variety of *combinators*, that is operations on functional processes.

If B is a basic class and if we have *reliable message delivery*, then a component may cause an event in B by placing a message with an appropriate header on its external part. A rule, $A \Rightarrow B$ is *programmable-basic* (PB) if A is programmable and B is basic. Thus, under the assumption of reliable message delivery, every PB-rule is realizable.

Reliable message delivery is an assumption about the environment. One weakening of this assumption allows some components to suffer *send omission faults*. Under this assumption, parameterized by a set of locations, F, called the *fail-set*, every message on the external part of a component whose location is not in F, will eventually be delivered.

If send omissions are allowed, not every PB-rule is realizable, but the restricted rule $A|(\neg F) \Rightarrow B$ is realizable, when $A \Rightarrow B$ is PB, and $A|(\neg F)$ is the class of A-events whose location is not in the fail-set. A fault-tolerant protocol like Paxos can be described by such restricted rules, and proved correct under appropriate assumptions on the size of the fail-set.

A PB-rule $A \Rightarrow B$ is also strongly realizable. This is because, essentially by definition, class A is programmable only if there is a system S that recognizes A-events in any context. So in a run of system, S', with $S \subseteq S'$, the components in S will still recognize A-events. Also, if message delivery is reliable then the addition of extra components will not prevent the required B-events from occurring.

Unfortunately, some desirable properties of protocols like leader election do not follow from conjunctions of PB-rules alone. We also need some propagation constraints, of the form $A \overset{f}{\Leftarrow} B@g$. The realizer we construct for $A \overset{f}{\Rightarrow} B@g$ generates B-events only from A-events, so it also realizes the propagation constraint $A \overset{f}{\Leftarrow} B@g$. But it is not a strong realizer of the constraint because, in an unrestricted larger system, other components may cause B-events.

Strong realizers will always compose to strong realizers. We can compose (nonstrong) realizers for propagation rules and propagation constraints if we can show that they do not "interfere" with one another. For example, the realizer we construct for $A \Rightarrow B$ will trivially realize $C \Leftarrow D$ if classes B and D are disjoint; and that can be trivially guaranteed if classes B and D are basic classes distinguished by different "message headers." That simple design rule reduces the proof of noninterference to a "compatibility check" that the message headers used by different rules are different.

Because a verified system may run in an environment that includes unverified, untrusted code for which we cannot perform the compatibility check, it would be desirable to ensure that the verified system is a strong realizer for a conjunction of propagation rules and propagation constraints. One way to modify a group of components in a realizer so that they form a strong realizer is to have them encrypt their messages with a shared key.

3 Coding 2/3 Consensus

Various languages have been designed to facilitate coding asynchronous protocols and systems. These systems have a special character because they interact with an environment which is fundamentally unpredictable. We do not have room in this course to deeply explore these languages and formalisms that were mentioned already in this lecture and the previous one. However, it is worth a brief look at the style of one such language developed here at Cornell called EventML [11]. It is similar in spirit to the Orc language (for Orchestration) developed at the University of Texas [5, 6].

Samples of code and definitions in EventML are available in the article used as resource material for this lecture [12] which can be accessed from the PRL project page of publications,

```
http://www.nuprl.org/Publications
```

. This material is included for interested students and will not be required reading. What is required is an understanding of the two simple protocols we have studied, leader election in a ring and 2/3 consensus as assigned in PS6.

One of the lessons for the Cornell researchers who worked together on this topic is that *asynchronous distributed computing* reveals practical computational ideas that appear to be closely related to Brouwer's notion of *free choice sequences*. Well before the idea of asynchronous distributed computing, Brouwer foresaw that computation would computing with *streams of data* that were not generated by any rule, but by potentially unbounded sequences of choices made during a computation. The collection of all such developing sequences is a precise mathematical concept which can be investigated rigorously. For Brouwer the concept was used to achieve a computational understanding of the continuum. For computer scientists, the exploration is also driven by the need to understand how to compute with data that arises from complex asynchronous interactions that we are unable to understand completely algorithmically because of the inherent complexity of interactions among algorithmic processes in the physical world. The continuum has posed a similar set of questions and issues as humans have tried for centuries to understand it computationally. We can even see this when we try to understand Euclidean geometry computationally, an agenda of the earliest geometers. Set theory is also a manifestation of our attempt to understand these ideas but without the constraints imposed by computational concerns. We see that a computational understanding of the continuum introduces new primitive notions into the set theoretic framework for understanding it. We use the term "type" instead of "set" to emphasize this computational aspect of the investigation (even though that was not the motivation for the development of type theory by Russell and Whitehead [14, 13, 16].

References

[1] Mark Bickford. Unguessable atoms: A logical foundation for security. In *Verified Software: Theories, Tools, Experiments, Second International Conference*, pages 30–53, Toronto, Canada, 2008. VSTTE 2008.

[2] Mark Bickford, Robert Constable, and David Guaspari. Generating event logics with higher-order processes as realizers. Technical Report http://hdl.handle.net/1813/23562, Cornell University, 2011.

[3] Mark Bickford and Robert L. Constable. Formal foundations of computer security. In *Formal Logical Methods for System Security and Correctness*, volume 14, pages 29–52, 2008.

[4] C. A. R. Hoare. How did software get so reliable without proof? In Marie-Claude Gaudel and James Woodcock, editors, *Proceedings FME'96: Industrial Benefit and Advances in Formal Methods*. Springer, 1996.

[5] David Kitchin, William R. Cook, and Jayadev Misra. A language for task orchestration and its semantic properties. pages 477–491, 2006.

[6] David Kitchin, Adrian Quark, William R. Cook, and Jayadev Misra. The Orc programming language. pages 1–25, 2009.

[7] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–65, 1978.

[8] Nancy Lynch and Mark Tuttle. An introduction to Input/Output automata. *Centrum voor Wiskunde en Informatica*, 2(3):219–246, September 1989.

[9] Donald Michie. *Machine Intelligence 3*. American Elsivier, New York, 1968.

[10] Robin Milner. *Communication and Concurrency*. Prentice-Hall, London, 1989.

[11] Vincent Rahli. Interfacing with proof assistants for domain specific programming using eventml. In *10th International Workshop On User Interfaces for Theorem Provers*, 2012.

[12] Vincent Rahli, David Guaspari, Mark Bickford, and Robert Constable. Formal Specification, Verification, and Implementation of Fault-Tolerant Systems using EventML. In *Proceedings of the 15th International Workshop on Automated Verification of Critical Systems (AVoCS 2015)*, September 2015.

[13] Bertrand Russell. Mathematical logic as based on a theory of types. *Am. J. Math.*, 30:222–62, 1908.

[14] Bertrand Russell. *The Principles of Mathematics.* Cambridge University Press, Cambridge, 1908.

[15] Nicolas Schiper, Vincent Rahli, Robbert van Renesse, Mark Bickford, and Robert L. Constable. Developing correctly replicated databases using formal tools. pages 395–406, 2014.

[16] A.N. Whitehead and B. Russell. *Principia Mathematica*, volume 1, 2, 3. Cambridge University Press, 2nd edition, 1925–27.