Generating Event Logics with Higher-Order Processes as Realizers

Mark Bickford, Robert Constable, and David Guaspari
Cornell University Computer Science and ATC-NY

Abstract

Our topic is broadening a practical "proofs-as-programs" method of program development to "proofs-as-processes". We extend our previous results that implement proofs-as-processes for the standard model of asynchronous message passing computation to a much wider class of process models including the \( \pi \)-calculus and other process algebras. Our first result is a general process model whose definition in type theory is interesting in itself both technically and foundationally. Process terms are type free lambda-terms. Typed processes are elements of a co-inductive type. They are higher-order in that they can take processes as inputs and produce them as outputs.

A second new result is a procedure to generate event structures over the general process model and then define event logics and event classes over these structures. Processes are abstract realizers for assertions in the event logics over them, and they extend the class of primitively realizable propositions built on the propositions-as-types principle. They also provide a basis for the third new result, showing when programmable event classes generate strong realizers that prevent logical interference as processes are synthesized.

1 Introduction

1.1 Background

Using a constructive Logic of Events based on Computational Type Theory (CTT) [CB08, Bic09, ABC06] we have been able to formally specify safety and liveness properties for distributed protocols and synthesize executable code from constructive proofs in NuPrl that the specifications are realizable [CB08, Bic09]. We have used this proofs-as-processes method to build fault-tolerant protocols, adaptive protocols, and provably secure protocols. Recently we have created versions of Paxos this way.

This system development capability is based on a constructive semantics for assertions in our "standard" Logic of Events using the concept of event structures [Win80, Win89] which are defined over executions of process in the standard model of asynchronous message passing computation. This semantics is expressed in CTT in such a way that proof terms contain distributed realizers. These realizers are state machines which can easily be compiled into appropriate programming languages such as Java, Erlang, F#, etc. Critical to the practical success of this methodology is the use of programmable event classes [Bic09] to specify computing tasks at a high level of abstraction that can be refined automatically to processes.

A motivating technical result of this paper is that we substantially extend this synthesis/verification/development method so that it applies to a very general notion of process of the kind used in process algebras (e.g. the higher-order \( \pi \)-calculus, Petri nets, CCS, CSP, etc.) as well for the standard process model used in the Logic of Events mentioned above, e.g. the standard textbook model for systems courses [AW04, FLP85]. Our generalization enables the synthesis of correct-by-construction processes over a wide variety of process models by extracting them as distributed realizers for specifications in the event logics generated by our method over these process models. Thus processes are automatically generated from constructive proofs that high-level event-based specifications are achievable and then compiled into a standard programming language. In due course we will execute them directly in the proof system itself because our general process model is elementary enough to be taken both as logically foundational and practically implementable. Moreover our test bed theorem prover, NuPrl, is a distributed system with an evaluation subsystem into which these process can be incorporated along with their corresponding operating environment.

A second technical result of this paper is that because our general process model is more abstract than the model used in the standard Logic of Events, it can directly support event classes and the key concept of a programmable event class. This allows us to move the entire development methodology and Logic of Events to a more abstract level, and that becomes critical in synthesizing complex protocols in a timely manner and decomposing their development into meaningful layers. A key practical advantage of this more abstract approach is that when we are developing a protocol
such as Paxos, we are creating a large number of variations depending on the way the proof is refined. This diversity is very useful in practice.

Our first new result is the general process model itself; it is both an extremely general and remarkably simple and can simulate process algebras as well as the standard model. We focus on the \( \pi \)-calculus to illustrate this generality. The higher-order process terms are type free lambda terms built with the \( \gamma \) combinator. The processes that can be typed belong to a co-inductive type we call Processes. In the case of Computational Type Theory (CTT) [ABC06], the co-inductive type can in fact be defined using intersection over a family of types, and we give the definition here, another small new result. These process terms are sufficiently elementary to serve as new computational primitives in typing theories built on the propositions-as-types principle such as the Calculus of Inductive Constructions (CIC), CTT, and ITT . We show that they are components of a new class of distributed realizers, thus enriching the expressiveness of these theories. Moreover, as a related aside we note that the definable co-inductive type constructor can be used to express new propositions such as those built with infinitary operators.

2 Overview and Example

We begin with an overview of our model of distributed computation and the concepts we use to reason about them. All the italicized nouns will be formalized in CTT in the next section.

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 this paper, this interface will be a list of messages, containing either data or a process, labeled with the location of the 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 infinite 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 [Lam78]. This allows us to define an event-ordering, a structure, \( \langle E, \text{loc}, <, \text{info} \rangle \), 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, \text{info}(e), associated with event \( e \) is the message input to \text{loc}(e) when the event occurred.

We have found that requirements for distributed systems can be expressed as (higher-order) logical propositions about event-orderings. To illustrate this and motivate the results in the rest of the paper we present a simple example of leader election in a group of processes arranged in a ring.

Example 1. Leader election in a ring

Each participating component will be a member of some groups and each group has a name, \( G \). 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 \( j = \text{neighbor}(G, i) \) is a simple cycle. When any component in a group \( G \) receives a message \( \langle \text{elect}, G \rangle \) 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 location of the leader (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 \text{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 \text{propose}. Every component that receives a proposal with a uid, \( p \), different than 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. In this protocol there 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
\( \langle \text{[elect]} , G \rangle \) at a location \( i \in xs \) is a member of an event
class \( \text{Start} \), with value \( \text{Start}(e) = G \). Such classes,
defined by a list of locations and a particular message header,
are the basic event classes. Likewise, we may define ba-
sic classes \( \text{Propose} \) and \( \text{Leader} \) with values of the form
\( \text{Propose}(e) = \langle G, p \rangle \) and \( \text{Leader}(e) = \langle G, x \rangle \).
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 re-
cent message from the environment. As we will see below,
this allows us to derive recognizable event classes \( \text{Start}^+, \)
\( \text{Propose}^+, \) and \( \text{Leader}^+ \) that assign values as follows:

\[
\begin{align*}
\text{Start}^+(e) &= \langle G, \text{uid}(i), j \rangle \\
\text{Propose}^+(e) &= \langle G, p, i, \text{uid}(i), j \rangle \\
\text{Leader}^+(e) &= \langle G, x, i, j \rangle
\end{align*}
\]

where \( i = \text{loc}(e) \), \( j = \text{neighbor}(G, i) \)

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

Clearly, these constraints (and the assumption that group
\( G \) forms a ring) imply that after a \( \text{Start} \) event, the member
\( \max \in G \) with the maximum \( \text{uid}_{\max} \) must eventually pro-
pose \( \text{uid}_{\max} \) and this will be proposed by all members of the
group, until component \( \max \) receives its own \( \text{uid}_{\max} \).
It will then cause a \( \text{Leader} \)-event with value \( \langle G, \max \rangle \) at its
neighbor and this will be forwarded around the ring, so ev-
ery 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 \( \text{Leader} \)-events for \( G \) have the same value, then we also
need constraints that say that \( \text{Propose} \) and \( \text{Leader} \) events
are caused only by the above rules.)

The general form of the algorithm in this example and
many other distributed algorithms is this: A component rec-
ognizes some basic event. It associates additional infor-
mation, which it computes as a function of its prior input his-
tory, with the basic event. As a function of this information
it computes a new message and a list of recipients and sends
the message to each of them, causing more basic events. We
describe the part of the algorithm that recognizes events and
associates additional information with them as components
that recognize a general \( \text{programmable} \) event class. We de-
scribe the part of the algorithm that sends the information
to other components in term of propagation rules and pro-
agation constraints.

Propagation rules and constraints

If \( A \) and \( B \) are event
classes, the propagation rule \( A \overset{\text{prop}}{\Rightarrow} B@q \) 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,

\[
\forall x: \text{Loc. } \exists p: \{ e: E(A) | x \in g(A(e)) \} \\
\Rightarrow \{ e': E(B@x) | \text{loc}(e') = x \},
\]

injection\( (p) \wedge \)

\[
\forall e: E(A). e < p(e) \wedge B(p(e)) = f(A(e))
\]

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

The propagation constraint \( A \overset{\text{surr}}{\Rightarrow} B@q \) is the
same proposition, but with injection\( (p) \) replaced by surjec-
tion\( (p) \). This says that every \( B \)-event “comes from”
and appropriate \( A \)-event.

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

\[
\begin{align*}
\text{Start}^+ & \overset{\text{prop}}{\Rightarrow} \text{Propose}@q, \text{ where} \\
f((G, \text{uid}(i))) &= (G, \text{uid}), \quad g((G, \text{uid}(i))) = [j] \\
\text{Leader}^+ & \overset{\text{prop}}{\Rightarrow} \text{Leader}@q, \text{ where} \\
f((G, x, i, j)) &= (G, x), \quad g((G, x, i, j)) = \text{if } x = i \text{ then } \text{nil} \text{ else } [j]
\end{align*}
\]

If \( \psi \) is a proposition about event orderings, we say that a
system realizes \( \psi \), if the event-ordering of any run of the
system satisfies \( \psi \). We extend the “proofs-as-programs”
paradigm to “proofs-as-processes” for distributed comput-
ing by making constructive proofs that requirements are re-
alizable. For compositional reasoning, it is desirable to cre-
ate, when possible, a strong realizer of requirement \( \psi \)—a
system that realizes $\psi$ in any context. Formally, system $S$ is a strong realizer of $\psi$ if the event-ordering of any run of a system $S'$ such that $S \subseteq S'$, satisfies $\psi$. If $S_1$ is a strong realizer of $\psi_1$ and $S_2$ is a strong realizer of $\psi_2$, then $S_1 \cup S_2$ is a strong realizer of $\psi_1 \land \psi_2$.

One of our main results is that propagation rules like those used in the leader election example have strong realizers. A realizer for a propagation rule $A \rightarrow B \oplus g$ is a set of components. Using a (computable) function of the history of inputs at its location, each of these components recognizes, and computes the value $v$ of, events in class $A$ that occur there. Whenever such events occur, the component sends messages that will eventually result in an events in class $B$ with value $f(v)$ at each location in $g(v)$. We call the classes $A$ that can be so recognized, programmable. Basic event classes are programmable, and the set of programmable event classes is closed under a variety of combinations. Thus, many classes can be automatically shown to be programmable, and their recognizers generated automatically. 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. In this case, the assumption is a fairness assumption on the choices the environment makes. It states that all messages in the external part of a component will eventually be chosen. 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 $A|(-F) \rightarrow B$ is realizable, when $A \rightarrow B$ is PB, and $A|(-F)$ 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 the fairness assumption is sufficient to guarantee that basic $B$ events will occur, then the addition of extra components will not interfere with this, either.

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 $A \rightarrow B \oplus g$. A realizer constructed for $A \rightarrow B \oplus g$ will generate $B$-events only from $A$-events, so it will also realize $A \rightarrow B \oplus g$. But it will not necessarily be a strong realizer of the propagation 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 realizers for $A \rightarrow B$ will trivially realize $C \equiv D$ 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 make the verified system be a strong realizer for a conjunction of propagation rules and propagation constraints. A method that modifies 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 Formalization

All the concepts in the overview are formalized as types. In this paper we present mainly the definitions and statements of results, and merely sketch the proofs, which have all been carried out in NuPrl.

Primitives We use the notation $[x, \ldots, z]$ for finite lists, $L_1 \oplus L_2$ for the result of appending list $L_1$ to list $L_2$, and $nil$ for the empty list. Functions $fst$ and $snd$ access the components of pairs, which we write $(x, y)$. The members of a disjoint union $A + B$ are $\{\text{inl}(a) | a \in A \} \cup \{\text{inr}(b) | b \in B\}$ where $\text{inl}$ and $\text{inr}$ are primitive constructors. The only member of type $\text{Unit}$ is $()$ and we write $\text{inr}() = \bot$.

Locations The type that represents locations is a primitive type $\text{Loc} \equiv_{\text{def}} \text{Atom}$. The members of the atom type are abstract "tokens" that have no structure and can only be tested for equality. We also use atoms as headers or tags on data or messages.

Data We want to allow any reasonable values as data in messages, e.g., integers, strings, booleans, tuples, records, lists, etc. For simplicity, we merely parameterize our definitions with a type parameter $T$ that represents the type of all data values. In applications, we can use a type like $T = \text{tg}: \text{Atom} \times M(\text{tg})$ where $M$ is a function $\text{Atom} \rightarrow \text{Type}$. 4
Then a data value is a pair \([tg, v]\) where \(tg\) is an atom and \(v\) is a member of type \(M(tg)\). For any particular application we can name all the relevant types of data values. In the rest of this document, the parameter \(T\) represents the type of data values. All of our definitions will include parameter \(T\), so to save space \(T\) will be implicit.

**Components, Systems, and Computation** Since, in this paper, we are not modeling state shared among components at the same location, the external part of a component is simply a list of labeled messages. A component is a triple of a location, a process, and an external part, and a system is a list of components. So, in terms of the process and message types, \(Process\) and \(Msg\), discussed below, we define:

\[
\begin{align*}
Ext & \equiv_{def} (\text{Loc} \times \text{Msg}) \text{ List} \\
\text{Component} & \equiv_{def} \text{Loc} \times \text{Process} \times Ext \\
\text{System} & \equiv_{def} \text{Component List}
\end{align*}
\]

When component \(\langle x, P, ext \rangle\) gets input message \(m\) it will become \(\langle x, P', ext \oplus ext' \rangle\) where \(\langle P', ext' \rangle = P(m)\). For this to be well defined we need the types \(Process\) and \(Msg\) to satisfy the following subtype relation:

\[
\text{Process} \subseteq \text{Msg} \rightarrow (\text{Process} \times Ext)
\]

This says that a process is a function that accepts an input message and produce a pair, a process and an external part.

Finite objects, such as lists, can be constructed as members of inductive (recursive) types, but processes are a kind of “infinite stream” and such objects are members of a co-inductive type. In NuPrl’s CTT, we can define co-inductive types using the *intersection type constructor*.

A term is a member of an intersection of a family of types if it is a member of each type, and two terms represent the same member the intersection, if they represent the same member of each type. In particular, all terms represent the same member of the intersection of an empty family, \(\text{Top} \equiv_{def} \bigcap_{k} \text{Void} \text{ Void}\), and a function of type \(\text{Top} \rightarrow T\) must be a constant function. (Terms in NuPrl do not have a single “best” type—for example, \(17 \in \mathbb{N}, 17 \in \text{Top}\), and \(17 \in \{m : \mathbb{N} | m > 5\}\).

Co-inductive types are defined, for \(F \in \text{Type} \rightarrow \text{Type},\)

\[
F^0(T) = T, \text{ and } F^{k+1}(T) = F(F^k(T)),
\]

by

\[
\text{corec}(P, F(P)) \equiv_{def} \bigcap_{k \in \mathbb{N}} F^k(\text{Top})
\]

Types \(A\) and \(B\) are extensionally equal, written \(A \equiv B\), if they have the same members, i.e. if \(A \subseteq B\) and \(B \subseteq A\). A type function \(F\) is *continuous* if for any sequence of types \(X \in \mathbb{N} \rightarrow \text{Type},\)

\[
\bigcap_{k} F(X(k)) \equiv F(\bigcap_{k} X(k))
\]

Function \(F\) is *weakly continuous* if

\[
\bigcap_{k} F(X(k)) \subseteq F(\bigcap_{k} X(k))
\]

If \(F\) is weakly continuous then

\[
\text{corec}(P, F(P)) \subseteq F(\text{corec}(P, F(P)))
\]

**Messages and Processes** We want “higher-order” processes that can input and output messages that contain processes, so we define the types of processes and messages simultaneously. According to our computation model, a process is a function that can accept an input message and produce a new process (possibly updating its internal state) and an external part (to be appended to its current external part). External parts (in this paper) contain only a list of output messages, labeled with their recipient. The body of an input message will be either data or a process, so we define\(^1\)

\[
\begin{align*}
M(P) & \equiv_{def} (\text{Atom List}) \times (T + P) \\
E(P) & \equiv_{def} (\text{Loc} \times M(P)) \text{ List} \\
F(P) & = M(P) \rightarrow (P \times E(P))
\end{align*}
\]

It is easy to show that \(M\) and \(E\) are continuous type functions and that \(F\) is weakly continuous\(^2\).

This implies that if we define

\[
\begin{align*}
\text{Process} & \equiv_{def} \text{corec}(P, F(P)) \\
\text{Msg} & \equiv_{def} M(\text{Process}) \\
\text{Ext} & \equiv_{def} E(\text{Process})
\end{align*}
\]

then

\[
\text{Process} \subseteq F(\text{Process}) = \text{Msg} \rightarrow \text{Process} \times \text{Ext}
\]

**Building Processes** There is a standard way to construct a process, embodied in the following recursive definition:

**Definition 1.**

\[
\text{RecPr}(\text{next}, \text{ext}, s) =_{rec} \lambda m. \text{ let } \langle s', e \rangle = \text{next}(s, m) \text{ in} \\
\text{ let } P = \text{RecPr}(\text{next}, \text{ext}, s') \text{ in} \\
\{P, \text{ ext}(e, m, P)\}
\]

The parameter \(s\) in \(\text{RecPr}(\text{next}, s)\) is the *internal state* of the process, and \(\text{next}\) is its *program*. On input \(m\) the process uses its program to compute a new internal state \(s'\) and an external part \(e\) and “becomes” the process \(P\) with the new internal state. It can put process \(P\) into the external part by using the function \(\text{ext}\). To build processes that never add themselves to messages we write just \(\text{RecPr}(\text{next}, s)\) and supply the default \(\text{ext}(e, m, P) = e\).

---

\(^1\)Recall that \(U + V\) is the disjoint union of types \(U\) and \(V\)

\(^2\)Because \(P\) occurs on the lefthand side of the arrow in the function type, \(F\) is not continuous.
Lemma 1. Suppose that $S$ is a continuous type function, and let $F'(T) \equiv \text{def } M(T) \rightarrow (S(T) \times E(T))$

If

\[ s \in S(\text{Process}) \]
\[ \text{ext} \in \bigcap_{T'} E(T) \rightarrow M(T) \rightarrow T \rightarrow E(T) \]
\[ \text{next} \in \bigcap_{T'} S(F(T)) \rightarrow F'(T) \]

then $\text{RecPr}(\text{next}, \text{ext}, s) \in \text{Process}$

Proof. We claim that

\[ \forall k. \forall s: S(F^k(\text{Top})), \text{RecPr}(\text{next}, \text{ext}, s) \in F^k(\text{Top}) \]

We prove this by induction on $k$; the base case is trivial. Let $T = F^k(\text{Top})$, assume that $st \in S(F(T))$ and $m \in M(T)$, and let $⟨s', e⟩ = \text{next}(st, m)$. Then $s' \in S(T)$ and $e \in E(T)$. Thus, by induction, $P = \text{RecPr}(\text{next}, \text{ext}, s') \in T$. So, $⟨e, m, P⟩ \in E(T)$. This shows that $\text{RecPr}(\text{next}, \text{ext}, s) \in F'(T)$ and completes the inductive proof of the claim. Because $S$ is continuous,

\[ s \in S(\text{Process}) = S(\bigcap_{k} F^k(\text{Top})) \subseteq \bigcap_{k} F^k(\text{Top}) \]

Hence, by the claim, for any $k$, $\text{RecPr}(\text{next}, \text{ext}, s) \in F^k(\text{Top})$, so

\[ \text{RecPr}(\text{next}, \text{ext}, s) \in \bigcap_{k} F^k(\text{Top}) = \text{Process} \]

Axillary definitions Messages are pairs of a “header”, a list of atoms, and a “body” that is either data or a process. For constructing and manipulating messages, we define

\[ \text{pmsg}(\text{hdr}, P) \equiv \text{def } ⟨\text{hdr}, \text{inr}(P)⟩ \]
\[ \text{header}(m) \equiv \text{def } \text{fst}(m) \]
\[ \text{body}(m) \equiv \text{def } \text{snd}(m) \]
\[ \text{rmheader}(m) \equiv \text{def } ⟨\text{tail}(\text{header}(m)), \text{body}(m)⟩ \]
\[ \text{addheader}(k, m) \equiv \text{def } ⟨\text{cons}(k, \text{header}(m)), \text{body}(m)⟩ \]

We define the iteration of a process on a list of messages by recursion on the list:

\[ P^*(\text{nil}) = ⟨P, \text{nil}⟩ \]
\[ P^*(L \oplus [m]) = \text{let } ⟨P', \text{ext}⟩ = P^*(L) \text{ in } P'(m) \]

So $P^*(L)$ is the pair consisting of the resulting process and the last external part that was produced.

Example 2. A bootstrap process.

To illustrate the use of Lemma 1, we make a bootstrap process, $\text{boot}$, that has an internal state of type $\text{Process} + \text{Unit}$. Initially it will have state $\bot$ and the first time it gets a process message, $\text{pmsg}(\text{hdr}, Q)$, it will change its internal state to $\text{inl}(Q)$. After that, it behaves like the process in its internal state, by passing its inputs to the internal process and using the resulting external part as its own. Thus,

\[ \text{boot} \equiv \text{def } \text{RecPr}(\text{next}, \bot) \text{ where} \]
\[ \text{next}(\bot, m) = ⟨G(m), \text{nil}⟩ \text{ where} \]
\[ G(m) = \cases{ \text{if } m = \text{pmsg}(\text{hdr}, Q) \text{ then } \text{inl}(Q) \text{ else } \bot } \]

The internal state has type $S(\text{Process})$ where $S(T) = T + \text{Unit}$, so $S$ is continuous. It is easy to see that the given process, $\text{next}$, has the polymorphic type given in Lemma 1. So, $\text{boot}$ is a process. We can use this process as the default boot-process in the run of a system.

Example 3. A “forkable” process.

For a given function $f$ of type $\text{Msg} \rightarrow (\text{Loc} \times \text{Atom List})?$, we can define a process, $\text{forkable}(P, f)$, that interprets a message $m$ with $f(m) = \text{inl}(x, \text{hdr})$ as the instruction send itself to location $x$ in a message with header $\text{hdr}$, but otherwise acts like a given process $P$. By Lemma 1, the following defines the desired process:

\[ \text{forkable}(P, f) \equiv \text{def } \text{RecPr}(\text{next}, \text{ext}, P) \text{ where} \]
\[ \text{next}(P, m) = \cases{ P(m) \text{ if } m = \text{pmsg}(\text{hdr}, P) \text{ then } [(x, \text{msg}(\text{hdr}, P))] \text{ else } e } \]

Environment and Runs The environment chooses which messages will be delivered, and that is the only nondeterminism in the model. We use the type $\text{Choice} = \mathbb{N} \times \mathbb{N}$ to represent the choices made by the environment. Then $(i, k) \in \text{Choice}$ represents the choice of the $i^{th}$ message on the external part of the $k^{th}$ component, if there is such a message, and no message otherwise. A run is determined by the initial system $S$, a boot-process $\text{boot}$ and an environment $\text{env} \in \mathbb{N} \rightarrow \text{Choice}$, an infinite sequence of choices. The run is the infinite sequence of pairs (in $\text{System} \times \text{Choice}$) defined by $\text{Run}(S, \text{boot}, \text{env}) \equiv \text{def } \lambda n. F(n)$, where

\[ F(n) = \text{rec } \cases{ \text{if } n = 0 \text{ then } S, \text{env}(0)) \text{ else } \text{Next}(F(n-1), \text{boot}, \text{env}(n)) } \]

The detailed definition of $\text{Next}((S_n, \text{choice}_n), \text{boot})$ is straightforward and has been formalized in NuPrl. The boot-process $\text{boot}$ is used to initialize a new component that is created when a message containing a process is delivered to a new location.

We call this formal model of distributed computation the general process model.
4 Event-orderings and realizers

We are now ready to define for any run, \( R \), a structure \( EO^+(R) = (E, \text{loc}, \prec, \text{info}) \) called the (extended) event-ordering of the run. This is an abstraction of the observable behavior of a distributed system. Extended event-orderings can also be derived from other process models.

In the event ordering derived from our model, the events are the pairs, \( e = (x, n) \), at which location \( x \) gets an input message at step \( n \)—this defines the type \( E_R \) of events in the run. Note that equality on \( E_R \) is decidable. The location \( \text{loc}(e) \) of event \( e \) is its first component, and \( \text{info}(e) \) is the input message it received.

Event \( e_1 = (x, n) \) is the local predecessor of \( e_2 = (y, m) \) if \( x = y \) and \( n < m \) and there are events \( (x, k) \) with \( n < k < m \). Event \( e_1 = (x, n) \) is an immediate predecessor of \( e_2 = (y, m) \) if it is either the local predecessor of \( e_2 \) or if a message for \( y \) was taken from a component at location \( x \) in step \( m \), and \( e_1 \) was the most recent prior event at location \( x \). The causal ordering \( \prec \) is, by definition, the transitive closure of the immediate-predecessor relation.

We formalized the details in NuPrl and proved that \( \prec \) is well-founded, transitive, locally-finite, decidable, and a total ordering of events at the same location. We call a structure \( (E, \text{loc}, \prec) \) that satisfies these properties (and has decidable equality) an event-ordering. An event ordering is an abstract model of causality and location in “space-time”. The extended event ordering adds the information function \( \lambda e. \text{info}(e) \) so that events are associated with some primitive information content. We define the type, \( \text{EventOrdering}^+ \), of extended event-orderings in appendix A using the dependent record type defined in NuPrl.

A less general model of distributed computing, called message automata\(^3\), that we have been using for several years, gives rise to what we call an event-structure—similar to an event-ordering but with additional structure and axioms\(^4\). We have made many definitional extensions and proved many properties of event-structures and used them to specify and prove properties of distributed algorithms such as leader-election, consensus, and authentication protocols. We recently “re-factored” our theory of event-structures to see how many of the results held for the more general event-orderings defined above. We found that over 1200 lemmas in our library could be re-proved using only the properties of an event-ordering.

Event history Some definitional extensions of (extended) event-orderings that we will need in the sequel are:

- \( \text{first}(e) \): A boolean, true iff \( e \) is the first event at its location.
- \( \text{pred}(e) \): An event (provided \( \neg \text{first}(e) \)), the local immediate predecessor of \( e \).
- \( \text{history}(e) \): A \( \text{Msg List} \), defined recursively by

\[
\text{(if } \text{first}(e) \text{ then } \text{nil} \text{ else } \text{history}(\text{pred}(e))) \oplus [\text{info}(e)]
\]

the list of all inputs up to and including the one received at event \( e \).

Such expressions are defined in the context of an event ordering \( eo \in \text{EventOrdering}^+ \), so they include \( eo \) as an implicit parameter. To make this explicit we sometimes write, e.g., \( \text{history}_{eo}(e) \), \( \text{loc}_{eo}(e) \), etc.

4.1 Event classes

To structure our specifications of distributed systems and our reasoning about them, we introduce the concept of an event class. An event class \( X \) is function on events that partitions the events into two sets, \( E(X) \) and \( E = E(X) \), and assigns a value \( X(e) \) to events \( e \in E(X) \).

We can choose to assign a type to each event class, a type to which the value of all its events must belong, but for simplicity, in this paper all event classes will assign values of type \( \text{Msg} \). Since the body of a message has type \( T + (m : \text{N} \times \text{Process}_m) \), and since the data type \( T \) is assumed to be sufficiently universal, we can put whatever information we need into a member of type \( \text{Msg} \).

Therefore we define:

\[
\text{EClass} \equiv \text{def} \ e o : \text{EventOrdering}^+ \rightarrow E(eo) \rightarrow \text{Msg}?
\]

where \( \text{Msg}? \equiv \text{def} (\text{Msg} + \text{Unit}) \). This says that an event class \( X \) is really a function of both an event ordering and an event in that ordering. It decides whether the event is in the class and if so produces a message.

Basic event classes If \( k \in \text{Atom} \) we say that event \( e \) has kind \( k \) if \( \text{head}(\text{header}(\text{info}(e))) = k \). Then, if \( xs \) is a list of locations,

\[
\text{Kind}(k, xs) \equiv \text{def} \ \lambda e. \text{if} \ \text{loc}_{eo}(e) \notin xs \text{ then } \bot \text{ if } \text{head}(\text{header}(\text{info}_{eo}(e))) \neq k \text{ then } \bot \text{ else } \text{inl}(\text{rmheader}(\text{info}_{eo}(e)))
\]

is an event class such that \( E(\text{Kind}(k, xs)) \) are the events of kind \( k \) whose location is in \( xs \), and \( \text{Kind}(k, xs)(e) \) is \( \text{rmheader}(\text{info}(e)) \). We call the classes \( \text{Kind}(k, xs) \) the basic event classes.

Example 4. Consensus specification

---

\(^3\)In particular, they are not “higher order” and reliable, FIFO, message delivery is built-in, which makes specifications that allow faulty behavior trickier.

\(^4\)For example, event structures include operations \( x \text{ when } e \) and \( x \text{ after } e \) for observing state variables before and after events, and an axiom \( \neg \text{first}(e) \Rightarrow (x \text{ when } e = x \text{ after } \text{pred}(e)) \)
If propose and decide are atoms, and xs lists the locations of the components in a consensus protocol, let $P = \text{Kind}(\text{propose}, \text{xs})$ and $D = \text{Kind}(\text{decide}, \text{xs})$. Then the specification of a consensus protocol is the conjunction of two propositions on (extended) event-orderings, called agreement (all decision events have the same value) and validity (the value decided on must be one of the values proposed):

$$\forall e_1, e_2 : E(D), D(e_1) = D(e_2)$$
$$\forall e : E(D), \exists e' : E(P), e' < e \land D(e) = P(e')$$

Simple combinators If $X_1, \ldots, X_n$ are event classes, and $F \in \text{Msg}^n \rightarrow \text{Msg}$, then

$$\hat{F}(X_1, \ldots, X_n) \equiv \lambda e_0. \lambda e. F(X_1(e_0, e), \ldots, X_n(e_0, e))$$

is an event class. When $F$ is strict, $F(\bot, \ldots, \bot) = \bot$, then we call $\hat{F}$ a simple combinator.

The “prime” combinator If $X$ is an event class, the class $(X')^\prime$ is the class which contains those events for which an earlier event at the same location was in class $X$ (call this a locally-prior $X$-event). The value assigned to an event in $(X')^\prime$ is the value if the most recent locally prior $X$-event. Formally,

$$e \in E(X') \Leftrightarrow \exists e' : E(X), \text{loc}(e') = \text{loc}(e) \land e' < e$$

$$e \in E(X') \Rightarrow X'(e) = X(\text{the most recent such } e')$$

Recursion combinators

Lemma 2. If $X_1, X_2, \ldots, X_n$ are event classes, and $H \in \text{Msg}^{n+1} \rightarrow \text{Msg}$, then there is a unique class $Z$ such that $Z = \hat{H}((Z')^\prime, X_1, \ldots, X_n)$

Proof. The equation $Z = \hat{H}((Z')^\prime, X_1, \ldots, X_n)$ is a recursive definition of $Z$. Using the well-foundedness of $\prec$, it can be shown to be well defined. Uniqueness is also proved by induction on $\prec$.

We write $\hat{H}((\text{self})', X_1, \ldots, X_n)$ for the $Z$ defined in lemma 2.

Example 5. Accumulators

Distributed algorithms usually require a participating component to maintain state information that is a function of the inputs the component has seen. In the leader election example, we suppose that there is a class of input events $e \in \text{Neighbor}$, where some function $f$ of the value $\text{Neighbor}(e)$ is a pair $(G, j)$, and this informs the component at $\text{loc}(e)$ that its neighbor in $G$ is $j$. Each component $i$ must maintain a table of such neighbor information, and this is a function of the history of $\text{Neighbor}$ inputs it receives. We represent such accumulated state information as the value of an event class defined using the recursion combinator. For example the table of neighbors is the value of the following class:

$$\text{NbrTab} \equiv \lambda e_0. \lambda e. \hat{H}((\text{self})', \text{Neighbor}), \text{where}$$

$$H(\bot, m) = [f(m)]$$

$$H(\text{tabl}, m) = \text{let } (G, j) = f(m) \text{ in } \text{tabl}[G := j]$$

(Actually, because all event classes have values of type $\text{Msg}$, we would “code” the values in the definition above into messages.) By using “accumulators” like this, we can describe the behavior of distributed algorithms without introducing state variables. This allows us to specify “implementations” abstractly.

4.2 Programmable event classes

Specifying an implementation abstractly is of little use if there is no reliable path from the abstract implementation to running, correct, code. Fortunately, all of the event classes we have discussed are programmable, which means that they can be recognized by a system of (computable) component processes. Because of this, we will be able to automatically construct a realizer (see section 5) for our abstract implementation.

Definition 2. Let $X$ be an event class. Component $(x, P, \text{ext}_0)$ recognizes $X$-events at $x$ if for any event $e$ with $\text{loc}(e) = x$,

$$\text{snd}(P^*(\text{history}(e))) =$$

$$\text{if } e \in E(X) \text{ then } [(x, X(e))] \text{ else nil}$$

A system $S$ recognizes event class $X$ if there is one component in $S$ at each location where $X$-events may occur, and it recognizes the $X$-events at that location. Class $X$ is programmable if there is a system $S$ that recognizes it.

Theorem 3. A basic class $\text{Kind}(k, \text{xs})$ is programmable. If $X_1, \ldots, X_n$ are programmable classes, and $F \in \text{Msg}^n \rightarrow \text{Msg}$ is strict and $H \in \text{Msg}^{n+1} \rightarrow \text{Msg}$ is strict, then

1. $F(X_1, \ldots, X_n)$ is programmable.
2. $(X'_i)'$ is programmable.
3. $\hat{H}((\text{self})', X_1, \ldots, X_n)$ is programmable.
Then if \((\bigwedge_i \phi_i \Rightarrow \psi)\) and \((\forall i. S_i \vdash_C \phi_i)\) then \(\bigcup_i S_i \vdash_C \psi\) provided all of the pairwise compatibility relations \(P_{S_i}(S_j)\) are true.

6 Realizers for propagation rules

The rule \(A \overset{f}{\rightarrow} B@g\) is programmable-basic (PB) if \(A\) is programmable and \(B\) is basic. Assuming reliable message delivery, we show that \(A \overset{f}{\rightarrow} B@g\) is strongly realizable.

The definition of a fairness condition, \(\phi_{\text{rd}}\), on environments that implies reliable message delivery is straightforward. The external parts of the components in a system are the messages that are in-transit, and, at each step in the run, the environment chooses which in-transit message will be delivered. The condition \(\phi_{\text{rd}}\) says that every message in-transit will eventually be delivered—we omit the formal definition to save space. If we want to allow send omission faults, then we modify \(\phi_{\text{rd}}\) to \(\phi_{\text{rd}}(F)\) that says every message in-transit from a component with a location not in the fail-set \(F\) will eventually be delivered.

Theorem 4. If event class \(A\) is programmable, \(B = \text{Kind}(k, zs)\) is basic, and \(\text{range}(g) \subseteq \text{zs}\), then

\[ \vdash_{\phi_{\text{rd}}} A \overset{f}{\rightarrow} B@g \]

Proof. Since \(A\) is programmable there is a system \(S_0\) that recognizes \(A\)-events. We simply modify each component of \(S_0\) (at a location \(z\)) so that instead of producing \([x, A(e)]\) it produces \(\text{map}(\lambda x. (x, \text{addheader}(k, f(A(e))),(g(A(e)))).\) That the modified system \(S\) strongly realizes \(A \overset{f}{\rightarrow} B@g\) follows easily from the assumption \(\phi_{\text{rd}}\).

The strong realizer \(S\) for \(A \overset{f}{\rightarrow} B@g\) constructed in the proof of theorem 4 is also a realizer (but not a strong realizer) for \(A \overset{f}{\rightarrow} B@g\) because \(S\) generates \(B\)-events only when it has recognized an \(A\)-event. Since it is not a strong realizer, we must provide a compatibility test \(P_S\) such that \(P_{S}(S') \Rightarrow S \oplus S' \vdash_{\phi_{\text{rd}}} (A \overset{f}{\rightarrow} B@g)\). Because the basic \(B\)-events all have header \(k\), the compatibility test \(P_S(S')\) can be \text{avoids}(k, S')—\(S'\) does not generate any messages with header \(k\). If \(S'\) is the realizer of another PB-rule, \(C \overset{D}{\rightarrow} D@g\), then it is easy to check whether \(\text{avoids}(k, S')\)—check whether the header for basic class \(D\) differs from \(k\).

Private names All of the preceding results have been proved using the NuPrl theorem prover, but the ideas in this paragraph are speculative.

It is not clear that it is always easy to prove \(\text{avoids}(k, S')\) for systems \(S'\) that are not derived from theorem 4. Also, we may want to run a verified, correct-by-construction, system \(S\) in an environment that includes untrusted code. For
these reasons, we would like to construct a strong realizer
for the propagation constraint \( A \leftarrow B \). This can be done
by using techniques from nominal logic or the \( \pi \)-calculus
and provide a set of components with a shared, fresh, private
name which they may use as a header on messages.

We structure a system into a list of groups of compo-
nents, each group having the form \( \text{private } k.S(k) \) where
\( S \) is a (sub)system with a free name parameter \( k \). A run of
such a system would begin by initializing each group with a
freshly generated name. Then, computation would proceed
as before except that, if the message chosen in a step has
a header that is the private name of a group, it is delivered
only to components in that group (even though there may
be other components at the recipient location). Using this
mechanism we can construct strong realizers for constraints
like \( A \leftarrow B \) provided that the definition of the basic class \( B \)
is \( \text{Kind}(k, xs) \) for a free parameter \( k \) (rather than a specific
constant).

We do not need to add any nominal binders to our
logic—the propagation rules and propagation constraints
have the same formal definitions as before. The “nominal-
” techniques are used only in the construction of the realizers,
and allow us to build strong realizers for the propagation
constraints. An implementation of such realizers has to
enforce the semantics of the private names. A plausible
mechanism to do this is to use the shared name as an en-
cryption key. For such names \( k \) the \( \text{addheader}(k, m) \) op-
eration is really \( \text{encrypt}_k(m) \) and the \( \text{rmheader}(m) \) is really
if \( \text{encrypted}_k(m) \) then \( \text{decrypt}_k(m) \) else \( \text{rmheader}(m) \).

7 Modeling the \( \pi \)-calculus

We informally describe an encoding of the \( \pi \)-calculus in
the general process model. Filling in the details is straight-
forward (and has been done formally in NuPrl). We use
the formulation of monadic \( \pi \)-calculus in which all sums are
guarded, so the syntax of a \( \pi \)-calculus process term is:

\[
P ::= 0 | \Sigma_{i=1}^n \pi_i.P_i | P|Q | !P | (\nu x)P
\]

where \( n > 0 \) and a prefix \( \pi_i \) is either the “get” operation
\( c(x) \) or the “put” operation \( \tau.x \). We will model the basic
\( \pi \)-calculus semantics that does not require communications
to be chosen “fairly.”

Several basic differences between \( \pi \)-calculus and the
general process model must be negotiated. The notion of
location, fundamental in the general process model, doesn’t
exist in \( \pi \)-calculus. In the general process model, a process
sends a message to a known recipient by labeling the mes-
 sage with the recipient’s location, whereas a \( \pi \)-calculus pro-
cess sends a message to a channel without knowing which
other processes might be reading from it. The general
process model is asynchronous and processes can act only on
local knowledge; whereas in \( \pi \)-calculus, communication is
synchronous and rendezvous are effected by the environ-
ment, using global information. Finally, a process in the
general process model is purely reactive; it acts only in re-
 sponse to an input. A \( \pi \)-calculus process is active at least
in the sense that it can, for example, “fork” replicas or ini-
tiate a communication (though an act of the environment is
needed for communications to complete).

From now on, “process” refers to a process in the sense
of the general process model. “Target processes” will
encode \( \pi \)-calculus process terms and “bookkeeping pro-
cesses” will be added to manage and constrain their interac-
tions.

7.1 Communication

It seems appealing, at first glance, to encode a channel
as a process, but \( \pi \)-calculus semantics requires global deci-
sions to determine what communications occur; the general
process model has no built-in magic to make such global
decisions.

Instead, we introduce a central bookkeeping process
\( \text{Comm} \), at location \( l_{\text{comm}} \), that manages all communication.
A \( \pi \)-calculus term that can immediately engage in commu-
nications has the form \( \pi_1.P_1 + \ldots + \pi_n.P_n \). A process
at location \( l \) that encodes this term will send to \( \text{Comm} \) a
message containing \( l \) (the return address) and the sequence of
possible communications. That is, it will place

\[
[l_{\text{comm}} : \langle l, [\pi_1, \ldots, \pi_n] \rangle]
\]
on its external part.\(^7\) The process will block until it receives
a reply from \( \text{Comm} \).

\( \text{Comm} \) accumulates these requests (its state can be
thought of as a partial function from locations to lists of
prefixes), decides which communications will occur (once
it has chosen one prefix from the request \( \langle l, \pi \rangle \) it deletes
that request from its state) and carries them out by sending
appropriate messages to the processes requesting the “get”
and the corresponding “put.”

The list of \( \pi_i \) could contain repetitions—e.g., the process
could be \( c(x).P + c(x).Q \), so that the corresponding \( \text{Req} \)
is \( \langle l, [c(x), c(x)] \rangle \). \( \text{Comm} \)'s replies must therefore contain
both a return value (or acknowledgement) and the index of
the request chosen. In this case it would send to location \( l \)
a message of the form \( \langle v, 1 \rangle \) or \( \langle v, 2 \rangle \). The process at \( l \) will
use the index to determine whether to continue as \( P \) or as
\( Q \).

The decisions about which communications occur must
be made nondeterministically from the full range of possi-
bilities. In the general process model, however, processes

\(^7\)This is slightly loose, since a process does not have an external part;
only a component does.
must be deterministic—nondeterminism comes only from
the environment. The trick for making Comm deterministic is simple: Introduce an additional bookkeeping process, Choose. When Comm wants to make a choice from some list of possibilities it sends each element of the list, in a separate message, to Choose. The environment will determine which of those messages reaches Choose first. Choose returns to Comm the first one it receives and ignores the others. Since Choose must make repeated choices, without being confused by old messages that the environment delivers belatedly, we must introduce some additional state into both Comm and Choose to do the bookkeeping (or create a new instance of Choose to handle each choice). Those details are straightforward; one version of them can be found at www.nuprl.org/documents/Guaspari/picalculus.html.

7.2 Encoding π-calculus terms and programs

If P is a π-calculus process term, we will define

\[ [P] : \text{Loc} \rightarrow \text{Process} \]
\[ \mathcal{M}(P) : \text{System} \]

For any location l, \([P](l)\) will represent the behavior of P if it is “installed” at location l. \(\mathcal{M}(P)\) is a system whose runs simulate the executions of P as a stand-alone π-calculus program. One component of the system \(\mathcal{M}(P)\) will be constructed from \([P]\) and the others will contain bookkeeping processes.

For readability, we’ll write \([P](l)\) as \([P]_l\). These target processes will have certain features in common:

- Each responds to a special message, called fire, that “activates” it. Processes in the general process model are purely reactive and this will provide a uniform way to create a system that, once set in motion, can keep going.

- Each communicates only with itself (to which it can send a fire message) and with three bookkeeping processes: Comm, the location server LServer, and the name server NServer.

Informally, we will describe \([P]\) in terms of primitive operations that “get a (globally) new location” and “get a (globally) new name.” Formally, these primitives will be implemented by sending a message to one of these servers and getting the location or name in reply.

Strictly speaking LServer is a function that outputs location servers (and NServer is analogous). If L is a finite set of locations, then LServer(L) is a process that, in response to an input returns a message with a name that differs from any location in L and any location it has previously returned.

The base case \([0]_l\) is the null process.

Guarded choice \([\pi_1.P_1 + \ldots + \pi_n.P_n]_l\) is the process that on receiving the fire message returns

\(\langle Q, l_{\text{comm}} : \langle l, [\pi_1 \ldots \pi_n]\rangle\rangle\)

That is, it sends a request to Comm and becomes process Q, which acts as follows: wait for a response \(\langle v, i \rangle\) from Comm and then return

\(\langle P_i, l : \text{fire}\rangle\)

That is, it becomes process \(P_i\) and tells \(P_i\) to fire. The definition of \(P_i\) depends on \(i\). If, for example, \(\pi_i = c(x)\),

\(P_i = [P_i[x := v]]_l\)

where “\([x := v]\)” denotes substitution of v for x.

Replication and parallel composition The obvious way to encode parallel composition or replication is to create new sub-processes. Thus, \([P|Q]_l\) responds to its fire message by obtaining new locations \(l_1\) and \(l_2\) from the location server and sending to these locations, respectively, messages containing the processes \([P]_{l_1}\) and \([Q]_{l_2}\). The effect of such “process messages” depends on the boot process to be applied. If we use the default boot process the newly installed processes will not be “active” because they’re waiting for the fire message. Thus we must both install them and send the activation messages. It will not suffice to have \([P|Q]_l\) send both the processes and the activation messages—for there is no guarantee that either process will be installed before the fire message arrives.

The solution we choose is to introduce a fancier boot process, boot⁺, that creates a component containing not the process in the message, but the result of applying that process to the message fire (and whose external part is initialized to the messages that result). It is a simple modification of the boot process defined in section 2.

Replication is just a variant of parallel composition. In response to a fire message the process \([P]_l\) will obtain a new location \(l_1\), install \([P]_{l_1}\) there, and resend the fire message to itself (which enables it, when the message is delivered, to install further models of \(P\) at other locations).

The ν operator \([\nu x]P\] responds to its fire message by obtaining a globally fresh name \(n\), becoming the process \([P[x := n]]_l\) and telling itself to fire.

7.3 Simulating a π-calculus program

If P is a π-calculus process term, the system \(\mathcal{M}(P)\) models execution of P as a stand-alone program. Let N be the set of all names occurring in P and choose five distinct locations: \(l_{\text{comm}}, l_C, l_N, l_L, l\). \(\mathcal{M}(P)\) consists of the following five components, together with the boot process boot⁺.
Our semantic model of $P$ as a stand-alone program is the collection of all runs of $\mathcal{M}(P)$ under the assumption of reliable message delivery. That assumption ensures that the model makes progress when it should. It ensures, for example, that a run of $\mathcal{M}(\mathcal{E})$ will do more than fork off replicas of $P$—the replicas themselves will also have a chance to act because their requests for communication will be delivered.

8 Related Work

Event Structures We built on the work of Winskel [Win80, Win89] and Lamport [Lam78] when we designed and implemented the “standard” Logic of Events in 2005 around the notion of event structures. Instead of reasoning about a conflict primitive we reason about logical interference. The book of Abraham [Abr99] also confirmed the importance of event structures, expressed as Tarski models. Although his methods are classical, we found we could do the proofs constructively as a basis for synthesis. We were also influenced by a long term collaboration with Birman and van Renesse e.g. [LKvR+99] who use an informal logic of events in discussions with us. Our work was expedited by the ease of building our first standard models using IO Automata [Lyn96] and expressing our realizers as IOA. However, we also wanted to compose realizers and introduced frame conditions to enable us to reason about logical interference. In the standard Logic of Events we define temporal operators in the spirit of [Pnueli81], but they are not the main vocabulary of expression, nor are the modal operators of process logics [HKP82, HM85].

Process Synthesis The value of writing abstract specifications was made especially clear by Vardi [Vardi95], Smith and Green [SG96], and by Meseguer and Winkler [MW92]. Lately we were encouraged by the results of Murphy, Crary, and Harper [MCH04] for distributed but not concurrent computing. Their modal lambda calculus also illustrates abstract realizers based on propositions-as-types.

Proofs-as-Processes The methodology of proofs-as-programs [BC85] has proven effective in functional and procedural programming, and has gained momentum as can be seen from some contemporary examples of verified programming [BB08, WMM09, XL09]. The work of Abramsky [Abr94] suggested what constructive results were possible using linear logic, but we were not able to build practical synthesis methods grounded in his logic nor in Pratt’s use of it [Pratt91]. We were led to our standard event logic by working on the verification of deployed practical distributed systems [LKvR+99] and looking at other abstract theoretical approaches, e.g. Abstract State Machines [BG03a, GGV04] which are closer to the standard model [AW04, FLP85].

Process Models and Mobility Robin Milner’s work on processes has been part of our background in thinking about concurrency since CCS, the pi-calculus, and now Bigraphs [Mil89, Mil09]. Connecting his work to other abstract models such as Abstract State Machines [BG03a, GGV04] was a motivation for our general process model, in particular we were keen to capture mobility using the ability to send processes in messages. We obtain the mechanisms of the $\pi$-calculus for mobility as well.

References


A Appendix: Definition of EventOrdering type

The definition of a type for event-orderings illustrates a useful piece of type theory.
A.1 Dependent record types

If \( A \in \text{Type} \), and \( B \in A \rightarrow \text{Type} \), then the dependent intersection \( x: A \cap B(x) \) is also a type [Kop03]. A term \( t \) is a member of type \( x: A \cap B(x) \) if \( t \in A \) and \( t \in B(t) \). This definition makes sense because if \( t \in A \), then \( B(t) \in \text{Type} \). The dependent intersection allows a certain kind of “self reference”, so we often use the variable \( \text{self} \) as the bound variable and write the type as \( \text{self} : A \cap B(\text{self}) \).

We use this type to define dependent record types as follows. Define the type \( T; z : B[\text{self}] \) to be

\[
\text{self} : T \cap (x : \text{Atom} \rightarrow \text{if } x = z \text{ then } B[\text{self}] \text{ else } \text{Top})
\]

If \( r \) is a member this type, then \( r \in T \) and also, \( r \) is a function from labels (atoms) to values such that \( r(z) \in B(r) \). So, if we define record-selection \( r.z \) as application \( r(z) \), we have \( r.z \in B(r) \). We start with type \( \text{Top} \) and iterate the record type constructor to build arbitrary dependent records. We write \( \text{Top}; z : A \) as simply \( x : A \), so that a type like \( \text{Top}; x : A; z : B[\text{self}.x] \) is written \( x : A; z : B[\text{self}.x] \). If \( r \in x : A; z : B[\text{self}.x] \) then \( r.x \in A \) and \( r.z \in B(r.x) \).

A.2 Mathematical Structures as Types

To represent a structure \( \langle A, \ldots, f, \ldots, R, \ldots \rangle \) with some sorts \( A, \ldots \), functions \( f, \ldots \), and relations \( R, \ldots \), we make a dependent record type, \( \text{Struct} \), like:

\[
A : \text{Type};
\]

\[
\ldots;
\]

\[
f : \text{self}.A \rightarrow \text{self}.A;
\]

\[
\ldots;
\]

\[
R : \text{self}.A \rightarrow \text{self}.A \rightarrow \text{P}
\]

\[
\ldots
\]

So, the sorts become types, the functions are “methods” whose type depends on the sorts, and relations are functions from the sorts to propositions.

In CTT, propositions are types, so to represent a structure as above that also satisfies axioms \( \psi_1(A, \ldots, f, \ldots, R, \ldots), \ldots, \psi_n(A, \ldots, f, \ldots, R, \ldots) \), we merely add the axioms to the record:

\[
\text{Struct};
\]

\[
a_1 : \psi_1(\text{self}.A, \ldots, \text{self}.f, \ldots, \text{self}.R, \ldots);
\]

\[
\ldots;
\]

\[
a_n : \psi_n(\text{self}.A, \ldots, \text{self}.f, \ldots, \text{self}.R, \ldots)
\]

An event-ordering is a structure \( \langle E, \text{loc}, < \rangle \) that satisfies six axioms stating that equality is decidable, and that \( < \) is well-founded, transitive, locally-finite, decidable, and a total ordering of events at the same location. The type

\[
\text{EventOrdering} \equiv_{\text{def}}
\]

\[
E : \text{Type};
\]

\[
< : \text{self}.E \rightarrow \text{self}.E \rightarrow \text{P};
\]

\[
\text{loc} : \text{self}.E \rightarrow \text{Loc};
\]

\[
deq : \forall e_1, e_2 : \text{self}.E. (e_1 = e_2) \lor (e_1 \neq e_2);
\]

\[
\text{wf} : \exists f : \text{self}.E \rightarrow \text{N}. \forall e_1, e_2 : \text{self}.E.
\]

\[
\quad (e_1 \text{self}. < e_2) \Rightarrow (f(e_1) < f(e_2));
\]

\[
\text{dco} : \forall e_1, e_2 : \text{self}.E.
\]

\[
\quad (e_1 \text{self}. < e_2) \lor \neg(e_1 \text{self}. < e_2);
\]

\[
\text{trans} : \forall e_1, e_2, e_3 : \text{self}.E.
\]

\[
\quad ((e_1 \text{self}. < e_2) \land (e_2 \text{self}. < e_3))
\]

\[
\Rightarrow (e_1 \text{self}. < e_3);
\]

\[
\text{fin} : \forall e : \text{self}.E. \exists L : \text{self}.E \text{ List}.
\]

\[
\forall e' : \text{self}.E. (e' \text{self}. < e)
\]

\[
\Rightarrow (e' \in L);
\]

\[
\text{total} : \forall e_1, e_2 : \text{self}.E.
\]

\[
\quad \text{self.loc}(e_1) = \text{self.loc}(e_2) \Rightarrow
\]

\[
\quad (e_1 = e_2) \lor (e_1 \text{self}. < e_2) \lor (e_2 \text{self}. < e_1)
\]

Figure 1. Definition of EventOrdering

EventOrdering is defined in figure 1. An extended event-ordering adds the operation \( \text{info}(e) \), so its type is

\[
\text{EventOrdering}^* \equiv_{\text{def}}
\]

\[
\text{EventOrdering};
\]

\[
\text{info} : \text{self}.E \rightarrow \text{Msg}
\]

B Appendix: Proof of Theorem 3

B.1 Parallel and sequential composition

Processes in separate components of a system operate in parallel, but to prove that the programmable event classes are closed under the simple and recursive combinators, it is useful to encapsulate the parallel composition of processes within a single component. This is easily done. Suppose \( Ps = \{P_1, \ldots, P_n\} \) is a list of processes. We can define a new process that starts with \( P_0 \) as its internal state. When it gets an input message, it passes it to each of the processes in its internal state, who each compute new internal and external parts. The new external part is then some combination \( F \) of the external parts of the internal state. The diagram in figure 2 is similar to a hardware circuit, but since the inner processes are “higher order” they could be “re-programmed” by sending them process messages. The formal definition
of the parallel composition is:

\[ \text{Par}(Ps, F) \equiv \text{RecPr}(\text{next}, Ps) \text{ where } \]
\[ \text{next}(Ps) = \lambda m. \text{let } \text{Pes}' = \text{map}(\lambda P. P(m), Ps) \text{ in } \]
\[ ((\text{map}(\text{fst}, \text{Pes}'), F(\text{map}(\text{snd}, \text{Pes}')))) \]

Similarly, we can define a sequential composition as in

\[ ((X_i)'', X_1, \ldots, X_n) \text{ is programmable.} \]

Proof. Each \( X_i = \text{Class}(S_i) \) for some system \( S_i \).

1. We make a system that recognizes \( \hat{F}(X_1, \ldots, X_n) \) as follows: At each location \( x \) in the union of the systems, \( S_i \), we pick the component \( C^x_i \) at location \( x \) from \( S_i \) (and supply a “null component” if \( S_i \) does not have a component at \( x \)). Then we make the list \( Ps = \text{map}(\text{snd}, [C^x_1, \ldots, C^x_n]) \), the process of each component, and use this as the internal state of the parallel composition operator from section B.1, shown in figure 2. From this we make a combined component \( C^x_{\text{comb}} \) and the resulting system of combined components does the job.

2. To make the system for \( (X)' \) from the system for \( X \), we use the sequential composition operator, shown in figure 3, with the second, \( Q \), box being the “buffer” process mentioned in example B.1.

3. To recognize \( \hat{H}((\text{self})'', X_1, \ldots, X_n) \), we combine the methods used in the previous two cases. The process construction uses a combination of parallel and sequential composition that results in the “circuit” with feedback, shown in figure 4.

Lemma 5. A basic class \( \text{Kind}(k, xs) \) is programmable.

Proof. Make a system with one component at each location \( x \in xs \). The component has a “stateless” process that act as follows: If its input message has kind \( k \), then it removes \( k \) from the header and puts the message into its external part, and otherwise makes the external part empty. □

To prove the rest of Theorem 3 we must show that if \( X_1, \ldots, X_n \) are programmable classes, and \( F \in \text{Msg}^n \rightarrow \text{Msg} \) is strict, and \( H \in \text{Msg}^{n+1} \rightarrow \text{Msg} \) is strict, then

1. \( \hat{F}(X_1, \ldots, X_n) \) is programmable.