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 π-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 π-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 π-calculus to illustrate this generality. The higher-order process terms are type free lambda terms built with the Y 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 type 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, $\langle E,\ loc,\ <,\ info\rangle$, 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, $info(e)$, associated with event e is the message input to $loc(e)$ 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 $j = neighbor(G, i)$ is a simple cycle. When any component in a group G receive a message $\langle [elect], G\rangle$ 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 $uid(i)$ 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 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 $\langle [elect], G\rangle$ 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. As we will see below, this allows us to derive recognizable event classes $Start^+$, $Propose^+$, and $Leader^+$ that assign values as follows:

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

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.)

The general form of the algorithm in this example and many other distributed algorithms is this: A component recognizes some basic event. It associates additional information, which it computes as a function of its prior input history, 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 *programmable* event class. We describe the part of the algorithm that sends the information to other components 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,

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

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" and 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:

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

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 our main results is that propagation rules like those used in the leader election example have strong realizers. A realizer for a propagation rule $A \overset{f}{\Rightarrow} B@g$ 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 send 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 *combinators*. 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 $A|(\neg F) \Rightarrow B$ 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 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 $A \overset{f}{\Leftarrow} B@g$. A realizer constructed for $A \overset{f}{\Rightarrow} B@g$ will generate B-events only from A-events, so it will also realize $A \overset{f}{\Leftarrow} B@g$. 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 $C \Leftarrow D$ 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 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 have 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 $\langle x, y \rangle$. The members of a *disjoint union* $A + B$ are $\{inl(a) | a \in A\} \cup \{inr(b) | b \in B\}$ where inl and inr are primitive constructors. The only member of type $Unit$ is $()$ and we write $inr(()) = \perp$.

Locations The type that represents locations is a primitive type $Loc \equiv_{def} Atom$. 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 $T = tg : Atom \times M(tg)$ where M is a function $Atom \to Type$.

Then a data value is a pair $\langle tg, v\rangle$ 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:

$$\begin{array}{rcl} \mathit{Ext} & \equiv_{def} & (\mathit{Loc} \times \mathit{Msg})\ \mathit{List} \\ \mathit{Component} & \equiv_{def} & \mathit{Loc} \times \mathit{Process}\ \times \mathit{Ext} \\ \mathit{System} & \equiv_{def} & \mathit{Component\ List} \end{array}$$

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

$$\mathit{Process} \subseteq \mathit{Msg} \to (\mathit{Process} \times \mathit{Ext})$$

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, $\mathit{Top} \equiv_{def} \bigcap_{x:\ \mathit{Void}} \mathit{Void}$, and a function of type $\mathit{Top} \to T$ must be a constant function. (Terms in NuPrl do not have a single “best” type—for example, $17 \in \mathbb{N}$, $17 \in \mathit{Top}$, and $17 \in \{m\colon \mathbb{N} | m > 5\}$.)

Co-inductive types are defined, for $F \in \mathit{Type} \to \mathit{Type}$, $F^0(T) = T$, and $F^{k+1}(T) = F(F^k(T))$, by

$$\mathit{corec}(P.F(P)) \equiv_{def} \bigcap_{k:\ \mathbb{N}} F^k(\mathit{Top})$$

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 $X \in \mathbb{N} \to \mathit{Type}$,

$$\bigcap_k F(X(k)) \equiv F(\bigcap_k X(k))$$

Function F is *weakly continuous* if

$$\bigcap_k F(X(k)) \subseteq F(\bigcap_k X(k))$$

If F is weakly continuous then

$$\mathit{corec}(P.F(P)) \subseteq F(\mathit{corec}(P.F(P)))$$

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]

$$\begin{array}{rcl} M(P) & \equiv_{def} & (\mathit{Atom\ List}) \times (T + P) \\ E(P) & \equiv_{def} & (\mathit{Loc} \times M(P))\ \mathit{List} \\ F(P) & = & M(P) \to (P \times E(P)) \end{array}$$

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

$$\begin{array}{rcl} \mathit{Process} & \equiv_{def} & \mathit{corec}(P.F(P)) \\ \mathit{Msg} & \equiv_{def} & M(\mathit{Process}) \\ \mathit{Ext} & \equiv_{def} & E(\mathit{Process}) \end{array}$$

then

$$\mathit{Process} \subseteq F(\mathit{Process}) = \mathit{Msg} \to \mathit{Process} \times \mathit{Ext}$$

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

Definition 1.

$$\begin{array}{l} \mathit{RecPr}(\mathit{next}, \mathit{ext},\ s) =_{rec} \\ \qquad \lambda m.\ \mathbf{let}\ \langle s', e\rangle = \mathit{next}(s,\ m)\ \mathbf{in} \\ \qquad\quad \mathbf{let}\ P = \mathit{RecPr}(\mathit{next}, \mathit{ext}, s')\ \mathbf{in} \\ \qquad\quad \langle P,\ \mathit{ext}(e, m, P)\rangle \end{array}$$

The parameter s in $\mathit{RecPr}(\mathit{next},\ s)$ is the *internal state* of the process, and 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 ext. To build processes that never add themselves to messages we write just $\mathit{RecPr}(\mathit{next}, s)$ and supply the default $\mathit{ext}(e, m, P) = e$.

[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 $F'(T) \equiv_{def} M(T) \to (S(T) \times E(T))$*

If
$s \in S(\mathit{Process})$
$\mathit{ext} \in \bigcap_T .\, E(T) \to M(T) \to T \to E(T)$
$\mathit{next} \in \bigcap_T .\, S(F(T)) \to F'(T)$
then $\mathit{RecPr}(\mathit{next}, \mathit{ext}, s) \in \mathit{Process}$

Proof. We claim that

$$\forall k.\, \forall st\colon S(F^k(\mathit{Top})).\ \mathit{RecPr}(\mathit{next}, \mathit{ext}, st) \in F^k(\mathit{Top})$$

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

$$s \in S(\mathit{Process}) = S(\bigcap_k F^k(\mathit{Top})) \subseteq \bigcap_k F(F^k(\mathit{Top}))$$

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

$$\mathit{RecPr}(\mathit{next}, \mathit{ext}, s) \in \bigcap_k F^k(\mathit{Top}) = \mathit{Process}\ \square$$

□

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

$$\begin{aligned}
\mathit{pmsg}(\mathit{hdr}, P) &\equiv_{def} \langle \mathit{hdr}, \mathit{inr}(P)\rangle \\
\mathit{header}(m) &\equiv_{def} \mathit{fst}(m) \\
\mathit{body}(m) &\equiv_{def} \mathit{snd}(m) \\
\mathit{rmheader}(m) &\equiv_{def} \langle \mathit{tail}(\mathit{header}(m)), \mathit{body}(m)\rangle \\
\mathit{addheader}(k, m) &\equiv_{def} \langle \mathit{cons}(k, \mathit{header}(m)), \mathit{body}(m)\rangle
\end{aligned}$$

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

$$\begin{aligned}
P^*(\mathit{nil}) &= \langle P, \mathit{nil}\rangle \\
P^*(L \oplus [m]) &= \textbf{let } \langle P', \mathit{ext}\rangle = P^*(L) \textbf{ in } P'(m)
\end{aligned}$$

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, *boot*, that has an internal state of type $\mathit{Process} + \mathit{Unit}$. Initially it will have state $\perp$ and the first time it gets a process message, $\mathit{pmsg}(\mathit{hdr}, Q)$, it will change its internal state to $\mathit{inl}(Q)$. 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,

$$\begin{aligned}
\mathit{boot} &\equiv_{def} \mathit{RecPr}(\mathit{next}, \perp) \textbf{ where} \\
\mathit{next}(\perp, m) &= \langle G(m),\ \mathit{nil}\rangle \textbf{ where} \\
G(m) &= \textbf{if } m = \mathit{pmsg}(\mathit{hdr}, Q) \\
& \ \ \textbf{then } \mathit{inl}(Q) \textbf{ else } \perp \\
\mathit{next}(\mathit{inl}(Q), m) &= \textbf{let } \langle Q',\ e\rangle = Q(m) \\
& \ \ \textbf{in } \langle \mathit{inl}(Q'),\ e\rangle
\end{aligned}$$

The internal state has type $S(\mathit{Process})$ where $S(T) = T + \mathit{Unit}$, so S is continuous. It is easy to see that the given function, *next*, has the polymorphic type given in lemma 1. So, *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 $\mathit{Msg} \to (\mathit{Loc} \times \mathit{Atom\ List})?$, we can define a process, $\mathit{forkable}(P, f)$, that interprets a message m with $f(m) = \mathit{inl}(x, \mathit{hdr})$ as the instruction send itself to location x in a message with header *hdr*, but otherwise acts like a given process P. By Lemma 1, the following defines the desired process:

$$\begin{aligned}
\mathit{forkable}(P, f) &\equiv_{def} \mathit{RecPr}(\mathit{next}, \mathit{ext}, P) \textbf{ where} \\
\mathit{next}(P, m) &= P(m) \\
\mathit{ext}(e, m, P) &= \textbf{if } f(m) = \mathit{inl}(x, \mathit{hdr}) \\
& \ \ \textbf{then } [\langle x, \mathit{pmsg}(\mathit{hdr}, P)\rangle)] \\
& \ \ \textbf{else } e
\end{aligned}$$

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

$$\begin{aligned}
F(n) \ =_{rec}\ & \textbf{if } n = 0 \textbf{ then } \langle S, \mathit{env}(0)\rangle \\
& \textbf{else } \langle \mathit{Next}(F(n-1), \mathit{boot}), \mathit{env}(n)\rangle
\end{aligned}$$

The detailed definition of $\mathit{Next}(\langle S_n, \mathit{choice}_n\rangle, \mathit{boot})$ is straightforward and has been formalized in NuPrl. The boot-process *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 $EO^+(R) = \langle E,\ loc,\ <,\ info\rangle$ 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, $e = \langle x, n\rangle$, at which location x gets a 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 $loc(e)$ of event e is its first component, and $info(e)$ is the input message it received.

Event $e_1 = \langle x, n\rangle$ is the *local predecessor* of $e_2 = \langle y, m\rangle$ if $x = y$ and $n < m$ and there are events $\langle x, k\rangle$ with $n < k < m$. Event $e_1 = \langle x, n\rangle$ is an *immediate predecessor* of $e_2 = \langle y, m\rangle$ 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 $<$ is, by definition, the transitive closure of the immediate-predecessor relation.

We formalized the details in NuPrl and proved that $<$ is well-founded, transitive, locally-finite, decidable, and a total ordering of events at the same location. We call a structure $\langle E, loc, <\rangle$ 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 $\lambda e.\ info(e)$ so that events are associated with some primitive information content. We define the type, $EventOrdering^+$, 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:

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

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

  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 $eo \in EventOrdering^+$, so they include eo as an implicit parameter. To make this explicit we sometimes write, e.g., $history_{eo}(e)$, $loc_{eo}(e)$, 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 Msg. Since the body of a message has type $T + (m\colon \mathbb{N} \times Process_m)$, and since the data type T is assumed to be sufficiently universal, we can put whatever information we need into a member of type Msg.

Therefore we define:

$$EClass \equiv_{def}\ eo\colon EventOrdering^+ \to E(eo) \to Msg?$$

where $Msg? \equiv_{def} (Msg + Unit)$. 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 $k \in Atom$ we say that event e has *kind* k if $head(header(info(e))) = k$. Then, if xs is a list of locations,

$$\begin{aligned} Kind(k, xs) \equiv_{def} \lambda eo.\lambda e.&\textbf{if } loc_{eo}(e) \notin xs \textbf{ then } \bot\\ &\textbf{if } head(header(info_{eo}(e))) \neq k \textbf{ then } \bot\\ &\textbf{else } inl(rmheader(info_{eo}(e)))\end{aligned}$$

is an event class such that $E(Kind(k, xs))$ are the events of kind k whose location is in xs, and $Kind(k, xs)(e)$ is $rmheader(info(e))$. We call the classes $Kind(k, xs)$ 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 **when** e and x **after** e for observing state variables before and after events, and an axiom $\neg \mathit{first}(e) \Rightarrow (x \textbf{ when } e = x \textbf{ after } pred(e))$

If $propose$ and $decide$ are atoms, and xs lists the locations of the components in a consensus protocol, let $P = Kind(propose, xs)$ and $D = Kind(decide, xs)$. 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):

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

Simple combinators If $X_1, \ldots, X_n$ are event classes, and $F \in Msg?^n \to Msg?$, then

$$\hat{F}(X_1, \ldots, X_n) \equiv_{def} \lambda eo.\lambda e.\ F(X_1(eo, e), \ldots, X_n(eo, e))$$

is an event class. When F is *strict*, $F(\perp, \ldots, \perp) = \perp$, then we call $\hat{F}$ a *simple* combinator[5].

The "prime" combinator If X is an event class, the class $(X)'$ 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 $(X)'$ is the value if the most recent locally prior X-event. Formally,

$$e \in E(X') \Leftrightarrow \exists e' : E(X).\ loc(e') = loc(e)\ \wedge\ e' < e$$

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

Recursion combinators

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

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

We write $\hat{H}((self)', X_1, \ldots, X_n)$ 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 $e \in Neighbor$, where some function f of the value $Neighbor(e)$ is a pair $\langle G, j\rangle$, and this informs the component at $loc(e)$ 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 $Neighbor$ 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:

$$\begin{array}{rcl} NbrTab & \equiv_{def} & \hat{H}((self)', Neighbor), \textbf{where} \\ H(\perp, m) & = & [f(m)] \\ H(tabl, m) & = & \textbf{let } \langle G, j\rangle = f(m) \textbf{ in } tabl[G := j] \end{array}$$

(Actually, because all event classes have values of type 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 $\langle x, P, ext_0\rangle$ recognizes X-events at x if for any event e with $loc(e) = x$,*

$$\begin{array}{c} snd(P^*(history(e))) = \\ \textbf{if } e \in E(X) \textbf{ then } [\langle x, X(e)\rangle] \textbf{ else } nil \end{array}$$

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 $Kind(k, xs)$ is programmable. If $X_1, \ldots, X_n$ are programmable classes, and $F \in Msg?^n \to Msg?$ is strict, and $H \in Msg?^{n+1} \to Msg?$ is strict, then*

1. *$\hat{F}(X_1, \ldots, X_n)$ is programmable.*
2. *$(X_i)'$ is programmable.*
3. *$\hat{H}((self)', X_1, \ldots, X_n)$ is programmable.*

[5] If F is not strict, $F(\perp, \ldots, \perp) = inl(m)$, for some message m. Then, any event e *not* in any of $E(X_1), \ldots, E(X_n)$ would be in $E(F(X_1, \ldots, X_n))$ and have value m. This would make it difficult to recognize such events, since they could occur at any location.

The constructive proof of theorem 3 is in appendix B. The program derived from the constructive proof of this theorem provides the core of a compiler for a language[6] that allows programmers to define combinators, basic classes, and classes composed from these.

5 Realizability

To limit the number of parameters, let's fix a boot-process, $boot$, say the one in example 2. A run R of system S has the form $Run(S, boot, env)$ for some $env \in \mathbb{N} \to Choice$. We say that the event ordering $EO^+(R)$ of a run of S is an ordering *consistent with* S. Most requirements for a distributed system can be expressed as constraints $\psi(eo)$ on the event-orderings consistent with it (ψ has type $EventOrdering^+ \to \mathbb{P}$). For example, the agreement and validity requirements for a consensus protocol (example 4) have this form. Generalizing the "proofs-as-programs" paradigm we view a system as *evidence* that such requirements are *realizable*. Usually, we assume some conditions about the environments (e.g. a fairness condition). So, if S is a system, and C is a proposition on environments, we define

$$\begin{array}{l} \text{“}S \textit{ realizes } \psi\text{”} \\ S \vdash_C \psi \Leftrightarrow \forall env.\ C(env) \Rightarrow \\ \qquad\qquad \psi(EO^+(Run(S, boot, env))) \\ \text{“}\psi \textit{ is realizable}\text{”} \\ \vdash_C \psi \Leftrightarrow \exists S.\ S \vdash_C \psi \\ \text{“}S \textit{ strongly realizes } \psi\text{”} \\ S \vDash_C \psi \Leftrightarrow \forall S'.\ S \subseteq S' \Rightarrow S' \vdash_C \psi \\ \text{“}\psi \textit{ is strongly realizable}\text{”} \\ \vDash_C \psi \Leftrightarrow \exists S.\ S \vDash_C \psi \end{array}$$

Realizability is not a useful concept unless it can be used in compositional reasoning. From a realizer of ψ_1 and a realizer of ψ_2 there must be an "easy" way to construct a realizer for $\psi_1 \wedge \psi_2$. Clearly, if $S_1 \vDash_C \psi_1$ and $S_2 \vDash_C \psi_2$ then $S_1 \oplus S_2 \vDash_C \psi_1 \wedge \psi_2$. Also, if $\phi \Rightarrow \psi$ and $S \vDash_C \phi$ then $S \vDash_C \psi$. So, strong realizers allow compositional, *logical refinement*

$$\textbf{if } ((\bigwedge_i \phi_i) \Rightarrow \psi) \textbf{ and } (\forall i.\ \vDash_C \phi_i) \textbf{ then } \vDash_C \psi$$

If property ϕ does not have a strong realizer, then for each realizer S, such that $S \vdash_C \phi$, there should at least be an "easily provable" *compatibility test* $P_S(S')$ such that

$$\forall S'.(P_S(S') \Rightarrow S \oplus S' \vdash_C \phi$$

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, ϕ_{rmd}, 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 ϕ_{rmd} 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 ϕ_{rmd} to $\phi_{rmd}(F)$ 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, $B = Kind(k, xs)$ is basic, and $range(g) \subseteq xs$, then* $\vDash_{\phi_{rmd}} A \overset{f}{\Rightarrow} B@g$

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 $[\langle z, A(e)\rangle]$ it produces $map(\lambda x.\ \langle x, addheader(k, f(A(e)))\rangle, g(A(e)))$. That the modified system S strongly realizes $A \overset{f}{\Rightarrow} B@g$ follows easily from the assumption ϕ_{rmd}. □

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 $A \overset{f}{\Leftarrow} B@g$ 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 $P_S(S') \Rightarrow S \oplus S' \vdash_{\phi_{rmd}} (A \overset{f}{\Leftarrow} B@g)$. Because the basic B-events all have header k, the compatibility test $P_S(S')$ can be $avoids(k, S')$—S' does not generate any messages with header k. If S' is the realizer of another PB-rule, $C \overset{f}{\Rightarrow} D@g$, then it is easy to check whether $avoids(k, S')$—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 $avoids(k, S')$ 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

[6] We have implemented a prototype called $E^{\#}$.

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 π-calculus to 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 components, each group having the form $\mathbf{private}\ k.S(k)$ 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 $Kind(k, xs)$ 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 encryption key. For such names k the $addheader(k, m)$ operation is really $encrypt_k(m)$ and the $rmheader(m)$ is really $\mathbf{if}\ encrypted_k(m)\ \mathbf{then}\ decrypt_k(m)\ \mathbf{else}\ rmheader(m)$.

7 Modeling the π-calculus

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

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

where $n > 0$ and a prefix π_i is either the "get" operation $c(x)$ or the "put" operation $\overline{c}\,x$. We will model the basic π-calculus semantics that does not require communications to be chosen "fairly."

Several basic differences between π-calculus and the general process model must be negotiated. The notion of location, fundamental in the general process model, doesn't exist in π-calculus. In the general process model, a process sends a message to a known recipient by labeling the message with the recipient's location, whereas a π-calculus process 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 π-calculus, communication is synchronous and rendezvous are effected by the environment, using global information. Finally, a process in the general process model is purely reactive; it acts only in response to an input. A π-calculus process is active at least in the sense that it can, for example, "fork" replicas or initiate 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 π-calculus process terms and "bookkeeping processes" will be added to manage and constrain their interactions.

7.1 Communication

It seems appealing, at first glance, to encode a channel as a process, but π-calculus semantics requires global decisions 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 *Comm*, at location l_{comm}, that manages all communication. A π-calculus term that can immediately engage in communications has the form $\pi_1.P_1 + \ldots + \pi_n.P_n$ A process at location l that encodes this term will send to *Comm* a message containing l (the return address) and the sequence of possible communications. That is, it will place

$$[l_{comm} : \langle l, [\pi_1, \ldots, \pi_n] \rangle]$$

on its external part.[7] The process will block until it receives a reply from *Comm*.

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 $\langle l, \vec{\pi} \rangle$ 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 π_i could contain repetitions—e.g., the process could be $c(x).P + c(x).Q$, so that the corresponding *Reqs* is $\langle l, [c(x), c(x)] \rangle$. *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 nonderministically from the full range of possibilities. 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

$$\begin{array}{rcl} [\![P]\!] & : & \mathit{Loc} \rightarrow \mathit{Process} \\ \mathcal{M}(P) & : & \mathit{System} \end{array}$$

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 bookeeping 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 $\mathit{LServer}(L)$ 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 $[\![\mathbf{0}]\!]_l$ 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

$$\langle \mathbf{Q}, [l_{comm} : \langle l, [\pi_1 \ldots, \pi_n] \rangle] \rangle$$

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

$$\langle \mathbf{P}_i, [l : \mathsf{fire}] \rangle$$

That is, it becomes process $\mathbf{P}_i$ and tells $\mathbf{P}_i$ to fire. The definition of $\mathbf{P}_i$ depends on π_i. If, for example, $\pi_i = c(x)$,

$$\mathbf{P}_i = [\![P_i[x \ := \ v]]\!]_l$$

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 $[\![(\nu x)P]\!]_l$ 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_{comm}, l_C, l_N, l_L, and l. $\mathcal{M}(P)$ consists of the following five components, together with the boot process boot^+.

- $\langle l_{comm}, Comm, nil \rangle$
- $\langle l_C, Choose, nil \rangle$
- $\langle l_N, NServer(N), nil \rangle$
- $\langle l_L, LServer(\{l_C, l_{comm}, l_N, l_L, l_0\}), nil \rangle$
- $\langle l, [\![P]\!]_l(\mathsf{fire}) \rangle$

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 $\mathcal{M}(!P)$ 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 Tarksi 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 π-calculus for mobility as well.

References

[ABC06] Stuart Allen, Mark Bickford, Robert Constable, Richard Eaton, Christoph Kreitz, Lori Lorigo, and Evan Moran. Innovations in Computational Type Theory using Nuprl, *Journal of Applied Logic*, Elsevier Science, 2006, 428-469.

[Abr94] S. Abramsky. Proofs-as-processes. *Journal of Theoretical Computer Science,* 135(1), 5-9, 1994.

[Abr99] Uri Abraham. *Models for Concurrency*, Gordon and Breach Science Publishers, 1999.

[AW04] Hagit Attiya and Jennifer Welch. *Distributed Computing*, 2nd Edition, Wiley, 2004.

[BB08] Bruno Barras and Bruno Bernardo. The Implicit Calculus of Constructions as a programming language with dependent types, In *Proc. FoSSaCS, LNCS* 4962, Springer, 2008.

[BC85] Joseph L. Bates and Robert L. Constable. Proofs as Programs, *TOPLAS*, Vol 7 (1), 1985, 53-71.

[BCHP05] Mark Bickford, Robert Constable, Joseph Halpern, and Sabina Petride, Knowledge-based synthesis of distributed systems using event structures, *Logic for Programming, Artificial Intelligence, and Reasoning, LNCS 3452,* NY,449-465,2005.

[Bic09] Mark Bickford. Component Specification Using Event Classes. CBSE '09: Proceedings of the 12th International Symposium on Component-Based Software

Engineering, *Lecture Notes in Computer Science*, Vol 5582,2009, 140-155.

[Bic08] Mark Bickford. Unguessable Atoms: A Logical Foundation for Security. *Verified Software:Theories, Tools, Experiments, Second International Conference, VSTTE 2008,* 30-53, 2008.

[BG03a] Andreas Blass and Yuri Gurevich. Abstract state machines capture parallel algorithms, *ACM Transactions on Computational Logic*, 4(4), 578-651, 2003.

[CB08] Robert Constable and Mark Bickford. Formal Foundations of Computer Security, *NATO Science for Peace and Security Series D: Information and Communication Security,* Vol 14, 29-52, 2008.

[DDMR07] A.Datta, A.Derek, J.C.Mitchell, A.Roy. Protocol Composition Logic, *Electronic Notes in Theoretical Computer Science*, 172, 311-358, 2007.

[GGV04] U.Glaesser, Y. Gurevich, and M. Veanes. Abstract Communication Model for Distributed Systems, IEEE Transactions on Software Engineering, 30(7), 458-472,2004.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faculty process. volume 32 of *JACM*, pages 374–382, 1985.

[HKP82] D. Harel, D. Kozen, and R. Parikh. Process logic: Expressiveness, decidability, completeness, *Journal of Computer and System Sciences* 25(2), pp. 144–170, 1982.

[HM85] M. C. B. Hennessy and R. Milner. Algebraic laws for non-determinism and concurrency, *Journal of the ACM* 32(1),137-161, 1985.

[Kop03] Alexei Kopylov Dependent Intersection: A New Way of Defining Records in Type Theory, *Proceedings of 18th Annual IEEE Symposium on Logic in Computer Science*, pp. 86-95, 2003.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system, *Comm ACM*, 21(7), 558-65, 1978.

[LKvR+99] X.Liu, C. Kreitz, R. van Renesse, J.Hickey, M. Hayden, K. Birman, R. Constable. Building reliable, high-performance communication systems from components, *ACM Symposium on Operating Systems Principles (SOSP)*, ACM Press, 80-92, 1999.

[Lyn96] Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, San Mateo, CA, 1996.

[MW92] Jose Meseguer and Tim Winkler. Parallel programming in Maude, LNCS 574, Springer, NY, 253-293, 1992.

[Mil89] Robin Milner. *Communication and Concurrency*, Prentice-Hall, London, 1989.

[Mil09] Robin Milner. *The Space and Motion of Communicating Agents*, Cambridge University Press, 2009.

[MCH04] T. Murphy, K. Crary, and R. Harper. A Symmmetric Modal Lambda Calculus for Distributed Computing, Carnegie Mellon University CS Technical Report 105, 2004.

[Pnueli81] Amir Pnueli. The temporal logic of concurrent programs, *Theoretical Computer Science* 13,45-60, 1981.

[Pratt91] Vaughan Pratt. Event Spaces and Their Linear Logic, *Proceedings of the Second International Conference on Methodology and Software Technology: Algebraic Methodology and Software Technology*, 3-25, 1991.

[SG96] Douglas Smith and Cordell Green. Toward practical applications of software synthesis, FMSSP, 31-39, 1996.

[Vardi95] Moshe Y. Vardi. An automata-theoretic approach to fair realizability and synthesis, CAV, LNCS,939, Springer, NY, 267-292, 1995.

[Win80] Glynn Winskel. *Events in Computation*, PhD Thesis, University of Edinburgh, 1980.

[Win89] Glynn Winskel. An introduction to event structures, LNCS 345, Springer, NY, 364-397, 1989.

[WMM09] R. Wisnesky, G. Malecha, and G. Morrisett. Certified web services in Ynot, *In Proceedings of the 5th International Workshop on Automated Specification and Verification of Web Systems*,5-20, 2009.

[XL09] Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant, In *Journal of Automated Reasoning* 43(4),363-446, 2009.

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 $A \in \mathit{Type}$, and $B \in A \to \mathit{Type}$, then the *dependent intersection* $x\colon A \cap B(x)$ is also a type [Kop03]. A term t is a member of type $x\colon A \cap B(x)$ if $t \in A$ and $t \in B(t)$. This definition makes sense because when $t \in A$, then $B(t) \in \mathit{Type}$. The dependent intersection allows a certain kind of "self reference", so we often use the variable *self* as the bound variable and write the type as $\mathit{self}\colon A \cap B(\mathit{self})$.

We use this type to define *dependent record types* as follows. Define the type $T; z\colon B[\mathit{self}]$ to be

$$\mathit{self}\colon T \cap (x\colon \mathit{Atom} \to \textbf{if } x = z \textbf{ then } B[\mathit{self}] \textbf{ else } \mathit{Top})$$

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 Top and iterate the record type constructor to build arbitrary dependent records. We write $\mathit{Top}; x\colon A$ as simply $x\colon A$, so that a type like $\mathit{Top}; x\colon A; z\colon B[\mathit{self}.x]$ is written $x\colon A; z\colon B[\mathit{self}.x]$. If $r \in x\colon A; z\colon B[\mathit{self}.x]$ 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, Struct, like:

$$\begin{array}{l} A\colon \mathit{Type}; \\ \ldots; \\ f\colon \mathit{self}.A \to \mathit{self}.A; \\ \ldots; \\ R\colon \mathit{self}.A \to \mathit{self}.A \to \mathbb{P} \\ \ldots \end{array}$$

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:

$$\begin{array}{l} \mathit{Struct}; \\ ax_1\colon \psi_1(\mathit{self}.A, \ldots, \mathit{self}.f, \ldots, \mathit{self}.R, \ldots); \\ \ldots; \\ ax_n\colon \psi_n(\mathit{self}.A, \ldots, \mathit{self}.f, \ldots, \mathit{self}.R, \ldots) \end{array}$$

An *event-ordering* is a structure $\langle E, \mathit{loc}, < \rangle$ 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 *EventOrdering* is defined in figure 1. An extended event-ordering adds the operation $\mathit{info}(e)$, so its type is

$$\begin{array}{l} \mathit{EventOrdering}^{+} \equiv_{def} \\ \mathit{EventOrdering}; \\ \mathit{info}\colon \mathit{self}.E \to \mathit{Msg} \end{array}$$

$$\begin{array}{l} \mathit{EventOrdering} \equiv_{def} \\ E\colon \mathit{Type}; \\ <\colon \mathit{self}.E \to \mathit{self}.E \to \mathbb{P}; \\ \mathit{loc}\colon \mathit{self}.E \to \mathit{Loc}; \\ \mathit{deq}\colon \forall e_1, e_2\colon \mathit{self}.E.\ (e_1 = e_2) \vee (e_1 \neq e_2); \\ \mathit{wf}\colon \exists f\colon \mathit{self}.E \to \mathbb{N}.\ \forall e_1, e_2\colon \mathit{self}.E. \\ \quad (e_1\ \mathit{self}.<\ e_2) \Rightarrow (f(e_1) < f(e_2)); \\ \mathit{dco}\colon \forall e_1, e_2\colon \mathit{self}.E. \\ \quad (e_1\ \mathit{self}.<\ e_2) \vee \neg(e_1\ \mathit{self}.<\ e_2); \\ \mathit{trans}\colon \forall e_1, e_2, e_3\colon \mathit{self}.E. \\ \quad ((e_1\ \mathit{self}.<\ e_2) \wedge (e_2\ \mathit{self}.<\ e_3)) \\ \quad \Rightarrow (e_1\ \mathit{self}.<\ e_3); \\ \mathit{fin}\colon \forall e\colon \mathit{self}.E.\ \exists L\colon \mathit{self}.E\ \mathit{List}. \\ \quad \forall e'\colon \mathit{self}.E.\ (e'\ \mathit{self}.<\ e) \Rightarrow (e' \in L); \\ \mathit{total}\colon \forall e_1, e_2\colon \mathit{self}.E. \\ \quad \mathit{self}.\mathit{loc}(e_1) = \mathit{self}.\mathit{loc}(e_2) \Rightarrow \\ \quad (e_1 = e_2) \vee (e_1\ \mathit{self}.<\ e_2) \vee (e_2\ \mathit{self}.<\ e_1) \end{array}$$

Figure 1. Definition of EventOrdering

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 $Ps = [P_1 \ldots P_k]$ is a list of processes. We can define a new process that starts with Ps as its internal state. When it get 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:

$$\begin{aligned} Par(Ps, F) &\equiv_{def} \; RecPr(next, Ps) \textbf{ where} \\ next(Ps) = \lambda m. \textbf{ let } & Pes' = map(\lambda P.\, P(m), Ps) \textbf{ in} \\ & \langle (map(fst, Pes'), F(map(snd, Ps'))\rangle \end{aligned}$$

Similarly, we can define a sequential composition as in figure 3. In the sequential case, the input to the second box Q is an external part rather than a message, so $Q \in Process'$ where $Process' \subseteq Ext \to (Process' \times Ext)$. A useful instance of this is the "buffer"

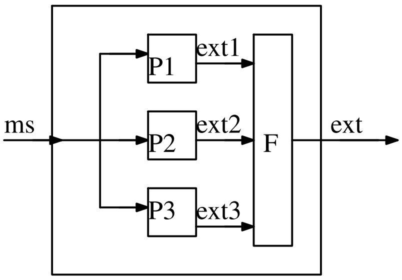

Figure 2. Parallel composition of processes

$$\begin{aligned} Q(s) \quad =_{rec} \quad & \lambda ext. \textbf{ let } s' = \textbf{if } null(ext) \textbf{ then } s \textbf{ else } ext \\ & \textbf{in } \langle Q(s'), s'\rangle \end{aligned}$$

In this case the sequential composition of P and Q "remembers" the last non-null external part produced by P.

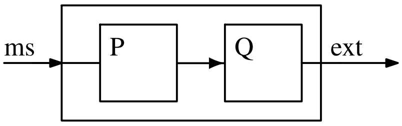

Figure 3. Sequential composition of processes

Lemma 5. *A basic class $Kind(k, xs)$ 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 $F \in Msg?^n \to Msg?$ is strict, and $H \in Msg?^{n+1} \to Msg?$ is strict, then

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

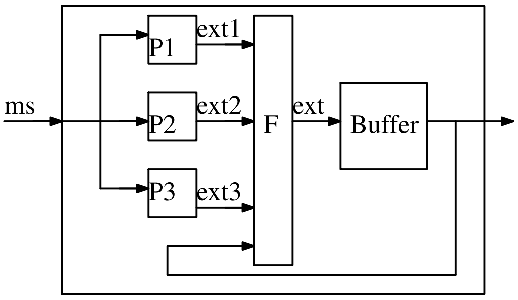

Figure 4. Parallel/Sequential composition with feedback

2. $(X_i)'$ is programmable.
3. $\hat{H}((self)', X_1, \ldots, X_n)$ is programmable.

Proof. Each $X_i = Class(S_i)$ 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_i^x 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 $Ps = map(snd, [C_1^x, \ldots, C_n^x])$, 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_{comb}^x 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 $\hat{H}((self)', X_1, \ldots, X_n)$, 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.

□