A Causal Logic of Events in Formalized Computational Type Theory *

Mark Bickford\textsuperscript{a} and Robert L. Constable\textsuperscript{b}

\textsuperscript{a}ATC, NY
\textsuperscript{b}Cornell University, Department of Computer Science

Abstract

We provide a logic for distributed computing that has the explanatory and technical power of constructive logics of computation. In particular, we establish a proof technology that supports correct-by-construction programming based on the notion that concurrent processes can be extracted from proofs that specifications are achievable.

1 Introduction

1.1 Historical Context

Models of computation have been important in mathematics since Greek geometry of 300 BC, and perhaps for much longer. We call these models formal if they can be implemented by (idealized) machines. The sustained development of formal computing models and their implementation is much more recent, a 20th century activity with some foreshadowing by Babbage in the late 19th century. The main focus is digital computation, and it has been revolutionary—creating a computational aspect of every science and giving birth to a new discipline called computer science, starting with Turing in 1936 [Tur37]. Digital computation has even been proposed as a new foundation for physics [Hey02, Whe82, Whe89].

In the late 20th century, the Internet and other networks of machines made distributed computing a transformative global resource. Reasoning about networks required a new model of computation. The resulting model of distributed computation is enormously rich,

\textsuperscript{*}This material is based upon work supported by the National Science Foundation under Grant No. 0208536.
and computer scientists are only beginning to create the concepts and tools to understand it deeply and exploit its potential in science, as well as in technology and commerce.

One of the critical challenges researchers have faced in understanding every model of computation is creating a declarative language that relates the dynamic nature of computation with the declarative basis of scientific theories. An illustrative example of this challenge already appears in Euclidean geometry.

Euclid’s propositions and postulates are a mixture of constructions and declarative statements. For example, he says essentially “given two points, we can draw a line (segment) connecting them”. He declares that in any triangle, the length of any two sides is greater than that of the remaining one. Corresponding to this is the problem of making a construction, namely ”given two line segments whose combined length is greater than a third, construct a triangle with these segments as sides.”

Euclidean geometry is a mixture of propositions, problems, postulates, and constructions. There are a finite number of basic postulates, and a finite number of atomic straight edge and compass construction methods (or schemes). This intuitive hybrid language sufficed for two thousand years. The geometric model of computing was far from formal, it was not even rigorous. Logicians then discovered how to make the declarative language rigorous, and eventually formal, using quantifiers, but “the quantifiers killed the constructions”. That is, instead of saying given points A and B we can construct a line segment between them, Hilbert said, given points A and B, there exists a line segment connecting them. Symbolically,

$$\forall A, B : \text{Points.} \exists L : \text{Line.} L = [AB].$$

1.2 Computational Logic

It took a few decades to sort out a declarative language with computational meaning. L.E.J. Brouwer showed the way, and in due course a computational (or constructive) interpretation of formal logic was achieved, called the Brouwer, Kolmogorov, Heyting (BKH) interpretation. We will use this below. See the book [GTL89] for the BKH interpretation.

This computational interpretation of the predicate calculus restored the balance between computation and assertion, and it became the basis for logics of computation that applied well to functional and procedural programs as demonstrated by deBruijn, Scott, Martin-Löf, Girard, Constable, Huet, Coquand, Paulin and others. These logics have enabled a very potent proof technology with applications both to mathematics and to software development. One of the key ideas in the logic of computation is the notion of proofs-as-programs [BC85], which will be of central concern here.

1.3 The Logical Challenge of Distributed Computing

The issue before us now is to find an adequate logic for distributed computing that has the explanatory and technical power of constructive logics of computation. In particular, we aspire to a proof technology that supports correct-by-construction programming based on
the notion that concurrent processes can be extracted from proofs that specifications are achievable. The goal has been elusive until now.

Equally elusive in the case of networked computation is finding a declarative language for specifying distributed computing problems at very high levels of abstraction. Languages such as TLA+ [Lam03] describe computation at the level of execution models, and even at their most general, such models are not sufficiently abstract to apply well in all the circumstances we have in mind.

We present a very abstract specification language which can be understood without direct reference to a computational model. As in the case of the language of Computational Type Theory (CTT [ABC+, CAB+86a, Con02]), there is a computation model behind it that is manifest in rules for reasoning. Likewise, in the setting of our computational theory of typed events (CTT-E), the inference rules will exploit the underlying computational interpretation. The computational interpretation through the inference rules is sufficiently strong that from a proof that a specification is achievable, we can automatically exact an executable distributed system.

We have formalized the logic and its implementation in the Nuprl system [ACE+00, CAB+86a] and the ScoRes distributed runtime environment [BG05] so that the creative steps of distributed system design and verification can be undertaken at a high logical level, and the detailed system programming can automated by the extractor/compiler. The extractor/compiler contains a large amount of detailed systems programming knowledge that is automatically applied. The designer can ignore many of these details. However, in a proof that Nuprl and ScoRes are correct, this knowledge must be made explicit. This has not yet been accomplished. Eventually, it could be done using formalizations of Java and of virtual machine models like JVM of the kind being formalized in Isabelle, HOL [GM93, NPW02, PN90, Pau88]. However, our focus is on the design and verification stage, and on the contributions possible at this level to computer science and to computing technology and software development.

Another aspect of our work that we only touch on briefly is the nature of formal interactive proof using the Nuprl 5 Logical Programming Environment. The entire theory of event structures on communication graphs has been formalized in Nuprl 5 by Mark Bickford and made available at the Nuprl web site www.nuprl.org. This theory contains over 2,500 definitions and theorems and is completely formally checked. It is a large knowledge base for understanding distributed computing at a fine level of detail.

The automated reasoning techniques implemented in the course of this formalization and supported by Stuart Allen and Richard Eaton, as well, represent a significant step in the implementation of the process of understanding distributed systems and designing protocols for communication, control, and security. This work is part of the long tradition begun by Newell, Simon, and Shaw [NSS57] of automating reasoning. Taken in its full extent, from pure mathematics to the verification of deployed systems, such work is one of the enduring contributions of computer science to intellectual history.
1.4 Formulas and problems
Here is how we interpret the statements of a typed predicate logic. For atomic predicates to assert or solve \( P(t_1, \ldots, t_n) \) means to provide a proof or a construction \( p(t_1, \ldots, t_n) \).

If \( P, Q \) are problem statements (predicate formulas), then to assert

\[
\begin{align*}
    P &\land Q \quad \text{means to find proofs or constructions} \ p \text{ and } q \text{ for } P, Q \text{ respectively.} \\
    P &\lor Q \quad \text{means to find a proof or construction} \ p \text{ for } P \text{ and mark it as applying to } \ P \text{ or to find a proof or construction} \ q \text{ for } Q \text{ and mark it as apply to } Q. \\
    P &\Rightarrow Q \quad \text{means to find an effective procedure} \ f \text{ that takes a proof or construction} \ p \text{ for } P \text{ and computes} \ f(p) \text{ a proof or construction for } Q. \\
    \neg P &\quad \text{means that there is no proof or construction for } P. \\
    \forall x:A.P &\quad \text{means that there is an effective procedure} \ f \text{ that takes any element of type } A, \text{ say } a, \text{ and computes a proof or construction} \ f(a) \text{ for } P[a/x]. \\
    \exists x:A.P &\quad \text{means that we can construct an object} \ a \text{ of type } A \text{ and find a proof or construction} \ p_a \text{ of } P[a/x], \text{ taken together, } <a, p_a> \text{ solves this problem or proves this formula.}
\end{align*}
\]

2 Event Systems

2.1 General
Our theory is designed to account for the behavior of a wide variety of systems, from interacting computers on the Internet to interacting components in a single computer or in a brain. It can also describe cause and effect behavior in physical systems on the scale of galaxies or subatomic particles. The right theory can be a unifying force in the study of computation in all its many forms. Our theory is another step toward a comprehensive account of distributed computing in its broadest sense. It is heavily influenced by the insights of Lamport[Lam78] and Winskel [Win80, Win89].

2.1.1 Events
Events are the atomic units of the theory. They are the occurrences of atomic actions in space/time. Although they have duration, we don’t speak of it, considering them to be instantaneous moments at which “things happen”. These events are causally ordered, \( e \) before \( e' \), denoted \( e < e' \). As Lamport postulated, causal order is the structure of time.
We abstract away the duration of an event, which would be related to the physical time that the action requires. The structure of event space is determined by the organization of events into discrete loci, each a separate locus of actions through time at which events are sequentially ordered. The entities (locations) are separate; for example, they do not share state, they can be distinguished by messages. All actions take place at these locations (or by these entities). Actions are “located at these entities”, and conversely, these entities are all (potentially) active. New entities can be created over time. At some locations, atomic actions produce random values. When seen as an entity, these loci can have properties such as physical coordinates. These are examples of observable properties of a locus of action.

2.1.2 Observables

We are interested in actions with observable results. Observables are known by identifiers and have types. For example, an observable might be a discrete value such as the spin of an electron, up or down; it might be the charge, positive or negative. We might observe the state of a device, on or off, or the values of a memory location, say an integer. The physical coordinates might be a quadruple of (computable) real numbers. The list of observables of an entity is its state.

Interaction among entities is determined by connections among them called communication links or interaction channels. These links form a discrete interaction topology. We allow that each entity is connected, perhaps by multiple links, to every other entity. The link structure can be dynamic.

Interaction is achieved by messages communicated on links. At each locus, every event can emit a signal (send a message). Sending a signal along a link to an entity will eventually cause that signal to be received by that entity, so the links are reliable, and reception cannot be blocked by the receiver. The action of detecting (or receiving) a signal is called an external event at the locus of reception. In addition, there can be internal events as the result of internal actions of the entity. All events are either external or internal, and either kind can emit a signal. The actions have names in the type Action.

Internal events can have preconditions or guards that determine the conditions under which they take place. The externally caused actions are not guarded; they happen whenever the signal arrives.

2.1.3 Computation and message automata

The universe is run by computation. It is the force that makes things happen. Computation is digital, built from discrete atomic actions. We can build the entire edifice on functional update of the state and of the message queues on the interaction links. The form of a state update is $s' := f(s, v)$ where $s$ is the current state, $v$ is a signal received or the value of an action and $s'$ is the new state. We take arbitrary computable functions $f$ as possible updating steps.

Ultimately we will describe the entities as automata, called message automata. Depending on the resolution at which we describe them, they can be as simple as atomic particles
or as complex as separate distributed systems, such as agents (human or robotic) or even large systems like a planet.

2.2 Event structures with order (EOrder)

It is possible to say a great deal without mentioning values, observables, and states; so we first axiomatize event structures with order but without values or states.

2.2.1 Signature of EOrder

The signature of these events requires two types, and two partial functions. The types are discrete, which means that their defining equalities are decidable. We assume the types are disjoint. We define $\mathbb{D}$ as $\{T : Type \mid \forall x, y : T. x = y \in T \lor \neg (x = y \in T)\}$, the large type of discrete types.

Events with order (EOrder)

$E : \mathbb{D}$

Loc:$\mathbb{D}$

pred?: $E \rightarrow E + \text{Loc}$

sender?: $E \rightarrow E + \text{Unit}$

The function $\text{pred?}$ finds the predecessor event of $e$ if $e$ is not the first event at a locus or it returns the location if $e$ is the first event. The $\text{sender?}(e)$ value is the event that sent $e$ if $e$ is a receive, otherwise it is a unit. We can define the location of an event by tracing back the predecessors until the value of $\text{pred}$ belongs to $\text{Loc}$. This is a kind of partial function on $E$. From $\text{pred?}$ and $\text{sender?}$ we can define these Boolean valued functions:

$$\text{first}(e) = \text{if is_left (pred?(e)) then true else false}$$

$$\text{rcv?}(e) = \text{if is_left (sender?(e)) then true else false}$$

The relation $\text{is_left}$ applies to any disjoint union type $A + B$ and decides whether an element is in the left or right disjunct (see Naive Computational Type Theory [Con02]). We can “squeeze” considerable information out of the two functions $\text{pred?}$ and $\text{sender?}$. In addition to $\text{first}$ and $\text{rcv?}$, we can define the order relation

$$\text{pred!}(e, e') == (\neg \text{first}(e') \Rightarrow e = \text{pred?}(e')) \lor e = \text{sender}(e').$$

We will axiomatize this as a strongly well-founded order relation.

The transitive closure of $\text{pred!}$ is Lamport’s causal order relation denoted $e < e'$. We can prove that it is also strongly well-founded and decidable; first we define it.

The $n$th power of relation $R$ on type $T$, is defined as
The transitive closure of $R$ is defined as $xR^*y$ iff \( \exists n : \mathbb{N}^+ \cdot (xR^ny) \).

Causal order is $x \text{ pred}!*y$, abbreviated $x < y$.

### 2.2.2 Axioms for event structures with order (EOrder)

There are only three axioms that constrain event systems with order beyond the typing constraints.

**Axiom 1** If event $e$ emits a signal, then there is an event $e'$ such that for any event $e''$ which receives this signal, $e'' = e'$ or $e'' < e'$.

\[ \forall e : \mathcal{E}. \exists e' : \mathcal{E}. \forall e'' : \mathcal{E}. \left( \text{rcv}?(e'') \land \text{sender}?(e'') = e \Rightarrow (e'' = e' \lor e'' < e) \right) \]

**Axiom 2** The $\text{pred?}$ function is injective.

\[ \forall e, e' : \mathcal{E}. \text{loc}(e) = \text{loc}(e') \Rightarrow \text{pred?}(e) = \text{pred?}(e') \Rightarrow e = e' \]

**Axiom 3** The $\text{pred!}$ relation is strongly well founded.

\[ \exists f : \mathcal{E} \rightarrow \mathbb{N}. \forall e, e' : \mathcal{E}. \text{pred!}(e, e') \Rightarrow f(e) < f(e') \]

To define $f$ in Axiom 3 we arrange a linear "tour" of the event space. We can imagine that space as a subset of $\mathbb{N} \times \mathbb{N}$ where $\mathbb{N}$ numbers the locations and discrete time. Events happen as we examine them on this tour, so a receive can't happen until we activate the send. Local actions are linearly ordered at each location. Note, we need not make any further assumptions.

We can define the finite list of events before a given event at a location, namely

\[ \text{before}(e) = \text{if first}(e) \text{ then} [] \text{ else } \text{pred?}(e) \text{ append before (pred?}(e)) \]

Similarly, we can define the finite tree of all events causally before $e$, namely

\[ \text{prior}(e) = \text{if first}(e) \text{ then} [] \text{ else if rcv?}(e) \text{ then } < e, \text{prior(sender?}(e)), \text{prior}(\text{pred?}(e)) > \text{ else } < e, \text{prior}(\text{pred?}(e)) > \]
2.2.3 Properties of events with order

We can prove many interesting facts about events with order. The basis for many of the proofs is induction over causal order. We prove this by first demonstrating that causal order is strongly well founded.

**Theorem 1** \( \exists f : \mathcal{E} \to \mathbb{N} \). \forall e, e' : \mathcal{E}. \ e < e' \Rightarrow f(e) < f(e')

The argument is simple. Let \( x < y \) denote \( \text{pred!}(x, y) \) and let \( x <^n y \) denote \( \text{pred}^n(x, y) \). Recall that \( x <^{n+1} y \) if \( \exists z : \mathcal{E}. \ x < z \& z <^n y \). From Axiom 3 there is function \( f_o : \mathcal{E} \to \mathbb{N} \) such that \( x < y \) implies \( f_o(x) < f_o(z) \). By induction on \( \mathbb{N} \) we know that \( f_o(z) < f_o(y) \). From this we have \( f_o(x) < f_o(y) \). So the function \( f_o \) satisfies the theorem. The simple picture of the argument is

\[
\begin{align*}
 x < z_1 < z_2 < \ldots < z_n < y
\end{align*}
\]

so

\[
\begin{align*}
f_o(x) < f_o(z_1) < \ldots < f_o(z_n) < f_o(y).
\end{align*}
\]

We leave the proof of the following induction principle to the reader.

**Theorem 2** \( \forall P : \mathcal{E} \to \text{Prop.} \forall e' : \mathcal{E}. \ ((\forall e : \mathcal{E}. \ e < e'. \ P(e)) \Rightarrow P(e')) \Rightarrow \forall e : \mathcal{E}. \ P(e) \)

Using induction we can prove that causal order is decidable.

**Theorem 3** \( \forall e, e' : \mathcal{E}. \ e < e' \lor \neg (e < e') \)

We need the lemma.

**Theorem 4** \( \forall e, e' : \mathcal{E}. \ (e < e' \lor \neg (e < e')) \)

This is trivial from the fact that \( \text{pred!}(x, y) \) is defined using a decidable disjunction of decidable relations, recall

\[
\begin{align*}
x < y \text{ is } \text{pred!}(x, z)
\end{align*}
\]

and

\[
\begin{align*}
\text{pred!}(x, y) = \neg \text{first}(y) \Rightarrow x = \text{pred?}(y) \lor x = \text{sender?}(y).
\end{align*}
\]

The local order given by \( \text{pred?} \) is a total order. Define \( x <_{\text{loc}} y \) is \( x = \text{pred?}(y) \).

**Theorem 5** \( \forall x, y : \mathcal{E}. \ (x <_{\text{loc}} y \lor x = y \lor y <_{\text{loc}} x) \)

8
References


Michel Charpentier and K. Mani Chandy. Towards a compositional approach to the design and verification of distributed systems. In Jeannette Wing, Jim


[Kre99] Christoph Kreitz. Automated fast-track reconfiguration of group communication systems. In Cleaveland [Cle99], pages 104–118.


