A Causal Logic of Events in Formalized Computational Type Theory *

Mark Bickford[a] and Robert L. Constable[b]

[a] *ATC, NY*
[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,

*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,

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

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** $P(t_1, \cdots, t_n)$ means to provide a proof or a construction $p(t_1, \cdots, t_n)$.

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

$P \,\&\, Q$ means to find **proofs** or **constructions** p and q for P, Q respectively.

$P \vee Q$ means to find a proof or construction p for P and **mark it** as applying to P or to find a proof or construction q for Q and mark it as apply to Q.

$P \Rightarrow Q$ means to find an **effective procedure** f that takes a proof or construction p for P and computes $f(p)$ a proof or construction for Q.

$\neg P$ means that there is no proof or construction for P.

$\forall x\!:\!A.P$ means that there is an **effective procedure** f that takes any element of type $A,$ say $a,$ and computes a proof or construction $f(a)$ for $P[a/x]$.

$\exists x\!:\!A.P$ means that **we can construct an object** a of type A and find a proof or construction p_a of $P[a/x],$ taken together, $< a, p_a >$ solves this problem or proves this formula.

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 $\{T : Type \mid \forall x, y : T.\ x = y \text{ in } T \vee \neg\ (x = y \text{ in } T)\}$, the large type of discrete types.

Events with order (EOrder)

E: $\mathbb{D}$
Loc:$\mathbb{D}$
pred?: $E \to E + Loc$
sender?: $E \to E + Unit$

The function *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 *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 *pred* belongs to *Loc*. This is a kind of partial function on E. From *pred? and sender?* we can define these Boolean valued functions:

$$first(e) = \textit{if is\_left } (pred?(e)) \textit{ then true else false}$$
$$rcv?(e) = \textit{if is\_left } (sender?(e)) \textit{ then true else false}$$

The relation *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 *pred?* and *sender?*. In addition to *first* and *rcv?*, we can define the order relation

$$pred!(e, e') == (\neg\ first(e') \Rightarrow e = pred?(e')) \vee e = sender(e').$$

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

The transitive closure of *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 nth power of relation R on type T, is defined as

$xR^o y$ iff $x = y$ *in* T
$xR^n y$ iff $\exists z : T.\ xRz\ \&\ zR^{n-1}y$

The *transitive closure* of R is defined as xR^*y iff $\exists n : \mathbb{N}^{+\cdot}(xR^n y)$.

Causal order is x *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 $e'' < e'$.*

$$\forall e : E.\exists e' : E.\ \forall e'' : E.\ (\ rcv?(e'')\ \&\ sender?(e'') = e) \Rightarrow (e'' = e' \vee e'' < e)$$

Axiom 2 *The pred? function is injective.*

$$\forall e, e' : E.\ loc(e) = loc(e') \Rightarrow pred?(e) = pred?(e') \Rightarrow e = e'$$

Axiom 3 *The pred! relation is strongly well founded.*

$$\exists f : E \to \mathbb{N}.\ \forall e, e' : E.pred!(e, e') \Rightarrow f(e) < f(e')$$

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

before$(e) ==$ if first(e) then[]
else *pred*?(e) append *before* (*pred*? (e))

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

prior$(e) ==$ if first(e) then[]
else if *rcv*?(e)
then $< e, prior(sender?(e)), prior(pred?(e)) >$
else $< e, prior(pred?(e)) >$

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 $\exists f : E \to \mathbb{N}.\, \forall e, e' : E.\;\; e < e' \Rightarrow f(e) < f(e')$

The argument is simple. Let $x \lhd y$ denote $pred!(x, y)$ and let $x \lhd^n y$ denote $pred!^n(x, y)$. Recall that $x \lhd^{n+1} y$ iff $\exists z : E.\, x \lhd z \;\&\; z \lhd^n y$. From Axiom 3 there is function $f_o : E \to \mathbb{N}$ such that $x \lhd y$ 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

$$x \lhd z_1 \lhd z_2 \lhd \ldots \lhd z_n \lhd y$$

so

$$f_o(x) < f_o(z_1) < \ldots < f_o(z_n) < f_o(y).$$

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

Theorem 2 $\forall P : E \to Prop.\, \forall e' : E.\, ((\forall e : E.\, e < e'.\, P(e)) \Rightarrow P(e')) \Rightarrow \forall e : E.\, P(e)$

Using induction we can prove that causal order is decidable.

Theorem 3 $\forall e, e' : E.\, e < e' \vee \neg\, (e < e')$

We need the lemma.

Theorem 4 $\forall e, e' : E.\, (e \lhd e' \vee \neg\, (e \lhd e'))$

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

$$x \lhd y \text{ is } pred!(x, z)$$

and

$$pred!(x, y) = \neg\, first(y) \Rightarrow x = pred?(y) \vee x = sender?(y).$$

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

Theorem 5 $\forall x, y : E.\;\; (x <_{loc} y \vee x = y \vee y <_{loc} x)$

2.2.4 Specifications using events with order

The language of events with order allows us to describe interesting event structures that we observe in nature or which have been created or which we wish to create. For example, when we observe two processes communicating, say S and R, we might see an alternating causal sequence of sends and receives which we would describe as a "two way alternating communication" of the form

$$s_1 < r_2 < s_3 < r_4 < ...$$

where s_1 is a send from S to R, r_{i+1} is a receive with sender s_i, and s_{i+2} is a send from R to S with r_{i+3} is a receive at S. We notice that such a sequence would produce a one-one correspondence between sends and receives, i.e.,

$$\forall e@S.\exists e'@R.\ sender(e') = e \text{ and}$$

$$\forall e@R.\exists e'@S.\ sender(e') = e.$$

The quantifiers $\forall e@i, \exists e@i$ are defined as follows:

$$\forall e@i.P == \forall e : E.\ (loc(e) = i \Rightarrow P)$$

$$\exists e@i.P == \exists e : E.\ (loc(e) = i \Rightarrow P).$$

Another way to describe a two way alternating communication between S and R is using quantifiers to directly translate informal descriptions such as this:

(i) every send s from S to R

(ii) is followed by a signal r from R to S

(iii) that must arrive at S before another message is sent from S to R.

Clause (i) is expressed as

$$\forall s@S.\ s \text{ sends to } R, \text{ that is}$$

$$\forall s@S.\ \exists r'@R.\ s = sender(r')$$

Clause (ii) is expressed as

$$\exists r@S.\ (s < r\ \&\ \exists s''@R.\ (r' \leq s''\ \&\ sender(r) = s''))$$

and clause (iii) is

$$\forall x@S.\ s < x < r \Rightarrow x \text{ is not a send to } R, \text{ i.e.,}$$

$$\forall x@S.\ s < x < r \Rightarrow \neg\ \exists x'@R.\ (x = sender(x')).$$

2.3 Event structures with value (EValue)

2.3.1 Signature of EValue

External events have values, namely the signal detected at the point of reception. This value is related to the signal emitted by the sender. In the distributed communication model, to be defined later, the signal detected will be the *message* sent by the sender over a communication link. The value of an external event is part of the signal. We also want to assign a value to internal events in a way that discriminates more finely than just being internal, otherwise all internal events would have the same value. To this end we subdivide internal actions further by giving them a name from a discrete type called Act. Thus the *kind* of an event can now be seen as an element of this type

$$kind == Act + Top$$

The left disjunct are the *internal actions* and the right the *external.* We add a function

$$kind : E \to Act + Top$$

and we add this typing function

$$Ty : Loc \to kind \to Type$$

$$val : E \to Ty(loc(e), kind(e)).$$

For internal events whose action is a in Act, the value can be any element of $Ty(e, a)$, perhaps chosen nondeterministically or randomly.

2.3.2 Sample specifications

We illustrate the new expressive power of EValue by describing *functionality* and *interaction.* Note, relations on a type A are defined in computational type theory as functions from A into the type of Propositions. Because the theory is *predicative*, these types are stratified by level and denoted $Prop_i$. For simplicity, when the index does not figure in the discussion, we write $Prop$.

Similarly, the universe of all types is stratified into levels, $Type_i$. We suppress the level index when it is not significant. In much of our writing, these universes of types are denoted $\mathbb{U}_i$ instead of $Type_i$.

1. Functionality
The atomic actions of an event structure can be the computation of an arbitrary computable function. For instance, there is an event structure es which takes inputs from an arbitrary type A and computes $f(x)$ for any function $f : A \to B$. Given $A, B,$ and$f$, the basic elements are a location, say R, internal actions a at R and receive events that accept values of type A. The specification we want is

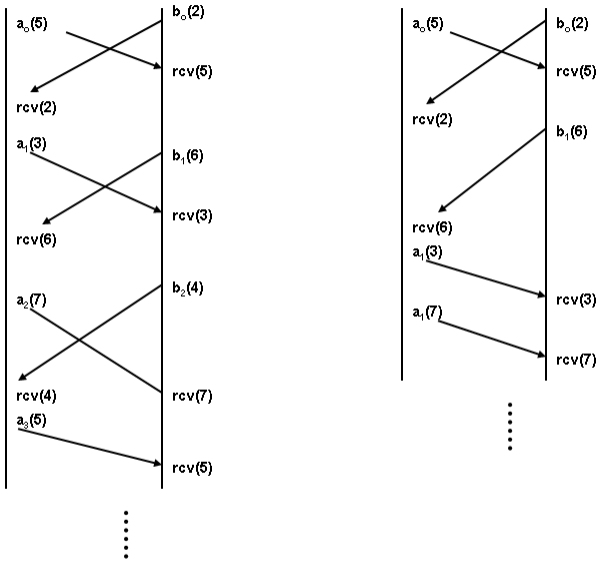
$$\forall A, B : Type. \forall f : A \to B.\ \forall x : A.$$

$$\forall r@R.rcv?(r)\ \&\ val(r) = x \Rightarrow \exists e'@R.\ kind(e') = a\ \&\ val(e') = f(val(r)).$$

To make the typing work, we need that the type of e is A and of e' is B. The latter is accomplished by having $Ty(R, a) = B$. The type of x is determined by the value of the sender, and an event structure requires that $sender?(r)$ is defined. We can do this by creating an event $e(x)$ of a kind s, the sender of r and stipulating that $val(e(x)) = x, T(loc(e(x)), s) = A$. This is a particular event structure depending on $x \in A$. We could make this more general by allowing a sequence of events $e(x_1), e(x_2), ...$ for each $x_i \in A$. This would generate a sequence of receives and actions $r_1, a_1, r_2, a_2, \ldots$.

2. Interaction

One of the simplest examples of concurrent computation is an interaction between two processes, say A and B. We look at the case where A and B send each other natural number values, and send back a value one larger. In this case, each process eventually receives a value larger than its initial value. If the actions continue indefinitely, each process will receive arbitrarily large values. Here are two possible interactions, each creating an event structure. The diagrams are called *message sequence diagrams.*



2.4 Events with State (EState)

2.4.1 Signature of EState

At every location, the value of an event e can depend on the entire history of internal and external events that precede e. The efficient way to organize action based on cumulative

history is to introduce memory in the form of a state. We add to events with values, the notion of state and three relations on state. To do this we need another discrete type, the names of the state variables. These are called *identifiers*, Id, in the nomenclature of programming. Each identifier can hold a value, and that value has a type; so we also need a function T from identifiers to types.

$$Id : \mathbb{D}$$

$$T : x : Id \to i : Loc \to Type.$$

The new relations are

initially: $x : Id \to i : Loc \to T(x, i)$
when: $x : Id \to e : E \to T(x, loc(e))$
after: $x : Id \to e : E \to T(x, loc(e))$

At each location i, the *state* is the map $Id \to T(x, i)$. Only finitely many of the identifiers map to a type which is not Top.

2.4.2 Axiom of EState

Axiom 4 *For any event except the first, the value of any observable when e is the value after the predecessor of e.*

$$\forall e : E.\neg\ first(e) \Rightarrow (x\ \textbf{\textit{when}}\ e) = (x\ \textbf{\textit{after}}\ pred(e))$$

2.4.3 Change operator

Definition:

$$\begin{array}{rcl} x\ \Delta\ e & = & (x\ \textbf{after}\ e \neq x\ \textbf{when}\ e) \\ \Delta(x, e) & = & \|[e_1 \in\ \textbf{before}(e) \mid x\ \Delta\ e_1]\| \\ & & \text{(only defined when } T(loc(e), x) \text{ has a decidable equality)} \\ x\ \Delta_n\ e & = & 0 < n\ \wedge\ \Delta(x, e) = n - 1\ \wedge\ x\ \Delta\ e \end{array}$$

The formula $x\ \Delta\ e$ is true when event e makes a change in state variable x. The formula $\Delta(x, e) = n$ is true when there have been exactly n changes to x strictly before event e. The formula $x\ \Delta_n\ e$ is true when there have been exactly n changes to x up to and including event e and one of the changes is at e.

Properties of $\triangle$:
Suppose that, $x \in X$, $i \in Loc$, and $T(i, x) \in \mathbb{D}$, i.e. the type of x at location i has decidable

equality. Then,

(1) $\forall e@i.\ \forall n : \mathbb{N}.$
$x\ \Delta\ e,\ \Delta(x,e) = n,$ and $x\ \Delta_n\ e$ are decidable

(2) $\forall e@i.\ e\ <_{loc}\ e'\ \Rightarrow\ \Delta(x,e) \leq \Delta(x,e')$

(3) $\forall e'@i.\ \Delta(x,e') = n\ \wedge\ e = pred(e')\ \Rightarrow$
$\Delta(x,e) = n\ \vee\ x\ \Delta_n\ e$

(4) $\forall e@i.\ \Delta(x,e) = 0\ \Rightarrow\ x$ **when** $e = x$ **initially** i

(5) $\forall e'@i.\ x$ **when** $e' \neq x$ **initially** $i\ \Rightarrow\ \exists e <_{loc} e'.\ x\ \Delta\ e$

Proof:

1. If $loc(e) = i$, then x **when** e and x **after** e have type $T(i,x)$ so if equality in $T(i,x)$ is decidable, then $x\ \Delta\ e$ is decidable. We can then prove that the other predicates, $\Delta(x,e) = n$ and $x\ \Delta_n\ e$ are decidable, by induction on $<_{loc}$. Essentially, they are defined by bounded quantification over the predecessors of e from the decidable $x\ \Delta\ e$.

2. follows by induction on $<_{loc}$.

3. Under the hypotheses,
$$n = \Delta(x,e') = \Delta(x,e) + \textbf{ if } x\ \Delta\ e \textbf{ then } 1 \textbf{ else } 0.$$
If $x\ \Delta\ e$ then $x\ \Delta_n\ e$ and otherwise $\Delta(x,e) = n$.

4. is proved by induction on $<_{loc}$. If e has no predecessors then x **when** $e' = x$ **initially** i so the assertion is true. If $e_1 = pred(e)$ and $\Delta(x,e) = 0$ then, by lemma 3, $\Delta(x,e_1) = 0$, so, by induction, x **when** $e_1 = x$ **initially** i. Also, $\neg(x\ \Delta\ e_1)$, so x **when** $e_1 = x$ **after** $e_1 = x$ **when** e, and hence x **when** $e = x$ **initially** i.

5. Under the decidability assumption, $\exists e <_{loc} e'.\ x\ \Delta\ e$ is decidable. If it is true then the assertion is true. If it is false, then $\Delta(x,e') = 0$, so by lemma 4, x **when** $e = x$ **initially** i, which contradicts the hypothesis.

Qed.

2.5 Event system communication typology (ECom)

Some specifications of computing systems depend on the underlying communication typology. We see this in consensus algorithms which work over rings or trees or graphs. The general structure we imagine is a directed graph whose nodes are the locations or agents. A link from i to j is a labeled pair $<i,j>$. The label allows us to distinguish among several possible communication channels from i to j. To include the topology we add a discrete type of *Links*. Each link has a *source* and a *destination* which are locations.

Link :$\mathbb{D}$
src :$Link \ \rightarrow \ Loc$
dst :$Link \ \rightarrow \ Loc$

Using links, it is possible to refine the emission and reception of signals. We say that a message m of $Type\, T$ is sent on a link l. Moreover, we allow that a location can emit more than one message on a link, and to identify them, they are tagged from the discrete type Tag. So when a message is received we can prescribe the link and tag on which we discriminate, i.e., $rcv(l,t)(e)$ means that event e is a receive on link l of a messages with tag t. We need a typing function for tagged messages on a link

$$M : Link \rightarrow Tag \rightarrow Type.$$

We require links to deliver messages in FIFO (first in first out) order, that is if e_1 and e_2 both receive messages on link l, then if $sender?(e_1) < sender?(e_2),$ then $e_1 < e_2$.

Axiom 5

$$\forall e_1, e_2 : E.rcv(l,t)(e_1)\ \&\ rcv(l,t)(e_2) \Rightarrow$$
$$sender?(e_1) < sender?(e_2) \Rightarrow e_1 < e_2.$$

2.6 Event systems with time (ETime)

To reason about realtime properties of distributed systems and embedded systems, we add to the event structure a map

$$time \ : \ E \rightarrow Time$$

(where $Time$ is currently the rational numbers) and the following axiom

$$e_1 \ \prec \ e_2 \ \Rightarrow \ time(e_1) \leq time(e_2)$$

In the realtime event structures, the state variables are all implicitly functions of time. The realtime event structure introduces another primitive component:

$$discrete \ : \ Id \rightarrow Id \rightarrow \mathbb{B}$$

If $discrete(i,x)$ then the state variable x at location i is constrained so that events can only change its value from one constant to another (so its behavior over time will be a step funtion). For discrete state variables the axiom

$$\neg first(e) \ \Rightarrow \ x \ \textbf{when} \ e = x \ \textbf{after} \ pred(e)$$

still holds but for general state variables the axiom becomes

$$\neg first(e) \ \Rightarrow \ \forall t \geq 0.\ (x \ \textbf{when} \ e)(t) = (x \ \textbf{after} \ pred(e))(t + time(pred(e)) - time(e))$$

As an example of the expressive power of this language, to specify the occurence of regularly spaced ticks at location i, we could write

$$\forall e, e'@i.\ \text{consecutive}(tick, e, e') \ \Rightarrow\ time(e') - time(e) = D \pm \epsilon$$

where

$$\begin{aligned}\text{consecutive}(tick, e, e') \ \equiv\ & kind(e) = kind(e') = tick \ \wedge\ e \prec e' \\ & \wedge\ \forall e''.\ e \prec e'' \prec e' \ \Rightarrow\ \neg kind(e'') = tick\end{aligned}$$

2.7 Event systems with transition function (ETrans)

For applications to security, it is useful to refine the event model further and specify that state changes are given by an explicit transition function. In the formal Nuprl definition this is the element *Trans*, the tenth element of the following sixteen couples. This is the version used in our work on security.

EventsWithState
$\equiv_{\text{def}}$ E:Type
$\times$EqDecider(E)
$\times$*pred*?:($E\rightarrow$(E+Unit))
$\times$*info*:($E\rightarrow$(Id$\times$Id+(IdLnk$\times E$)$\times$Id))
$\times$EOrderAxioms(E; *pred*?; *info*)
$\times T$:(Id$\rightarrow$Id$\rightarrow$Type)
$\times V$:(Id$\rightarrow$Id$\rightarrow$Type)
$\times M$:(IdLnk$\rightarrow$Id$\rightarrow$Type)
$\times$*init*:(i,x:Id$\rightarrow T(i,x)$)
$\times$*Trans*:(i:Id$\rightarrow k$:Knd$\rightarrow$kindcase(k; $a.V(i,a)$; l,$t.M(l,t)$)$\rightarrow$(x:Id$\rightarrow T(i,x)$)$\rightarrow$
(x:Id$\rightarrow T(i,x)$))
$\times$*val*:(e:$E\rightarrow$kindcase(kind(e); $a.V$(loc(e),a); l,$t.M(l,t)$))
$\times$*Send*:(i:Id$\rightarrow k$:Knd$\rightarrow$kindcase(k; $a.V(i,a)$; l,$t.M(l,t)$)$\rightarrow$(x:Id$\rightarrow T(i,x)$)$\rightarrow$
(Msg(M) List))
$\times$*Choose*:(i,a:Id$\rightarrow$(x:Id$\rightarrow T(i,x)$)$\rightarrow V(i,a)$+Unit))
$\times$val-axiom(E;V;M;*info*;*pred*?;
init;*Trans*;*Choose*;
Send;*val*)
$\times$(($\forall i$, x:Id. AtomFree(Type;$T(i,x)$))
& ($\forall i$, a:Id. AtomFree(Type;$V(i,a)$))
& ($\forall l$:IdLnk, tg:Id. AtomFree(Type;$M(l,tg)$)))
$\times$Top

3 Computational Models

3.1 A network model of distributed computing

One kind of computing model whose features are expressed naturally in event systems is a network of machines acting asynchronously. Another might be a computational model of the universe, a model we won't explicitly consider. As for the network model, it will prove to be a good basis for our work on process synthesis.

The nodes of the network are processes, each with a unique name; the type of names is discrete. So the type of names is like that of locations, and we adopt a specific countable instance of *Loc*. The canonical names of elements are $loc\{i:nat\}$ for *nat* the intuitive natural numbers.

There are directed links between any two processes (locations), each link has a unique name from the discrete type *Links*. The canonical names are $link\{l:nat\}$. With each link l is associated its *source*, $src(l)$ and *destination*, $dst(l)$, which are locations.

Processes communicate by sending tagged messages on a link. The tags are from the discrete type Tag whose canonical elements are $tag\{t:nat\}$. The use of a tag depends on the link. With each link l and tag t we associated the type of message being sent $msg\ type(l,t)$.

Our model is abstract in that elements of any type T can be sent as message values, e.g., we can send natural numbers ($\mathbb{N}$) or real numbers ($\mathbb{R}$) or functions from reals to reals ($\mathbb{R} \to \mathbb{R}$) or even types ($Type$), etc.

We assume that communication links are *reliable*, i.e., all messages sent on l are eventually received at $dst(l)$, and they are *FIFO*, i.e., the messages are sent in the order received. During a computation, the state of the link is given by a queue of messages that are in transit. As they are received, they are removed from the head of the queue, and as new messages are generated, they join the tail of it.

Each process computes by receiving messages, sending messages, updating its state, and choosing random values. The state is a mapping of identifiers, Id, to values. The values are types as in the definition of $EState$. We use the same type of identifiers at each location.

The atomic steps of computation are either receives of messages or local actions. Each can also send a list of tagged messages on one link. For ease of description, each internal action has a unique name from the type *Act* of actions whose canonical names are $act\{i:nat\}$. An action might be guarded by a decidable predicate P of the state.

Although computation is asynchronous and there is no global clock, for the sake of the mathematical description, we assume discrete time measured by natural numbers. Thus time is discrete and linearly ordered. It would be possible to measure time using the rationale, $\mathbb{Q}$, which are discrete and densely ordered, but we take the simpler approach of using $\mathbb{N}$. At any point of discrete time t, a process can perform an action or do nothing.

The global state of the computation at time t consists of the states at each location, $s(i,t)$, and the message queues at each link, $msgs(l,t)$. The next state is given by the action to be taken next at each location, $act(i,t)$, including a possible null action, which we take to be $act\{0:nat\}$ by convention.

We will arrange that computation is *fair* in that messages are eventually delivered. Just

below we will refine fairness such that if there is a choice of multiple actions that are able to execute because their guards are true, then eventually the action will be taken or the guard will become false. This notion of fairness is constructive in that a global schedule for actions will predict when an action must be taken if the guard remains true. In any real network the schedules and delivery protocols will ensure this.

We consider this model to be *universal*. In it we can embed any distributed system, from the global internet to any local area network, or a multiprocessor piece of hardware, or even a totally isolated sequential machine computing a specific function.

On the discrete view of physics [Hey02, Whe82, Whe89], the progression of time could be in Planck units.

3.2 Signature of the Network Model

The signature of the *network computing model* is

$$
\begin{array}{l}
Loc : \mathbb{D} \times Link : \mathbb{D} \times src, dst : Link \to Loc \\
\times Id : \mathbb{D} \times Act : \mathbb{D} \times Tag : \mathbb{D} \\
T : Loc \to (Id \to Type) \\
TA : Loc \to Act \to Type \\
M : Link \to Tag \to Type \\
msgs : Loc \to Link \to \mathbb{N} \to List(Msg(Link, Tag, M)) \\
s : i : Loc \to \mathbb{N} \to State(Id, T(i)) \\
a : i : Loc \to \mathbb{N} \to Action(Act, Link, Tag, kindcase(TA(i), M))
\end{array}
$$

The elements of this type are *computations* (or sometimes *worlds*) of the model. We can state our assumptions on the model in terms of these elements. We denote a computation by w.

1. Process i can only send messages on links whose source is i.

$$\forall i : Loc.\, \forall t : \mathbb{N}.\, \forall l : link.src(l) \neq i \Rightarrow msgs(i, l, t) = nil$$

2. A *null action* does not sent or receive messages or change the state.

$$\forall i : Loc.\, \forall t : \mathbb{N}.\, \text{is null } (a(i, t)) \Rightarrow s(i, t + 1) = s(i, t)\ \&$$

$$msgs(i, l, t) = msgs(i, l, t + 1).$$

3. A receive action at i must be on a link whose destination is i and whose value is the head of the message queue.

$$\forall i : Loc.\, \forall t : \mathbb{N}.\, \forall t : Tag.\ isrcv(l, t, act(i, t)) \Rightarrow$$

$$dst(l) = i\ \&\ val(act(i, t)) = hd(msgs(i, l, t))$$

4. The final assumption is *fairness of delivery*; it says that for every message queue that is not empty, the message at the head of the queue is delivered and removed from the queue.

$$\forall l : Link.\, \forall t : \mathbb{N}.\, msgs(src(l), l, t) \neq nil \Rightarrow$$
$$\exists t' : \mathbb{N}.\, msgs(src(l), l, t') = tl(msgs(src(l), l, t)).$$

3.3 Distributed systems

In a realistic network, each process is running a finite program; so there are only finitely many possible internal atomic actions on the state. We might as well classify them as *receives* of tagged messages on a link, and *internal actions*. The internal actions can be given unique names, say $a_1, ..., a_n$. These names are the *kinds* of internal actions.

Each action can update the state by a computable function f of the state and the value of the action. An external action that receives a message value m tagged by t can update the state by a function $f(s, m)$. An internal action a with value v can update by $f(s, v)$.

Moreover, some internal actions might be guarded by a decidable predicate called the *precondition* for the action. If the precondition is true, the action can be taken, otherwise it cannot. At any moment, several actions might be *enabled* by having true guards. In our model, receive actions are always enabled.

If several actions are enabled, at most one can be taken (possibly none). We imagine that the execution is *fair* in the sense that if a guard remains true long enough, its action will be taken. More precisely, if the guard is true at t and remains true until a time $sch(t)$ determined by a schedule, then the action will be taken.

In this distributed system model, we will also imagine that each process can stipulate initial values of its state and of its first action.

3.4 Realizers

We have said enough about the behavior of processes that we can isolate the atomic actions.

1. Set the state to some initial value s_0.
2. Check whether precondition p of the state is true, if so, possibly select a random value v and update the state by $f(s, v)$.
3. Receive a tagged value v on a link and update the state by $f(s, v)$.
4. Send a list of tagged messages on one specific link l $send(l, [< t_1, m_1 >, ..., < t_n, m_n >])$.

Below we will adopt a rudimentary syntax for these action schemes and call them *realizers*.

3.5 Frame Conditions

One humorous definition of distributed computing is that it allows a computer you never knew existed to crash yours. In order to reason about a process in a distributed system, it must be possible to limit the affect of other processes on it. This issue is more acute in a setting where processes are built from components or where new actions are created at a location, say by adding a new layer to a protocol stack. For example, a process might be incrementing a counter x_0 upon each arrival of a message on link l, assuming that no other action affects x_0. If a later piece of code is installed that resets x_0 to 0, it *interferes* with our implicit assumptions about x_0.

To prevent unexpected or unwanted interactions, we will include in the programming notation for processes statements of the form

$$\textit{only actions in list } L \textit{ affect } x \textit{ and}$$

$$\textit{only actions in list } L \textit{ send on } e$$

for L a list of actions and x a state variable (identifiers). We call these *frame conditions*.

3.6 Compatibility and Composition

Frame conditions place constraints on how we combine realizers. If action a changes x and frame conditions c prevents a from changing x, then these two realizers are incompatible. Processes are built by combining realizers. The composition operation, $\oplus$, is very simple, namely $R_1 \oplus R_2$ is the union of the actions of R_1 and R_2.

3.7 Message Automata

Here we provide a specific syntax for the atomic clauses and their composition.

1. Initial clauses
   $@i$ *state* $x : T$ *initially* $P(x)$.

2. Guarded actions (effects)
   $@i$ *action* $k(v)$: T_1; *state* $x : T_2$;
   precondition $P(s, v)$
   effect $x := f(s, v)$.

3. Receive actions
   $@i$ *action* $rcv_l\ (v) : T_1$; *state* $x : T_2$
   effect $x := f(s, v)$.

4. Send Clause
   $@i$ *sends on link* $l\ [< t_1,\ f_1\ (s, v)\ >, ...,\ < t_n,\ f_n\ (s, v) >]$.

5. Frame clause
   $@i$ *only* L *affects* x.

6. Send frame clause
   $@i$ *only* L *sends* $< l, tg >$.

4 A Logic of Events

4.1 A Logical Perspective

In Sections 1 and 2, we motivated our selection of the conceptual primitives that describe a wide range of computational phenomena that arise when processes interact concurrently and asynchronously. In Section 3, we looked at the computational model in sufficient detail to distinguish six kinds of clauses that are atomic primitives from which many fundamental processes can be constructed. We considered issues of logical reasoning only in a passing manner. Here we explore the logical issues in enough detail to provide the means to prove interesting assertions in a logic of events whose primitives are those defined in Section 2, the axioms, and section three, the possible constructions. In a sense, Section 2 is analogous to the Euclidean postulates and Section 3 to the ruler and compass constructions. We will start by looking at logical descriptions of the constructions. These are like Euclid's assertions of the form "given two distinct points, we can draw exactly one and only one straight line between them".

4.2 Realizable Event Specifications

For each of the six atomic *process construction schemes* of section 3.7, there is an *event specification scheme* that describes it. These six specification schemes along with the event system axioms constitute the logical bases for reasoning. All other other reasoning matters are general and are richly provided in the Nuprl, MetaPRL, and Coq implementations of constructive typed logic.

1. **Initial Clause**
   $@i$ p $(x$ *initially* $i)$

   realizes

   $\forall e@i.\ first(e)\ \Rightarrow\ p(x$ **when** $e).$

2. **Guarded actions (effects)**
   $@i$ *action* $k(v) : T_1$ *state* $x : T_2$
   precondition $p(s, v)$; *effect* $x := f(s, v)$.

   realizes

   $\forall e@i.\ kind\ (e)\ =\ k\ \Rightarrow\ (p(s$ **when** $e,\ val(e))\ \&$
   x **after** $e\ = f(s, val(e))\ \&$
   $\forall\ e@i.\ \exists e'@i.\ e\ \leq_{loc} e'\ \&\ (kind(e')\ =\ k \vee\ \neg\ p(s,\ $ **after** $e',\ val(e')))\ \&$
   $\exists v : T_1.\ p(s\ initial\ i, v)\ \Rightarrow\ \exists e : E.\ loc(e)\ =\ i.$

3. **Receive actions**
   $@i$ *action* $rcv_l(v) : T_1$; *state* $x : T_2$
   effect $x := f(s, v)$

   realizes

   $\forall v : T_1.\ \forall l : link\ \forall e@i.\ kind(e)\ =\ rcv(l, v)\ \Rightarrow\ x$ **after** $e\ =\ f(s, v)$.

4. **Sends Clause (send one message version)**
   $@i$ *action* $k(v)$ $v : T$ *sends on* l, $< tag,\ f(s$ **when** $e, v) >$

   realizes

   $\forall e@i.\ kind(e)\ =\ k\ \Rightarrow\ \exists e'@dst(l).\ kind(e')\ =\ rcv(l, tag)$
   $\&\ sender(e') = e\ \&\ val(e')\ =\ f(s$ **when** $e,\ val(e))$.

   The action which sends can be a receive or a guarded effect. We let k be either kind and let the value be v of type T. The full event specification says more about typing information which we discuss later.

5. **Frame Clause**
   $@i$ *only* L *affects* x *for* L *a list of actions*

   realizes

   $\forall e@i\ (kind(e)\ \notin\ L\ \Rightarrow\ \neg\ (x \triangle e))\ \&\ (x \triangle e\ \Rightarrow\ kind(e) \in L)$

6. **Sends Frame Clause**
   $@i$ *only* L *sends* $< l, tg >$

   realizes

   $\forall e@dst(l).\ kind(e)\ =\ rcv(l, tg)\ \Rightarrow\ kind(sender(e))\ \in\ L$

4.3 Purely Logical Reasoning

Example 1. The Last Change Lemma
It is important in many arguments to find *the last event before an event* e at which a state variable x changes. This is easily done by simply looking at all the predecessors of e, working backwards, $pred?(e)$, $pred?(pred?(e)), \ldots$. It is decidable whether $x \triangle e'$ for all these e'. If there is no change by the time we reach first(e'), then we conclude that x does not change before e.

We state this fact as a theorem from which we extract a function *last* which maps from E into $E?$. We use the predicate $event(e)$ on $E?$ to decide whether $last(e)$ belongs to E or to Unit.

Theorem 6 *(Last Change):*
$\forall i : Loc.\forall x : Id.\forall e@i.\exists\ y : E?\ Last\ (i, x, y, e)$,
where $Last\ (i, x, y, e)\ ==\ event(y) \Rightarrow (\ y\ < e\ \&\ x \triangle y\ \&\ \forall e'@i.\ y <\ e'\ < e\ \Rightarrow\ \neg\ x \triangle e')\ \&$
$\neg\ event(y)\ \Rightarrow\ \forall e'@i.\ e' <\ e\ \Rightarrow\ \neg\ x \triangle e'$.

Proof: Let $i,\ x,\ e$ be given, argue by complete induction on $<_{loc}$.

Base Case: Suppose first(e).
There is no event at i less than e, so y is the unit value, i.e. $\neg\ event(y)$, and there is no $e' < e$.

Induction case:
Assume $\exists\ y : E?Last\ (i, x, y, pred?(e)),\ show\ Last(i, x, y_0, e)$ for some y_0.
Either $x \triangle\ pred?\ (e)\ \vee\ \neg\ x \triangle\ pred?(e)$, since $\triangle$ is decidable.
case $x \triangle\ pred?(e)\ then\ take\ y_0\ =\ pred?(e)$
case $\neg\ x \triangle\ pred?(e)$
Use the induction hypothesis to find a y, $Last\ (i, x, y, pred?(e))$.
Note $event(y)\ \vee\ \neg\ event(y)$
case $event(y)$ then
$y < pred?(e)\ \&\ x \triangle y\ \&\ \forall\ e'@i.\ y < e' <\ pred?(e)\ \Rightarrow\ \neg\ x \triangle e'$.
Since $y < e\ \&\ x \triangle\ y\ implies\ \forall e'@i.\ y < e' <\ e.\ \neg\ x \triangle\ e'$,
we know, $Last\ (i, x, y_0, e)$, with $y_0 = y$.
case $\neg\ event(y)$. Take $y_0 = y$, since $\forall e'@i.\ e' < e \Rightarrow\ \neg\ x \triangle\ e'$.
Thus $Last(i, x, y_0, e)$ as required.
Qed.

Example 3. The Once Lemma

Theorem 7 *Let Hyp be this list of five hypotheses:*

1. $\exists e : E.loc(e) = i.$
2. $\forall e@i.\ first(e) \Rightarrow (done\ \textbf{when}\ \mathbf{e} = false).$
3. $\forall\ e@i\ kind(e) = k \Rightarrow (done\ \textbf{when}\ e\ =\ false)\ \&\ (done\ \textbf{after}\ e\ =\ true\ ).$
4. $\forall e@i.\ \exists e'@i.\ e \leq_{loc} e'\ \&\ (kind(e')\ =\ k \vee\ done\ \textbf{when}\ e'\ =\ true).$
5. $\forall e@i.\ (done \triangle\ e)\ \Rightarrow\ kind(e)\ =\ k.$

Hyp $\vdash\ \exists e@i.\ kind(e)\ =\ k\ \&\ \forall e'@i.\ kind(e') = k \Rightarrow e = e'$

Proof: By 1, $\exists e@i.\ loc(e) = i$. Let e_0 be this event.

Apply 4 to e_0 and let e_1 be such that $e_0 \leq_{loc} e_1$ and $kind\ (e_1) = k\ \vee\ done$ **when** $e_1 =$ true. First consider the case $done$ **when** $e_1 =$ true. Notice that initially $done\ =\ false$. Find the last change $done$ before e_1, say e', so $done \triangle\ e'$. By 5, $kind(e') = k$. Hence in both cases, we conclude $\exists e@i.\ kind(e) = k$. Let this event be e. Now to show $\forall e@i.\ kind(e') = k$ implies $e' = e$. Suppose $kind(e')\ =\ k$. If $e'\ =\ e$ we are done. So suppose $e' < e\ or\ e < e_1$. In

either case, the Exclusion Lemma below is violated. Hence $e' = e$.
Qed.

Exclusion Lemma
Hyp $\vdash \neg \exists\, e_1@i,\ e_2@i.\ e_1 < e_2$ & $kind(e_1) = k$ & $kind(e_2) = k$.

Proof: Suppose e_1, e_2 exist. Note, by 3 *(done* **after** e_1) = true, also *(done* **when** e_2) = $false$. Find the last change of *done* before e_2 , call it e' , note $e_1 \leq e' < e_2$. By 5, $kind(e') = k$, thus by 3 (done **after** e') = true. But *done* must change to false before e_2 , yet there is no change before e_2 . Thus *done* **when** e_2 = true and *done* **when** e_2 = false. This is a contradiction.
Qed.

The Recognizer Lemma
Another computational ability we need is the ability to set a designated variable to true when a decidable condition p becomes true. We want an action k that recognizes $p(s)$. Change *found* to *true* as soon as we recognize $p(s, v)$.

Theorem 8 *Let Hyp be these five assumptions.*

1. **Initially** *(found, i) = false.*
2. $\forall e@i$. *found* **when** e = *false* & *found* **after** e = true $\Rightarrow rcv?(e)$ & $p(s, val(e))$.
3. $rcv?(e)$ & $p(s, val(e)) \Rightarrow$ *found* **after** e = true.
4. $rcv?(e)$ & $\neg\, p(s, val(e)) \Rightarrow$ *found* **after** e = *found when* e.
5. $\forall e@i$. *found* $\triangle e \Rightarrow rcv?(e)$ & $p(s, val(e))$.

Hyp $\vdash \forall e@i$. *found* **when** e = *true* $\Leftrightarrow\ \exists e'@i.\ e' <_{loc} e.\ kind(e') = rcv$ & $p(s, val(e'))$.

Proof:

1. If *found* **when** (e) then $\exists\ e'@i\ (e' < e$ & $found \triangle e)$ by the lemma below.

Lemma: $\forall e@i.\ \neg\, x \triangle e \Rightarrow x$ **when** $e = x$ ***initially*** i.
The lemma is easily proved by induction on $<_{loc}$.

To finish step 1, note, by hypothesis 4 $\forall e@i$. found $\triangle e \Rightarrow kind(e) = rcv$. Hence the *only if* direction follows.

2. Now suppose $\exists\ e'@i.\ e' <_{loc} e$ & $rcv?\ (e')$ & $p(s, val(e'))$ then $found \triangle e')$ and *found* **when** e' = true. We claim that no e'' $e' < e'' \leq e$ can change found to false. The only e'' that can change it is $rcv(e'')$ and that only changes found to be true, and if it is true, it remains true. Hence *(found* **when** e) = true. This proves the *if directions.*

Qed.

In both of the previous two arguments, we see that the hypotheses can be made true by certain conditions on the processes at i that can be achieved by the finite program fragments we discussed. Let us examine these.

For the *Recognizer Lemma*, cause 3 is just a frame condition for an external action whose affect is to set found to true, specifically

state found : $\mathbb{B}$
$rcv(v)$: *effect* found := if $p(sv)$ *then* true *else* found.

Clause 1: is realized by an initial clause.
found ***initially*** = false.

Clause 2: is the effect clause of the above receive action.

For the *Once Lemma*, the first clause, $\exists e : E.\ loc(i)$ can be made true by initializing done to false and then adding an effect guarded by $\neg$ *done*. Then, either $\neg$ *done* remains true until the action is taken by e or it changes to *done* by some other event e'.

5 Proof as Processes

5.1 Introduction

There is a well-established theory and practice for creating *correct-by-construction* functional programs by extracting them from constructive proofs of assertions of the form $\forall x : A.\exists y : B.R(x, y)$ [CAB$^+$86b, BC85, CH88, Con71, ML82, NPS90, BGS03, GPS01, GSW96, PS03a]. We have used this method in the previous sections; for example in extracting the *last* function from the *Last Change Theorem*. There have been several efforts to extend this methodology to concurrent programs, for instance by using linear logic, but there is no practice and the results are limited. In this section, we explain a practical method for creating correct-by-construction processes (protocols).

Several implementations of extraction have been built based on the concept of *proofs-as-programs* (e.g. Alf, MetaPRL, Nuprl, Coq, Lego), and many interesting examples are well-known, including solutions of Higman's lemma [Mur91] and a recent program for Buchberger's Gröbner basis algorithm [Thé01]. The extracted functional programs are called *realizers* for propositions. In this paper we deal with constructive type theory, in which all provable assertions have realizers.

For many years researchers have tried to apply this methodology to concurrent programs by extending the proofs-as-programs principle to something worthy of the name *proofs-as-processes* principle. In 1994 Samson Abramsky wrote an article [Abr94] under this title in which linear logic was the basic logic and certain nondeterministic programs in [BB90] were considered as realizers. Robin Milner and his students also took up this challenge, and there

are now a number of results along these lines [BGHP98, Mil94].

In this article we take a different approach to the problem. We can extract distributed systems from proofs that system specifications that arise in practice are achievable. The realizers are the Message Automata; they resemble the IO automata of Lynch and Tuttle [LT89], and the active objects of Chandy [CK93].

5.2 Synthesis of Two-Phased Handshake Protocol

In Section 2, we examined two-way alternating communication; now we refine this example by using the link structure. Suppose that process S sends messages to process R on link l_1, and R replies on link l_2; thus $src(l_1) = S$, $dist(l_1) = R$, src(l_2) = R, dist(l_2) = S. For simplicity, we assume that only one type of message is sent of type T with tag t and the acknowledgment is tagged ack with value t.

Let

$Snd_{S,l_1} = \{x : E \mid loc(x) = S \ \&\ x \ sends \ (l_1, t)\}$
$Snd_{R,l_2} = \{x : E \mid loc(x) = R \ \&\ x \ sends \ (l_2, < ack, t >)\}$
$Rcv_{S,l_2} = \{x : E \mid loc(x) = S \ \&\ x \ receives \ on \ l_2 \ with \ tag \ ack\}$
$Rcv_{R,l_1} = \{x : E \mid loc(x) = \ R \ \&\ x \ receives \ on \ l_1 \ with \ tag \ t\}$

The two-way alternating character of the communication behavior we seek is described by these theorems.

Theorem 9 *(S-behavior):*
$\forall e_1, e_2 : Snd_{s,l_1}.\exists r : Rcv_{s,l_2} \ (e_1 < e_2 \Rightarrow e_1 < r < e_2)$

Theorem 10 *(R-behavior):*
$\forall e_1, e_2 : Snd_{R,l_2}.\ \exists r_1, r_2 : Rcv_{R,l_1}.\ (e_1 < e_1 \Rightarrow r_1 \leq e_1 < r_2 \leq e_2)$

We will prove the S-behavior result in a purely logical way using the rules of type theory, the axioms of event structures and clauses that can be realized. The proof will implicitly create the constraints on S needed to achieve its role in this two way interaction.

Proof: Assume $e_1 < e_2$ for e_1, e_2 in Snd_{s,l_1}. Thus S sent two messages. We now make a *design decision* in the argument by relating the send events to knowledge of a Boolean variable that keeps track of whether S is ready to send. We call the variable rdy and stipulate

- rdy must be true in order for S to send
- rdy will be set to false by events in Snd_{s,l_1}.

We state these facts as lemmas

L1. $\forall e : Snd_{s,l_1}\ .\ rdy$ **when** $e =$ true
L2. $\forall e : Snd_{s,l_1}\ .\ rdy$ **after** $e =$ false.

From L2 we know rdy **after** $e_1 =$ false, and from L1 rdy **when** $e_2 =$ true. Thus, some event e' between e_1 and e_2 must set rdy to true.

We can constrain e' by limiting access to rdy with a frame condition, and stipulating that only a receive on l_2 of $<ack, t>$ can set ready to true.

L3. $\forall e@s.\ rdy$ **when** $e =$ false & rdy **after** $e =$ true $\Rightarrow rcv(l_2, ack)(e)$

From L3 we conclude that $rcv(l_2, ack)(e')$ as needed.
Qed.

The three lemmas are instances of realizable clauses. L1 is a precondition clause on the send action. To realize it, we create an internal action $a1$ guarded by $rdy =$ true.
The clause L2 is an affect clause for action $a1$ that sets rdy to false. L1 and L2 are realized by this action of a message automaton at S.

$$\begin{aligned} @s \quad & action\ a1\ state\ rdy\ :\ Boolean \\ & precondition\ rdy = true;\ sends\ on\ l_1\ tag\ t \\ & effect\ rdy := false. \end{aligned}$$

The clause L3 is realized by a received action $a2$ whose effect is to set rdy to true.

$$\begin{aligned} @s \quad & action\ a2\ rcv\ (l_2, t)\ (v)\ state\ v\ :\ T \\ & effect\ rdy\ :=\ true. \end{aligned}$$

The proof also depended on the frame condition that only $a1$ and $a2$ affect rdy.

$$@s \quad only\ [a1, a2]\ affect\ rdy.$$

5.2.1 Example: Simple Leader Election

In this section we show how to use the Logic of Events to specify a well-known protocol in a natural way. We intend this example to illustrate several of the ideas behind logical design and correct-by-construction programming.

The *leader election problem* is to have exactly one member of a group announce that it is the leader. If we choose to have the announcement be the occurrence of the action "leader" at a location, then the specification of the leader election for a group R is the following

$$Leader(R)\ \equiv\ \exists ldr :\ R.\ (\exists e@ldr\ =\ leader)\ \&\ (\forall i : R.\ \forall e@i\ =\ leader.\ i = ldr).$$

R is a ring if we have links $in(i)$ and $out(i)$ for each $i \in R$ and $in(i) = out(src)(in(i))$ and the subgraph is connected, so that by following the out links we may reach every node in R. In this case we can define next and predecessor function n and p on R and also a distance metric $d(i, j)$ that is the number of hops needed to get from i to j.

If R is a ring, and we have a one-to-one function, $uid : R \to N$, then we claim that the following specification is derivable and refines $Leader(R)$.

$LE(R, uid, in, out) \equiv \forall i \in R.$

(1) $\exists e = rcv_{out(i)}(vote)(uid(i))$.

(2) $\forall e = rcv_{in(i)}(vote)(v).\ v > uid(i) \ \Rightarrow\ \exists e' = rcv_{out(i)}(vote)(v)$.

(3) $\forall e' = rcv_{out(i)}(vote)(v).\ v = uid(i) \ \vee\ \exists e = rcv_{in(i)}(vote)(v).\ e \prec e' \ \wedge\ v > uid(i)$

(4) $\forall e = rcv_{in(i)}(vote)(uid(i)).\ \exists e'@i = leader.$

(5) $\forall e'@i = leader.\ \exists e = rcv_{in(i)}(vote)(uid(i)).\ e \prec e'$

Theorem 11 *If (R, in, out, n, p) is a ring and $uid : R \to \mathbb{N}$ is 1 -1, then $LE(R, uid, in, out) \ \Rightarrow$ $Leader(R)$.*

Proof: Assuming the hypotheses, we let $m = max\{uid(i) \mid i \in R\}$ and let $ldr = uid^{-1}(m)$. Then the conclusion, $Leader(R)$ follows from the following four lemmas.
Qed.

Lemma 1: $\forall i : R.\ \exists e = rcv_{in(i)}(vote)(uid(ldr))$. .
Proof: By induction on $d(ldr, i)$. If $d = 1$ then $in(i) = out(ldr)$, so by (1)

$$\exists e = rcv_{in(i)}(vote)(uid(ldr)).$$

If $d > 1$ then $p(i) \neq ldr$ and $d(ldr, i) > d(ldr, p(i))$, so by induction,

$$\exists e = rcv_{in(p(i))}(vote)(uid(ldr)).$$

Then by (2), since $uid(ldr) > uid(p(i))$, $\exists e = rcv_{out(p(i))}(vote)(uid(ldr))$,
and $out(p(i)) = in(i)$.
Qed.

Lemma 2: $\forall i, j : R.\ \forall e = rcv_{in(i)}(vote)(uid(j)).\ j = ldr \ \vee\ d(ldr, j) < d(ldr, i)$
Proof: By induction on $\prec$. If $e = rcv_{in(i)}(vote)(uid(j))$ then by (3)

$$uid(j) = uid(p(i)) \ \vee\ \exists e = rcv_{in(p(i))}(vote)(uid(j)).\ e \prec e' \ \wedge\ uid(j) > uid(p(i)).$$

In the first case, we have $j = p(i)$ and this implies $j = ldr \ \vee\ d(ldr, j) < d(ldr, i)$. In the second case, $uid(j) > uid(p(i))$ so $p(i) \neq ldr$ and, by induction, we have

$$j = ldr \ \vee\ d(ldr, j) < d(ldr, p(i))$$

But $d(ldr, p(i)) < d(ldr, i)$, since $p(i) \neq ldr$.
Qed.

Lemma 3: $\forall i : R.\ \forall e'@i = leader.\ i = ldr.$

Lemma 4: $\exists e'@ldr = leader.$

Theorem 12 $LE(R, uid, in, out)$.

Proof: We have to "implement" each of the five clauses, by deriving them from the rules for message automata and event structures. Instantiate the Constant Lemma to get a state variable "me" such that $\forall e@i.\ me$ **when** $e = uid(i)$. Instantiate the Send Once Lemma using $tg = vote, f(s) = s.me, l = out(i)$. This gives

$$\exists e, e'.\ kind(e') = rcv_{out(i)}(vote)\ \wedge\ val(e) = me\ \textbf{when}\ e$$

and also

$$\forall e_1@i = vote.\ sends(e_1) = [msg(out(i), vote, me\ \textbf{when}\ e_1)],$$

which implies $\exists e'.\ kind(e') = rcv_{out(i)}(vote)\ \wedge\ val(e) = uid(i)$, which is clause (1) of $LE(R, uid, in, out)$, and also $\forall e_1@i = vote.\ sends(e_1) = [msg(out(i), vote, uid(i))]$. Instantiate the Trigger lemma (not shown) $k = rcv_{in(i)}(vote),\ k' = leader,\ p(s, v) = (me = v)$ to get

$$\begin{array}{l}\forall i : Loc.\ (\forall e'@i = leader.\ \exists e <_{loc} e'.\ kind(e) = rcv_{in(i)}(vote)\ \wedge\ uid(i) = val(e)) \\ \wedge\ (\forall e@ = rcv_{in(i)}(vote).\ uid(i) = val(e)\ \Rightarrow\ \exists e'.\ kind(e') = leader).\end{array}$$

This gives us clauses (4) and (5) of $LE(R, uid, in, out)$.

The rule for the sends clause

$$sendsi\ rcv_{in(i)}(vote)\mathbb{N}\ \textbf{if}\ v > me\ \ \textbf{then}\ [msg(out(i), vote, v)]\ \ \textbf{else}\ []$$

gives, (since $(me\ \ \textbf{when}\ e_i) = uid(i)$)

$$\forall e@ = rcv_{in(i)}(vote)(v).\ sends(e)\ =\ \ \textbf{if}\ v > uid(i)\ \ \textbf{then}\ [msg(out(i), vote, v)]\ \ \textbf{else}\ []$$

So

$$\forall e@ = rcv_{in(i)}(vote)(v).\ v > uid(i)\ \Rightarrow\ msg(out(i), vote, v) \in sends(e)$$

By a simple fact on sending and receiving (not shown), this implies clause (2)

$$\forall e = rcv_{in(i)}(vote)(v).\ v > uid(i)\ \Rightarrow\ \exists e' = rcv_{out(i)}(vote)(v).$$

Finally, to derive clause (3) we need a send frame clause to constrain the actions that can send vote messages. In what we have derived so far , the only actions that send vote messages

are the $rcv_{in(i)}(vote)$ action and also the action $vote$ from the Send Once Lemma. So we use the rule for the send frame clause

$@i$ $\mathbf{o}nly[rcv_{in(i)}(vote); vote]\mathbf{s}ends\langle out(i), vote\rangle$:
$\forall e' = rcv_{out(i)}(vote) \Rightarrow kind(sender(e')) = rcv_{in(i)}(vote) \vee kind(sender(e'))vote.$

From this we can prove clause (3) since if $e' = rcv_{out(i)}(vote)(v)$ then

$$emsg(e') = msg(out(i), vote, v) \in sends(out(i), sender(e')).$$

Then either $kind(sender(e')) = vote$, in which case

$$sends(sender(e')) = [msg(out(i), vote, uid(i))], sov = uid(i),$$

or, for some v, $sender(e') = rcv_{in(i)}(vote)(v)$, in which case

$$sends(sender(e')) = \textbf{if } v > uid(i) \textbf{ then } [msg(out(i), vote, v)] \textbf{ else } [],$$

so we must have $v > uid(i)$.
Qed.

At this point we have proved the leader election specification, so we can extract from our proof a distributed system as an assignment of message automata to locations (see Section 3.3).

5.2.2 Message Automata Realizer

Leader automaton consists of the following clauses for each $i \in R$:

$@i$ $me : \mathbb{N}$ **initially** $= uid(i)$
$@i$ $done : \mathbb{B}$ **initially** $= false$
$@i$ $x : \mathbb{B}$ **initially** $= false$
$@i$ **precondition** $vote$ **is** $\neg done$
$@i$ **effect** $vote$ **is** $done : \mathbb{B} := true$
$@i$ **action** $vote$ **sends on** $out(i)$ $[vote, me]$
$@i$ $rcv_{in(i)}(vote)(v : \mathbb{N})$ **sends on** $out(i)$
if $v > me$ **then** $[vote, v]$ **else** $[]$
$@i$ **effect** $rcv_{in(i)}(vote)(v : \mathbb{N})$ **is**
$x : \mathbb{B} :=$ **if** $me = v$ **then** $true$ **else** x
$@i$ **precondition** $leader$ **is** $x = true$
$@i$ **only** $[rcv_{in(i)}(vote); vote]$ **sends** $\langle out(i), vote\rangle$
$@i$ **only** $[]$ **affects** me
$@i$ **only** $[vote]$ **affects** $done$
$@i$ **only** $[rcv_{in(i)}(vote)]$ **affects** x

5.3 Extraction of Process Code

Our current Logical Programming Environments supports proof and program development by top down refinement of goals into subgoals and bottom up synthesis of programs from fragments of code derived from proofs of subgoals. We are extending this mechanism, called *program extraction*, to the synthesis of *processes* to support *process extraction*.

Our library of declarative knowledge about distributed systems contains many theorems that state that some property ϕ of event structures is *realizable* (which we write here as $\models \phi$). A property ϕ is realizable if and only if there is a distributed system D that is both *feasible*—which implies that there is at least one world and, hence, at least one event structure, consistent with D—and realizes the property; every event structure consistent with D satisfies the property.

The basic rules for message automata provide initial knowledge of this kind—all the properties of single clauses of message automata are realizable. We add to our knowledge by proving that more properties are realizable. In these proofs, the system will automatically make use of the knowledge already in the library when we reach a subgoal that is known to be realizable.

To make this automated support possible, some new features, which we believe are unique to the Nuprl system, have been added. In order to motivate a discussion of these features, let us examine in detail the steps a user takes in proving a new realizability result.

Suppose that we want to prove that ϕ is realizable, and we start a proof of the top-level goal $\models \phi$. From the form of the goal, the proof system knows that we must produce a feasible distributed system D that realizes ϕ so it adds a new abstraction $D(x, \ldots, z)$ to the library (where $x, \ldots, z$ are any parameters mentioned in ϕ). The new abstraction has no definition initially—that will be filled in automatically as the proof proceeds. This initial step leads to a goal where from the hypothesis that an event structure es is consistent with $D(x, \ldots, z)$ we must show the conclusion that $\phi(es)$, i.e., that es satisfies ϕ .

Now, suppose that we can prove a lemma stating that in any event structure, es,

$$\psi_1(es) \ \& \ \psi_2(es) \Rightarrow \phi(es)$$

(the proof of this might be done automatically by a good decision procedure for event structures or interactively in the standard way). In this case, the user can refine the initial goal $\phi(es)$ by asserting the two subgoals $\psi_1(es)$ and $\psi_2(es)$ (and then finishing the proof of $\phi(es)$ using the lemma).

If ψ_1 is already known to be realizable, then there is a lemma $\models \psi_1$ in the library and, since the proofs are constructive, there is a realizer A_1 for ψ_1. Thus to prove $\psi_1(es)$, it is enough to show that es is consistent with A_1, and since this follows from the fact that es is consistent with $D(x, \ldots, z)$ and that $A_1 \subset D(x, \ldots, z)$, the system will automatically refine the goal $\psi_1(es)$ to $A_1 \subset D(x, \ldots, z)$. If ψ_2 is also known to be realizable with realizer A_2 then the system produces the subgoal $A_2 \subset D(x, \ldots, z)$, and if not, the user uses other lemmas about event structures to refine this goal further.

Whenever the proof reaches a point where the only remaining subgoals are $D(x, \ldots, z)$ is feasible or have the form $A_i \subset D(x, \ldots, z)$, then the proof can be completed automatically

by defining $D(x, \ldots, z)$ to be the join of all the A_i. In this case, all the subgoals of the form $A_i \subset D(x, \ldots, z)$ are automatically proved, and only the feasibility proof remains. Since each of the realizers A_i is feasible, the feasibility of their join follows automatically from the pairwise compatibility of the A_i and the system will prove the pairwise compatibility of the realizers A_i automatically if they are indeed compatible, in which case the proof of the realizability of ϕ is complete and its realizer has been constructed and stored in the library.

How might realizer A_1 and A_2 be incompatible? For instance, A_1 might contain a clause that initializes a state variable x to true while A_2 contains a clause that initializes the same variable to false. Or, A_1 might declare that an action of *kind* k_1 has an effect on x while A_2 declares that only actions of *kinds* k_2 or k_3 may affect x.

If the variable x occurs explicitly in the top goal ϕ then the user has simply made incompatible design choices in his attempt to realize ϕ and must change the proof. However, if the variable x is not explicitly mentioned in ϕ then it is the case that x can be renamed to y without affecting ϕ. It is often the case that x can be renamed independently in the proofs of the subgoals ψ_1 and ψ_2 (say to y and z) and hence the realizers $A_1(y)$ and $A_2(z)$ will no longer be incompatible.

Incompatibilities such as these can arise when names for variables, local actions, links, locations, or message tags that may be chosen arbitrarily and independently, happen to clash. Managing all of these names is tedious and error prone, so we have added automatic support for managing these names.

By adding some restrictions to the definition mechanism, we are able to ensure that the names inherent in any term are always visible as explicit parameters. We have also defined the semantics of Nuprl in such a way that the *permutation rule* is valid. The permutation rule says that if proposition $\phi(x, y, \ldots, z)$ is true, where $x, y, \ldots, z$ are the names mentioned in ϕ, then proposition $\phi(x', y', \ldots, z')$ is true, where $x', y', \ldots, z'$ is the image of $x, y, \ldots, z$ under a permutation of all names.

Using the permutation rule, our automated proof assistant will always permute any names that occur in realizers brought in automatically as described above, so that any names that were chosen arbitrarily in the proof of the realizability lemma but were not explicit in that lemma will be renamed to fresh names that will not clash with any names chosen so far. This strategy is supported by the primitive rules of the logic and it guarantees that any incompatibility in the realizer built by the system was inherent in the user's design choices and was not just an unfortunate accident in the choice of names.

6 Conclusion

6.1 Origins

The event system formalism described here resulted from several years of experience in using formal methods in the design, implementation and verification of distributed system components [BCH+00, HvR97, Kre03, HLvR99, Kre99, KHH98, vRBH+98, LKvR+99, BH99, BCH+00, BKvRC01, BKvRL01]. We needed a formalism capable of expressing all of the

features important in real world examples including fault tolerance, performance, security, both asynchronous and synchronous communication, quality of service requirements and so forth. Programming notations such as *IO-automata* [Lyn96] turned out to be a good abstract computing framework for expressing these concepts and for building reference implementations of important construction patterns in systems such as UAV [LRSA02]. However, even on this level of idealized code it is tedious to reason about the behavior of software components, so we have worked toward *abstracting even further* in such a way that these computing notations would become the means of realizing distributed computing behaviors described in a declarative language that directly captures Lamport's insights.

6.2 Related Work

Event Systems Winskel considered event systems in his 1980 Ph.D. thesis [Win80] and in other publications [Win89]. He considered relationships to Petri nets and to domain theory and established the generality of event systems. As in our work, he abstracted from Lamport [Lam78] where events are local transitions and message passing. Results on *knowledge in multi-agent systems* [FHMV97, Hal00, HF89, HS99] use models with some of the properties of worlds in our event systems.

Protocol Verification Hoare [Hoa85] and Milner [Mil89] created extremely influential *process calculi* and their work is the basis for exploring verification of processes. Milner's approach has been extended to mobile processes and *action calculi* [Mil93b, Mil93a, Mil96].

Many logics used for practical reasoning and formal verification are based on programming logics [ZdRvEB85, Sch97, GT97] or on *temporal logic* [MP92, MP95], especially Unity [CM88] and *TLA*[Lam03]. One of the richest is Lamport's first-order TLA [Lam94] system which has been embedded in theorem provers such as Isabelle [Isa] and the Larch Prover LP. PVS is used extensively as well [KRS99, QS98]. TLA+ is a system with primitives for specifying real-time properties [Lam93, Lam03]. We do not regard this logic as sufficiently expressive for direct communication across the range of people who need to use it. It is also not oriented to the goal of extracting executable code from declarative specifications.

Temporal logics provide a way to reason about how states evolve over time, but in temporal logics, *only* state variables can be directly referred to; time and events are built into the temporal operators but are not things that are explicitly named. This means that designers using a temporal logic based method are forced to focus on state variables and their invariants when often it would be more natural to focus on events and their ordering.

Abraham [Abr95, Abr99] uses classical multi-sorted first order logic to model processes as defined by Lamport where state transitions are events. Our approach is related to his in that we use a higher-order *constructive logic* to define the models. His logic and ours deal explicitly with collections of events and with functions on these collections — another feature missing from temporal logic.

Synthesis One of the most direct attempts to synthesize concurrent programs using proofs as processes is the work of Abramsky [Abr94, Abr00] directed toward linear logic. These

results are of considerable theoretical interest, but they have not been connected to practical verification. Milner's process calculi have also been used to explore process realizability of logical formulas [BGHP98, Mil94, Mil96]. The disadvantage of these well known and deeply studied methods is that they have not been applied to real systems of the kind this effort is focused on and they do not support code synthesis from formal proofs. See also Vardi[Var95], Clarke, Emerson [CE82], and Manna, Wolper [MW84], and Koskimies, Makinen [KM94] for different notions of synthesis that reference the meaning we intend. Temporal logic has a limited role in synthesis [EC82]. For knowledge-based synthesis, we have shown that in principle our current implementation of event theory with knowledge operators can implement the rules and calculus of Engelhardt, v.d. Meyden, and Moses [EvdMM98]. The Kestrel group has also done a great deal of work on synthesis using Refine, Specware, and Planware [SL90, SG96, PS03b]. They use category theory to ground their methods rather than the proofs-as-programs paradigm.

IO-Automata Our message automata are related to the IO-automata of Nancy Lynch. The IO-automata provide a convenient abstract concurrent programming notation and there is a large body of work describing, analyzing, and proving properties of distributed algorithms expressed as IO-automata, much of which is described in the book [Lyn96]. Verification based on IO Automata [Lyn96] has been directly modeled in Nuprl [BH99] and PVS [AHR02] and it is subsumed here as the special case where we reason directly about message automata. We have worked with the IO-automata formalism and proof method for several years and have found ways to improve it by building message passing into the semantics of message automata and adding the *sends* clauses to message automata.

Alfaro and Henzinger [dAH01] make the distinction between *interface models* of components, which assert the existence of a helpful environment in which the component operates properly, and *component models*, which assert that the component operates properly in every environment. Our notion of realizability is both an interface model and a component model since it asserts both the existence of an event structure consistent with the program and that every event structure consistent with the program satisfies the specification. Also see [KW01] for a use of automata in the semantics of *message sequence diagrams* which are also featured in [HM03].

Abstract State Machines The abstract state machines approach to distributed systems has let to interesting semantic models [GRS05, GGV04, BG03a, BG03b, Mes03, Esc01].

Active Objects The *active objects* model is similar to our semantics of distributed systems of message automata: objects communicate by passing messages over peer-to-peer channels. The info-spheres work also has a formal component – distributed algorithms can be expressed and reasoned about in the UNITY programming logic and extensions of UNITY [CC99, Mis01]. The UNITY logic is used to prove properties of abstract concurrent programs. It does not have a method for extracting code from proofs, nor does it have a tactic mechanism. Some work has been done to embed UNITY into theorem provers like Isabelle [Pau99].

The greatest ideas are the greatest events.
Nietzsche

Acknowledgements We want to thank Rebecca Rich-Goldweber and Cindy Robinson for their excellent help in preparing the manuscript. They both worked under challenging conditions to meet the deadline for this article.

References

[ABC+] Stuart F. Allen, Mark Bickford, Robert Constable, Richard Eaton, Christoph Kreitz, Lori Lorigo, and Evan Moran. Innovations in computational type theory using Nuprl. To appear in Journal of Applied Logic 2006.

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

[Abr95] Uri Abraham. On interprocess communication and the implementation of multi-writer atomic registers. *Journal of Theoretical Computer Science*, 149:257–298, 1995.

[Abr99] Uri Abraham. *Models for Concurrency*, volume 11 of *Algebra, Logic and Applications Series*. Gordon and Breach, 1999.

[Abr00] S. Abramsky. Process realizability. In F. L. Bauer and R. Steinbrüggen, editors, *Foundations of Secure Computation: Proceedings of the 1999 Marktoberdorf Summer School*, pages 167–180. IOS Press, 2000.

[ACE+00] Stuart Allen, Robert Constable, Richard Eaton, Christoph Kreitz, and Lori Lorigo. The Nuprl open logical environment. In David McAllester, editor, *Proceedings of the 17th International Conference on Automated Deduction*, volume 1831 of *Lecture Notes in Artificial Intelligence*, pages 170–176. Springer Verlag, 2000.

[AHR02] Myla Archer, Constance Heitmeyer, and Elvinia Riccobene. Proving invariants of I/O automata with TAME. *Automated Software Engineering*, 9(3):201–232, 2002.

[BB90] G. Berry and G. Boudol. The chemical abstract machine. In *Conference Record of the 17th Annual ACM Symposium on Principles of Programming Languages*, pages 81–94, 1990.

[BC85] J. L. Bates and Robert L. Constable. Proofs as programs. *ACM Transactions on Programming Languages and Systems*, 7(1):53–71, 1985.

[BCH+00] K. Birman, R. Constable, M. Hayden, J. Hickey, C. Kreitz, R. van Renesse, O. Rodeh, and W. Vogels. The Horus and Ensemble projects: Accomplishments and limitations. In *DARPA Information Survivability Conference and Exposition (DISCEX 2000)*, pages 149–161, Hilton Head, SC, 2000. IEEE Computer Society Press.

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

[BG03b] Andreas Blass and Yuri Gurevich. Algorithms: A quest for absolute definitions. *Bulletin of the European Association for Theoretical Computer Science*, (81):195–225, 2003.

[BG05] Mark Bickford and David Gauspari. Scores: A programming logic for distributed systems. Technical Report Technical Report 05-0007, ATC-NY, 2005.

[BGHP98] Andrew Barber, Philippa Gardner, Masahito Hasegawa, and Gordon D. Plotkin. From action calculi to linear logic. In Mogens Nielsen and Wolfgang Thomas, editors, *Computer Science Logic, 11th International Workshop, Annual Conference of the EACSL, Aarhus, Denmark, August 23–29, 1997, Selected Papers*, volume 1414 of *Lecture Notes in Computer Science*, pages 78–97. Springer, 1998.

[BGS03] Marcel Becker, Limei Gilham, and Douglas R. Smith. Planware II: Synthesis of schedulers for complex resource systems. Submitted for publication, 2003.

[BH99] Mark Bickford and Jason J. Hickey. Predicate transformers for infinite-state automata in Nuprl type theory. In *Proceedings of 3rd Irish Workshop in Formal Methods*, 1999.

[BKvRC01] Mark Bickford, Christoph Kreitz, Robbert van Renesse, and Robert Constable. An experiment in formal design using meta-properties. In J. Lala, D. Mughan, C. McCollum, and B. Witten, editors, *DARPA Information Survivability Conference and Exposition II (DISCEX-II)*, volume II of *IEEE Computer Society Press*, pages 100–107, Anaheim, CA, 2001.

[BKvRL01] Mark Bickford, Christoph Kreitz, Robbert van Renesse, and Xiaoming Liu. Proving hybrid protocols correct. In Richard Boulton and Paul Jackson, editors, *14th International Conference on Theorem Proving in Higher Order Logics*, volume 2152 of *Lecture Notes in Computer Science*, pages 105–120, Edinburgh, Scotland, September 2001. Springer-Verlag.

[CAB+86a] Robert L. Constable, Stuart F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, NJ, 1986.

[CAB+86b] Robert L. Constable, Stuart F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, NJ, 1986.

[CC99] Michel Charpentier and K. Mani Chandy. Towards a compositional approach to the design and verification of distributed systems. In Jeannette Wing, Jim

Woodcock, and J. Davies, editors, *FM99: The World Congress in Formal Methods in the Development of Computing Systems*, volume 1708 of *Lecture Notes in Computer Science*, pages 570–589. Springer Verlag, 1999.

[CE82] Edmund M. Clarke and E. Allen Emerson. Synthesis of synchronization skeletons from branching time temporal logic. In *Proc. Workshop on Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer–Verlag, 1982.

[CH88] Thierry Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.

[CK93] K. Mani Chandy and Carl Kesselman. CC++: A declarative concurrent object oriented programming notation. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, chapter 11, pages 281–313. MIT Press, Boston, 1993.

[Cle99] W. Rance Cleaveland, editor. *5^{th} International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1579 of *Lecture Notes in Computer Science*. Springer, 1999.

[CM88] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.

[Con71] Robert L. Constable. Constructive mathematics and automatic program writers. In *Proceedings of the IFIP Congress*, pages 229–233. North-Holland, 1971.

[Con02] Robert L. Constable. Naïve computational type theory. In H. Schwichtenberg and R. Steinbrüggen, editors, *Proof and System-Reliability, Proceedings of International Summer School Marktoberdorf, July 24 to August 5, 2001*, volume 62 of *NATO Science Series III*, pages 213–260, Amsterdam, 2002. Kluwer Academic Publishers.

[dAH01] Luca de Alfaro and Thomas A. Henzinger. Interface theories for component-based design. In *Proceedings of the First International Workshop on Embedded Software(EMSOFT)*, volume 2211 of *Lecture Notes in Computer Science*, pages 148–165. Springer-Verlag, 2001.

[EC82] E. Allen Emerson and Edmund M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2(3):241–266, 1982.

[Esc01] Robert Eschbach. A verification approach for distributed abstract state machines. In *PSI'02: Revised Papers from the 4th International Andrei Ershov Memorial Conference on Perspectives of System Informatics*, volume 2244 of *Lecture Notes in Computer Science*, pages 109–115. Springer-Verlag, London, UK, 2001.

[EvdMM98] Kai Engelhardt, Ron van der Meyden, and Yoram Moses. A program refinement framework supporting reasoning about knowledge and time. In Jerzy Tiuryn, editor, *Proc. Foundations of Software Science and Computation Structures (FOSSACS 2000)*, pages 114–129, Berlin/New York, 1998. Springer-Verlag.

[FHMV97] Ronald Fagin, Joseph Y. Halpern, Yoram Moses, and Moshe Y. Vardi. Knowledge-based programs. *Distributed Computing*, 10(4):199–225, 1997.

[GGV04] Uwe Glaesser, Yuri Gurevich, and Margus Veanes. Abstract communication model for distributed systems. *IEEE Transactions on Software Engineering*, 30(7):458–472, 2004.

[GM93] Michael Gordon and Tom Melham. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic*. Cambridge University Press, Cambridge, 1993.

[GPS01] Cordell Green, Dusko Pavlovic, and Douglas R. Smith. Software productivity through automation and design knowledge. In *Software Design and Productivity Workshop*, 2001.

[GRS05] Yuri Gurevich, Benjamin Rossman, and Wolfram Schulte. Semantic essence of asml. *Theoretical Computer Science*, 343(3):370–412, October 2005.

[GSW96] Carla P. Gomes, Douglas R. Smith, and Stephen J. Westfold. Synthesis of schedulers for planned shutdowns of power plants. In *Proceedings of the Eleventh Knowledge-Based Software Engineering Conference*, pages 12–20. IEEE Computer Society Press, 1996.

[GT97] V. K. Garg and A. I. Tomlinson. Using the causal domain to specify and verify distributed programs. *Acta Informatica*, pages 667–686, 1997.

[GTL89] J-Y. Girard, P. Taylor, and Y. Lafont. *Proofs and Types*, volume 7 of *Cambridge Tracts in Computer Science*. Cambridge University Press, 1989.

[Hal00] Joseph Y. Halpern. A note on knowledge-based programs and specifications. *Distributed Computing*, 13(3):145–153, 2000.

[Hey02] Anthony J.G. Hey. *Feynman & Computation*. Westview Press, Boulder, 2002.

[HF89] Joseph Y. Halpern and Ronald Fagin. Modeling knowledge and action in distributed systems. *Distributed Computing*, 3(4):159–177, 1989.

[HLvR99] Jason J. Hickey, Nancy Lynch, and Robbert van Renesse. Specifications and proofs for Ensemble layers. In Cleaveland [Cle99], pages 119–133.

[HM03] D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer-Verlag, New York, 2003.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[HS99] Joseph Y. Halpern and Richard A. Shore. Reasoning about common knowledge with infinitely many agents. In *Proceedings of the 14th IEEE Symposium on Logic in Computer Science*, pages 384–393, 1999.

[HvR97] Mark Hayden and Robbert van Renesse. Optimizing layered communication protocols. In *Proceedings of the High Performance Distributed Computing*, Portland, Oregon, August 1997.

[Isa] Isabelle home page. `http://www.cl.cam.ac.uk/Research/HVG/Isabelle`.

[KHH98] Christoph Kreitz, Mark Hayden, and Jason J. Hickey. A proof environment for the development of group communications systems. In *Fifteen International Conference on Automated Deduction*, number 1421 in Lecture Notes in Artificial Intelligence, pages 317–332. Springer, 1998.

[KM94] K. Koskimies and E. Makinen. Automatic synthesis of state machines from trace diagrams. *Software–Practice and Experience*, 24(7):643–658, 1994.

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

[Kre03] Christoph Kreitz. The FDL navigator: Browsing and manipulating formal content. Cornell University, Ithaca, NY, 2003. `http://www.nuprl.org/documents/Kreitz/03fdl-navigator.html`.

[KRS99] S. S. Kulkarni, J. Rushby, and N. Shankar. A case-study in component-based mechanical verification of fault-tolerant programs. In A. Arora, editor, *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems Workshop on Self-Stabilizing Systems, Austin, TX*, pages 33–40. IEEE Computer Society Press, 1999.

[KW01] J. Klose and H. Wittke. An automata based interpretation of live sequence charts. In *Proceedings of Seventh International Coference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*, 2001.

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

[Lam93] Leslie Lamport. Hybrid systems in TLA+. In Grossman, Nerode, Ravn, and Rischel, editors, *Hybrid Systems*, volume 736 of *Lecture Notes in Computer Science*, 1993.

[Lam94] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.

[Lam03] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, Boston, 2003.

[LKvR+99] Xiaoming Liu, Christoph Kreitz, Robbert van Renesse, Jason J. Hickey, Mark Hayden, Kenneth Birman, and Robert Constable. Building reliable, high-performance communication systems from components. In David Kotz and John Wilkes, editors, *17th ACM Symposium on Operating Systems Principles (SOSP'99)*, volume 33(5) of *Operating Systems Review*, pages 80–92. ACM Press, DIGITAL Systems Research Center 1999.

[LRSA02] J. Loyall, P. Rubel, R. Schantz, and M. Atighetc. Emerging patterns in adaptive, distributed real-time, embedded middleware. In Weerasak Witthawaskul, editor, *9th Conference of Pattern Language of Programs*, 2002.

[LT89] Nancy Lynch and Mark Tuttle. An introduction to Input/Output automata. *Centrum voor Wiskunde en Informatica*, 2(3):219–246, September 1989.

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

[Mes03] Jos'e Meseguer. Software specification and verification in rewriting logic. Unpublished, 2003.

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

[Mil93a] Robin Milner. Higher-order action calculi. In Egon Börger, Yuri Gurevich, and Karl Meinked, editors, *Computer Science Logic*, volume 832 of *Lecture Notes in Computer Science*, pages 238–260, 1993.

[Mil93b] Robin Milner. Structures for the λ-calculus action structures. University of Edinburgh, Edinburgh, UK, August 1993. Marktoberdorf.

[Mil94] R. Milner. Action structures and the π-calculus. In Helmut Schwichtenberg, editor, *Proof and Computation*, volume 139 of *NATO Advanced Study Institute, International Summer School held in Marktoberdorf, Germany, July 20–August 1, 1993, NATO Series F*, pages 219–280. Springer, Berlin, 1994.

[Mil96] Robin Milner. Calculi for interaction. *Acta Informatica*, 33(8):707–737, 1996.

[Mis01] Jayadev Misra. *A Discipline of Multiprogramming*. Springer Verlag, 2001.

[ML82] Per Martin-Löf. Constructive mathematics and computer programming. In *Proceedings of the Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175, Amsterdam, 1982. North Holland.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, Berlin, 1992.

[MP95] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety.* Springer-Verlag, Berlin, 1995.

[Mur91] Chetan Murthy. An evaluation semantics for classical proofs. In *Proceedings of Sixth Symposium on Logic in Comp. Sci.*, pages 96–109. IEEE, Amsterdam, The Netherlands, 1991.

[MW84] Z. Manna and P. Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM Trans. Program. Lang. and Syst.*, 6(1):68–93, 1984.

[NPS90] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory.* Oxford Sciences Publication, Oxford, 1990.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[NSS57] A. Newell, J.C. Shaw, and H.A. Simon. Empirical explorations with the logic theory machine: A case study in heuristics. In *Proceedings West Joint Computer Conference*, pages 218–239, 1957.

[Pau88] L.C. Paulson. A preliminary user's manual for Isabelle. Technical Report 133, University of Cambridge, Cambridge, England, 1988.

[Pau99] Lawrence C. Paulson. Mechanizing UNITY in isabelle. *ACM Transactions on Computational Logic*, 1999.

[PN90] L. Paulson and T. Nipkow. Isabelle tutorial and user's manual. Technical report, University of Cambridge Computing Laboratory, 1990.

[PS03a] Dusko Pavlovic and Douglas R. Smith. Software development by refinement. In Bernhard K. Aichernig and T. S. E. Maibaum, editors, *UNU/IIST 10th Anniversary Colloquium, Formal Methods at the Crossroads: From Panaea to Foundational Support*, volume 2757 of *Lecture Notes in Computer Science*, pages 267–286. Springer, 2003.

[PS03b] Dusko Pavlovic and Douglas R. Smith. Software development by refinement. In Bernhard K. Aichernig and T. S. E. Maibaum, editors, *UNU/IIST 10th Anniversary Colloquium, Formal Methods at the Crossroads: From Panaea to Foundational Support*, volume 2757 of *Lecture Notes in Computer Science*, pages 267–286. Springer, 2003.

[QS98] Shaz Qadeer and Natarajan Shankar. Verifying a self-stabilizing mutual exclusion algorithm. In David Gries and Willem-Paul de Roever, editors, *IFIP International Conference on Programming Concepts and Methods: PROCOMET'98*, pages 424–443, Shelter Island, NY, June 1998. Chapman & Hall.

[Sch97] Fred B. Schneider. *On Concurrent Programming*. Springer-Verlag, New York, 1997.

[SG96] Douglas R. Smith and Cordell Green. Toward practical applications of software synthesis. In *FMSP'96, The First Workshop on Formal Methods in Software Practice*, pages 31–39., 1996.

[SL90] D.R. Smith and M.R. Lowry. Algorithm theories and design tactics. *Science of Computer Programming*, 14(2–3):305–321, October 1990. Report KES.U.89.3, Kestrel Institute.

[Thé01] Laurent Théry. A machine-checked implementation of Buchberger's algorithm. *Journal of Automated Reasoning*, 26(2):107–137, February 2001.

[Tur37] A. M. Turing. On computable numbers, with an application to the Entscheidungs problem. In *Proceedings London Math Society*, pages 116–154, 1937.

[Var95] M. Y. Vardi. An automata-theoretic approach to fair realizability and synthesis. In P. Wolper, editor, *Computer Aided Verification, Proceedings of the 7th International Conference*, volume 939 of *Lecture Notes in Computer Science*, pages 267–292. Springer-Verlag, 1995.

[vRBH+98] Robbert van Renesse, Kenneth P. Birman, Mark Hayden, Alexey Vaysburg, and David Karr. Building adaptive systems using Ensemble. *Software: Practice and Experience*, 28(9):963–979, July 1998.

[Whe82] John A. Wheeler. The computer and the universe. *Int. J. Theoretical Physics*, 21:557–571, 1982.

[Whe89] John A. Wheeler. Information, physics, quantum: The search for links. In *Proc. 3d Int. Symp. Foundations of Quantum Mechanics*, Tokyo, 1989.

[Win80] G. Winskel. *Events in Computation*. PhD thesis, University of Edinburgh, 1980.

[Win89] G. Winskel. An introduction to event structures. In J. W. de Bakker et al., editors, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, number 345 in Lecture Notes in Computer Science, pages 364–397. Springer, 1989.

[ZdRvEB85] Job Zwiers, Willem P. de Roever, and Peter van Emde Boas. Compositionality and concurrent networks: Soundness and completeness of a proofsystem. In *ICALP 1985*, pages 509–519, 1985.