Proof Assistants and the Dynamic Nature of Formal Theories[*]

Robert L. Constable

Cornell University

Abstract

This article shows that *theory exploration* arises naturally from the need to progressively modify *applied formal theories*, especially those underpinning deployed systems that change over time or need to be attack-tolerant. Such formal theories require us to *explore a problem space* with a proof assistant and are naturally dynamic.

The examples in this article are from our on-going decade-long effort to formally synthesize critical components of modern distributed systems. Using the Nuprl proof assistant we created *event logic* and its protocol theories. I also mention the impact over this period of extensions to the constructive type theory implemented by Nuprl. One of them led to our solution of a long standing open problem in constructive logic.

Proof exchange among theorem provers is promising for improving the "super tactics" that provide *domain specific reasoners* for our protocol theories. Both theory exploration and proof exchange illustrate the dynamic nature of applied formal theories built using modern proof assistants. These activities dispel the false impression that formal theories are rigid and brittle artifacts that become less relevant over time in a fast moving field like computer science.

1 Introduction

I believe that one of the major scientific accomplishments of computer science is the creation of software systems that provide indispensable help in performing complex reasoning tasks and in automating abstract intellectual processes. Among such systems are *proof assistants.* They have helped people solve open mathematical problems, build provably correct hardware and software, create attack-tolerant distributed systems, and implement foundational theories of mathematics and computer science. These assistants are also finding a place in advanced technical education [53]. This development fits John McCarthy's early definition of computer science as the subject concerned with the *automation of intellectual processes* [44], and I think it would have fit Turing's, who I regard as the founder of the field. In recognition of this historical Turing celebration year, this article will include some bits of history related to Turing's role in logic and computer science and some bits related to constructive type theory.

These accomplishments with proof assistants are well documented in academic journals and confirmed by industrial uptake. Nevertheless, many otherwise well-informed and influential scientists believe that proof assistants are very difficult to use in these ways and that their solutions are expensive, rigid, and inflexible [21]. I claim that in many cases the formal solutions provided by proof assistants are flexible and easily modified. In some cases that I will discuss, as in creating attack-tolerant networked systems, it is imperative that we automatically modify the supporting formal theories.

[*] This paper appears also, by agreement, in J. Fleuriot, P. Höfner, A. McIver, and A. Smaill (eds.), Proceedings ATx/WInG 2012.

The fact that we can easily modify, reuse, and replay formal theories might be the main driving force spreading their widening use, just as it is a driving force in the universal spread of text editors, compilers, programming environments, and other commonly used "tools for precise thought". I predict that we will see in the near future dynamic formal theories that are developed and maintained by a community of researchers using proof assistants to support applied formal theories that in turn support deployed software systems "on the fly". We already see communal activity in joint efforts to build mathematical theories or settle open problems, and that is a first step.

1.1 Overview

I explain these claims by referring to proof assistants that implement *constructive type theories* – because these are the proof assistants I know best and because they also connect directly to programming practice. Indeed, they support what has come to be called *correct-by-construction* programming and *formal system evolution* [39, 36, 38]. This is because critical elements of the executing code in deployed systems built this way are synthesized from proofs. To reliably change that code, one must change the proofs and extract new code. Such *applied formal theories* commonly evolve with the application. Moreover, we see another interesting phenomenon, namely the underlying foundational type theories also continue to evolve, perhaps faster than foundational theories that mainly support established mathematics. Those deep theory extensions can be brought to bear on improving the applied theories embedded in them. I will illustrate this and claim that type theories will continue to evolve for quite some time.

I mention briefly just one example of the gradual extension of Constructive Type Theory (CTT), the theory implemented by Nuprl. This example led my colleague Mark Bickford and me to solve an open problem in logic by finding a completeness theorem for intuitionistic first-order logic [14]. We systematically, but gradually over a few years, expanded the use of the intersection type over a family of propositions – at times treating it as a polymorphic universal quantifier because it made program extraction simpler; this led us to the idea of *uniform validity* which became the key to solving the open problem in logic.

Additions to the underlying foundational type theory can be indexed by the years in which they were made, up to the one we used to solve the problem, say from CTT00 to CTT10. The base theory is CTT84, and we are now using CTT12.[1] By and large each theory is an extension of the previous one, although there was a major improvement from CTT02 onwards along with a corresponding change in the proof assistant from Nuprl4 to Nuprl5.

In the special case of proof assistants based on the *LCF tactic mechanism* [29], there is an interesting way to create *domain specific reasoners.* This is simple to describe and remarkably effective in the case of proof assistants with distributed computing capabilities as is the case with Nuprl in support of *Constructive Type Theory* circa 2012 (CTT12). We turn briefly to that example later.

These scientific contributions are enabled by a powerful and ubiquitious information technology that integrates programming languages, interactive provers, automatic provers, model checkers, and databases of formal knowledge. I believe that this integrated technology is rapidly moving toward a point at which it will provide tools for thought able to accelerate scientific research and enrich education in ways computer scientists have only imagined before. The practical integration of computing and logic seen for the first time in this research area has made us aware in detail that there is a *computational reality to logic* which Turning glimpsed in general outline in 1936.

[1] In contrast, I believe that the formalized Tarski-Grothendieck set theory used by the Mizar system [43] to support the *Journal of Formalized Mathematics* has been stable from 1989 until now.

1.2 Historical Background

According to Andrew Hodges [34], one of Alan Turing's deepest insights about computing was that his universal machines would compute with logical formulas as well as numbers. Hodges says: "It put logic, not arithmetic, in the driving seat." Turing believed that this extension would allow computers to participate in the whole range of rational human thought. Turing also understood that we would reason precisely about programs.

Turing also made the notion of a *computable function* a central concept in science. This in turn made the notion of *data types* fundamental and linked them to Russell's conception of a type. Now we see types in the curriculum from the school level through university education. Sets are taught in mathematics, types in programming. In modern type theory, we see sets (in their infinite variety) as a special data type that appeals to mathematicians. So we take a brief historical look at types to explain their use in proof assistants.[2]

1.3 Types in programming and logic

The notion of type is a central organizing concept in the design of programming languages, both to define the *data types* and also to determine the range of significance of procedures and functions.[3] Types feature critically in reasoning about programs as Hoare noted in his fundamental paper on data types [33]. The role of types in programming languages is evident in Algol 60 [59] and its successors such as Pascal and Algol 68 (where types were called *modes*). One of the most notable modern examples is the language ML, standing for MetaLanguage, designed by Milner as an integral part of the *Edinburgh LCF* mechanized *Logic for Computable Functions* [29]. This ML programming language with its remarkably effective *type inference algorithm* and its recursive data types is widely taught in computer science and is central to a large class of proof assistants.

Type theory serves as the logical language of several interactive theorem provers including Agda [12], Coq [19, 4], HOL [28], Isabelle [48], MetaPRL [32], Minlog [3], Nuprl [3, 1], PVS [51], and Twelf [52]. Among these provers, the constructive ones build *correct by construction* software. All of them formalize mathematical theories whose logical correctness is assured to the highest standards of certainty ever achieved. Interactive provers have also been used to solve open mathematical problems, e.g. definitively proving the Four Color Theorem [27]. The accumulation of large libraries of formalized mathematical knowledge using proof assistants has led to the field of *mathematical knowledge management.* Constructive type theories for *constructive and intuitionistic mathematics* serve as practical programming languages, a role imagined forty years ago [47, 30, 15] yet only recently made practical in several settings. The programming language ML also provides the metalanguage for several of the interactive proof assistants in service today, including Agda, Coq, HOL, Isabelle, MetaPRL, Nuprl among others.

2 Automated Reasoning in Constructive Type Theories

The gap between data types in programming languages and those in *foundational type theory* and from there to *constructive type theory* are large. That first gap can be traced to the influence of *Principia Mathematica* and the second to the influence of the mathematician L.E.J. Brouwer

[2] I wrote a longer unpublished paper in 2010 for the 100 year anniversary of *Principia Mathematica* entitled "The Triumph of Types: *Principia Mathematica's* Impact on Computer Science". It describes the influence of type theory on computer science and is available at the celebration web page. These notes rephrase a few examples used there.

[3] This use matches Russell's definition of a type as the *range of significance* of a propositional function.

and his intuitionistic program for mathematics. We look very briefly at these historical gaps to establish the context for the work of modern proof assistants supporting applied formal theories and correct-by-construction programming.

2.1 Comprehensive Mathematical Theories

Near the turn of the last century, Frege [24, 22], Russell [55] and Whitehead [58] strove to *design of a logical foundation for mathematics* free from the known paradoxes and able to support an extremely precise comprehensive treatment of mathematics in an axiomatic logical language. *Principia Mathematica (PM)* was created with those goals in mind, intended to be *safe enough* to avoid paradox and *rich enough* to express all of the concepts of modern pure mathematics of its time in a language its authors regarded as *pure logic.*

For Russell and Whitehead, type theory was not introduced because it was interesting on its own, but because it served as a tool to make logic and mathematics safe. According to *PM* page 37: Type theory "only recommended itself to us in the first instance by its ability to solve certain contradictions. ... it has also a certain consonance with common sense which makes it inherently credible". This common sense idea was captured in Russell's definition of a type in his *Principles of Mathematics, Appendix B The Doctrine of Types* [56] where he says "Every propositional function $\phi(x)$ – so it is contended – has, in addition to its range of truth, a range of significance, i.e. a range within which x must lie if $\phi(x)$ is to be a proposition at all,...." It is interesting that later in computer science, types are used precisely in this sense: to define the *range of significance of functions* in programming languages. It is also interesting that Frege devised a judgment form: $-A$ to indicate that the syntactic term A made sense or was a proposition. The notation $\vdash A$ is the judgment that proposition A is provable. Both of these judgment forms were used in ITT82 and then adopted in CTT84. These forms have proved to be extremely important in practice. In CTT84 they are used to exploit the Turing completeness of the computation system and thus produce efficient extracted code, suitable for practical use.

According to *PM*, statements of *pure mathematics* are inferences of pure logic. All commitments to "reality" (Platonic or physical) such as claims about infinite totalities, the interpretation of implication as a relation, the existence of Euclidean space, etc. were taken as hypotheses. At the time of *PM* it appeared that there would emerge settled agreement about the nature of pure inferences and their axiomatization. That was not to be. Even the notion of pure logic would change.

2.2 Computation in logic and mathematics

While Russell was working on his *theory of types* [55], circa 1907-1908, another conception of logic arose in the writings of L.E.J. Brouwer [31, 57] *circa* 1907, a conception that would depart radically from the vision of Frege and Russell, as they departed from Aristotle. By the early 1930's a mature expression of a new semantics of the logical operators emerged from the work of Brouwer, Heyting, and Kolmogorov; it is now called the *BHK semantics* for intuitionistic versions of formalisms originally developed based on truth-functional semantics. BKH semantics is also called the *propositions-as-types principle.* By 1945 Kleene captured this semantics for first-order logic and Peano arithmetic in his notion of recursive *realizability* based on general recursive functions [35]. By 1968, a formal version of a comprehensive theory of types based on the *propositions as types principle* was implemented in the *Automath* theories of de Bruijn and his colleagues [20]. Unlike Kleene's work, these theories did not take advantage of the computational interpretation of the logical primitives made possible by BHK, instead treating them formally as rules in the style of *PM*.

This line of research eventually led to two extensional type theories, Intuitionistic Type Theory (ITT82, ITT84) [41, 42] and *Constructive Type Theory (CTT84)* [17] related to ITT82. Subsequently the type theory of the *Calculus of Constructions (CoC)* [18] was designed based on Girard's logic [25, 26]. Over the next few years, CoC moved closer to the ITT82 conception and also built on the CTT82 recursive types [45, 46, 17] to produce the widely used (intensional) type theory *Calculus of Inductive Constructions (CIC)* [19] implemented in the Coq prover [4]. Then an intensional version of ITT82 was defined [49], say ITT90, and implemented much later in the Alf and Agda provers [12]. All three of these efforts, but especially CTT84 and ITT82, were strongly influenced by *Principia* and the work of Bishop [11] presented in his book *Foundations of Constructive Analysis* [11] who set about doing real analysis in the constructive tradition.

Computational content is present in the provable assertions of the constructive type theories, and one of the features of CIC, CTT, and ITT is that implementing the proof machinery makes it possible to *extract* this content and execute it. This discovery has led to a new proof technology for *extracting computational content* from assertions. That technology has proven to be very effective in practice, where it is called correct-by-construction programming. There is a growing literature on this subject with references in survey articles such as [1].

2.3 Effectively Computable, Turing Computable, and Subrecursive Computation Systems

Brouwer's notion of computability was not formal and not axiomatic. It was intuitive and corresponds to what is called *effective computability.* The *Church/Turing Thesis* claims that all effectively computable functions are computable by Turing machines (or any equivalent formalism, e.g. the untyped λ-calculus). There is no corresponding formalism for *Brouwer Computable.* However, I believe that this notion can be captured in intuitionistic logics by leaving a Turing complete computation system for the logic *open-ended* in the sense that new primitive terms and rules of reduction are possible. This method of capturing effective computability may be unique to CTT in the sense that the computation system of CTT is open to being "Brouwer incomplete" as a logic. We have recently added a primitive notion of *general process* to formalize distributed systems whose potentially nonterminating computations are not entirely effective because they depend on asynchronous message passing over a network which can only be modeled faithfully by allowing unpredictable choices by the *environment*, e.g. the internet.

3 A Case Study in Theory Exploration: the Logic of Events

The PRL group's most practical results in formal methods might be the optimization of protocol stacks [39, 36]. The optimization system was adopted by Nortel as a product because it could significantly improve the performance of a distributed system by using automatic compression techniques that we proved were safe. That is, the optimized protocols produced identical results to the original but over five times faster.

Based on our work in this area, we came to see that writing correct distributed protocols was an extremely difficult business. Basically no one can write these protocols completely correctly "by hand", so they are all gradually debugged in the field. This became an excellent target on which to deploy formal methods, in particular correct-by-construction programming. To start this we needed a specification language for distributed system behavior. We noticed that our collaborators used *message sequence diagrams* as their main informal method of description and reasoning. They did not use temporal logic in any form, they did not use any process algebra, they did not use a logic of actions. So we decided to create a logic that matched their way of thinking as closely as possible. As we spoke it with them, it gradually evolved. The tools of the Nuprl proof assistant and its Logical Programming Environment greatly aided that evolution in ways that I will explain *as an example of theory exploration in action.*

3.1 A sequence of theories

The interested reader could track the evolution of the logic of events from the first publications in 2003 to the latest publications in 2012, [8, 13, 10, 9, 5, 6, 7][4] All stages of this evolution were formalized and applied. They illustrate *theory exploration* in detail. I will summarize the nature of the changes briefly here. In the lecture associated with this article, I will tell some of the stories about why the changes were made and the formal tools used to manage them.

3.2 Summarizing an example of theory exploration and evolution

In *A Logic of Events* [8] circa 2003, we defined together a logic and a computing model. The computing model was based closely on the IO Automata from the book of Nancy Lynch, *Distributed Algorithms*[40]. We called our machines *Message Automata* (MA) because they differed from IOA in the way they composed; we added *frame conditions* to the automata to facilitate reasoning about composition. We also provide addresses which identified the machines and explicit communication channels. We intermixed axioms about events with axioms about the machines, and proved that these axioms were validated by the standard model of computing for the automata. This model was close to that presented by Lynch and also the model used in other textbooks [2]. We used axioms about sending and receiving messages, about state change, and ordering of events, about kinds of events, about how processes represented as Message Automata were composed, and how their computations validated the axioms. Later in *A Causal Logic of Events* [10] circa 2005, we classified the event logic axioms more finely in a progression, starting with axioms for event ordering, then events with values, then events relative to states, and so on.

We learned that the key axiom for many proofs was that *causal ordering* [37] of events is well founded. We could justify this property based on the computation model, and we made that an explicit axiom. We used it to formally prove many important theorems and derive several simple protocols. As we began to specify and prove properties about complex consensus protocols, we discovered that small changes in the protocol would cause disproportionate expansion of our proofs which had to be redone by hand. So we modified the theory to raise the level of abstraction and increase the amount of automation connecting layers of abstraction. The discovery of the notion of an *event class* by 2008 [5] allowed us to postpone reasoning about specific automata until we had the abstract argument finished at this much higher level. The event classes eventually led us to a more abstract programming notation called *event combinators* that we have now implemented directly [7].

By 2010 we could automatically generate proofs from the higher level event classes to the lowest level process code, now implemented in our new programming interface to Nuprl built

[4] This style has the unintended consequence of citing a very large number of our own papers.

by Vincent Rahli called EventML [54]. We were able to create domain specific tactics that would automate in a matter of a day or two the work that previously took us a month or two of hand proofs. By exploiting the distributed implementation of Nuprl, we can now run these domain specific proof tactics in a matter of a few hours by using a cluster of many multi-core processors.

During the period from 2005 to 2009 we learned to modify and extend the entire theory step by step and replay all the proofs. Mark Bickford and Richard Eaton developed mechanisms to replay the proofs, find the broken steps, and modify the tactics and then replay the entire event theory library.

I can only cover one point of this long and rich technical story in this short article. We turn to that one point next. In due course we will publish more "chapters" of the story now that we have deployed our synthesized consensus protocols in a working system called ShadowDB and have started collecting data about its performance [50]. We will also be studying how this system adapts to attack by exploiting *synthetic code diversity*, discussed briefly later in this article.

3.3 Evolving understanding – causal order

We now look briefly at how our understanding of causal order changed as the theories evolved. The same kind of story could be told about minimizing the role of state, about the need for event combinators, and about the need for sub-processes that live only for short periods. All of this would take a monograph. So I will focus on only one such aspect.

Lamport discovered in 1978 that causal order was the right notion of time in distributed systems [37]. Intuitively, an event e comes before e' if e can influence e'. We give the definition below.

To show that causal order is well-founded, we notice that in any execution of a distributed system given by reduction rules, we can conduct a "tour of the events" and number them so that events caused by other events have a higher number. Thus causal order is well founded and if equality of events is decidable, then so is causal order. We show this below.

3.3.1 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?: $E \rightarrow E + Loc$
sender?: $E \rightarrow E + Unit$

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. 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

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

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

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

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

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

Causal order is $x\ pred!^*y,\ abbreviated\ x <\ y.$

3.3.2 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 justify f in Axiom 3 we arrange a linear "tour" of the event space in any computation. We 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 $< e, prior(pred?(e)) >$
```

3.3.3 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 3.1. $\exists f : E \rightarrow \mathbb{N}. \forall e, e' : E.\ \ e < e' \Rightarrow f(e) < f(e')$

The argument is simple. Let $x \lhd y$ denote $\mathit{pred!}(x, y)$ and let $x \lhd^n y$ denote $\mathit{pred!}^n(x, y)$. Recall that $x \lhd^{n+1} y$ iff $\exists z : E.\ x \lhd z \ \&\ z \lhd^n y$. From Axiom 3 there is function $f_o : E \rightarrow \mathbb{N}$ 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 3.2. $\forall P : E \rightarrow \mathit{Prop}.\ \forall e' : E.\ ((\forall e : E.\ e < e'.\ P(e)) \Rightarrow P(e')) \Rightarrow \forall e : E.\ P(e)$

Using induction we can prove that causal order is decidable.

Theorem 3.3. $\forall e, e' : E.\ e < e' \vee \neg\ (e < e')$

We need the lemma.

Theorem 3.4. $\forall e, e' : E.\ (e \lhd e' \vee \neg\ (e \lhd e'))$

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

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

and

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

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

Theorem 3.5. $\forall x, y : E.\ \ (x <_{loc} y \vee x = y \vee y <_{loc} x)$

The environment as well as the processes are players in the execution, determining whether protocols converge or not and whether they converge sufficiently fast. However, unlike the processes, the environment is not a Turing computable player. It's behavior was not lawlike. This means that the reduction rule semantics should not be taken as the whole story, and the justification of causal order based on the computations is not the fundamental explanation, only a suggestive approximation. The best explanation are the axioms that define what the environment can contribute, and the induction principle on causal order.

3.4 The formal distributed computing model - circa 2012

Here is a brief overview of our General Process Model circa 2010 [6]. From it we can generate structures we call *event orderings*. We mention key concepts for reasoning about event orderings from these computations. A *system* consists of a set of *components*. Each component has a *location*, an *internal* part, and an *external* part. Locations are just abstract identifiers. There may be more than one component with the same location.

The internal part of a component is a *process*—its program and internal (possibly hidden) state. The external part of a component is its interface with the rest of the system. In this account, the interface will be a list of *messages*, containing either *data* or processes, labeled with the location of the recipient. The "higher order" ability to send a message containing a process allows a system to grow by "forking" or "bootstrapping" new components.

A system executes as follows. At each step, the *environment* may choose and remove a message from the external component. If the chosen message is addressed to a location that is not yet in the system, then a new component is created at that location, using a given *boot-process*, and an empty external part. Each component at the recipient location receives the message as input and computes a pair that contains a process and an external part. The process becomes the next internal part of the component, and the external part is appended to the current external part of the component.

A potentially infinite sequence of steps, starting from a given system and using a given boot-process, is a *run* of that system. From a run of a system we derive an abstraction of its behavior by focusing on the *events* in the run. The events are the pairs, $\langle x, n \rangle$, of a location and a step at which location x gets an input message at step n, i.e. information is transferred. Every event has a location, and there is a natural *causal-ordering* on the set of events, the ordering first considered by Lamport [37]. This allows us to define an *event-ordering*, a structure, $\langle E,\ loc,\ <,\ info \rangle$, in which the causal ordering $<$ is transitive relation on E that is well-founded, and locally-finite (each event has only finitely many predecessors). Also, the events at a given location are totally ordered by $<$. The information, $info(e)$, associated with event e is the message input to $loc(e)$ when the event occurred.

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

Example 1. *A simple consensus protocol: TwoThirds*

Each participating component will be a member of some groups and each group has a name, G. A message $\langle G, i \rangle$ from the environment to component i informs it that it is in group G. The groups have $n = 3f + 1$ members, and they are designed to tolerate f failures. When any component in a group G receives a message $\langle [start], G \rangle$ it starts the consensus protocol whose goal is to decide on values received by the members from *clients*. We assume that once the protocol starts, each process has received a value v_i or has a default non-value.

The simple TwoThirds consensus protocol is this: A process P_i that has a value v_i of type T starts an *election* to choose a value of type T (with a decidable equality) from among those received by members of the group from clients. The elections are identified by natural numbers, el_i initially 0, and incremented by 1, and a Boolean variable $decide_i$ is initially $false$. The function from lists of values, Msg_i to a value is a parameter of the protocol. If the type T of values is Boolean, we can take f to be the majority function.

```
**Begin**
**Until** $decide_i$ **do**:
```

```
1. Increment $el_i$; 2. Broadcast vote $\langle el_i, v_i \rangle$ to $G$;
3. Collect in list $Msg_i$ $2f+1$ votes of election $el_i$;
4. Choose $v_i := f(Msg_i)$;
5. If $Msg_i$ is unanimous then $decide_i := true$
End
```

We describe protocols like this by classifying the events occurring during execution. In this algorithm there are *Input*, *Vote*, *Collect*, and *Decide* events. The components can recognize events in each of these *event classes* (in this example they could all have distinctive headers), and they can associate information with each event (e.g. $\langle e_i, v_i \rangle$ with Vote, Msg_i with Collect, and $f(Msg_i)$ with Decide). Events in some classes *cause* events with related information content in other classes, e.g. Collect causes a Vote event with value $f(Msg_i)$.

In general, an *event class* X is function on events in an event ordering that *effectively partitions* events into two sets, $E(X)$ and $E - E(X)$, and assigns a value $X(e)$ to events $e \in E(X)$.

Example 2. *Consensus specification*

Let P and D be the classes of events with headers *propose* and *decide*, respectively. Then the *safety specification* of a consensus protocol is the conjunction of two propositions on (extended) event-orderings, called *agreement* (all decision events have the same value) and *responsiveness* (the value decided on must be one of the values proposed):

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

It is easy to prove about TwoThirds the safety property that if there are two decide events, say $Decide(e)$ and $Decide(e')$, then $Decide(e) = Decide(e')$. We can also show that if $Decide(e_1) = v$, then there is a prior input event, e_0 such that $Input(e_0) = Decide(e_1)$.

We can prove safety and the following *liveness property* about TwoThirds. We say that activity in the protocol *contracts to a subset* S of exactly $2f+1$ processes if these processes all vote in election n say at $vt(n)_1, ..., vt(n)_k$ for $k = 2f{+}1$ and collect these votes at $c(n)_1, ..., c(n)_k$, and all vote again in election $n{+}1$ at $vt(n{+}1)_1, ..., vt(n{+}1)_k$, and collect at $c(n{+}1)_1, ..., c(n{+}1)_k$. In this case, these processes in S all decide in round $n+1$ for the value given by f applied to the collected votes. This is a *liveness* property.

If exactly f processes fail, then the activity of the group G contracts to some S and decides. Likewise if the message traffic is such that f processes are delayed for an election, then the protocol contracts to S and decides. This fact shows that the TwoThirds protocol is *non-blocking*, i.e. from any state of the protocol, there is a path to a decision. We can construct the path to a decision given a set of f processes that we delay.

We also proved safety and liveness of a variant of this protocol that may converge to consensus faster. In this variant, if P_i receives a vote $\langle e, v \rangle$ from a higher election, $e > el_i$, then P_i joins that election by setting $el_i := e; v_i := v$; and then going to step 2.

3.5 Attack Tolerant Distributed Systems

Networked systems will be attacked. One way to protect them is to enable them to adapt to attacks. One way to adapt is to change the protocol, but this is dangerous, especially for the most critical protocols. The responsible system administrators don't ever want to modify the most critical protocols such as consensus. One alternative is to switch to another provably equivalent protocol, a task familiar to us from the stack optimization work. So we proposed to formally generate a large number of *logically equivalent variants*, store them in an *attack response library*, and switch to one of these variants when an attack is detected or even pro-actively. Each variant uses distinctly different code which a system under attack can install *on-the-fly* to replace compromised components. Each variant is known to be equivalent and correct.

We express threatening features of the environment formally and discriminate among the different types. We can do this in our new GPM model because *the environment is an explicit component about which we can reason.* This capability allows us to study in depth how diversity works to defend systems. We can implement attack scenarios as part of our experimental platform. It is interesting that we can implement a *constructive version* [16] of the famous FLP result [23] that shows how an attacker can keep any deterministic fault-tolerant consensus protocol from achieving consensus.

Synthetic Code Diversity We introduce diversity at all levels of the formal code development (synthesis) process starting at a very high level of abstraction. For example, in the TwoThirds protocol, we can use different functions f, alter the means of collecting Msg_i, synthesize variants of the protocol, alter the data types, etc. We are able to create multiple provably correct versions of protocols at each level of development, e.g. compiling TwoThirds into Java, Erlang, and $F^{\#}$. The higher the starting point for introducing diversity, the more options we can create.

Acknowledgement The ideas reported here are based on the work of the PRL research group, in particular our research on the logic of events by Mark Bickford, Richard Eaton, and Vincent Rahli, and by our collaboration with David Guaspari of ATC-NY corporation. Our formal work was strongly influenced by a long term collaboration with Robbert van Renesse and Ken Birman of the Cornell systems group and the recent work of Nicolas Schiper.

References

[1] S. Allen, M. Bickford, R. L. Constable, R. Eaton, C. Kreitz, L. Lorigo, and E. Moran. Innovations in computational type theory using Nuprl. *Journal of Applied Logic*, 4(4):428–469, 2006.

[2] H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics.* Wiley Interscience, New York, 2nd edition, 2004.

[3] H. Benl, U. Berger, H. Schwichtenberg, M. Seisenberger and W. Zuber Proof theory at work: Program development in the Minlog system. In W. Bibel and P. G. Schmitt, editors, *Automated Deduction*, volume II. Kluwer, 1998.

[4] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development; Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. Springer-Verlag, 2004.

[5] M. Bickford. Component specification using event classes. In *Lecture Notes in Computer Science 5582*, pages 140–155. Springer, 2009.

[6] M. Bickford, R. L. Constable, and D. Guaspari. Constructing event structures over a general process model. Department of Computer Science TR1813-23562, Cornell University, Ithaca, NY, 2011.

[7] M. Bickford, R. L. Constable, and V. Rahli. The logic of events, a framework to reason about distributed systems. Technical report.

[8] M. Bickford and R. L. Constable. A logic of events. Technical Report TR2003-1893, Cornell University, 2003.

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

[10] M. Bickford and R. L. Constable. A causal logic of events in formalized computational type theory. In *Logical Aspects of Secure Computer Systems, Proceedings of International Summer School Marktoberdorf 2005*, to Appear 2006. Earlier version available as Cornell University Technical Report TR2005-2010, 2005.

[11] E. Bishop. *Foundations of Constructive Analysis.* McGraw Hill, NY, 1967.

[12] A. Bove, P. Dybjer, and U. Norell. A brief overview of Agda – a functional language with dependent types. In C. Urban M. Wenzel S. Berghofer, T. Nipkow, (eds.), *LNCS 5674, Theorem Proving in Higher Order Logics*, pages 73–78. Springer, 2009.

[13] R. L. Constable. Information-intensive proof technology. In H. Schwichtenberg and K. Spies, editors, *Proof Technology and Computation.* Kluwer, Amsterdam, 2005. 42 pages to appear.

[14] R. L. Constable and M. Bickford. Intuitionistic Completeness of First-Order Logic. Technical Report arXiv:1110.1614v3, Computing and Information Science Technical Reports, Cornell University, 2011.

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

[16] R. L. Constable. Effectively nonblocking consensus procedures can execute forever: a constructive version of flp. Technical Report Tech Report 11512, Cornell University, 2008.

[17] R. L. Constable, S. 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, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the* NUPRL *Proof Development System.* Prentice-Hall, NJ, 1986.

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

[19] T. Coquand and C. Paulin-Mohring. Inductively defined types, preliminary version. In *COLOG '88, International Conference on Computer Logic*, volume 417 of *LNCS*, pages 50–66. Springer, 1990.

[20] N. G. de Bruijn. The mathematical language Automath: its usage and some of its extensions. In J. P. Seldin and J. R. Hindley, editors, *Symposium on Automatic Demonstration*, volume 125 of *Lecture Notes in Mathematics*, pages 29–61. Springer-Verlag, 1970.

[21] R. A. De Millo, R. J. Lipton, and A. J. Perlis. Social processes and proofs of theorems and programs. *Communications of the Association of Computing Machinery*, 22:271–280, 1979.

[22] M. Dummett. *Frege Philosophy of Mathematics.* Harvard University Press, Cambridge, MA, 1991.

[23] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faculty process. *JACM*, 32:374–382, 1985.

[24] G. Frege. Begriffsschrift, a formula language, modeled upon that for arithmetic for pure thought. In J. van Heijenoort, (ed), *From Frege to Gödel: A Source Book in Mathematical Logic, 1879–1931*, pages 1–82. Harvard University Press, Cambridge, MA, 1967.

[25] J-Y. Girard. Une extension de l'interpretation de Gödel a l'analyse, et son application a l'elimination des coupures dans l'analyse et la theorie des types. In *2nd Scandinavian Logic Symposium*, pages 63–69. Springer-Verlag, NY, 1971.

[26] J-Y. Girard. The system F of variable types: Fifteen years later. *Journal of Theoretical Computer Science*, 45:159–192, 1986.

[27] G. Gonthier. Formal proof – the four color theorem. *Notices of the American Math Society*, 55:1382–1392, 2008.

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

[29] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF: a mechanized logic of computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, NY, 1979.

[30] C. C. Green. An application of theorem proving to problem solving. In *IJCAI-69—Proceedings of the 1^{st} International Joint Conference on Artificial Intelligence*, pages 219–239, Washington, DC, May 1969.

[31] A. Heyting (ed) *L. E. J. Brouwer. Collected Works*, volume 1. North-Holland, Amsterdam, 1975. (see On the foundations of mathematics 11-98.).

[32] J. J. Hickey. *The MetaPRL Logical Programming Environment*. PhD thesis, Cornell University, Ithaca, NY, January 2001.

[33] C. A. R. Hoare. Notes on data structuring. In *Structured Programming*. Academic Press, New York, 1972.

[34] A. Hodges. Beyond Turing's Machines. *Science*, 336, April 2012.

[35] S. C. Kleene. On the interpretation of intuitionistic number theory. *J. of Symbolic Logic*, 10:109–124, 1945.

[36] C. Kreitz. Building reliable, high-performance networks with the Nuprl proof development system. *JFP*, 14(1):21–68, 2004.

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

[38] X. Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Proceedings of the 33d ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 42–54. ACM Press, 2006.

[39] X. Liu, C. Kreitz, R. van Renesse, J. J. Hickey, M. Hayden, K. Birman, and R. L. Constable. Building reliable, high-performance communication systems from components. In D. Kotz and J. Wilkes (eds), *17^{th} ACM Symposium on Operating Systems Principles (SOSP'99)*, volume 33(5) of *Operating Systems Review*, pages 80–92. ACM Press, December 1999.

[40] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, San Mateo, CA, 1996.

[41] P. 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.

[42] P. Martin-Löf. *Intuitionistic Type Theory*. Number 1 in Studies in Proof Theory, Lecture Notes. Bibliopolis, Napoli, 1984.

[43] R. Matuszewski and P. Rudnicki. Mizar: the first 30 years. *Mechanized Mathematics and its Applications*, 4:8–14, 2005.

[44] J. McCarthy. A basis for a mathematical theory of computation. In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems*, pages 33–70. North-Holland, Amsterdam, 1963.

[45] P. F. Mendler. Recursive types and type constraints in second-order lambda calculus. In D. Gries, editor, *Proceedings of the 2^{nd} IEEE Symposium on Logic in Computer Science*, pages 30–36. IEEE Computer Society Press, June 1987.

[46] P. F. Mendler. *Inductive Definition in Type Theory*. PhD thesis, Cornell University, Ithaca, NY, 1988.

[47] D. Michie. *Machine Intelligence 3*. American Elsivier, New York, 1968.

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

[49] B. Nordström, K. Petersson, and J. M. Smith. *Programming in Martin-Löf's Type Theory*. Oxford Sciences Publication, Oxford, 1990.

[50] N. Schiper, V. Rahli, R. van Renesse, and R. L. Constable M. Bickford. Shadowdb: A replicated database on a synthesized consensus core. Technical report, Computer Science Department, Cornell University, Ithaca, NY, 2012.

[51] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *Proceedings of the 11th International Conference on Automated Deduction*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.

[52] F. Pfenning and C. Schürmann. Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction*, volume 1632, pages 202–206. Trento, Italy, July 7–10 1999.

[53] B. C. Pierce, C. Casinghino, M. Greenberg, V. Sjberg, and B. Yorgey. *Software Foundations.* Electronic, 2011.

[54] V. Rahli. New tools for domain specific verified programming. Technical Report 1000, Cornell Computing and Information Science, 2012.

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

[56] B. Russell. *The Principles of Mathematics.* Cambridge University Press, Cambridge, 1908.

[57] Wa. P. van Stigt. *Brouwer's Intuitionism.* North-Holland, Amsterdam, 1990.

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

[59] N. Wirth and C. A. R. Hoare. A contribution to the development of ALGOL. *Communications of the ACM*, 9:413–432, 1966.