Nuprl’s Inductive Logical Forms

Mark Bickford, Robert L. Constable, Rich Eaton and Vincent Rahli∗
Cornell University

Abstract

For more than a decade, we have been working on specifying, verifying, and synthesizing asynchronous distributed protocols using the Nuprl proof assistant. Only recently we have been able to do so for complicated protocols such as Paxos. This has been made possible thanks to the level of abstraction of our specification language called the Logic of Events (LoE). This paper discusses our main automation tool, namely our Inductive Logical Forms (ILFs), which are first order formulas that characterize the responses of a system to events in terms of observations made at causally prior events.

Nuprl’s type theory. The Nuprl proof assistant [12, 3] implements a type theory called Constructive Type Theory (CTT), which is an extensional dependent type theory à la Martin-Löf. It is based on an untyped functional programming language à la Curry. Nuprl has a rich type theory including dependent product and sum types, identity (or equality) types, a hierarchy of universes, disjoint union types, W types, quotient types [14], set types, union and (dependent) intersection types [15, 22], image types [25], PER types [4], approximation and computational equivalence types [19, 28], and partial types [16].

Type checking is undecidable but in practice this is mitigated by type inference and checking heuristics implemented as tactics. Moreover, to avoid proving well-formedness conditions (i.e., doing type checking) altogether, we often do untyped reasoning using Howe’s computational equivalence relation (an observational congruence, which we write as ∼) [19, 28]. For example, we can prove that for all terms f and t (t need not be a list), map(f, t) @ nil ∼ map(f, t). We can then rewrite map(f, t) @ nil into map(f, t) anywhere in a sequent without having to prove any well-formedness condition.

Allen developed a semantics for Nuprl where types are defined as partial equivalence relations (PERs) on closed terms [2, 1, 16]. We have implemented this semantics in Coq and verified a large number of Nuprl’s inference rules [6, 7] (Nuprl’s consistency follows from the fact that its inference rules are valid w.r.t. Allen’s PER semantics and that False is not inhabited).

We have also recently turned Nuprl into a fully intuitionistic theory following Brouwer’s principles. Using our Coq framework, we have proved Brouwer’s continuity principle for numbers [27]. Following Dummett’s “standard” classical proof [17, pp.55], we have also proved (in Prop and using classical axioms) the truth of several bar induction rules (see, e.g., [21, 11, 31]), such as bar induction on decidable bars for free choice sequences of numbers [26].

By default the Nuprl system is distributed and runs in the cloud. Alternatively, Nuprl can run locally using one of our virtual machines available at http://www.nuprl.org/vms/ Nuprl is composed of several processes: database and library processes to store and access definitions, lemmas, and proofs; refiner processes to apply inference rules; and editor processes, which are user interfaces. It also allows several users to simultaneously edit a proof, and users to simultaneously refine several subgoals in a proof.

Nuprl led to other similar proof assistants such as MetaPRL [18] and JonPRL [20]. It has been used over the years to both formalize mathematical results and develop verified software.

∗This work was partially supported by the DARPA CRASH project, award number FA8750-10-2-0238, by the SnT, and by the National Research Fund Luxembourg (FNR), through PEARL grant FNR/F14/8149128.
For example, we have recently proved a completeness result of intuitionistic first-order logic \cite{13}, and we have built distributed systems including a verified ordered broadcast service \cite{33,30,32}.

**The Logic of Events.** To specify, verify, and synthesize asynchronous distributed protocols, we use two languages, both implemented in Nuprl: (1) a high-level specification language called the Logic of Events (LoE) \cite{8,10} to specify and reason about the information flow of distributed program runs; and (2) a low-level programming language, the General Process Model (GPM) \cite{9}, to implement these information flows.

LoE, related to Lamport’s notion of causal order \cite{23}, was developed to reason about events occurring in the execution of a distributed system, where an event is an abstract entity corresponding to the receipt of a message; the message is called the *primitive information* of the event. An event happens at a specific point in space/time. The space coordinate of an event is called its location, and the time coordinate is given by a well-founded causal ordering on events that totally orders all events at the same location. Using LoE one can describe systems in terms of the causal relations among events and (ultimately) their primitive information. LoE has been used among other things to verify consensus protocols \cite{33} and cyber-physical systems \cite{3}.

We have also developed a programming language called EventML, to automatically generate both LoE specifications and GPM programs from EventML specifications \cite{29}. Once we have extracted the semantic meaning of an EventML specification in terms of a LoE formula \(F\) and a GPM program \(P\), we automatically prove that \(P\) satisfies \(F\). It remains to interactively prove that the LoE formula \(F\) satisfies the desired correctness properties.

To reason about a protocol in LoE, we reason about its possible runs. An *event ordering* is an abstract representation of one run of a distributed system; it provides a formal definition of a *message sequence diagram* as used by systems designers. It is a structure consisting of: (1) a set of events; (2) a function \(\text{loc}\) that associates a *location* with each event; (3) a function \(\text{info}\) that associates primitive information with each event; and (4) a well-founded *causal ordering* relation, \(<\), on events \cite{23}. We express system properties as predicates on event orderings. A system satisfies such a property if every execution satisfies the predicate.

The message sequence diagram on the right depicts a simple event ordering with events \(e_1\) and \(e_3\) happening at location \(L_1\), and \(e_2\) at location \(L_2\). Event \(e_1\) happens causally before \(e_2\), which happens causally before \(e_3\). We write \(e_1 < e_2, e_2 < e_3,\) and \(e_1 <_{\text{loc}} e_3\).

In LoE, we specify systems by defining and combining event observers \cite{8} (which can be regarded as the combinations of event recognizers and event handlers). An event observer is a function that assigns to any event ordering \(eo\) and event \(e\) in that event ordering \(eo\), an unordered bag of outputs observed (or produced) at \(e\). For example, the following observer recognizes every event and observes its location: \(\lambda eo.\lambda e.\{\text{loc}(e)\}\). We also have primitive observers to, e.g., run two processes in parallel or to build state machines.

We reason about event observers in terms of the *event observer relation*, which relates events, observers, and observations: we say that the observer \(X\) observes \(v\) at event \(e\) (in an event ordering \(eo\)), and write \(v \in X(e)\), if \(v\) is a member of the bag \((X eo e)\).

Formally verifying distributed protocols is not trivial and can be time consuming. Our main automation tool to assist us in this task is called an Inductive Logical Form (ILF).

**Inductive Logical Forms.** An ILF is a first order formula that characterizes the responses of a system to events in terms of observations made at causally prior events. For example, in Paxos \cite{24,34}—a state machine replication protocol, if a leader \(L\) decides that slot \(n\) is to be filled with command \(c\), then that means that \(c\) was proposed to the leader \(L\) at an earlier event.

ILFs are automatically generated from observers using logical simplifications, and charac-
terizations of the LoE combinator. For example, one of the simplest but subtle such characterizations is the one for our parallel combinator “\(-\mid -\)”, which allows one to run two processes in parallel: $v \in X \upharpoonright Y(e) \iff (v \in X(e) \lor v \in Y(e))$. This says that $v$ is produced by $X \upharpoonright Y$ if it is produced by either of its components (see 29 for more details).

Given an observer $X$, i.e. an LoE specification, we wrote a program that starts with a formula of the form $v \in X(e)$, and keeps on rewriting it using equivalences such as the one provided above (and also applies various logical simplifications), to finally generate a formula of the form $v \in X(e) \iff C$, where $C$ is a complete declarative characterization of $X$’s outputs. Finally, we have built a proof tactic that automatically proves such double implications.

Rewriting using such formulas we can easily trace back the outputs of a distributed system to the states of its state machines and to its inputs. It turns out that when proving safety properties of distributed systems, much of the effort is spent tracing back outputs to inputs. ILFs helped us automate this reasoning process.

References


