# Knowledge-Based Synthesis of Distributed Systems Using Event Structures

Mark Bickford Robert C. Constable Joseph Y. Halpern
Sabina Petride
Department of Computer Science
Cornell University
Ithaca, NY 14853
{markb,rc,halpern,petride}@cs.cornell.edu

**Abstract**

To produce a program guaranteed to satisfy a given specification one can synthesize it from a formal constructive proof that a computation satisfying that specification exists. This process is particularly effective if the specifications are written in a high-level language that makes it easy for designers to specify their goals. We consider a high-level specification language that results from adding *knowledge* to a fragment of Nuprl specifically tailored for specifying distributed protocols, called *event theory*. We then show how high-level *knowledge-based programs* can be synthesized from the knowledge-based specifications using a proof development system such as Nuprl. Methods of Halpern and Zuck [1992] then apply to convert these knowledge-based protocols to ordinary protocols. These methods can be expressed as heuristic transformation tactics in Nuprl.

## 1 Introduction

Errors in software are extremely costly and disruptive. NIST (the National Institute of Standards and Technology) estimates the cost of software errors to the US economy at $59.5 billion per year. One approach to minimizing errors is to synthesize programs from specifications. Synthesis methods have produced highly reliable moderate-sized programs in cases where the computing task can be precisely specified. One of the most elegant synthesis methods is the use of so-called *correct-by-construction* program synthesis [Bates and Constable 1985; Constable et al. 1986; Smith and Green 1996; Kreitz 1998; Paulin-Mohring and Werner 1993; Geuvers et al. 2001; Geuvers et al. 2001]. Here programs are constructed from *proofs* that the specifications are satisfiable. That is, a constructive proof that a specification is satisfiable gives a program that satisfies the specification. This method has been successfully used by several research groups and companies to construct large complex *sequential* programs, but it has not yet been used to create substantial realistic distributed programs.

The Cornell Nuprl proof development system was among the first tools used to create correct-by-construction functional and sequential programs [Constable et al. 1986]. Nuprl has also been used extensively to optimize distributed protocols [Liu et al. 1999; Birman et al. 2000], and to specify them in the language of I/O Automata [Bickford et al. 2001; Bickford et al. 2001; Liu et al. 2001]. Recent work by two of the authors has resulted in the definition of a fragment of the higher-order logic used by Nuprl tailored to specifying distributed protocols, called *event theory*, and the extension of Nuprl methods to synthesize distributed protocols from specifications written in event theory [Bickford 2003; Bickford and Constable 2003].

Event logic is a specification language closely related to I/O automata. As has long been recognized [Halpern and Moses 1990], designers typically think of specifications at a high level, which often involves knowledge-based statements. For example, the goal of a program might be to guarantee that a

certain process knows certain information. It has been argued [Fagin et al. 1995; Fagin et al. 1997] that a useful way of capturing these high-level knowledge-based specifications is by using high-level *knowledge-based programs*. Knowledge-based programs are an attempt to capture the intuition that what an agent does depends on what it knows. For example, a knowledge-based program may say that process 1 should stop sending a bit to process 2 once process 1 knows that process 2 knows the bit. Such knowledge-based programs and specifications can be given precise semantics [Fagin et al. 1995; Fagin et al. 1997; Halpern 1999]. They have already met with some degree of success, having been used in papers such as [Dwork and Moses 1990; Hadzilacos 1987; Halpern et al. 2001; Halpern and Zuck 1992; Mazer and Lochovsky 1990; Mazer 1990; Moses and Tuttle 1988; Neiger and Toueg 1993; Stulp and Verbrugge 2002] both to help in the design of new protocols and to clarify the understanding of existing protocols.

In this paper, we add knowledge operators to event theory raising its level of abstraction and show by example that knowledge-based programs can be synthesized from constructive proofs that specifications in event theory with knowledge operators are satsifiable. Our example uses the *sequence-transmission problem*, where a sender must transmit a sequence of bits to a receiver in such a way that the receiver eventually knows abitrarily long prefixes of the sequence. Halpern and Zuck [1992] provide two knowledge-based programs for the sequence-transmission, prove them correct, and show that many standard programs for the problem in the literature can be viewed as implementations of their high-level knowledge-based program. Here we show that these two knowledge-based programs can be synthesized from the specifications of the problem, expressed in event theory augmented by knowledge. We can then translate the arguments of Halpern and Zuck to Nuprl, to show that the knowledge-based programs can be transformed to the standard programs in the literature.

Engelhardt, van der Meyden, and Moses [1998, 2001] have also provided techniques for synthesizing knowledge-based programs from knowledge-based specifications, by successive refinement. We see their work as complementary to ours. Since our work is based on Nuprl, we are able to take advantage of the huge library of tactics provided by Nuprl to be able to generate proofs. The expressive power of Nuprl also allows us to express all the high-level concepts of interest (both epistemic and temporal) easily. Engelhardt, van der Meyden, and Moses do not have a theorem-proving engine for their language. However, they do provide useful refinement rules that can easily be captured as tactics in Nuprl.

# 2 Synthesizing Distributed Programs From Constructive Proofs

## 2.1 Nuprl: a brief overview

Much current work on formal verification using theorem proving, including Nuprl, is based on type theory (see [Constable 2002] for a recent overview). A type can be thought of as a set with structure that facilitates its use as a data type in computation; this structure also supports constructive reasoning. The set of types is closed under constructors such as $\times$ and $\rightarrow$, so that if $A$ and $B$ are types, so are $A \times B$ and $A \rightarrow B$. where, intuitively, $A \rightarrow B$ represents the computable functions from $A$ into $B$. *Constructive* type theory, upon which Nuprl is based, was developed to provide a foundation for constructive mathematics. The key feature of constructive mathematics is that "there exists" is interpreted as "we can construct (a proof of)". A consequence of this approach is that, for example, the law of excluded middle does not hold.

At an abstract level, a program in Nuprl is just an object of some type `Pgm`. A *program semantics* is a function `S` $\in$ `Pgm` $\rightarrow$ `Sem` assigning to each *program* `pr` $\in$ `Pgm` a *meaning* in type `Sem`. A *semantic property* is a predicate `X` on meanings. We say that a *program* `pr` *satisfies a semantic property* `X` if `X` holds of the semantic meaning of `pr`. Formally, we write `pr |= X` as an abbreviation of `X (S pr)`; thus, this fact can be expressed in Nuprl. A semantic property `X` is *satisfiable* if there is some program that satisfies it. Satisfiability can also be expressed in Nuprl: we take `Sat(X)` to be an abbrevation for $\exists$`pr:Pgm.pr |= X`. The key point for the purposes of this paper is that from a

constructive proof of `Sat(X)`, we can *extract* a program that satisfies `X`.

For an instance of this general constructive framework to be useful *in practice*, the parameters `Pgm`, `Sem`, and `S` must be chosen so that (a) programs are concrete enough to be compiled, and (b) specifications are naturally expressed as predicates over `Sem`, and (c) there is a small set of *rules* for producing proofs of satisfiabilty. To use this general framework for synthesis of *distributed*, *asynchronous* algorithms, we choose the programs in `Pgm` to be *distributed message automata*.

Message automata are closely related to Lynch's *IO-Automata* [Lynch and Tuttle 1987; Lynch and Tuttle 1989] and are roughly equivalent to *UNITY* programs [Chandy and Misra 1988] (but with message-passing rather than shared-variable communication). We describe distributed message automata in Section 2.3. As we shall see, they satisfy criterion (a) above.

The semantics of a program is the *system*, or set of *runs*, consistent with it. Typical specifications in the literature are predicates on runs. We can view a specification as a predicate on systems by saying that a system satisfies a specification exactly if all the runs in the system satisfy it.

To satisfy criterion (b) above, we choose a formal definition of runs that builds in the fundamental order structure and provides the operators for appropriately abstract specifications. To do this we formalize runs as structures that we call *event structures*, much in the spirit of Lamport's [1978] model of events in distributed systems. Event structures are explained in more detail in the next section.

## 2.2 Event Structures

Following Lamport, we characterize a run of a program as a set of events. Each event is associated with a unique *agent* or *process* (we use the two words interchangeably in this paper). Formally, events are elements of a type `E`, and there is a function `agent` of type `E` $\rightarrow$`AG`, where `AG` is some set of agents. For each `i` $\in$ `AG`, the set of events `e` such that `agent(e) = i` is totally ordered. Intuitively, this set of events is the *history* of events at agent `i`. If `first(e)` holds, then `e` is the first event in the history associated with `agent(e)`; if not, then `e` has a predecessor `pred(e)`.

Events are further partitioned by their *kinds*. The `kind(e)` of event `e` is either `rcv(l)`—a receive event on *link* `l`, or else `local(a)`—a local event of kind `a`.[1] Every receive event has a *sender event* `sender(e)`. The sender event of a message is the event when the message was sent.

Following Lamport [1978], we can define a *causal order* on events as the transitive closure of the sender-receiver and predecessor relations. Thus, $\rightarrow$ is the least relation on events such that $\mathtt{e} \rightarrow \mathtt{e}'$ if

- $\mathtt{e}'$ is a receive event and `e` is the corresponding send event,
- $\mathtt{agent}(\mathtt{e}) = \mathtt{agent}(\mathtt{e})'$ and `e` precedes $\mathtt{e}'$ in the total order associated with `agent(e)`, or
- for some event $\mathtt{e}''$ we have $\mathtt{e} \rightarrow \mathtt{e}''$ and $\mathtt{e}'' \rightarrow \mathtt{e}'$.

Intuitively, if $\mathtt{e} \rightarrow \mathtt{e}'$, then `e` is guaranteed to happen before $\mathtt{e}'$.

The local state of an agent is represented as the values of a collection of state variables. Formally, state variables are just identifiers that are assigned a value at each event.[2] There are binary functions `when` and `after` that describe the values of state variables before and after an event takes place. We typically write these functions using infix notation. Thus, if `agent(e) = i` then `(x when e)` describes the value of the state variable `x` at agent `i` just before `e`, and `(x after e)` describes its value after `e`. Note that state variables are local variables and two agents may have state variables with the same name.

[1] Receive events are further partitioned by a *tag* so that we can restrict the kinds of events sending messages on a given link with a given tag without restricting other uses of that link.To simplify the discussion in this paper we have supressed all mention of these tags.

[2] State variables are typed, but to simplify our discussion we suppress all type declarations.

Every event `e` also has a *value* `val(e)`. The value of a receive event is the message that is received, and the value of a local event represents a value (satisfying some constraints) chosen (non-deterministically) by the agent when the local event is generated. For example, if whenever a local event of kind `a` occurs the agent chooses an integer value and sends twice that value on link `l`, then events with `kind(e)=rcv(l)` and `kind(sender(e))=local(a)` will have `val(e)=2*val(sender(e))`.

The axioms of the event structure say that an event is the sender of only a finite number of messages; the predecessor function is one to one; causal order is well-founded; the local predecessor of an event has the same agent; the sender of an event has the agent of the source of the link on which the message was received; and the observation of state variables is related to the order structure in the obvious way, namely: `(x after pred(e)) = (x when e)`

### 2.3 Distributed message automata

Programs are built from a small set of basic clauses. With each basic clause `c` we associate a formula $\varphi_c$ in the language of event structures, and the event structures consistent with `c` are the ones satisfying $\varphi_c$. If we prove a specification $\psi$ using a set of assumptions $\{\varphi_c | c \in C\}$, then this set $C$ of the clauses used in the proof will be a program realizing $\psi$. A finite set $C$ of basic clauses is *feasible* if there is an event structure (a run) consistent with all the clauses in $C$ (i.e. satisfying all the $\varphi_c$). Every basic clause is feasible and we have defined a syntactically checkable notion of *compatibility* that guarantees that the union of compatible feasible sets is again feasible.

Accordingly, a distributed message automaton is a finite set of basic clauses. Each basic clause has an agent, and as part of that agent it does one of six things:

1. defines the initial value of one state variable,
2. defines the effect of one kind of event on one state variable,
3. defines the messages sent on one link when events of one kind occur,
4. defines the precondition for one kind of local event,
5. lists all the kinds of event that affect one state variable,
6. lists all the kinds of event that send on a given link.

The first four basic clauses correspond to lines of code (and can easily be compiled into code). The last two basic clauses are called *frame conditions*, and correspond to promises *not* to add code. A frame condition and an effect clause or a send clause may be *incompatible*. We can form more complicated programs (i.e., automata) from simpler automata by composition. The composition `A` $\oplus$ `B` of two programs `A` and `B` is just the union of the clauses from `A` and `B`. The rules restrict composition of message automata to automata whose clauses are pairwise compatible. The class of distributed message automata (defined as a recursive type in Nuprl) is the smallest class containing the six basic clauses above and closed under $\oplus$.

The semantics of a distributed message automaton is the set of event structures that are consistent with it. This is formally defined as a relation `Consistent(R;es)` between a program (i.e., message automaton) `R` and event structure `es`. Let `Sys-R` consist of all event structures consistent with `R`. A specification for us is a predicate on sets of runs. A program `R` is said to satisfy a specification if `Sys-R` does. As we show in the full paper, the relation `Consistent(R;es)` is expressible in Nuprl. Programs are also expressible in Nuprl. Thus, we can talk about the system consistent with a program `R` in Nuprl.

Recall that a (standard) specification is a predicate on runs. A program `R` is said to satisfy a specification if every run in `Sys-R` does.

## 2.4 Realizability Rules

If P is a predicate on runs, R is a program, and the set of runs consistent with R is non-empty and every run consistent with R satisfies P, then we write R ||- P and say that R *realizes* P. Note that R ||- P is just an abbreviation for the formula ∀ es. (Consistent(R;es) ⇒P(es)) ∧∃es. Consistent(R;es).

We have derived from the formal semantics of distributed message automata a set of nine rules for proving the realizability of a specification P[es]. There are six base rules, one for each basic clause, an additional rule for the combination of precondition and initialization clauses, a composition rule, and a refinement rule. We now briefly explain these rules.

The refinement rule says that if P refines Q (that is, if a run satisfies P, then it also satisfies Q) and A realizes P, then A also realizes Q:

$$\texttt{A ||- P} \Rightarrow (\forall \texttt{es:ES. (P[es]} \Rightarrow \texttt{Q[es])} \Rightarrow \texttt{A ||- Q .}$$

The composition rule requires the notion of *compatibility*. Two programs A and B are *compatible*, denoted A || B, if the sends and effect clauses of one obey the constraints imposed by the frame clauses of the other. The composition rule just captures the fact that A ⊕B combines the constraints of A and B:

$$\texttt{(A ||- P} \wedge \texttt{B ||- Q} \wedge \texttt{A || B)} \Rightarrow \texttt{A} \oplus \texttt{B ||- P} \wedge \texttt{Q .}$$

Each basic rule says that a basic clause c realizes its associated realizability rule $\varphi_c$. For the purposes of this paper we don't need the details of the syntax of the six basic clauses and the full listing of the six rules with all of their parameters. Instead, we give some examples.

The basic clause "at agent i initialize x to 5" realizes

$$\forall \texttt{e@i.}\neg\texttt{first(e)} \Rightarrow \texttt{(x when e) = 5,}$$

where ∀e@i.P is an abbreviation of ∀e.agent(e)=i ⇒ P.

A more important example for this paper is the precondition clause of the form "at agent i the precondition for a local action a(v) is P(v,x,...,z)". In this clause, the expression P(v,x,...,z) is a predicate on the values of the state variables x,...,z and a value v. The intended meaning is that "infinitely often" agent i decides whether there is a value v satisfying predicate P in the current state, and if so the agent chooses some such v and performs a local action of kind a and value v.[3] Also, an action of kind a may occur only when its value satisfies the precondition. The realizability rule associated with this precondition clause is the conjunction of a liveness property

$$\forall \texttt{e@i.}\exists \texttt{e'}\geq\texttt{e. kind(e')=local(a)} \vee \forall \texttt{v.}\neg\texttt{P(v) after e'}$$

and a safety property

$$\forall \texttt{e@i.kind(e)=local(a)} \Rightarrow \texttt{P@e,}$$

where P(v) after e ≡ P(v,x after e,...,z after e) and P@e ≡ P(val(e),x when e,...,z when e)

## 2.5 Example

As an example of a parameterized specification that we will need later, consider the following predicate Fair(P, f, l) on event structures, where P is a precondition, f is a function, and l is a link. Fair(P, f, l) is a conjunction of a safety condition and a liveness condition. The safety condition asserts that every

[3] Performing an action of kind k means updating all the state variables in accordance with any effect clauses for k and sending all messages in accordance with any sends clauses for k.

receive event on link `l` has a value that is `f` of the state of the sender and that state satisfies the precondition `P`. The liveness condition says that "infinitely often" either a receive event on $l$ occurs or else the precondition `P` fails.

```
Fair(P,f,l)  ≡
(∀e'.kind(e')=rcv(l) ⇒ P@sender(e') ∧ val(e') = f@sender(e')
∧ ∀e@source(l).∃e'≥e.kind(e')=rcv(l) ∨ ∀v:A. (¬(P(v) after e' )
```

This specification is realized by the following program `Fair-Pg`$(\mathtt{P}, \mathtt{f}, \mathtt{l})$ consisting of the three basic clauses at agent `source(l)`:

```
precondition for a(v) is P
local(a)(v) sends  [f v] on l
only events in [local(a)] send on l
```

We can prove that the program realizes the specification using the realizability rules. The intuitive reason is that the precondition is checked infinitely often. When it is true, the local event `a` occurs and sends the message. If this happens infinitely often, eventually a receive event will occur. Since only local action `a` sends this kind of message, the sender's precondition must be the given one.

# 3 Adding knowledge to Nuprl

## 3.1 Consistent cut semantics for knowledge

To reason about knowledge in event structures we use a standard first-order modal logic of knowledge and time. Assume that there are $n$ processes. Consider a propositional logic of knowledge, where formulas are formed by starting with a set $\Phi$ of functions symbols, predicate symbols, and constant symbols of various arities. We form atomic predicates and terms as usual in first-order logic, and close under conjunction, negation, universal quantification, the temporal operator $\Box$, and the modal operators $K_i$, $i = 1, \ldots, n$, one for each process $i$.[4]

Typically semantics for knowledge are given with respect to a pair $(r, m)$ consisting of a run $r$ and a time $m$, assumed to be the time on some external global clock (that none of the processes necessarily knows about) [Fagin et al. 1995]. In event structures, there is no external notion of time. Fortunately, Panangaden and Taylor [1992] give a variant of the standard definition with respect to what they call *asynchronous runs*, which are essentially identical to event structures. Thus, we just apply their definition in our framework.

The truth of formulas is defined relative to a triple (`Sys, E, c`), consisting of a system `Sys` (i.e., a set of event structures), and event structure `E` in `Sys`, and a *consistent cut* `c` of `E`, where a *consistent cut* `c` in `E` is a set of events in `E` closed under the causality relation. That is, if $\mathtt{e}'$ is an event in `c` and `e` is an event in `E` that precedes $\mathtt{e}'$ (i.e., $\mathtt{e} \rightarrow \mathtt{e}'$), then `e` must also be in `c`.

Define the equivalence relations $\sim_i$, $i = 1, \ldots, n$, on consistent cuts by taking $\mathtt{c} \sim_i \mathtt{c}'$ if $i$'s history is the same in `c` and $\mathtt{c}'$. Intuitively, $\mathtt{c} \sim_i \mathtt{c}'$ if process $i$ cannot tell `c` and $\mathtt{c}'$ apart, given its information. Given two consistent cuts `c` and $\mathtt{c}'$, we say that $\mathtt{c} \preceq \mathtt{c}'$ if, for each process $i$, process $i$'s history in `c` is a prefix of process $i$'s history in $\mathtt{c}'$.

Given a nonempty set of objects $D$ and a system $Sys$, an interpretation function $\pi$ associates to each cut $c$ and symbol $s$ in $\Phi$ its interpretation, denoted $\pi(c, s)$, which is a predicate or function on $D$ of the right arity. To extend this interpretation to terms, we start with a valuation $V$, which associates with each variable an element of $D$. For each variable $x$, we define $\pi(c, x) = V(x)$. We then define $\pi(c, f(t_1, \ldots, t_k))$ by induction on the structure of terms, taking $\pi(c, f(t_1, \ldots, t_k)) =$

[4] We can also define other standard modal operators, such as common knowledge, but we do not need them for the discussion in this paper. We can also define temporal operators such as *until*.

$\pi(c, f)(\pi(c, t_1), \ldots, \pi(c, t_k))$. Using $V$ and $\pi$, we define what it means for a formula $\varphi$ to be true at the consistent cut $\mathtt{c}$ in event structure $\mathtt{E}$ in system $\mathtt{Sys}$, denoted $(\mathtt{Sys}, \mathtt{E}, \mathtt{c}, \pi, V) \models \varphi$, by induction on the structure of $\varphi$, in the usual way:

- if $P$ is a predicate symbol in $\Phi$ of some arity $k$, and $t_1, \ldots, t_k$ are terms, then

$$(\mathtt{Sys}, \mathtt{E}, \mathtt{c}, \pi, V) \models P(t_1, \ldots, t_k) \text{ iff } \pi(c, P)(\pi(c, t_1), \ldots, \pi(c, t_k))$$

- $(\mathtt{Sys}, \mathtt{E}, \mathtt{c}, \pi, V) \models \neg\varphi$ iff $(\mathtt{Sys}, \mathtt{E}, \mathtt{c}, \pi, V) \not\models \varphi$
- $(\mathtt{Sys}, \mathtt{E}, \mathtt{c}, \pi, V) \models \varphi_1 \wedge \varphi_2$ iff $(\mathtt{Sys}, \mathtt{E}, \mathtt{c}, \pi, V) \models \varphi_1$ and $(\mathtt{Sys}, \mathtt{E}, \mathtt{c}, \pi, V) \models \varphi_2$
- $(\mathtt{Sys}, \mathtt{E}, \mathtt{c}, \pi, V) \models \forall x.\varphi$ iff, for all $d \in D$, $(\mathtt{Sys}, \mathtt{E}, \mathtt{c}, \pi, V[x/d]) \models \varphi$, where $V[x/d]$ is the valuation that agrees with $V$ on all variables except possible $x$, and $V[x/d](x) = d$
- $(\mathtt{Sys}, \mathtt{E}, \mathtt{c}, \pi, V) \models K_i\varphi$ iff for all $\mathtt{E}' \in \mathtt{Sys}$ and cuts $\mathtt{c}'$ of $\mathtt{E}'$ such that $\mathtt{c}' \sim_{\mathtt{i}} \mathtt{c}$, $(\mathtt{Sys}, \mathtt{E}', \mathtt{c}', \pi, V) \models \varphi$
- $(\mathtt{Sys}, \mathtt{E}, \mathtt{c}, \pi, V) \models \Box\varphi$ iff for all cuts $\mathtt{c}'$ of $\mathtt{E}$ such that $\mathtt{c} \preceq \mathtt{c}'$, $(\mathtt{Sys}, \mathtt{E}, \mathtt{c}', \pi, V) \models \varphi$.

As usual, we take $\Diamond\varphi$ to be an abbreviation of $\neg\Box\neg\varphi$, so that $\Diamond\varphi$ is true at $(\mathtt{E}, \mathtt{c})$ if there is some cut $\mathtt{c}'$ extending $\mathtt{c}$ where $\varphi$ is true.

Just as programs and the property of a program satisfying a specification can be expressed as a formula in Nuprl, the satisfaction relation $\models_V$ can be expressed as a formula in Nuprl. More precisely, we would like to define a translation $\mathtt{T}$ such that for all tuples $(\mathtt{Sys}, \mathtt{E}, \mathtt{c})$, domains $\mathtt{D}$, interpretations $\pi$, valuations $\mathtt{V}$, and formulas $\varphi$, $\mathtt{T}(\mathtt{Sys}, \mathtt{E}, \mathtt{c}, \mathtt{D}, \pi, V, \varphi)$ is provable iff $(\mathtt{Sys}, \mathtt{E}, \mathtt{c}, V, \pi) \models \varphi$.

To present the translation, some notation is necessary. The type of systems (sets of event structures) is $\mathtt{Set}(\mathtt{ES})$ and the type of all consistent cuts in event structure $\mathtt{E}$ is written as $\mathtt{CC}(\mathtt{E})$. The translation $\mathtt{T}$ is defined inductively on the structure of formula $\varphi$ as follows:

- if $\mathtt{P}$ is a predicate symbol in $\Phi$ of some arity $k$, and $\mathtt{t_1}, \ldots, \mathtt{t_k}$ are terms, then

$$\mathtt{T}(\mathtt{Sys}, \mathtt{E}, \mathtt{c}, \mathtt{D}, \pi, \mathtt{V}, \mathtt{P}(\mathtt{t_1}, \ldots, \mathtt{t_k})) = \pi(\mathtt{c}, \mathtt{P})(\pi(\mathtt{c}, \mathtt{t_1}), \ldots, \pi(\mathtt{c}, \mathtt{t_k})$$

- $\mathtt{T}(\mathtt{Sys}, \mathtt{E}, \mathtt{c}, \mathtt{D}, \pi, \mathtt{V}, \neg\varphi) = \neg(\mathtt{T}(\mathtt{Sys}, \mathtt{E}, \mathtt{c}, \mathtt{D}, \pi, \mathtt{V}, \varphi))$
- $\mathtt{T}(\mathtt{Sys}, \mathtt{E}, \mathtt{c}, \mathtt{D}, \pi, \mathtt{V}, (\varphi_1 \wedge \varphi_2)) = (\mathtt{T}(\mathtt{Sys}, \mathtt{E}, \mathtt{c}, \mathtt{D}, \pi, \mathtt{V}, \varphi_1)) \wedge (\mathtt{T}(\mathtt{Sys}, \mathtt{E}, \mathtt{c}, \mathtt{D}, \pi, \mathtt{V}, \varphi_2))$
- $\mathtt{T}(\mathtt{Sys}, \mathtt{E}, \mathtt{c}, \mathtt{D}, \pi, \mathtt{V}, (\forall \mathtt{x}.\varphi)) = \forall \mathtt{x} : \mathtt{D}.(\mathtt{T}(\mathtt{Sys}, \mathtt{E}, \mathtt{c}, \mathtt{D}, \pi, \mathtt{V}, (\varphi(\mathtt{V}\,\mathtt{x}))))$
- $\mathtt{T}(\mathtt{Sys}, \mathtt{E}, \mathtt{c}, \mathtt{D}, \pi, \mathtt{V}, K_i\varphi) = \forall \mathtt{E}' : \mathtt{Set}(\mathtt{ES}).\mathtt{E}' \in \mathtt{Sys} \Rightarrow \forall \mathtt{c}' : \mathtt{CC}(\mathtt{E}').\mathtt{c}' \sim_{\mathtt{i}} \mathtt{c} \Rightarrow (\mathtt{T}(\mathtt{Sys}, \mathtt{E}', \mathtt{c}', \mathtt{D}, \pi, \mathtt{V}, \varphi))$
- $\mathtt{T}(\mathtt{Sys}, \mathtt{E}, \mathtt{c}, \mathtt{D}, \pi, \mathtt{V}, \Box\varphi) = \forall \mathtt{c}' : \mathtt{CC}(\mathtt{E}).\mathtt{c} \preceq \mathtt{c}' \Rightarrow (\mathtt{T}(\mathtt{Sys}, \mathtt{E}, \mathtt{c}', \mathtt{D}, \pi, \mathtt{V}, \varphi))$

As we said, we would now like to show that $\mathtt{T}(\mathtt{Sys}, \mathtt{E}, \mathtt{c}, \mathtt{D}, \pi, \mathtt{V}, \varphi)$ is provable iff $(\mathtt{Sys}, \mathtt{E}, \mathtt{c}, V, \pi) \models \varphi$. However, since first-order epistemic logic relies on the principle of excluded middle, and Nuprl is a constructive type theory that does not, in general, assume the law of the excluded middle, we can prove that $\mathtt{T}$ has the desired effect only if we assume the law of the excluded middle. We remark that this assumption is necessary only for the purpose of this translation and not for the proofs we present in Section 4. With this assumption, we get the desired result.

**Proposition 3.1:** *Assuming the principle of excluded middle, for all tuples* $(\mathtt{Sys}, \mathtt{E}, \mathtt{c})$*, domains* $\mathtt{D}$*, interpretations* $\pi$*, valuations* $\mathtt{V}$*, and formulas* $\varphi$*, we have that* $\mathtt{T}(\mathtt{Sys}, \mathtt{E}, \mathtt{c}, \mathtt{D}, \pi, \mathtt{V}, \varphi)$ *is provable iff* $(\mathtt{Sys}, \mathtt{E}, \mathtt{c}, V, \pi) \models \varphi$.

To simplify the notation, we abbreviate $\mathtt{T}(\mathtt{Sys}, \mathtt{E}, \mathtt{c}, \mathtt{D}, \pi, \mathtt{V}, \varphi)$ as $\varphi@^{\mathtt{Sys}, \mathtt{E}, D, \pi, V}$; we often abbreviate this further as $\varphi@\mathtt{c}$, when the other components are clear from context.

### 3.2 Knowledge-based programs and specifications

In this section we show how we can extend the notions of program and specification presented in Section 2 to knowledge-based programs and specifications. This will allow us to employ the large body of tactics and libraries already developed in Nuprl to synthesize knowledge-based programs from knowledge-based specifications.

We have identified programs with distributed message automata, where a distributed message automaton is characterized by a set of clauses. We take a *knowledge-based message automaton* to be a function that associates to each system (i.e., set of event structures) a message automaton; intuitively, a knowledge-based message automaton allows preconditions on actions to depend on the knowledge of processes about the whole system. For the purposes of this paper, we take knowledge-based programs (hereafter abbreviated *kb programs*) to be knowledge-based message automata. Note that each standard program Pg corresponds to the kb program that associates to each system the program Pg.

What should the semantics of a kb program be? As discussed in Section 2, in the case of standard programs, a program semantics is a function of type $\mathtt{S} \in \mathtt{Pgm} \longrightarrow \mathtt{Sem}$; S associates with every program $\mathtt{Pg} \in \mathtt{Pgm}$ the system S(Pg) consisting of all the runs consistent with Pg. (Recall that Sem is the type consisting of all systems.) As we have seen, the truth of a knowledge test in a kb program depends on the whole system. Once we have a system, we can determine the truth of the knowledge tests. A kb program then reduces to a standard program. Thus, a kb program has type KbPgm = Sem $\rightarrow$Pgm. Note that composing the semantic function S with a knowledge-based program yields a function from systems to systems. A system Sys is said to *represent* a kb program kbPg if it is fixed point of this function. That is, Sys represents the kb program kbPg if $\mathtt{S}(\mathtt{kbPg})(\mathtt{Sys}) = \mathtt{Sys}$. Following Fagin et al. [1995, 1997], we take the semantics of a kb program kbPg to be the set of systems that represent kbPg. That is, a kb program semantics $\mathtt{S}^{\mathtt{kb}}$ is a function of type KbPgm $\rightarrow$set-of-systems, where set-of-systems is the type whose elements are sets of systems. As observed by Fagin et al. [1995, 1997], it is possible to construct kb programs that are represented by no systems, exactly one system, or more than one system. It is also possible to construct sufficient conditions (which are often satisfied in practice) that guarantee that a kb program is represented by exactly one system. Note that, in particular, standard programs when viewed as knowledge-based programs are represented by a unique system; indeed, $\mathtt{S}^{\mathtt{kb}}(\mathtt{Pg}) = \{\mathtt{S}(\mathtt{Pg})\}$. Thus, we can view $\mathtt{S}^{\mathtt{kb}}$ as extending S.

We next consider knowledge-based specifications (hereafter abbreviated *kb specifications*). Recall that a (standard) specification is a predicate on runs. Following [Halpern 1999], we take a kb specification to be a predicate on systems. That is, a kb specification has type Sem $\Rightarrow$P. If kbX is a kb specification, then kbX($Sys$) holds if Sys satisfies the specification kbX.

As in [Fagin et al. 1995; Halpern 1999], we say that a kb program kbPg satisfies the kb specification kbX if all the systems representing kbPg satisfy X. Thus, we take $\mathtt{kbPg} \mid = \mathtt{kbX}$ as an abbreviation of $\forall \mathtt{Sys} \in (\mathtt{S}^{\mathtt{kb}}\ \mathtt{kbPg}).\ \mathtt{kbX}(\mathtt{Sys})$.

**Example 3.2:** Recall that in Section 2 a specification Fair(P,f,l) was considered that requires that, infinitely often, either a precondition P fails at the state of the source of some link or a message is received on the link; the message is constructed by applying the function f at the source of the link. The specification is satisfied by a standard program $\mathtt{Fair\text{-}Pg}(\mathtt{P}, \mathtt{f}, \mathtt{l})$.

$\mathtt{Fair}(\mathtt{P}, \mathtt{f}, \mathtt{l})$ can be generalized to a kb specification $\mathtt{Fair}^{\mathtt{kb}}(\mathtt{P}^{\mathtt{kb}}, \mathtt{f}^{\mathtt{kb}}, \mathtt{l})$, where, instead of using a precondition P and function f, we use a *knowledge-based* predicate $\mathtt{P}^{\mathtt{kb}}$ and a *knowledge-based* function $\mathtt{f}^{\mathtt{kb}}$, both of which take a system as an extra argument (in addition to the other arguments of P and f). $\mathtt{Fair}^{\mathtt{kb}}(\mathtt{P}^{\mathtt{kb}}, \mathtt{f}^{\mathtt{kb}}, \mathtt{l})$ asserts that, in every run of the system, infinitely often either the kb precondition fails or a receive event with the value given by $\mathtt{f}^{\mathtt{kb}}$ occurs on line l. $\mathtt{Fair}^{\mathtt{kb}}(\mathtt{P}^{\mathtt{kb}}, \mathtt{f}^{\mathtt{kb}}, \mathtt{l})$ is satisfied by a kb program $\mathtt{Fair\text{-}Pg}^{\mathtt{kb}}(\mathtt{P}^{\mathtt{kb}}, \mathtt{f}^{\mathtt{kb}}, \mathtt{l})$, which associates to each system Sys the program $\mathtt{Fair\text{-}Pg}(\mathtt{P}^{\mathtt{kb}}(\mathtt{Sys}), \mathtt{f}^{\mathtt{kb}}(\mathtt{Sys}), \mathtt{l})$; in system Sys, a process following $\mathtt{Fair\text{-}Pg}^{\mathtt{kb}}(\mathtt{P}^{\mathtt{kb}}, \mathtt{f}^{\mathtt{kb}}, \mathtt{l})$ sends a message with value determined by $\mathtt{f}^{\mathtt{kb}}(\mathtt{Sys})$ exactly when predicate $\mathtt{P}^{\mathtt{kb}}(\mathtt{Sys})$ holds.

# 4 The sequence transmission problem

The main motivation for formalizing epistemic logic in event theory is that it allows us to write kb specifications in Nuprl, and then to synthesize kb programs. In this section, we consider one example of how this can be done. We show how a kb program that solves the *sequence transmission problem* (stp from now on) discussed by Halpern and Zuck [1992] can be synthesized from a kb specification of the problem. In fact, Halpern and Zuck consider two different kb programs that solve the problem; we show how both can be synthesized.

The stp involves a sender $S$ who has an input tape with a (possibly infinite) sequence $X = [x_0, x_1, \ldots]$ of bits, and wants to transmit $X$ to a receiver $R$; $R$ must write this sequence on an output tape $Y$. A solution to the problem must satisfy two conditions:

1. (safety): at all times, the sequence $Y$ of bits written by $R$ is a prefix of $X$, and
2. (liveness): every bit $x_k$ is eventually written by $R$ on the output tape.

If messages cannot be lost, duplicated, reordered or corrupted, then $S$ could simply send the bits in $X$ to $R$ in order; however, just as Halpern and Zuck, we are interested in solutions to the stp in contexts where communication is not reliable. Following Halpern and Zuck, we assume (a) that all corruptions are detectable and (b) a weak fairness condition: all messages sent infinitely often are eventually received.

The safety and liveness conditions above are run-based specifications. As argued in [Fagin et al. 1995], it is often better to think in terms of knowledge-based specifications for this problem. The real goal of the stp is to get the receiver to know the bits. That is, writing $K_R(x_i)$ as an abbreviation for $K_R(x_i = 0) \vee K_R(x_i = 1)$, we really want a knowledge-based liveness condition of the form $\forall i \Diamond K_R(x_i)$. If we further require $\forall i \forall k \Box(Y[i] = k \Rightarrow K_R(x_i = k))$ (that is, the receiver never sets the $i$th bit of the output to $k$ unless it knows that the $k$ bit of the sequence $X$ is actually $k$) and $\forall i, j, k \Box((i \geq j \Rightarrow K_R(x_j)) \wedge K_R(x_i = k) \Rightarrow \Diamond Y[i] = k)$ (that is, if the receiver knows the first $k$ bits, then it eventually writes them out), then clearly the safety and liveness specifications will hold. The formula $\forall j.i \geq j \Rightarrow K_R(x_j)$ is abbreviated as $K_R X\,[0 \ldots i]$.

Recall that a kb specification is a predicate on systems; the condition $\forall i, k.\Box(Y\,[i] = k \Rightarrow K_R(x_i = k))$ is written in Nuprl as the kb-specification $\mathtt{STP_1}$ that associates to each system $\mathtt{Sys}$ a predicate that ensures that for any $i$ and $k$, at any run $\mathtt{E}$ in the system and consistent cut $\mathtt{c}$, the condition $(\mathtt{Y}\,[\mathtt{i}] = \mathtt{k}) \Rightarrow K_R(\mathtt{x_i} = \mathtt{k})$ holds. Similarly, the specification $\forall i k \Box(K_R X[0 \ldots i] \wedge \wedge K_R(x_i = k) \Rightarrow \Diamond Y[i])$ is translated into Nuprl as the following kb specification $\mathtt{STP_2}$, which requires, for each system $\mathtt{Sys}$, natural number $\mathtt{i}$, bit $\mathtt{k}$, run $\mathtt{E}$ of $\mathtt{Sys}$, and consistent cut $\mathtt{c}$ in $\mathtt{E}$,

$$(K_R\mathtt{X}\,[\mathtt{0} \ldots \mathtt{i}] \Rightarrow K_R(\mathtt{x_i} = \mathtt{k}))@\mathtt{c} \Rightarrow \exists \mathtt{c}' \succeq \mathtt{c}.(\mathtt{Y}\,[\mathtt{i}] = \mathtt{k})@\mathtt{c}'.$$

Our goal is to prove using Nuprl that $\mathtt{STP_1} \wedge \mathtt{STP_2}$ is satisfiable. As we said before, the constructive proof will actually yield a kb program satisfying this specification. The proof that $\mathtt{STP_1} \wedge \mathtt{STP_2}$ is satisfiable is carried out in Nuprl by refining it to subgoals that are, intuitively, easier to prove. The system can often suggest appropriate subgoals using its library of tactics. In addition, the system has a library of standard proofs that can be applied at appropriate times. We are extending the library so that in includes standard facts about knowledge. For example, the well-known *knowledge axiom* that whatever is known must be true ($K_i\varphi \Rightarrow \varphi$) becomes a short lemma in Nuprl that can be invoked in any subsequent proof and used for refining a goal $\varphi$ to $K_i\varphi$. Similarly, the library includes the fact that, in event structures, there is *perfect recall* [Fagin et al. 1995]: processes do not forget *stable facts* (that is, facts which, once true, remain true), so the formula $K_i\varphi \Rightarrow \Box K_i\varphi$ is valid if $\varphi$ is stable. Including these results in the library facilitates reuse of code. These lemmas can be invoked repeatedly in the course of a proof.

Reusability of code can also be exploited at the level of kb specifications; having formally defined distributed programs in Nuprl, once we prove that a certain kb specification $\mathtt{kbX}$ is satisfiable we also

construct a program Pg that satisfies kbX that can be manipulated and reused during the proof. Ingenuity is required to identify the specifications likely to appear in many proofs. But once good choices are made, they can significantly simplify proofs. As we shall see, the specification in Section 3.2 is invoked a number of times in the course of synthesizing a program that satisfies $\mathtt{STP}_1 \wedge \mathtt{STP}_2$.

Returning to our problem, we remark that performing a few standard Nuprl refinement steps allows us to reduce the proof of $\mathtt{Sat}(\mathtt{STP}_1 \wedge \mathtt{STP}_2)$ to a few subgoals, the most interesting of which is that

$$K_R\mathtt{X}\,[\mathtt{0}\ldots\mathtt{k}-\mathtt{1}]\,@\mathtt{c} \Rightarrow \exists \mathtt{c}' \succeq \mathtt{c}.K_R\mathtt{X}\,[\mathtt{0}\ldots\mathtt{k}]\,@\mathtt{c}' \tag{1}$$

is satisfiable. (To simplify notation, we have omitted the universal quantification over runs E and cuts c in E.) This subgoal says that we want to find a program that ensures that R makes progress: if at some cut R knows the first $k$ bits, then later on R knows all these bits, and the next one in the sequence.

The fact that R continues to know the bits it already knows is an instance of the rule we mentioned before regarding perfect recall in event structures. Thus, what we really have to do is find a program that ensures that, if R knows the first $k$ bits, it will eventually know the $(k+1)$st bit.

Up to this point in the proof, we have used only standard facts about logic, standard refinement rules for first-order logic, and natural induction. All of this is built into Nuprl. At this stage in the refinement, ingenuity is required to find suitable rules specific to the particular kb specification of interest. It turns out that only one new tactic is needed; we are guided to it by the weak fairness condition we have imposed on communication. Observe that the fairness condition guarantees that if at S a message is sent infinitely often, then $R$ will eventually get it.

Intuitively, R learns $\mathtt{x_k}$ from S; we would like to be the case that infinitely often either there is no value for which S doesn't know that R knows it, or if such a value exists, then R receives it. This is just an instance of the kb specification $\mathtt{Fair}^{\mathtt{kb}}(\mathtt{P}^{\mathtt{kb}}, \mathtt{f}^{\mathtt{kb}}, \mathtt{l})$ presented in example 3.2 and satisfied by the program denoted by $\mathtt{Fair\text{-}Pg}(\mathtt{P}^{\mathtt{kb}}, \mathtt{f}^{\mathtt{kb}}, \mathtt{l})$. We just need to choose the $\mathtt{P}^{\mathtt{kb}}$, $\mathtt{f}^{\mathtt{kb}}$, and l appropriate for our problem. Let $\mathtt{P}^{\mathtt{kb}}_{\mathtt{S}}$ be the kb predicate that tests whether $\neg K_S K_R \mathtt{X}$ holds; let $\mathtt{f}^{\mathtt{kb}}_{\mathtt{S}}$ be the kb function that, given S's state, if $\mathtt{P}^{\mathtt{kb}}_{\mathtt{S}}$ holds, returns the maximum index i such that $K_S K_R \mathtt{X}\,[\mathtt{0}\ldots\mathtt{i}-\mathtt{1}] \wedge \neg K_S K_R \mathtt{X}\,[\mathtt{i}]$ holds; let l be the link between R and S. $\mathtt{Fair}^{\mathtt{kb}}(\mathtt{P}^{\mathtt{kb}}_{\mathtt{S}}, \mathtt{f}^{\mathtt{kb}}_{\mathtt{S}}, \mathtt{l})$ is the kb specification that requires that infinitely often either $K_S K_R \mathtt{X}$, or R receives $\langle \mathtt{i}, \mathtt{x_i} \rangle$.

Recall that $\mathtt{Fair\text{-}Pg}^{\mathtt{kb}}(\mathtt{P}^{\mathtt{kb}}_{\mathtt{S}}, \mathtt{f}^{\mathtt{kb}}_{\mathtt{S}}, \mathtt{l})| = \mathtt{Fair}^{\mathtt{kb}}(\mathtt{P}^{\mathtt{kb}}_{\mathtt{S}}, \mathtt{f}^{\mathtt{kb}}_{\mathtt{S}}, \mathtt{l})$. If S uses the kb program $\mathtt{Fair\text{-}Pg}^{\mathtt{kb}}(\mathtt{P}^{\mathtt{kb}}_{\mathtt{S}}, \mathtt{f}^{\mathtt{kb}}_{\mathtt{S}}, \mathtt{l})$, and S knows that R knows all the bits, then S sends no message; otherwise, S sends to R the maximum value i such that S knows that R knows the first i bits, but does not know that R knows $\mathtt{x_i}$. Thus, S is essentially using the following kb program, also used in [Halpern and Zuck 1992] (which we write using their notation):

**if** $K_S K_R X\,[0\ldots i-1] \wedge \neg K_S K_R X\,[i]$ **then** send$(\langle i, x_i \rangle)$ **else** skip.

It is clear that if S uses this kb program and does not know that R knows the $k$th bit, then S will send the $k$th bit repeatedly as long as S does not know that R knows the $k$th bit. Since a message is eventually received if sent infinitely often, R will eventually know the $k$th bit. Thus, we have reduced the problem to finding a way to guarantee that, if R knows the first $i$ bits and does not know $x_i$, then S eventually knows that R knows the first $i$ bits (and so will send $x_i$). This is again just an instance of the kb specification $\mathtt{Fair}^{\mathtt{kb}}(\mathtt{P}^{\mathtt{kb}}, \mathtt{f}^{\mathtt{kb}}, \mathtt{l})$. Let $\mathtt{P}^{\mathtt{kb}}_{\mathtt{R}}$ be the kb predicate that tests whether there is a bit that R does not know; let $\mathtt{f}^{\mathtt{kb}}_{\mathtt{R}}$ the kb function that, given R's state that, if $\mathtt{P}^{\mathtt{kb}}_{\mathtt{R}}$ holds, returns the maximum index j such that R knows the first j bits in the sequence; again, l is the link between R and S. In other words, $\mathtt{f}^{\mathtt{kb}}_{\mathtt{R}}$ returns the (unique) value j such that $K_R \mathtt{X}[\mathtt{0}\ldots\mathtt{j}-\mathtt{1}] \wedge \neg K_R \mathtt{X}[\mathtt{j}]$ holds in R's state when interpreted with respect to system Sys. $\mathtt{Fair}^{\mathtt{kb}}(\mathtt{P}^{\mathtt{kb}}_{\mathtt{R}}, \mathtt{f}^{\mathtt{kb}}_{\mathtt{R}}, \mathtt{l})$ is the kb specification that requires that infinitely often either $\mathtt{P}^{\mathtt{kb}}_{\mathtt{R}}$ fails, so the receiver knows all the bits, or S receives from R a message and S knows that $\mathtt{f}^{\mathtt{kb}}_{\mathtt{R}}$ holds at all points where R's last action was to send this message. Now, when R follows $\mathtt{Fair\text{-}Pg}^{\mathtt{kb}}(\mathtt{P}^{\mathtt{kb}}_{\mathtt{R}}, \mathtt{f}^{\mathtt{kb}}_{\mathtt{R}}, \mathtt{l})$, if R knows all the bits, then R sends no message; otherwise R sends to S the maximum value j such that R knows the first $j$ bits but does not know $x_j$. Thus, R is essentially

using the following kb program used in [Halpern and Zuck 1992]:

$$\textbf{if } K_R X\,[0 \ldots j-1] \wedge \neg K_R X\,[j] \textbf{ then } \mathrm{send}(j) \textbf{ else } \mathrm{skip}.$$

Recall goal (1); from $\mathtt{Fair}^{\mathtt{kb}}(\mathtt{P}_{\mathtt{R}}^{\mathtt{kb}}, \mathtt{f}_{\mathtt{R}}^{\mathtt{kb}}, \mathtt{l})$ we infer that either $K_R \mathtt{X@c}$, which implies $K_R \mathtt{X}\,[\mathtt{0} \ldots \mathtt{k}]\,\mathtt{@c}$, or there is some $\mathtt{j}$ such that $\mathtt{R}$ knows all the bits up to but excluding $\mathtt{j}$. Since $\mathtt{R}$ knows all bits up to $\mathtt{k}$, it must be that $\mathtt{j} \geq \mathtt{k}$; if $\mathtt{j} > \mathtt{k}$ then, since $\mathtt{R}$ knows all bits up to $\mathtt{j}$, $\mathtt{R}$ also knows $\mathtt{x_k}$; if $\mathtt{j} = \mathtt{k}$ then $\mathtt{Fair}^{\mathtt{kb}}(\mathtt{P}_{\mathtt{R}}^{\mathtt{kb}}, \mathtt{f}_{\mathtt{R}}^{\mathtt{kb}}, \mathtt{l})$ assures us that there is some cut $\tilde{\mathtt{c}}$ extending $\mathtt{c}$ such that $\mathtt{S}$ receives $\mathtt{k}$ at $\tilde{\mathtt{c}}$. At this moment it must be that $\mathtt{S}$ knows that $\mathtt{R}$ knows at least the bits up to $\mathtt{k}$. From $\mathtt{Fair}^{\mathtt{kb}}(\mathtt{P}_{\mathtt{S}}^{\mathtt{kb}}, \mathtt{f}_{\mathtt{S}}^{\mathtt{kb}}, \mathtt{l})$ we infer that either $K_S K_R \mathtt{X@}\tilde{\mathtt{c}}$ holds, from which the knowledge axiom can be applied to deduce $K_R \mathtt{X@}\tilde{\mathtt{c}}$, or there is some $\mathtt{i}$ such that $\mathtt{S}$ knows that $\mathtt{R}$ knows the bits up to $\mathtt{i}$, but $\neg K_S K_R \mathtt{X}\,[\mathtt{i}]\,@\tilde{\mathtt{c}}$ holds. Thus, $\mathtt{i} \geq \mathtt{k}$; if $\mathtt{i} > \mathtt{k}$ then $\mathtt{S}$ knows that $\mathtt{R}$ knows $\mathtt{x_k}$, and so $\mathtt{R}$ knows $\mathtt{x_k}$. If $\mathtt{i} = \mathtt{k}$, than $\mathtt{Fair}^{\mathtt{kb}}(\mathtt{P}_{\mathtt{S}}^{\mathtt{kb}}, \mathtt{f}_{\mathtt{S}}^{\mathtt{kb}}, \mathtt{l})$ assures us that there must be a later cut $\mathtt{c}'$ at which $\mathtt{R}$ receives $\langle \mathtt{k}, \mathtt{x_k} \rangle$, and so $\mathtt{R}$ learns $\mathtt{x_k}$. From the perfect recall rule, we deduce that $\mathtt{R}$ knows all the bits up to and including $\mathtt{x_k}$, which completes our proof.

It is now straightforward to prove in Nuprl that if $\mathtt{R}$ uses the kb program $\mathtt{Fair\text{-}Pg}^{\mathtt{kb}}(\mathtt{P}_{\mathtt{R}}^{\mathtt{kb}}, \mathtt{f}_{\mathtt{R}}^{\mathtt{kb}}, \mathtt{l})$, then $\mathtt{S}$ gets the requisite knowledge regarding which bit to send. The composition of the two programs

$$(\mathtt{Fair\text{-}Pg}^{\mathtt{kb}}(\mathtt{P}_{\mathtt{S}}^{\mathtt{kb}}, \mathtt{f}_{\mathtt{S}}^{\mathtt{kb}}, \mathtt{l})\ \mathtt{A}) \oplus (\mathtt{Fair\text{-}Pg}^{\mathtt{kb}}(\mathtt{P}_{\mathtt{R}}^{\mathtt{kb}}, \mathtt{f}_{\mathtt{R}}^{\mathtt{kb}}, \mathtt{l})\ \mathtt{A})$$

is thus essentially the kb program from [Halpern and Zuck 1992] that solves the stp. What we have shown here is that this program can be synthesized in Nuprl, using just standard facts in the Nuprl library, and the kb program `Fair-Pg`, which we will build into the Nuprl library, since it seems that it will be generally useful in all systems where communication is fair.

Halpern and Zuck show that a number of standard programs used in the literature are implementations of this kb programs. We can prove this in Nuprl, essentially by emulating the Halpern-Zuck proof. We omit the details in this abstract. Halpern and Zuck also present another kb program which is *receiver-driven*: rather then S sending the $i$th bit when it S does not know that R knows the bit, S sends the bit only when S knows that R does *not* know it. That is, the precondition of the sender's program has the test $K_S \neg K_R \mathtt{X}[\mathtt{i}]$ rather than $\neg K_S K_R \mathtt{X}[\mathtt{i}]$. In the full paper, we show how this kb program can be synthesized in Nuprl, using a relatively minor variant of the argument given above.

# 5 Conclusion

We have shown how to embed epistemic logic notions in constructive type theory. This allows us to express knowledge-based specifications and knowledge-based programs in Nuprl. We then showed by example how, from the constructive proof that a knowledge-based specification is satisfiable, we can extract a knowledge-based program that is guaranteed to satisfy that specification. What is notable about our proof is that it uses only standard logical arguments in Nuprl involving straightforward first-order logic and induction (as is the case for the synthesis of many sequential programs that have been carried out in Nuprl), standard arguments in epistemic logic (the knowledge axiom and the fact that in event structures agents have perfect recall), and one new argument involving the knowledge-based specification $\mathtt{Fair}^{\mathtt{kb}}$ and the knowledge-based program $\mathtt{Fair\text{-}Pg}^{\mathtt{kb}}$ that implements it.

The structure of the argument gives us reason to believe that we will be able to incorporate into the Nuprl library standard tactics that will apply to a wide variety of knowledge-based specifications, and that we will be able to reuse these tactics in many proofs. We are currently in the process of synthesizing other knowledge-based programs to verify that, and to deepen our understanding of the techniques needed to knowledge-based verification. We believe that, once we have shown the power of this approach, the ideas will quickly spread to other verification approaches based on higher-order logic. We are optimistic about the prospects.

## References


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

Bickford, M. (2003). Experiments with theory modification in the FDL. *In Progress*.

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

Bickford, M., C. Kreitz, R. van Renesse, and R. Constable (2001). An experiment in formal design using meta-properties. In J. Lala, D. Mughan, C. McCollum, and B. Witten (Eds.), *DARPA Information Survivability Conference and Exposition II (DISCEX-II)*, Volume II of *IEEE Computer Society Press*, Anaheim, CA, pp. 100–107.

Bickford, M., C. Kreitz, R. van Renesse, and X. Liu (2001, September). Proving hybrid protocols correct. In R. Boulton and P. Jackson (Eds.), *$14^{th}$ International Conference on Theorem Proving in Higher Order Logics*, Volume 2152 of *Lecture Notes in Computer Science*, Edinburgh, Scotland, pp. 105–120. Springer-Verlag.

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

Chandy, K. M. and J. Misra (1988). *Parallel Program Design: A Foundation*. Reading, Mass.: Addison-Wesley.

Constable, R. L. (2002). Naïve computational type theory. In H. Schwichtenberg and R. Steinbrüggen (Eds.), *Proof and System-Reliability, Proceedings of International Summer School Marktoberdorf, July 24 to August 5, 2001*, Volume 62 of *NATO Science Series III*, Amsterdam, pp. 213–260. Kluwer Academic Publishers.

Constable, R. L. et al. (1986). *Implementing Mathematics with the Nuprl Proof Development System*. NJ: Prentice-Hall.

Dwork, C. and Y. Moses (1990). Knowledge and common knowledge in a Byzantine environment: crash failures. *Information and Computation 88*(2), 156–186.

Engelhardt, K., R. v. d. Meyden, and Y. Moses (1998). A program refinement framework supporting reasoning about knowledge and time. In J. Tiuryn (Ed.), *Proc. Foundations of Software Science and Computation Structures (FOSSACS 2000)*, pp. 114–129. Berlin/New York: Springer-Verlag.

Engelhardt, K., R. v. d. Meyden, and Y. Moses (2001). A refinement theory that supports reasoning about knowledge and time for synchronous agents. In *Proc. Int. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning*, pp. 125–141. Berlin/New York: Springer-Verlag.

Fagin, R., J. Y. Halpern, Y. Moses, and M. Y. Vardi (1995). *Reasoning about Knowledge*. Cambridge, Mass.: MIT Press.

Fagin, R., J. Y. Halpern, Y. Moses, and M. Y. Vardi (1997). Knowledge-based programs. *Distributed Computing 10*(4), 199–225.

Geuvers, H., F. Wiedijk, and J. Zwanenburg (2001). A constructive proof of the fundamental theorem of algebra without using the rationals. In P. Callaghan, Z. Luo, J. McKinna, and R. Pollack (Eds.), *Types for Proofs and Programs, Proceedings of the International Workshop TYPES 2000*, Volume 2277 of *Lecture Notes in Computer Science*, pp. 96–111. Springer.

Geuvers, J. H. et al. (2001). The 'fundamental theorem of algebra' project. FTA web page. See http://www.cs.kun.nl/gi/projects/fta/.

Hadzilacos, V. (1987). A knowledge-theoretic analysis of atomic commitment protocols. In *Proc. 6th ACM Symp. on Principles of Database Systems*, pp. 129–134.

Halpern, J. Y. (1999). Hypothetical knowledge and counterfactual reasoning. *International Journal of Game Theory 28*(3), 315–330.

Halpern, J. Y. and Y. Moses (1990). Knowledge and common knowledge in a distributed environment. *Journal of the ACM 37*(3), 549–587.

Halpern, J. Y., Y. Moses, and O. Waarts (2001). A characterization of eventual Byzantine agreement. *SIAM Journal on Computing 31*(3), 838–865.

Halpern, J. Y. and L. D. Zuck (1992). A little knowledge goes a long way: knowledge-based derivations and correctness proofs for a family of protocols. *Journal of the ACM 39*(3), 449–478.

Kreitz, C. (1998). Program synthesis. In W. Bibel and P. Schmitt (Eds.), *Automated Deduction – A Basis for Applications*, Volume III, Chapter III.2.5, pp. 105–134. Kluwer.

Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM 21*(7), 558–565.

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

Liu, X., R. van Renesse, M. Bickford, C. Kreitz, and R. Constable (2001). Protocol switching: Exploiting meta-properties. In L. Rodrigues and M. Raynal (Eds.), *International Workshop on Applied Reliable Group Communication (WARGC 2001)*, pp. 37–42. IEEE.

Lynch, N. A. and M. R. Tuttle (1987). Hierarchical correctness proofs for distributed algorithms. Technical Report MIT/LCS/TR-387, MIT.

Lynch, N. A. and M. R. Tuttle (1989). An introduction to input/output automata. *CWI Quarterly 2*(3), 219–246. Also available as MIT Technical Memo MIT/LCS/TM-373.

Mazer, M. S. (1990). A link between knowledge and communication in faulty distributed systems. In *Theoretical Aspects of Reasoning about Knowledge: Proc. Third Conference*, pp. 289–304.

Mazer, M. S. and F. H. Lochovsky (1990). Analyzing distributed commitment by reasoning about knowledge. Technical Report CRL 90/10, DEC-CRL.

Moses, Y. and M. R. Tuttle (1988). Programming simultaneous actions using common knowledge. *Algorithmica 3*, 121–169.

Neiger, G. and S. Toueg (1993). Simulating real-time clocks and common knowledge in distributed systems. *Journal of the ACM 40*(2), 334–367.

Panangaden, P. and S. Taylor (1992). Concurrent common knowledge: defining agreement for asynchronous systems. *Distributed Computing 6*(2), 73–93.

Paulin-Mohring, C. and B. Werner (1993). Synthesis of ML programs in the system Coq. *Journal of Symbolic Computations 15*, 607–640.

Smith, D. R. and C. Green (1996). Toward practical applications of software synthesis. In *FMSP'96, The First Workshop on Formal Methods in Software Practice*, pp. 31–39.

Stulp, F. and R. Verbrugge (2002). A knowledge-based algorithm for the Internet protocol (TCP). *Bulletin of Economic Research 54*(1), 69–94.