

Knowledge-Based Synthesis of Distributed Systems Using Event Structures

Mark Bickford, Robert C. Constable, Joseph Y. Halpern, and 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 [15] 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 [4, 9]. 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 [9]. Nuprl has also been used extensively to optimize distributed protocols, and to specify them in the language of I/O Automata [6]. Recent work by two of the authors [5] 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 [5]. Event logic is a specification language closely related to I/O automata. As has long been recognized [13], 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 that a useful way of capturing these high-level knowledge-based specifications is by using high-level *knowledge-based programs* [13, 14]. 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 [13, 14]. They have already met with some degree of success, having been used both to help in the design of new protocols and to clarify the understanding of existing protocols [10, 15, 21].

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 satisfiable. 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 arbitrarily long prefixes of the sequence. Halpern and Zuck [15] 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 den Meyden, and Moses [11, 12] 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 den 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 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 [8] 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 the law of excluded middle does not hold.

Definition 1. A program in Nuprl is an object of some type Pgm . A program semantics is a function S of type $Pgm \rightarrow Sem$ assigning to each program Pg of type Pgm a meaning of type Sem . A specification is a predicate X on Sem .

We take $Pg \models X$ to be an abbreviation of $X(S(Pg))$, and $Sat(X)$ to be an abbreviation for $\exists Pg(Pg \models X)$. Thus, the fact that a specification is satisfiable is expressible in Nuprl. 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 .

Constructive type logic is highly undecidable, so we cannot hope to construct a proof completely automatically. However, experience has shown that, by having a large library of lemmas and proof tactics, it is possible to “almost” automate quite a few proofs, so that with a few hints from the programmer, correctness can be proved. In any case, 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 satisfiability.

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 *IO-Automata* [17] and are roughly equivalent to *UNITY* programs [7] (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 [16] model of events in distributed systems. Event structures are explained in more detail in the next section.

2.2 Event Structures

Consider a set AG of processes or *agents*; associated with each agent in AG is a set of *local variables*. Agent i ’s local state at a point in time is defined as the values of its local variables at that time. There are no shared variables.

Information is communicated by message passing. Sending a message on some link l is understood as enqueueing the message on l , while receiving corresponds to dequeuing the message. Communication is asynchronous and point-to-point: for each link l there is a unique agent $source(l)$ that can send messages on l , and a unique agent $destination(l)$ that can receive message on l .

Following Lamport [16], changes in the local state of an agent are modeled as *events*. Intuitively, when an event “happens”, an agent either receives a message or chooses some values (perhaps nondeterministically). As a result of receiving the message or the (nondeterministic) choice, some of the agent’s local variables are changed. Formally, events are elements of a type E . There is a one-to-one function $agent$ such that for any event e , $agent(e)$ is the agent whose local state changes during event e . The values of state variables before and after e happens are described by binary functions *when* and *after*, typically written using infix notation: if $agent(e) = i$ and x is one of i ’s variables, then $(x \text{ when } e)$ describes the value of x before e , and $(x \text{ after } e)$ describes its value after e .¹ To each event we associate a *value* and a *kind*. If the event happens as a result of a receiving a message on some link l , then the event has kind $rcv(l)$, and its value is the corresponding message. Any other event has kind $local(a)$, where a is the label of the event, and its value is the set of values (nondeterministically) chosen by the agent. The label of an event is just a syntactic identifier that makes it easier to do proofs and state conditions. Note that, unlike Lamport [16], we do not have events of kind *send*. We model the sending of a message on a link l by changing the value of the local variable that describes the message enqueueing on l . This variable can be changed during any event; that is, any event can involve sending a message. This way of modeling events has proved to be convenient.

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 agent i ’s *history*. If $first(e)$ holds, then e is the first event in the history associated with $agent(e)$; if not, then e has a predecessor $pred(e)$.

Every receive event e has a corresponding *send event*, denoted $send(e)$; this is the event where the message received at e was enqueueing. Following Lamport [16], 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 $e \rightarrow e'$ if

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

Intuitively, if $e \rightarrow e'$, then e is guaranteed to happen before e' . We write $e \geq e'$ if $e = e'$ or $e \rightarrow e'$.

Formally, an *event structure* consist of a collection of events satisfying some natural properties: only finitely many messages can be sent when an event happens; the predecessor function is one to one; the causal order is well-founded;

¹ State variables are typed, but to simplify our discussion we suppress all type declarations.

$pred(e)$ is associated with the same agent as e ; if e has kind $rcv(l)$, then the agent associated with $send(e)$ is $source(l)$; and, finally, the local variables of agent $agent(e)$ do not change between $pred(e)$ and e ; that is, $(x \text{ after } pred(e)) = (x \text{ when } e)$. The type (set) of event structures is definable in Nuprl.

2.3 Distributed Message Automata

A *message automaton* is a nondeterministic state machine associated with some agent i ; it specifies when i can take actions and which actions it can take. The actions in programs are essentially events in event structures. We view a *receive* action as being out of the control of the agent; all other actions have associated preconditions. At each point in time i nondeterministically decides which actions to perform, among those whose precondition is satisfied.

Message automata are built from a small set of basic clauses. With each basic clause cl in an automaton we associate a formula ϕ_{cl} in the language of event structures. The event structures consistent with cl are the ones satisfying ϕ_{cl} . If we prove a specification X is satisfiable using a set $\{\phi_{cl} \mid cl \in C\}$ of assumptions, then the set C of the clauses used in the proof is a *program* satisfying X . This is how we extract a program satisfying a specification from a proof that the specification is satisfiable.

A basic clause does one of the following:

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 precondition for one kind of local event;
4. lists all the kinds of events that can change the value of one particular state variable.

For convenience, we represent each of the basic clauses as a simple program in a programming language. We give some examples here:

- A basic clause of the second type that says the message $f(s, v)$ is sent by i on link l when a local event of kind $local(a)$ and value v occurs and i 's local state is s , where f is some function (recall that, in our framework, sending a message on link l amounts to changing the value of a local variable that encodes messages enqueued on l) is represented by the program

$$a(v) \text{ sends } f(\text{state}, v) \text{ on } l.^2$$

- A basic clause of the third type that says that an event a with value v occurs only if precondition P holds is represented by the program

$$a(v) \text{ only if } P(\text{state}, v).$$

² Here the notation $a(v)$ is just a compact way of saying that the value of an event of kind $local(a)$ is v , and does not refer to function application.

- The program representing an instance of the last clause that says that all the events that result in sending a message on a link l are on the list L is

only events in L send on l .

The programs representing instances of the first three clauses can be easily compiled into JAVA. The last clause is called a *frame condition*; it corresponds to a promise *not* to add code.

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 clauses ϕ_{cl} such that $cl \in C$). Every basic clause is feasible. A *distributed message automaton* is a collection of message automata, one for each agent in the system. A frame condition and an effect clause may be *incompatible*; that is, they may not be simultaneously satisfiable. We can form more complicated programs from simpler programs 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 set (type) of distributed message automata is the smallest set of feasible clauses containing the four basic clauses and closed under \oplus . By adding the appropriate constants to Nuprl, we can ensure that each program is a term in the language.

The semantics of a distributed message automaton is the set of event structures that are consistent with it. In terms of the language used in Section 2.1, an event structure is a run, and a set of event structures is a system. As we show in the full paper, the semantics can formally be defined in Nuprl as a relation $Consistent(Pg, es)$ between a program (i.e., message automaton) Pg and event structure es . The set of event structures consistent with Pg is denoted $Sys(Pg)$. The fact that $Consistent$ is definable in Nuprl is critical: it means that we can talk about whether an event structure is consistent with a program in Nuprl.

Recall that a specification is a predicate on systems, i.e., on the meaning of programs. Many specifications that arise in practice are of a special type called *run-based* specifications [13]. A run-based specification is given as a predicate on runs (i.e., event structures). We can view a predicate P on runs as a predicate on systems by taking P to hold of a system if it holds of every run of the system. If P is a run-based specification, then $Pg \models P$ exactly if

$$\forall es.(Consistent(Pg, es) \Rightarrow P(es)) \wedge \exists es.Consistent(Pg, es).$$

We have derived from the formal semantics of distributed message automata a set of seven useful Nuprl axioms for proving the satisfiability of a run-based specification (see [5]). There are four base axioms, one for each basic clause, an additional axiom for the combination of precondition and initialization clauses, a composition axiom, and a refinement axiom. We now briefly discuss these axioms.

The refinement axiom says that if P refines Q (that is, if $P(es)$ implies $Q(es)$ for all event structures es) and A satisfies P , then A also satisfies Q :

$$A \models P \Rightarrow \forall es.(P(es) \Rightarrow Q(es)) \Rightarrow A \models Q.$$

The composition axiom says that if two programs A and B are *compatible* (that is, their clauses are pairwise compatible), denoted $A||B$, then $A \oplus B$ combines the constraints of A and B :

$$(A \models P \wedge B \models Q \wedge A||B) \Rightarrow A \oplus B \models P \wedge Q.$$

The axiom for each basic clause cl is just the corresponding formula ϕ_{cl} . We give two examples of the axioms here; see [5] for the others.

The axiom corresponding to the basic clause “agent i initializes x to 5” is $\forall e@i.first(e) \Rightarrow (x \text{ when } e) = 5$, where $\forall e@i.P$ is an abbreviation of the formula $\forall e.agent(e) = i \Rightarrow P$. For the axiom corresponding to the precondition clause, let $state(e)$ be the state associated with event e ; that is, the values of all the local variables in event e . Similarly, let $state(after(e))$ be the values of the variables after e . (This, of course, is just $state(e')$ where e' is the successor of e in the history, if there is a successor.) The intended meaning of the precondition clause $a(v)$ **only if** $P(state, v)$ in a program for agent i is that, infinitely often, agent i checks whether there is a some value v such that $P(state(e), v)$ holds; if so then, infinitely often, i chooses such a value v and an event of kind $local(a)$ and value v occurs. Moreover, an event of kind $local(a)$ occurs only when its value satisfies the precondition. Finally, the clause rules out finite event sequences where an event of kind $local(a)$ could be performed after the last event, but is not. The axiom ϕ_{cl} for this clause is the conjunction of two formulas. The first is

$$\forall e@i[\exists e' \geq e(kind(e') = local(a)) \vee \forall v'(\neg P(state(after(e)), v'))],$$

which says that either infinitely often an event of kind $local(a)$ occurs, or infinitely often P is false, or the sequence is finite and P is false after the last event. The second conjunct gives the obvious safety condition, namely, that P is a precondition for an event of kind $local(a)$: $\forall e@i.kind(e) = local(a) \Rightarrow P(state(e), val(e))$.

2.4 Example

As an example of a parameterized specification that we use 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 the value of every receive event on link l is given by 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. In the specification, we abbreviate $P(state(e), val(e))$ by $P@e$; we similarly use $f@e$. The specification is

$$\begin{aligned} Fair(P, f, l) &\equiv \\ &\forall e'.kind(e') = rcv(l) \Rightarrow P@send(e') \wedge val(e') = f@send(e') \\ &\wedge \forall e@source(l).\exists e' \geq e.kind(e') = rcv(l) \vee \forall v'.\neg(P(state(after(send(e'))), v')). \end{aligned}$$

This specification is satisfied by the following program $Fair-Pg(P, f, l)$ for agent $source(l)$, which combines three basic clauses:

$a(v)$ only if $P(state, v) \oplus a(v)$ sends $f(state, v)$ on $l \oplus$ only events in $local(a)$ send on l .

Lemma 1. *Fair-Pg(P, f, l) satisfies the specification Fair(P, f, l). Moreover, this can be proved using the seven axioms.*

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 first-order logic of knowledge and time, where formulas are formed by starting with a set Φ 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 \square , and the operators K_i , $i = 1, \dots, n$, one for each process i .

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) [13]. In event structures, there is no external notion of time. Fortunately, Panangaden and Taylor [18] 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 e' is an event in c and e is an event in E that precedes e' (i.e., $e \rightarrow e'$), then e must also be in c .

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

Given a nonempty set of objects D and a system Sys , an interpretation function π associates to each cut c and symbol s in Φ 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 $\pi(c, f(t_1, \dots, t_k))$ by induction on the structure of terms, taking

$$\pi(c, f(t_1, \dots, t_k)) = \pi(c, f)(\pi(c, t_1), \dots, \pi(c, t_k)).$$

Using V and π , we define what it means for a formula ϕ to be true at the consistent cut c in event structure E in system Sys , denoted $(Sys, E, c, \pi, V) \models \phi$, by induction on the structure of ϕ , in the usual way. For example

- $(Sys, E, c, \pi, V) \models K_i \varphi$ iff $\forall E' \in Sys$, and c' cut of E' such that $c' \sim_i c$, we have $(Sys, E', c', \pi, V) \models \varphi$
- $(Sys, E, c, \pi, V) \models \Box \varphi$ iff for all cuts c' of E such that $c \preceq c'$, we have $(Sys, E, c', \pi, V) \models \varphi$.

As usual, we take $\Diamond \phi$ to be an abbreviation of $\neg \Box \neg \phi$, so that $\Diamond \phi$ is true at (E, c) if there is some cut c' extending c where ϕ is true. Similarly we write $\Diamond \varphi$ if there was a time in past when φ was true, or φ holds at cut c . The complete definition of \models is given in the appendix.

The satisfaction relation \models can be expressed as a formula in Nuprl. More precisely, there is a translation T (which we define in the appendix) such that for all tuples (Sys, E, c) , domains D , interpretations π , valuations V , and formulas φ , $T(Sys, E, c, D, \pi, V \varphi)$ is true iff $(Sys, E, c, V, \pi) \models \varphi$. There is a subtlety though. First-order epistemic logic assumes the principle of excluded middle; Nuprl is a constructive type theory that does not. To prove that T has the desired properties, we need to 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.

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 allows 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 S that associates with every program Pg of type Pgm the system $S(Pg)$ consisting of all the runs consistent with Pg . 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 $Pgm^{kb} = Sem \rightarrow Pgm$. 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 Pg^{kb} if it is a fixed point of this function: Sys represents the kb program Pg^{kb} if $S(Pg^{kb} Sys) = Sys$. Following Fagin et al. [13, 14], we take the semantics of a kb program Pg^{kb} to be the set of systems that represent Pg^{kb} .

Definition 2. A *kb program semantics* S^{kb} is a function of type $Pgm^{kb} \rightarrow \mathcal{P}(Sem)$, where $\mathcal{P}(Sem)$ is the type whose elements are sets of systems.

As observed by Fagin et al. [13, 14], 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 kb programs, are represented by a unique system; indeed, $S^{kb}(Pg) = \{S(Pg)\}$. Thus, we can view S^{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 [13, 14], we take a kb specification to be a predicate on systems.

Definition 3. A *kb specification* is an object of type $Sem \rightarrow \mathcal{P}$. A kb program Pg^{kb} satisfies a kb specification X^{kb} , written $Pg^{kb} \models X^{kb}$, if all the systems representing Pg^{kb} satisfy X and $S^{kb}(Pg^{kb}) \neq \emptyset$; i.e., $\forall Sys \in S^{kb}(Pg^{kb}). X^{kb}(Sys) \wedge S^{kb}(Pg^{kb}) \neq \emptyset$.

3.3 Example

Recall that in Section 2 a specification $Fair(P, f, l)$ was defined 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 $Fair-Pg(P, f, l)$. $Fair(P, f, l)$ can be generalized to a kb specification $Fair^{kb}(P^{kb}, f^{kb}, l)$ where, instead of using a precondition P and function f , we use a *kb predicate* P^{kb} and a *kb function* f^{kb} , both of which take a system as an extra argument (in addition to the other arguments of P and f). $Fair^{kb}(P^{kb}, f^{kb}, l)$ asserts that, in every run of the system, infinitely often either the kb precondition fails or a receive event with the value given by f^{kb} occurs on link l :

$$\begin{aligned} Fair^{kb}(P^{kb}, f^{kb}, l) &\equiv \\ \forall e'. kind(e') = rcv(l) &\Rightarrow P^{kb}@send(e') \wedge val(e') = f^{kb}@send(e') \\ \wedge \forall e@source(l). \exists e' \geq e. &kind(e') = rcv(l) \vee \forall v'. \neg(P^{kb}(state(after(e'))), v') \end{aligned}$$

$Fair^{kb}(P^{kb}, f^{kb}, l)$ is satisfied by a kb program $Fair-Pg^{kb}(P^{kb}, f^{kb}, l)$:

$$\begin{aligned} a(v) \text{ only if } P^{kb}(state, v) &\oplus \\ a(v) \text{ sends } f^{kb}(state, v) \text{ on } l &\oplus \\ \text{only events in local}(a) \text{ send on } l & \end{aligned}$$

$Fair-Pg^{kb}(P^{kb}, f^{kb}, l)$ associates to each system Sys the program $Fair-Pg(P^{kb}(Sys), f^{kb}(Sys), l)$; in system Sys , a process following $Fair-Pg^{kb}(P^{kb}, f^{kb}, l)$ sends a message with value determined by $f^{kb}(Sys)$ exactly when $P^{kb}(Sys)$ is true.

Lemma 2. $Fair-Pg^{kb}(P^{kb}, f^{kb}, l)$ satisfies the specification $Fair^{kb}(P^{kb}, f^{kb}, l)$.

4 The Sequence Transmission Problem

In this section, we illustrate how programs can be extracted from knowledge-based specifications using the Nuprl system. We do the extraction in two stages. In the first stage, we use Nuprl to prove that the specification is satisfiable. The proof proceeds by refinement: at each step, a rule or tactic (i.e. sequence of rules invoked under a single name) is applied, and new subgoals are generated; when there are no more subgoals to be proved, the proof is complete. The proof is automated, in the sense that subgoals are generated by the system upon tactic invocation. From the proof, we can extract a knowledge-based program Pg^{kb} that satisfies the specification. In the second stage, we find standard programs that implement Pg^{kb} .

We illustrate this methodology by applying it to one of the problems that has received considerable attention in the context of knowledge-based programming, *the sequence transmission problem* (stp). The stp involves a sender S that has an input tape with a (possibly infinite) sequence $X = X[0], X[1], \dots$ 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.

Halpern and Zuck [15] define two kb programs that solve this problem, and show that a number of standard programs in the literature, like Stenning's algorithm [20], the alternating bit protocol [3], and Aho, Ullman and Yannakakis's [1] algorithms are all particular instances of these programs. Sanders [19] derives a number of kb and standard programs for the same problem, with a focus on the more practical aspects of program development. Our method uses ideas from both of these earlier works.

If messages cannot be lost, duplicated, reordered, or corrupted, then S could simply send the bits in X to R in order. However, we are interested in solutions to the stp in contexts where communication is not reliable. Without some constraints, the stp is unsolvable; following [15], 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 by Fagin et al. in [13], 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. 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 condition of the form

$$\varphi^{kb} \stackrel{def}{=} \forall i \diamond K_R(X[i]).$$

One way to achieve this condition is by requiring that the receiver makes progress: for each i , if for all $j < i$ there was a time when R knew $X[j]$, then eventually R knows $X[i]$.

Intuitively, the sender is responsible for R 's progress. But how can S ensure that R learns the i^{th} bit? For any finite number n , it is possible that a message sent n times is not received. Fortunately, the fairness condition ensures that if $X[i]$ is sent an unbounded number of times, R will receive it. Thus, S can ensure that R learns the i^{th} bit if, infinitely often, either S sends $X[i]$ or S knows that R knew $X[i]$ at some time in the past. This is similar in spirit to the specification $\text{Fair}^{kb}(P^{kb}, f^{kb}, l)$ described in Example 3.3. In this case, l is the communication link from S to R , f^{kb} encodes the least bit that S does not know that R knew at some point in the past, i.e., the pair $(i, X[i])$ for the least index i such that $\neg K_S \diamond K_R X[i]$ holds. P^{kb} is instantiated with a test on S 's knowledge such that whenever P^{kb} fails, $\forall i \diamond K_R X[i]$ holds. Thus, we take $P^{kb} \equiv \exists i \neg K_S (\diamond K_R X[i])$. Note that, unless at some point S knows that R knows the whole sequence P^{kb} will be *true*. (Indeed, in many settings, we can just take P^{kb} to be the formula *true*.) We abbreviate this specification as $\text{Fair}^{kb}(P_S^{kb}, f_S^{kb}, l_{SR})$.

How does the sender learn which bits the receiver knows? One possibility is for S to receive from R a request to send $X[i]$. This can be taken by S to be a signal that R knows all the preceding bits. R 's program for sending this request to S can again be viewed as an instance of the specification $\text{Fair}^{kb}(P^{kb}, f^{kb}, l)$, this time for l the communication link from R to S , and f^{kb} returning the least index i such that R never knew $X[i]$. We take P^{kb} to be $\exists i \neg \diamond K_R X[i]$ (again, in many contexts, we can take it to be simply *true*) and abbreviate this specification as $\text{Fair}^{kb}(P_R^{kb}, f_R^{kb}, l_{RS})$.

Up to this point we have only used our intuition to guess a plausible refinement for our initial specification φ^{kb} . We can now use the system to verify this intuition. That is, we prove that the satisfiability of φ^{kb} follows from the satisfiability of each of $\text{Fair}^{kb}(P_R^{kb}, f_R^{kb}, l_{RS})$ and $\text{Fair}^{kb}(P_S^{kb}, f_S^{kb}, l_{SR})$ separately:

$$\begin{aligned} \text{Goal:} & \models \varphi^{kb} \\ \text{Subgoal 1:} & \models \text{Fair}^{kb}(P_R^{kb}, f_R^{kb}, l_{RS}) \\ \text{Subgoal 2:} & \models \text{Fair}^{kb}(P_S^{kb}, f_S^{kb}, l_{SR}) \\ \text{Subgoal 3:} & (\models \text{Fair}^{kb}(P_R^{kb}, f_R^{kb}, l_{RS})) \wedge (\models \text{Fair}^{kb}(P_S^{kb}, f_S^{kb}, l_{SR})) \Rightarrow (\models \varphi^{kb}) \end{aligned}$$

The proof is carried out in Nuprl in two steps. Using Lemma 2, we can prove that there exist kb programs, $\text{Fair-Pg}^{kb}(P_R^{kb}, f_R^{kb}, l_{RS})$ and $\text{Fair-Pg}^{kb}(P_S^{kb}, f_S^{kb}, l_{SR})$, respectively, that satisfy both fairness specifications; i.e.,

$$\begin{aligned} \text{Fair-Pg}^{kb}(P_R^{kb}, f_R^{kb}, l_{RS}) & \models \text{Fair}^{kb}(P_R^{kb}, f_R^{kb}, l_{RS}) \\ \text{Fair-Pg}^{kb}(P_S^{kb}, f_S^{kb}, l_{SR}) & \models \text{Fair}^{kb}(P_S^{kb}, f_S^{kb}, l_{SR}). \end{aligned}$$

Finally we have to check that the combination of the two programs satisfies the conjunction of the specifications satisfied by each program separately. This means that we have to inspect the frame conditions for both programs, i.e. the conditions that allow only R to send messages of the form i to S , and only S to send pairs $\langle i, X[i] \rangle$ to R . Since messages sent by the programs go on separate links, the frame conditions are easily seen to be compatible with the

effect clauses. Thus, we can prove

$$\text{Fair-Pg}^{kb}(P_R^{kb}, f_R^{kb}, l_{RS}) \oplus \text{Fair-Pg}^{kb}(P_S^{kb}, f_S^{kb}, l_{SR}) \models \varphi^{kb}.$$

Using the program notation of Fagin et al. [13], $\text{Fair-Pg}^{kb}(P_S^{kb}, f_S^{kb}, l_{SR})$ is the following program:

```

if  $\exists i K_S(\diamond K_RX[0] \wedge \dots \wedge \diamond K_RX[i-1]) \wedge \neg K_S(\diamond K_RX[i])$ 
then  $\text{send}_{1_{sr}}(\langle i, X[i] \rangle)$  else skip
    
```

and $\text{Fair-Pg}^{kb}(P_R^{kb}, f_R^{kb}, l_{RS})$ is

```

if  $\exists i \diamond K_RX[0] \wedge \dots \wedge \diamond K_RX[i-1] \wedge \neg \diamond K_RX[i]$  then  $\text{send}_{1_{rs}}(i)$  else skip.
    
```

Notice that whenever S sends a message $\langle i, X[i] \rangle$, i is the minimum index for which $\neg K_S(\diamond K_RX[i])$; similarly, R always sends the minimum i for which $\neg \diamond K_RX[i]$. We can make this explicit by letting S and R have local variables, say i and j , to keep track of these minimum indices. i and j are initially set to 0; S increments i from the current value v to $v+1$ whenever he learns that $\diamond K_RX[v]$ holds, and similarly, R increments j from v to $v+1$ whenever he learns $X[v]$. The knowledge test in the sender program can then be rewritten as $\neg K_S(\diamond K_RX[v])$, for v the current value of i , and the knowledge test for the receiver becomes $\neg \diamond K_RX[j]$; thus the derived program is essentially one of the knowledge-based programs considered by Halpern and Zuck in [15].

This is not surprising, since our derivation followed much the same reasoning as that of Halpern and Zuck. However, note that we did not first give a kb program and then verify that it satisfied the specification. Rather, we derived the kb programs for the sender and receiver from the proof that the specification was satisfiable. And, while Nuprl required “hints” from us in terms of what to prove, the key ingredients of the proof, namely, the specification $\text{Fair}(P, f, l)$ and the proof that $\text{Fair-Pg}(P, f, l)$ realizes it, were already in the system, having been used in other contexts. Thus, this suggests that we may be able to apply similar techniques to derive programs satisfying other specifications in communication systems with only weak fairness guarantees.

This takes care of the first stage of the synthesis process. We now want to find a standard program that implements the knowledge-based program. As discussed by Halpern and Zuck [15], the exact standard program that we use depends on the underlying assumptions about the communications systems. Here we sketch an approach to finding the standard program.

The first step is to identify the exact properties of knowledge that are needed for the proof. We can inspect the proof and identify which properties of the knowledge operators K_S and K_R seem to be used. We replace $\diamond(K_R(X[i] = v))$ by an abstract predicate $Q(X[i] = v)$ and $K_S(\diamond K_R(X[i] = v))$ by $P(X[i] = v)$. As before, we abbreviate $P(X[i] = 0) \vee P(X[i] = 1)$ as $P(X[i])$, and $Q(X[i])$ for $Q(X[i] = 0) \vee Q(X[i] = 1)$. We add as hypotheses all the identified properties, now as properties of Q and P , and check whether the former proof still applies. If not, we add whatever additional properties are needed. Note that we can use Nuprl to automate these checks.

This approach enables us to prove that the specification is satisfiable in a more general setting. The specification is now written in terms of P and Q , and denoted $\tilde{\varphi}^{kb}$. $Fair^{kb}(P_R^{kb}, f_R^{kb}, l_{RS})$ is replaced by $Fair^{kb}(\tilde{Q}, \tilde{f}_R, l_{RS})$ with $\tilde{Q} \stackrel{def}{=} \exists i \neg Q(X[i])$, and similarly, $Fair^{kb}(P_S^{kb}, f_S^{kb}, l_{SR})$ becomes $Fair^{kb}(\tilde{P}, \tilde{f}_S, l_{SR})$ with $\tilde{P} \stackrel{def}{=} \exists i \neg P(X[i])$; whenever \tilde{Q} and \tilde{P} hold, \tilde{f}_R and \tilde{f}_S return the minimum index i such that $\neg Q(X[i])$, and $\neg P(X[i])$, respectively.

The new theorem states that, under suitable hypotheses about P and Q , $\tilde{\varphi}^{kb}$ is satisfiable since both $Fair^{kb}(\tilde{Q}, \tilde{f}_R, l_{RS})$ and $Fair^{kb}(\tilde{P}, \tilde{f}_S, l_{SR})$ are satisfiable. One hypothesis we require is that P and Q must be *sound* tests, this is $P(X[k] = v)$ and $Q(X[k] = v)$ both imply $X[k] = v$; this is clearly true of knowledge predicates. We must also assume that after R receives a message of the form $\langle i, v \rangle$, $Q(X[i] = v)$ holds; similarly, after S receives some message i , $\forall j < i P(X[j])$ holds. The extracted program is

$$Fair-Pg^{kb}(\tilde{Q}, \tilde{f}_R, l_{RS}) \oplus Fair-Pg^{kb}(\tilde{P}, \tilde{f}_S, l_{SR}),$$

where $Fair-Pg^{kb}(\tilde{P}, \tilde{f}_S, l_{SR})$, written using Fagin et al. notation, is

if $\exists i P(X[0]) \wedge \dots \wedge P(X[i-1]) \wedge \neg P(X[i])$ then $\text{send}_{1_{SR}}(\langle i, X[i] \rangle)$ else skip,

and $Fair-Pg^{kb}(\tilde{Q}, \tilde{f}_R, l_{RS})$ is

if $\exists i Q(X[0]) \wedge \dots \wedge Q(X[i-1]) \wedge \neg Q(X[i])$ then $\text{send}_{1_{RS}}(i)$ else skip.

Clearly, this is a generalization of the first kb-program for the stp. Furthermore, it is clear that other predicates P and Q satisfy these hypotheses. For example, suppose that S has a state variable i_S such that $P(X[0]) \wedge \dots \wedge P(X[i_S - 1]) \wedge \neg P(X[i_S])$ holds at the current state of S ; similarly, R has a state variable i_R such that $Q(X[0]) \wedge \dots \wedge Q(X[i_R - 1]) \wedge \neg Q(X[i_R])$ holds. We can then simply define $P'(X[k]) \stackrel{def}{=} i_S > k$ and $Q'(X[k]) \stackrel{def}{=} i_R > k$; the resulting program is exactly Stenning's [20] protocol.

The key point here is that by replacing the knowledge tests by weaker predicates that imply them and do not explicitly mention knowledge, we can derive standard programs that implement the knowledge-based program. We believe that other standard implementations of the knowledge-based program can be derived in a similar way, although we have not yet concluded the derivation. We hope to report on this shortly.

References

1. A. V. Aho, J. D. Ullman, A. D. Wyner, and M. Yannakakis. Bounds on the size and transmission rate of communication protocols. *Computers and Mathematics with Applications*, 8(3):205–214, 1982. This is a later version of [2].
2. A. V. Aho, J. D. Ullman, and M. Yannakakis. Modeling communication protocols by automata. In *Proc. 20th IEEE Symp. on Foundations of Computer Science*, pages 267–273. 1979.

3. K. A. Bartlett, R. A. Scantlebury, and P. T. Wilkinson. A note on reliable full-duplex transmission over half-duplex links. *Communications of the ACM*, 12:260–261, 1969.
4. J. L. Bates and Robert L. Constable. Proofs as programs. *ACM Transactions on Programming Languages and Systems*, 7(1):53–71, 1985.
5. Mark Bickford and Robert L. Constable. A logic of events. Technical Report TR2003-1893, Cornell University, 2003.
6. Mark Bickford, Christoph Kreitz, Robbert van Renesse, and Xiaoming Liu. Proving hybrid protocols correct. In Richard Boulton and Paul Jackson, editors, *14th International Conference on Theorem Proving in Higher Order Logics*, volume 2152 of *Lecture Notes in Computer Science*, pages 105–120, Edinburgh, Scotland, September 2001. Springer-Verlag.
7. K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, Reading, Mass., 1988.
8. Robert L. Constable. Naïve computational type theory. In H. Schwichtenberg and R. Steinbrüggen, editors, *Proof and System-Reliability, Proceedings of International Summer School Marktoberdorf, July 24 to August 5, 2001*, volume 62 of *NATO Science Series III*, pages 213–260, Amsterdam, 2002. Kluwer Academic Publishers.
9. Robert L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, NJ, 1986.
10. C. Dwork and Y. Moses. Knowledge and common knowledge in a Byzantine environment: crash failures. *Information and Computation*, 88(2):156–186, 1990.
11. K. Engelhardt, R. van der Meyden, and Y. Moses. A program refinement framework supporting reasoning about knowledge and time. In J. Tiuryn, editor, *Proc. Foundations of Software Science and Computation Structures (FOSSACS 2000)*, pages 114–129. Springer-Verlag, Berlin/New York, 1998.
12. K. Engelhardt, R. van der Meyden, and Y. Moses. 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*, pages 125–141. Springer-Verlag, Berlin/New York, 2001.
13. R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. *Reasoning about Knowledge*. MIT Press, Cambridge, Mass., 1995.
14. R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. Knowledge-based programs. *Distributed Computing*, 10(4):199–225, 1997.
15. J. Y. Halpern and L. D. Zuck. 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, 1992.
16. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.
17. Nancy Lynch and Mark Tuttle. An introduction to Input/Output automata. *Centrum voor Wiskunde en Informatica*, 2(3):219–246, September 1989.
18. P. Panangaden and S. Taylor. Concurrent common knowledge: defining agreement for asynchronous systems. *Distributed Computing*, 6(2):73–93, 1992.
19. B. Sanders. A predicate transformer approach to knowledge and knowledge-based protocols. In *Proc. 10th ACM Symp. on Principles of Distributed Computing*, pages 217–230, 1991. A revised report appears as ETH Informatik Technical Report 181, 1992.
20. M. V. Stenning. A data transfer protocol. *Comput. Networks*, 1:99–110, 1976.
21. F. Stulp and R. Verbrugge. A knowledge-based algorithm for the Internet protocol (TCP). *Bulletin of Economic Research*, 54(1):69–94, 2002.

A Translating \models into Nuprl

Using valuation V and interpretation π , the fact that formula ϕ is true at the consistent cut c in event structure E in system Sys is denoted $(Sys, E, c, \pi, V) \models \phi$ and defined by induction on the structure of ϕ as follows:

- if P is a predicate symbol in Φ of some arity k , and t_1, \dots, t_k are terms, then $(Sys, E, c, \pi, V) \models P(t_1, \dots, t_k)$ iff $\pi(c, P)(\pi(c, t_1), \dots, \pi(c, t_k))$
- $(Sys, E, c, \pi, V) \models \neg\phi$ iff $(Sys, E, c, \pi, V) \not\models \phi$
- $(Sys, E, c, \pi, V) \models \varphi_1 \wedge \varphi_2$ iff $(Sys, E, c, \pi, V) \models \varphi_1$ and $(Sys, E, c, \pi, V) \models \varphi_2$
- $(Sys, E, c, \pi, V) \models \forall x.\varphi$ iff, for all $d \in D$, $(Sys, E, c, \pi, V[x/d]) \models \varphi$, where $V[x/d]$ is the valuation that agrees with V on all variables except possible x , and $V[x/d](x) = d$
- $(Sys, E, c, \pi, V) \models K_i\varphi$ iff for all $E' \in Sys$ and cuts c' of E' such that $c' \sim_i c$, $(Sys, E', c', \pi, V) \models \varphi$
- $(Sys, E, c, \pi, V) \models \Box\varphi$ iff for all cuts c' of E such that $c \preceq c'$, we have $(Sys, E, c', \pi, V) \models \varphi$
- $(Sys, E, c, \pi, V) \models \Diamond\varphi$ iff there exists a cut c' of E such that $c \preceq c'$ and $(Sys, E, c', \pi, V) \models \varphi$
- $(Sys, E, c, \pi, V) \models \heartsuit\varphi$ iff there exists a cut c' of E such that $c' \preceq c$ and $(Sys, E, c', \pi, V) \models \varphi$.

We can now define the translation T by induction on the structure of formulas:

- if P is a predicate symbol in Φ of some arity k , and t_1, \dots, t_k are terms, then $T(Sys, E, c, D, \pi, V, P(t_1, \dots, t_k)) = \pi(c, P)(\pi(c, t_1), \dots, \pi(c, t_k))$
- $T(Sys, E, c, D, \pi, V, \neg\varphi) = \neg(T(Sys, E, c, D, \pi, V, \varphi))$
- $T(Sys, E, c, D, \pi, V, \varphi_1 \wedge \varphi_2) = (T(Sys, E, c, D, \pi, V, \varphi_1)) \wedge (T(Sys, E, c, D, \pi, V, \varphi_2))$
- $T(Sys, E, c, D, \pi, V, (\forall x.\varphi)) = \forall x.(T(Sys, E, c, D, \pi, V, (\varphi(V x))))$
- $T(Sys, E, c, D, \pi, V, K_i\varphi) = \forall E'. E' \in Sys \Rightarrow \forall c'. c' \sim_i c \Rightarrow (T(Sys, E', c', D, \pi, V, \varphi))$
- $T(Sys, E, c, D, \pi, V, \Box\varphi) = \forall c'. c \preceq c' \Rightarrow (T(Sys, E, c', D, \pi, V, \varphi))$

From this definition it follows that, assuming the principle of excluded middle, $T(Sys, E, c, D, \pi, V\varphi)$ is provable iff $(Sys, E, c, V, \pi) \models \varphi$.

Proposition 1. *Assuming the principle of excluded middle, for all tuples (Sys, E, c) , domains D , interpretations π , valuations V , and formulas φ , $T(Sys, E, c, D, \pi, V, \varphi)$ is provable iff $(Sys, E, c, V, \pi) \models \varphi$.*