Knowledge-Based Synthesis of Distributed Systems Using Event Structures*

Mark Bickford†
Cornell University
Ithaca, NY 14853
markb@cs.cornell.edu

Robert C. Constable‡
Cornell University
Ithaca, NY 14853
rc@cs.cornell.edu

Joseph Y. Halpern§
Cornell University
Ithaca, NY 14853
halpern@cs.cornell.edu

Sabina Petride§
Cornell University
Ithaca, NY 14853
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]. Here programs are constructed

*A preliminary version of this paper appeared in Proceedings of the 11th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning LPAR 2004, pp. 449-465.

†Supported in part by AF-AFOSR F49620-02-1-0170.

‡Supported in part by ONR N00014-02-1-0455 and NSF 0208535.

§Supported in part by NSF under grants ITR-0325453, CCR-0208535, IIS-0534064, and IIS-0812045, by ONR under grant N00014-02-1-0455, by the DoD Multidisciplinary University Research Initiative (MURI) program administered by ONR under grants N00014-01-1-0795 and N00014-04-1-0725, and by AFOSR under grants F49620-02-1-0101 and FA9550-05-1-0055.

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, and to specify them in the language of I/O Automata [Bickford, Kreitz, Renesse, and Liu 2001] Recent work by two of the authors [Bickford and Constable 2003] 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 and Constable 2003]. However, as has long been recognized [Fagin, Halpern, Moses, and Vardi 1995], 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* [Fagin, Halpern, Moses, and Vardi 1995; Fagin, Halpern, Moses, and Vardi 1997]. 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 have been given precise semantics by Fagin et al. [1995, 1997]. 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 [Dwork and Moses 1990; Halpern and Zuck 1992; Stulp and Verbrugge 2002].

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 [1992] provide 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 programs. Here we show that one of these 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 program 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.

The paper is organized as follows. In the next section we give a brief overview of the Nuprl system, review event theory, discuss the type of programs we use (distributed message automata), and show how automata can be synthesized from a specification. In Section 3 we review epistemic logic, show how it can be translated into Nuprl, and show how knowledge-based automata can be captured in Nuprl. The sequence transmission problem is analyzed in Section 4. We conclude with references to related work and a discussion on future research in Section 5.

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 [Constable] 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, on 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)". Reasoning in the Nuprl type theory is intuitionistic [Brouwer 1923], in the sense that proving a certain fact is understood as constructing evidence for that fact. For example, a proof of the fact that "there exists x of type A" builds an object of type A, and a proof of the fact "for any object x of type A there exists an object y of type B such that the relation $R(x, y)$ holds" builds a function f that associates with each object a of type A an object b of type B such that $R(a, b)$ holds.

One consequence of this approach is that the principle of excluded middle does not apply: while in classical logic, $\varphi \vee \neg\varphi$ holds for all formulas φ, in constructive type theory, it holds exactly when we have evidence for either φ or $\neg\varphi$, and we can tell from this evidence which of φ and $\neg\varphi$ it supports. A predicate $Determinate$ is definable in Nuprl such that $Determinate(\varphi)$ is true iff the principle of excluded middle holds for formula φ. (From here on in, when we say that a formula is true, we mean that it is constructively true, that is, provable in Nuprl.)

In this paper, we focus on synthesizing programs from specifications. Thus we must formalize these notions in Nuprl. As a first step, we define a type Pgm in Nuprl and take programs to be objects of type Pgm. Once we have defined Pgm, we can define other types of interest.

Definition 2.1: A *program semantics* is a function S of type $Pgm \rightarrow Sem$ assigning to each *program* Pg of type Pgm a *meaning* of type $Sem = 2^{Sem'}$. Sem' is the type of *executions* consistent with the program Pgm under the semantics S. A *specification* is a predicate X on Sem'. A program Pg *satisfies* the specification X if $X(e)$ holds for all e in $S(Pg)$. A specification X is *satisfiable* if there exists a program that satisfies X. ∎

As Definition 2.1 suggests, all objects in Nuprl are typed. To simplify our discussion, we typically suppress the type declarations. Definition 2.1 shows that the satisfiability of a specification is definable in Nuprl. The key point for the purposes of this paper is that from a constructive proof that X is satisfiable, 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. For 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,

(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* [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).

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), 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. We have shown [Bickford and Constable 2003] that, although satisfiability is undecidable, there is indeed a small set of rules from which we can prove satisfiability in many cases of interest; these rules are discussed in Section 2.3.

2.2 Event structures

Consider a set AG of processes or *agents*; associated with each agent i in AG is a set X_i 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. We assume that the sets of local variables of different agents are disjoint. Information is communicated by message passing. The set of links is $Links$. Sending a message on some link $l \in Links$ is understood as enqueuing the message on l, while receiving a message corresponds to dequeuing the message. Communication is point-to-point: for each link l there is a unique agent $source(l)$ that can send messages on l, and a unique agent $dest(l)$ that can receive message on l. For each agent i and link l with $source(l) = i$, we assume that $msg(l)$ is a local variable in X_i.

We assume that communication is asynchronous, so there is no global notion of time. Following Lamport [1978], changes to the local state of an agent are modeled as *events*. Intuitively, when an event "happens", an agent either sends a message, 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.

Lamport's theory of events is the starting point of our formalism. To help in writing concrete and detailed specifications, we add more structure to events. Formally, an event is a tuple with three components. The first component of an event e is an agent $i \in AG$, intuitively the agent whose local state changes during event e. We denote i as $agent(e)$. The second component of e is its *kind*, which is either a link l with $dest(l) = i$ or a local action a, an element of some given set Act of local actions. The only actions in Act are those that set local variables to certain values. We denote this component as $kind(e)$. We often write $kind(e) = rcv(l)$ rather than $kind(e) = l$ to emphasize the fact that e is a receive event; similarly we write $kind(e) = local(a)$ rather than $kind(e) = a$ to emphasize the fact that a is a local action. The last component of e is its *value* v, a tuple of elements in some domain Val; we denote this component as $val(e)$. If e is a receive event, then $val(e)$ is the message received when e occurs; if e is a local event a, then $val(e)$ represents the tuple of values to which the variables are set by a. (For more details on the reasons that led to this formalism, see [Bickford and Constable 2005].)

Rather than having a special kind to model send events, we model the sending of a message on link

l by changing the value of a local variable $msg(l)$ that describes the message sent on l. A special value $\perp$ indicates that no message is sent when the event occurs; changing $msg(l)$ to a value other than $\perp$ indicates that a message is sent on l. This way of modeling send events has proved to be convenient. One advantage is that we can model multicast: the event e of i broadcasting a message m to a group of agents just involves a local action that sets $msg(l)$ to m for each link l from i to one of the agents in the group. Similarly, there may be an action in which agent i sends a message to some agents and simultaneously updates other local variables.

Following Lamport [1978], we model an execution of a distributed program as a sequence of events satisfying a number of natural properties. We call such a sequence an event structure. We take an event structure es to be a tuple consisting of a set E of events and a number of additional elements that we now describe. These elements include the functions $dest$, $source$, and msg referred to above, but there are others. For example, Lamport assumes that every receive event e has a corresponding (and unique) event where the message received at e was sent. To capture this in our setting, we assume that the description of the event structure es includes a function $send$ whose domain is the receive events in es and whose range is the set of events in es; we require that $agent(send(e)) = source(l)$. Note that, since we allow multicasts, different receive events may have the same corresponding send event.

For each $i \in AG$, we assume that the set of events e in es associated with i is totally ordered. This means that, for each event e, we can identify the sequence of events (*history*) associated with agent i that preceded e. To formalize this, we assume that, for each agent $i \in AG$, the description of es includes a total order $\prec_i$ on the events e in es such that $agent(e) = i$. Define a predicate first and function $pred$ such that $\mathit{first}(e)$ holds exactly when e is the first event in the history associated with $agent(e)$ in es; if $\mathit{first}(e)$ does not hold, then $pred(e)$ is the unique predecessor of e in es. Following Lamport [1978], we take $\prec$ to be the least transitive relation on events in es such that $send(e) \prec e$ if e is a receive event and $e \prec e'$ if $e \prec_i e'$. We assume that $\prec$ is well-founded. We abbreviate $(e' \prec e) \vee (e = e')$ as $e' \preceq e$, or $e \succeq e'$. Note that $\prec_i$ is defined only for events associated with agent i: we write $e \prec_i e'$ only if $agent(e) = agent(e') = i$.

The local state of an agent defines the values of all the variables associated with the agent. While it is possible that an event structure contains no events associated with a particular agent, for ease of exposition, we consider only event structures in which each agent has at least one local state, and denote the initial local state of agent i as $initstate_i$. In event structures es where at least one event associated with a given agent i occurs, $initstate_i$ represents i's local state before the first event associated with i occurs in es. Formally, the *local state* of an agent i is a function that maps X_i and a special symbol $\mathbf{val}_i$ to values. (The role of $\mathbf{val}_i$ will be explained when we give the semantics of the logic.) If $x \in X_i$, we write $s(x)$ to denote the value of x in i's similarly, $s(\mathbf{val}_i)$ is the value of $\mathbf{val}_i$ in s. If $agent(e) = i$, we take $state\ before\ e$ to be the local state of agent i before e; similarly, $state\ after\ e$ denotes i's local state after event e occurs. The value $(state\ after\ e)(x)$ is in general different from $(state\ before\ e)(x)$. How it differs depends on the event e, and will be clarified in the semantics. We assume that $(state\ after\ e)(\mathbf{val}_i) = val(e)$; that is, the value of the special symbol $\mathbf{val}_i$ in a local state is just the value of the event that it follows. If $x \in X_i$, we take $x\ before\ e$ to be an abbreviation for $(state\ before\ e)(x)$; that is, the value of x in the state before e occurs; similarly, $x\ after\ e$ is an abbreviation for $(state\ after\ e)(x)$.

Example 2.2: Suppose that Act contains $send$ and $send{+}inc(x)$, where $x \in X_i$, and that Val contains the natural numbers. Let n and v be natural numbers. Then

- the event of agent i receiving message m on link l in the event structure es is modeled by the tuple $e = (i, l, m)$, where $agent(e) = i$, $kind(e) = rcv(l)$, and $val(e) = m$;

- the event of agent i sending message n on link l in es is represented by the tuple $e = (i, send, m)$, where $msg(l)$ $after$ $e = m$;

- the event e of agent i sending m on link l and incrementing its local variable x by v in es is represented by the tuple e such that $agent(e) = i$, $kind(e) = send{+}inc(x)$, and $val(e) = \langle m, v\rangle$, where $msg(l)$ $after$ $e = m$ and x $after$ $e = x$ $before$ $e + v$.

Definition 2.3: An event structure is a tuple $es = \langle AG, Links, source, dest, Act, \{X_i\}_{i\in AG}, Val,$ $\{initstate_i\}_{i\in AG}, E,$ $agent, send, first, \{\prec_i\}_{i\in AG}, \prec\rangle$ where AG is a set of agents, $Links$ is a set of links such that $source : Links \longrightarrow AG$, $dest : Links \longrightarrow AG$, Act is a set of actions, X_i is a set of variables for agent $i \in AG$ such that, for all links $l \in Links$, $msg(l) \in X_i$ if $i = source(l)$, Val is a set of values, $initstate_i$ is the initial local state of agent $i \in AG$, E is a set of events for agents AG, kinds $Kind = Links \cup Act$, and domain Val, functions $agent$, $send$ and $first$ are defined as explained above, $\prec_i$s are local precedence relations and $\prec$ is a causal order such that the following axioms, all expressible in Nuprl, are satisfied:

- if e has kind $rcv(l)$, then the value of e is the message sent on l during event $send(e)$, $agent(e) = dest(l)$, and $agent(send(e)) = source(l)$:

$$\begin{array}{l}\forall e \in es.\forall l.\ (kind(e) = rcv(l)) \Rightarrow \\ \quad (val(e) = msg(l)\ after\ send(e)) \wedge (agent(e) = dest(l)) \wedge (agent(send(e)) = source(l))\end{array}$$

- for each agent i, events associated with i are totally ordered:

$$\forall e \in es.\forall e' \in es.(agent(e) = agent(e') = i \Rightarrow e{\prec_i}e' \vee e'{\prec_i}e \vee e = e').$$

- e is the first event associated with agent i if and only if there is no event associated with i that precedes e:

$$\forall e \in es\ \forall i.\ (agent(e) = i) \Rightarrow (first(e) \Leftrightarrow \forall e' \in es.\ \neg(e'{\prec_i}e)).$$

- the initial local state of agent i is the state before the first event associated with i, if any:

$$\forall i.\ (\forall e \in es.\ (agent(e) = i \Rightarrow (first(e) \Leftrightarrow (state\ before\ e = initstate_i)))).$$

- the predecessor of an event e immediately precedes e in the causal order:

$$\begin{array}{l}\forall e \in es.\ \forall i.\ ((agent(e) = i) \wedge \neg first(e)) \Rightarrow \\ \quad ((pred(e){\prec_i}e) \wedge (\forall e' \in es.\ \neg(pred(e){\prec_i}e'{\prec_i}e))).\end{array}$$

- the local variables of agent $agent(e)$ do not change value between the predecessor of e and e:

$$\begin{array}{l}\forall e \in es.\ \forall i.\ (agent(e) = i \wedge \neg first(e)) \Rightarrow \\ \qquad\qquad \forall x \in X_i.\ (x\ after\ pred(e) = x\ before\ e).\end{array}$$

- the causal order $\prec$ is well-founded:

$$\forall P. \, (\forall e. \, (\forall e' \prec e. \, P(e')) \Rightarrow P(e)) \Rightarrow (\forall e. \, P(e)),$$

  where P is an arbitrary predicate on events. (It is easy to see that this axiom is sound if $\prec$ is well-founded. On the other hand, if $\prec$ is not well-founded, then let P be a predicate that is false exactly of the events e such that there there is an infinite descending sequence starting with e. In this case, the antecedent of the axiom holds, and the conclusion does not.)

In our proofs, we will need to argue that two events e and e' are either causally related or they are not. It can be shown [Bickford and Constable 2003] that this can be proved in constructive logic iff the predicate *first* satisfies the principle of excluded middle. We enforce this by adding the following axiom to the characterization of event structures:

$$\forall e \in es. \, Determinate(first(e)).$$

The set of event structures is definable in Nuprl (see [Bickford and Constable 2003]). We use event structures to model executions of distributed systems. We show how this can be done in the next section.

2.3 Distributed message automata

As we said, the programs we consider are *message automata*. Roughly speaking, we can think of message automata as nondeterministic state machines, though certain differences exist. Each basic message automaton is associated with an agent i; a message automaton associated with i essentially says that, if certain preconditions hold, i can take certain local actions. (We view $receive$ actions as being out of the control of the agent, so the only actions governed by message automata are local actions.) At each point in time, i nondeterministically decides which actions to perform, among those whose precondition is satisfied. We next describe the syntax and semantics of message automata.

2.3.1 Syntax We consider a first-order language for tests in automata. Fix a set AG of agents, a set X_i of local variables for each agent i in AG, and a set X^* of variables that includes $\cup_{i \in AG} X_i$ (but may have other variables as well). The language also includes special constant symbols $\mathbf{val}_i$, one for each agent $i \in AG$, predicate symbols in some finite set $\mathcal{P}$, and function symbols in some finite set $\mathcal{F}$. Loosely speaking, $\mathbf{val}_i$ is used to denote the value of an event associated with agent i; constant symbols other than $\mathbf{val}_1, \ldots, \mathbf{val}_n$ are just 0-ary function symbols in $\mathcal{F}$. We allow quantification only over variables other than local variables; that is, over variables $x \notin \cup_{i \in AG} X_i$.

Message automata are built using a small set of *basic programs*, which may involve formulas in the language above. Fix a set Act of local actions and a set $Links$ of links between agents in AG.[1] There are five types of basic programs for agent i:

- $@i$ **initially** ψ;
- $@i$ **if** $kind = k$ **then** $x := t$, where $k \in Act \cup Links$ and $x \in X_i$;

[1] We are being a little sloppy here, since we do not distinguish between an action a and the name for the action that appears in a program, and similarly for links and the variables in X_i.

- $@i$ $\mathit{kind} = \mathit{local}(a)$ **only if** φ;
- $@i$ **if necessarily** φ **then i.o.** $\mathit{kind} = \mathit{local}(a)$; and
- $@i$ **only** L **affects** x, where L is a list of kinds in $\mathit{Act} \cup \mathit{Links}$ and $x \in X_i$.

Note that all basic program for agent i are prefixed by $@i$.

We can form more complicated programs from simpler programs by *composition*. We can compose automata associated with different agents. Thus, the set (type) Pgm of programs is the smallest set that includes the basic programs such that if Pg_1 and Pg_2 are programs, then so is $Pg_1 \oplus Pg_2$.[2]

2.3.2 Semantics We give semantics by associating with each program the set of event structures consistent with the program. Intuitively, a set of event structures is consistent with a distributed message automaton if each event structure in the set can be seen as an execution of the automaton. The semantics can be defined formally in Nuprl as a relation between a distributed program Pg and an event structure es. In this section, we define the consistency relation for programs and give the intuition behind these programs.

In classical logic, we give meaning to formulas using an interpretation. In the Nuprl setting, we are interested in *constructive interpretations* I, which can be characterized by a formula φ_I. We can think of φ_I as characterizing a domain Val_I and the meaning of the fuction and predicate symbols. If I is an interpretation with domain Val_I, an *I-local state for i* maps $X_i \cup \{\mathbf{val}_i\}$ to Val_I; an *I-global state* is a tuple of I-local states, one for each agent in AG. Thus, if $s = (s_1, \ldots, s_n)$ is an I-global state, then s_i is i's local state in s. (Note that we previously used s to denote a local state, while here s denotes a global state. We will always make it clear whether we are referring to local or global states.)

For consistency with our later discussion of knowledge-based programs, we allow the meaning of some predicate and function symbols that appear in tests in programs to depend on the global state. We say that a function or predicate symbol is *rigid* if it does not depend on the global state. For example, if the domain is the natural numbers, we will want to treat $+$, $\times$, and $<$ as rigid. However, having the meaning of a function or predicate depend on the global state is not quite as strange as it may seem. For example, we may want to talk about an array whose values are encoded in agent 1's variables x_1, x_2, and x_3. An array is just a function, so the interpretation of the function may change as the values of x_1, x_2, and x_3 change. For each nonrigid predicate symbol P and function symbol f in $\mathcal{P} \cup \mathcal{F}$, we assume that there is a predicate symbol P^+ and function symbol f^+ whose arity is one more than that of P (resp., f); the extra argument that is a global state. We then associate with every formula φ and term t that appears in a program a formula φ^+ and term t^+ in the language of Nuprl. We define φ^+ by induction on the structure of φ. For example, for an atomic formula such as $P(c)$, if P and c are rigid, then $(P(c))^+$ is just $P(c)$. If P and c are both nonrigid, then $(P(c))^+$ is $P^+(c^+(\mathbf{s}), \mathbf{s})$, where $\mathbf{s}$ is a variable interpreted as a global state.[3] We leave to the reader the straightforward task of defining φ^+ and t^+ for atomic formulas and terms. We then take $(\varphi \wedge \psi)^+ = \varphi^+ \wedge \psi^+$, $(\neg\varphi)^+ = \neg\varphi^+$, and $(\forall x \varphi)^+ = \forall x \varphi^+$.

An *I-valuation* V associates with each non-local variable (i.e., variable not in $\cup_{i \in AG} X_i$) a value in Val_I. Given an interpretation I, an I-global state s, and an I-valuation V, we take $I_V(\varphi)(s)$ to be an

[2] Here we are deliberately ignoring the difference between sets and types.

[3] Since Nuprl is a higher-order language, there is no problem having a variable ranging over global states that is an argument to a predicate.

abbreviation for the formula (expressible in Nuprl) that says φ_I together with the conjunction of atomic formulas of the form $x = V(x)$ for all non-local variables x that appear in φ, $x = s_i(x)$ for variables $x \in X_i$, $i \in AG$, that appear in φ, and $\mathbf{s} = s$ implies φ^+. Thus, $I_V(\varphi)(s)$ holds if there is a constructive proof that the formula that characterizes I together with the (atomic) formulas that describe $V(x)$ and s, and a formula that says that $\mathbf{s}$ is represented by s, imply φ^+. It is beyond the scope of this paper (and not necessary for what we do here) to discuss constructive proofs in Nuprl; details can be found in [Constable et al. 1986]. However, it is worth noting that, for a first-order formula φ, if $I_V(\varphi)(s)$ holds, then φ^+ is true in state s with respect to the semantics of classical logic in I. The converse is not necessarily true. Roughly speaking, $I_V(\varphi)(s)$ holds if there is evidence for the truth of φ^+ in state s (given valuation V). We may have evidence for neither φ^+ nor $\neg\varphi^+$. We also take $I_V(t)(s)$ to be the value v such that there is a constructive proof of $I_V(t = v)(s)$. Note that this says that, just as we may not have evidence for either φ nor $\neg\varphi$ in constructive logic, not all terms are computable in Nuprl and $I_V(t)(s)$ may not be defined for all terms and states s.

A formula φ is an *i-formula in interpretation I* if its meaning in I depends only in i's local state; that is, for all global states s and s' such that $s_i = s'_i$, $I_V(\varphi)(s)$ holds iff $I_V(\varphi)(s')$ does. Similarly, t is an *i-term in I* if $x = t$ is an i-formula in I, for x a non-local variable. It is easy to see that φ is an i-formula in all interpretations I if all the predicate and function symbols in φ are rigid, and φ does not mention variables in X_j for $j \neq i$ and does not mention the constant symbol $\mathbf{val}_j$ for $j \neq i$. Intuitively, this is because if we have a constructive proof that φ holds in s with respect to valuation V, and φ is an i-formula, then all references to local states of agents other than i can be safely discarded from the argument to construct a proof for φ based solely on s_i. If φ is an i-formula, then we sometimes abuse notation and write $I_V(\varphi)(s_i)$ rather than $I_V(\varphi)(s)$. Note that the valuation V is not needed for interpreting formulas whose free variables are all local; in particular, V is not needed to interpret i-formulas. For the rest of this paper, if the valuation is not needed, we do not mention it, and simply write $I(\varphi)$. Given a formula φ and term t, we can easily define Nuprl formulas *i-formula*(φ,I) and *i-term*(t,I) that are constructively provable if φ is an i-formula in I (resp., t is an i-term in I).

We define a predicate $Consistent_I$ on programs and event structures such that, intuitively, $Consistent_I$ (Pg, es) holds if the event structure es is consistent with program Pg, given interpretation I. We start with basic programs. The basic program $@i$ $\mathbf{initially}$ ψ is an *initialization* program, which is intended to hold in an event structure es if ψ is an i-formula and i's initial local state satisfies ψ. Thus,

$$Consistent_I(@i \ \mathbf{initially}\ \psi, es) =_{\text{def}} \textit{i-formula}(\psi, I) \wedge I(\psi)(initstate_i).$$

(This notation implicitly assumes that $initstate_i$ is as specified by es, according to Definition 2.1. For simplicity, we have opted for this notation instead of $es.initstate_i$.)

We call a basic program of the form $@i$ $\mathbf{if}$ $kind = k$ $\mathbf{then}$ $x := t$ an *effect* program. It says that, if t is an i-term, then the effect of an event e of kind k is to set x to t. We define

$$\begin{array}{l} Consistent_I(@i \ \mathbf{if}\ kind = k \ \mathbf{then}\ x := t, es) =_{\text{def}} \\ \quad \textit{i-term}(t, I) \wedge \forall e@i \in es.\ (kind(e) = k \Rightarrow (state\ after\ e)(x) = I(t)(state\ before\ e)), \end{array}$$

where we write $\forall e@i \in es.\ \varphi$ as an abbreviation for $\forall e \in es.agent(e) = i \Rightarrow \varphi$. As above, the notation above implicitly assumes that *before* and *after* are as specified by es. Again, this expression is an abbreviation for a formula expressible in Nuprl whose intended meaning should be clear; $Consistent_I(@i \ \mathbf{if}\ kind = k \ \mathbf{then}\ x := t, es)$ holds if there is a constructive proof of the formula.

We can use a program of this type to describe a message sent on a link l. For example,

$$@i\ \mathit{kind} = \mathit{local}(a)\ \mathbf{then\ msg}(l) := \mathbf{f}(\mathbf{val}_i)$$

says that for all events e, $f(v)$ is sent on link l if the kind of e is a, the local state of agent i before e is s_i, and $v = s_i(\mathbf{val}_i)$.

The third type of program, $@i\ \mathit{kind} = \mathit{local}(a)$ **only if** φ, is called a *precondition* program. It says that an event of kind a can occur only if the precondition φ (which must be an i-formula) is satisfied:

$$\begin{array}{l} \mathit{Consistent}_I(@i\ \mathit{kind} = \mathit{local}(a)\ \mathbf{only\ if}\ \varphi, \mathit{es}) =_{\text{def}} \\ \quad \mathit{i\text{-}formula}(\varphi, I)\ \wedge \forall e@i \in \mathit{es}.\ (\mathit{kind}(e) = \mathit{local}(a) \Rightarrow I(\varphi)(\mathit{state\ before\ e})). \end{array}$$

Note that we allow conditions of the form $\mathit{kind}(e) = \mathit{local}(a)$ here, not the more general condition of the form $\mathit{kind}(e) = k$ allowed in effect programs. We do not allow conditions of the form $\mathit{kind}(e) = \mathit{rcv}(l)$ because we assume that receive events are not under the control of the agent.

Standard formalizations of input-output automata (see [Lynch and Tuttle 1989]) typically assume that executions satisfy some fairness constraints. We assume here only a weak fairness constraint that is captured by the basic program $@i$ **if necessarily** φ **then i.o.** $\mathit{kind} = \mathit{local}(a)$, which we call a *fairness program*. Intuitively, it says that if φ holds from some point on, then an event with kind $\mathit{local}(a)$ will eventually occur. For an event sequence with only finitely many states associated with i, we take φ to hold "from some point on" if φ holds at the last state. In particular, this means that the program cannot be consistent with an event sequence for which there are only finitely many events associated with i if φ holds of the last state associated with i. Define

$$\begin{array}{l} \mathit{Consistent}_I(@i\ \mathbf{if\ necessarily}\ \varphi\ \mathbf{then\ i.o.}\ \mathit{kind} = \mathit{local}(a), \mathit{es}) =_{\text{def}} \\ \quad \mathit{i\text{-}formula}(\varphi, I) \wedge \\ \quad [((\exists e@i \in \mathit{es}) \wedge \forall e@i \in \mathit{es}.\ \exists e' \succeq_i e.\ I(\neg\varphi)(\mathit{state\ after\ e'}) \vee (\mathit{kind}(e') = \mathit{local}(a))) \\ \quad \vee (\neg(\exists e@i \in \mathit{es}) \wedge I(\neg\varphi)(\mathit{initstate}_i))]. \end{array}$$

The last type of basic program, $@$**i only L affects** x, is called a *frame program*. It ensures that only events of kinds listed in L can cause changes in the value of variable x. The precise semantics depends on whether x has the form $\mathit{msg}(l)$. If x does not have the form $\mathit{msg}(l)$, then

$$\begin{array}{l} \mathit{Consistent}_I(@i\ \mathbf{only\ L\ affects}\ x, \mathit{es}) =_{\text{def}} \\ \quad \forall e@i \in \mathit{es}.\ ((x\ \mathit{after}\ e) \neq (x\ \mathit{before}\ e) \Rightarrow (\mathit{kind}(e) \in L)). \end{array}$$

If x has the form $\mathit{msg}(l)$, then we must have $\mathit{source}(l) = i$. Recall that sending a message m on l is formalized by setting the value of $\mathit{msg}(l)$ to m. We assume that messages are never null (i.e., $m \neq \bot$). No messages are sent during event e if $\mathit{msg}(l)\,\mathit{after}\ e = \bot$. If x has the form $\mathit{msg}(l)$, then

$$\begin{array}{l} \mathit{Consistent}_I(@i\ \mathbf{only\ L\ affects\ msg}(l), \mathit{es}) =_{\text{def}} \\ \quad \forall e@i \in \mathit{es}.\ ((\mathit{msg}(l)\ \mathit{after}\ e \neq \bot) \Rightarrow (\mathit{kind}(e) \in L)). \end{array}$$

Finally, an event structure es is said to be consistent with a distributed program Pg that is not basic if es is consistent with each of the basic programs that form Pg:

$$\mathit{Consistent}_I(Pg_1 \oplus Pg_2, \mathit{es}) =_{\text{def}} \mathit{Consistent}_I(Pg_1, \mathit{es}) \wedge \mathit{Consistent}_I(Pg_2, \mathit{es}).$$

Definition 2.4: Given an interpretation I, the semantics of a program Pg is the set of event structures consistent with Pg under interpretation I. We denote by S_I this semantics of programs: $S_I(Pg) = \{es \mid Consistent_I(Pg, es)\}$. We write $Pg \approx_I X$ if Pg satisfies X with respect to interpretation I; that is, if $X(es)$ is true for all $es \in S_I(Pg)$). ▮

Note that $S_I(Pg_1 \oplus Pg_2) = S_I(Pg_1) \cap S_I(Pg_2)$. Since the $Consistent_I$ predicate is definable in Nuprl, we can formally reason in Nuprl about the semantics of programs.

A specification is a predicate on event structures. Since our main goal is to derive from a proof that a specification X is satisfiable a program that satisfies X, we want to rule out the trivial case where the derived program Pg has no executions, so that it vacuously satisfies the specification X.

Definition 2.5: Program Pg is *consistent (with respect to interpretation I)* if $S_I(Pg) \neq \emptyset$. The specification X is *realizable (with respect to interpretation I)* if it is not vacuously satisfied, that is, if $\exists Pg.(Pg \approx_I X \wedge S_I(Pg) \neq \emptyset)$. *$Pg$ realizes* X (with respect to I) if $Pg \approx_I X$ and Pg is consistent (with respect to I). ▮

Thus, a specification is realizable if there exists a consistent program that satisfies it, and, given an interpretation I, a program is realizable if there exists an event structure consistent with it (with respect to I). Since we reason constructively, this means that a program is realizable if we can *construct* an event structure consistent with the program. This requires not only constructing sequences of events, one for each agent, but all the other components of the event structure as specified in Definition 2.3, such as AG and Act.

All basic programs other than initialization and fairness programs are vacuously satisfied (with respect to every interpretation I) by the empty event structure es consisting of no events. The empty event structure is consistent with these basic programs because their semantics in defined in terms of a universal quantification over events associated with an agent. It is not hard to see that an initialization program $@i$ $\textbf{initially}$ ψ is consistent with respect to interpretation I if and only if ψ is satisfiable in I; i.e., there is some global state s such that $I(\psi)(s_i)$ holds. For if es is an event structure with $initstate_i = s_i$, then clearly es realizes $@i$ $\textbf{initially}$ ψ.

Fair programs are realizable with respect to interpretations I where the precondition φ satisfies the principle of excluded middle (that is, $\varphi_I \Rightarrow Determinate(\varphi^+)$ is provable in Nuprl), although they are not necessarily realized by a finite event structure. To see this, note that if φ satisfies the principle of excluded middle in I, then either there is an I-local state s_i^* for agent i such that $I(\neg\varphi)(s_i^*)$ holds, or $I(\varphi)(s_i)$ holds for all I-local states s_i for i. In the former case, consider an empty event structure es with domain Val_I and $initstate_i = s_i^*$; it is easy to see that es is consistent with $@i$ $\textbf{if necessarily}$ φ $\textbf{then i.o.}$ $kind = local(a)$. Otherwise, let $Act = \{a\}$. Let es be an event structure where Act is the set of local actions, Val_I is the set of values, the sequence of events associated with agent i in es is infinite, and all events associated with agent i have kind $local(a)$. Again, it is easy to see that es is consistent with $@i$ $\textbf{if necessarily}$ φ $\textbf{then i.o.}$ $kind = local(a)$.

If φ does not satisfy the principle of excluded middle in I, then $@i$ $\textbf{if necessarily}$ φ $\textbf{then i.o.}$ $kind = local(a)$ may not be realizable with respect to I. For example, this would be the case if for example, neither $I(\varphi)(s_i)$ nor $I(\neg\varphi)(s_i)$ holds for any local state s_i.

Note that two initialization programs may each be consistent although their composition is not. For example, if both ψ and $\neg\psi$ are satisfiable i-formulas, then each of $@i$ $\textbf{initially}$ ψ and $@i$ $\textbf{initially}$ $\neg\psi$

is consistent, although their composition is not. Nevertheless, all programs synthesized in this paper can be easily proven consistent.

2.3.3 Axioms Bickford and Constable [2003] derived from the formal semantics of distributed message automata some Nuprl axioms that turn out to be useful for proving the satisfiability of a specification. We now present (a slight modification of) their axioms. The axioms have the form $Pg \models\!\approx_I X$, where Pg is a program and X is a specification, that is, a predicate on event structures; the axiom is sound if all event structures es consistent with program Pg under interpretation I satisfy the specification X. We write $\models\!\approx_I$ to make clear that the program semantics in given with respect to an interpretation I. There is an axiom for each type of basic program other than frame programs, two axioms for frame programs (corresponding to the two cases in the semantic definition of frame programs), together with an axiom characterizing composition and a refinement axiom.

Ax-init:

$$@i \ \mathbf{initially} \ \psi \models\!\approx_I \lambda es.\ \textit{i-formula}(\psi, I) \wedge I(\psi)(\textit{initstate}_i).$$

(Note that the right-hand side of $\models\!\approx$ is a specification; given an event structure es, it is true if $\textit{i-formula}(\psi, I) \wedge I(\psi)(\textit{initstate}_i)$ holds in event structure es.)

Ax-cause:

$$@i \ \mathbf{if} \ \textit{kind} = k \ \mathbf{then} \ x{:=}t \models\!\approx_I \quad \lambda es.\ \textit{i-term}(t, I) \wedge \forall e@i \in es.\ (\textit{kind}(e) = k \Rightarrow (\textit{state after } e)(x) = I(t)(\textit{state before } e)).$$

Ax-if:

$$@i \ \textit{kind} = \textit{local}(a) \ \mathbf{only\ if} \ \varphi \models\!\approx_I \quad \lambda es.\ \textit{i-formula}(\varphi, I) \wedge \forall e@i \in es.\ (\textit{kind}(e) = \textit{local}(a) \Rightarrow I(\varphi)(\textit{state before } e)).$$

Ax-fair:

$$\begin{aligned} &@i \ \mathbf{if\ necessarily} \ \varphi \ \mathbf{then\ i.o.}\ \textit{kind} = \textit{local}(a) \models\!\approx_I \\ &\quad \lambda es.\quad \textit{i-formula}(\varphi, I) \wedge \\ &\qquad\quad [((\exists e@i \in es) \wedge \forall e@i \in es.\ \exists e' \succeq_i e.\ I(\neg\varphi)(\textit{state after } e') \vee (\textit{kind}(e') = \textit{local}(a))) \\ &\qquad\quad \vee(\neg(\exists e@i \in es) \wedge I(\neg\varphi)(\textit{initstate}_i))]. \end{aligned}$$

Ax-affects:

$$@i \ \mathbf{only\ L\ affects} \ x \models\!\approx_I \lambda es.\ \forall e@i \in es.\ (x \textit{ after } e \neq x \textit{ before } e) \Rightarrow (\textit{kind}(e) \in L).$$

Ax-sends:

$$@i \ \mathbf{only\ L\ affects\ msg}(l) \models\!\approx_I \lambda es.\ \forall e@i \in es.\ (\textit{msg}(l) \textit{ after } e \neq \bot) \Rightarrow (\textit{kind}(e) \in L).$$

Ax-$\oplus$: $(Pg_1 \models\!\approx_I P) \wedge (Pg_2 \models\!\approx_I Q) \Rightarrow (Pg_1 \oplus Pg_2 \models\!\approx_I P \wedge Q)$.

Ax-ref: $(Pg \models\!\approx_I P) \wedge (P \Rightarrow Q) \Rightarrow (Pg \models\!\approx_I Q)$.

Lemma 2.6: *Axioms* **Ax-init**, **Ax-cause**, **Ax-if**, **Ax-fair**, **Ax-affects**, **Ax-sends**, **Ax-$\oplus$**, *and* **Ax-ref** *hold for all interpretations I.*

Proof: This is immediate from Definitions 2.1 and 2.4, and the definition of $\textit{Consistent}_I$. ∎

2.3.4 A general scheme for program synthesis Recall that, given a specification φ and an interpretation I, the goal is to prove that φ is satisfiable with respect to I, that is, to show that $\exists Pg.\ (Pg \mathrel{|\!\!\approx_I} \varphi)$ holds. We now provide a general scheme for doing this. Consider the following scheme, which we call $\mathcal{GS}$:

1. Find specifications $\varphi_1, \varphi_2, \ldots, \varphi_n$ such that $\forall es.\ (\varphi_1(es) \wedge \varphi_2(es) \wedge \ldots \wedge \varphi_n(es) \Rightarrow \varphi(es))$ is true under interpretation I.
2. Find programs $Pg_1, Pg_2, \ldots, Pg_n$ such that $Pg_i \mathrel{|\!\!\approx_I} \varphi_i$ holds for all $i \in \{1, \ldots n\}$.
3. Conclude that $Pg \mathrel{|\!\!\approx_I} \varphi$, where $Pg = Pg_1 \oplus Pg_2 \oplus \ldots \oplus Pg_n$.

Step 1 of $\mathcal{GS}$ is proved using the rules and axioms encoded in the Nuprl system; Step 2 is proved using the axioms given in Section 2.3.3. It is easy to see that $\mathcal{GS}$ is *sound* in the sense that, if we can show using $\mathcal{GS}$ that Pg satisfies φ, then Pg does indeed satisfy φ. We formalize this in the following proposition.

Proposition 2.7: *Scheme $\mathcal{GS}$ is sound.*

2.4 Example

As an example of a specification that we use later, consider the run-based specification $\mathit{Fair}_I(\varphi, t, l)$, where $i \neq j$, l is a link with $\mathit{source}(l) = i$ and $\mathit{dest}(l) = j$, φ is an i-formula, and t is an i-term. $\mathit{Fair}_I(\varphi, t, l)$ is a conjunction of a safety condition and a liveness condition. The safety condition asserts that if a message is received on link l, then it is the term t interpreted with respect to the local state of the sender, and that φ, evaluated with respect to the local state of the sender, holds. The liveness condition says that, if (there is a constructive proof that) condition φ is enabled from some point on in an infinite event sequence, then eventually a message sent on l is delivered. (Thus, the specification imposes a weak fairness requirement.) We define $Fair_I(\varphi, t, l)$ as follows:

$$
\begin{array}{l}
Fair_I(\varphi, t, l) =_{\mathrm{def}} \lambda es.\ \textit{i-formula}(\varphi, I) \wedge \textit{i-term}(t, I) \\
\quad (\forall e' \in es.\ (\mathit{kind}(e') = \mathit{rcv}(l) \Rightarrow \\
\quad\quad I(\varphi)(\mathit{state\ before\ send}(e')) \wedge \mathit{val}(e') = I(t)(\mathit{state\ before\ send}(e'))) \wedge \\
((\exists e@i \in es \wedge \forall e@i \in es.\ \exists e' \succeq_i e.\ I(\neg\varphi)(\mathit{state\ after\ } e'))\ \vee\ (\neg(\exists e@i \in es) \wedge I(\neg\varphi)(\mathit{initstate}_i)) \\
\quad \vee (\exists e@i \in es \wedge \forall e@i \in es.\ \exists e' \succeq_i e.\ \mathit{kind}(e') = \mathit{rcv}(l) \wedge \mathit{send}(e') \succeq_i e)).
\end{array}
$$

We are interested in this fairness specification only in settings where communication satisfies a (strong) fairness requirement: if infinitely often an agent sends a message on a link l, then infinitely often some message is delivered on l. We formalize this assumption using the following specification:

$$
\begin{array}{rl}
\mathit{FairSend}(l) =_{\mathrm{def}} \lambda es. & (\forall e@i \in es.\ \exists e' \succ_i e.\ \mathit{msg}(l)\ \mathit{after}\ e' \neq \bot) \\
& \Rightarrow (\forall e@i \in es.\ \exists e'.\quad \mathit{kind}(e') = \mathit{rcv}(l) \wedge \mathit{send}(e') \succ_i e).
\end{array}
$$

We explain below why we need communication to satisfy strong fairness rather than weak fairness (which would require only that if a message is sent infinitely often, then a message is eventually delivered).

For an arbitrary action a, let $\mathit{Fair}\text{-}\mathit{Pg}(\varphi, t, l, a)$ be the following program for agent i:

$$\begin{array}{l} @i\ \mathit{kind} = \mathit{local}(a)\ \textbf{only if}\ \varphi \oplus \\ @i\ \textbf{if}\ \mathit{kind} = \mathit{local}(a)\ \textbf{then}\ \textbf{msg}(l){:=}t \oplus \\ @i\ \textbf{only events in}\ [a]\ \textbf{affect}\ \textbf{msg}(l) \oplus \\ @i\ \textbf{if necessarily}\ \varphi\ \textbf{then i.o.}\ \mathit{kind} = \mathit{local}(a). \end{array}$$

The first basic program says that i takes action a only if φ holds. The second basic program says that the effect of agent i taking action a is for t to be sent on link l; in other words, a is i's action of sending t to agent j. The third program ensures that only action a has the effect of sending a message to agent j. With this program, if agent j (the receiver) receives a message from agent i (the sender), then it must be the case that the value of the message is t and that φ was true with respect to i's local state when it sent the message to j. The last basic program ensures that if φ holds from some point on in an infinite event sequence, then eventually an event of kind a holds; thus, i must send the message t infinitely often. The fairness requirement on communication ensures that if an event of kind a where i sends t occurs infinitely often, then t is received infinitely often.

Lemma 2.8: *For all actions a, $\mathit{Fair}\text{-}\mathit{Pg}(\varphi,t,l,a)$ satisfies $\lambda es.\mathit{FairSend}(l)(es) \Rightarrow \mathit{Fair}_I(\varphi,t,l)(es)$ with respect to all interpretations I such that φ is an i-formula and t is an i-term in I.*

Proof: We present the key points of the proof here, omitting some details for ease of exposition. We follow the scheme $\mathcal{GS}$. We assume that $\mathit{i\text{-}formula}(\varphi, I)$ and $\mathit{i\text{-}term}(t, I)$ both hold.

Step 1. For each event structure es, $\mathit{Fair}_I(\varphi,t,l)(es)$ is equivalent to a conjunction of three formulas:

$$\begin{array}{ll} \varphi_1(es): & \forall e' \in es.\ (\mathit{kind}(e') = \mathit{rcv}(l)) \Rightarrow I(\varphi)(\mathit{state\ before\ send}(e')) \\ \varphi_2(es): & \forall e' \in es.\ (\mathit{kind}(e') = \mathit{rcv}(l)) \Rightarrow \mathit{val}(e') = I(t)(\mathit{state\ before\ send}(e')) \\ \varphi_3(es): & (\exists e@i \in es \wedge \forall e@i \in es.\ \exists e' \succeq_i e.\ I(\neg\varphi)(\mathit{state\ after}\ e')) \vee \\ & (\neg(\exists e@i \in es) \wedge I(\neg\varphi)(\mathit{initstate}_i) \vee \\ & (\exists e@i \in es \wedge \forall e@i \in es.\ \exists e'.\mathit{kind}(e') = \mathit{rcv}(l) \wedge \mathit{send}(e') \succ_i e). \end{array}$$

We want to find formulas $\psi_1(es), \ldots, \psi_4(es)$ that follow from the four basic programs that make up $\mathit{Fair}\text{-}\mathit{Pg}(\varphi,t,l,a)$ and together imply $\varphi_1(es) \wedge \varphi_2(es) \wedge \varphi_3(es)$. It will simplify matters to reason directly about the events where a message is sent on link l. We thus assume that, for all events e, agent i sends a message on link l during event e iff $\mathit{kind}(e) = \mathit{local}(a)$. This assumption is expressed by:

$$\psi_1(es) =_{\mathrm{def}} \forall e@i \in es.\ (\mathit{msg}(l)\ \mathit{after}\ e \neq \bot) \Rightarrow (\mathit{kind}(e) = \mathit{local}(a)).$$

It is easy to check that $(\psi_1(es) \wedge \psi_2(es)) \Rightarrow \varphi_1(es))$ is true, where $\psi_2(es)$ is

$$\forall e@i \in es.\ (\mathit{kind}(e) = \mathit{local}(a)) \Rightarrow I(\varphi)(\mathit{state\ before}\ e).$$

Similarly, using the axiom of event structures given in Section 2.2 that says that the value of a receive event e on l is the value of $\mathit{msg}(l)$ after $\mathit{send}(e)$, it is easy to check that $(\psi_1(es) \wedge \psi_3(es)) \Rightarrow \varphi_2(es))$ is true, where $\psi_3(es)$ is

$$\forall e@i \in es.(\mathit{kind}(e) = \mathit{local}(a)) \Rightarrow \mathit{msg}(l)\ \mathit{after}\ e = I(t)(\mathit{state\ before}\ e).$$

We can show that $(\psi_3(es) \wedge \psi_4(es) \wedge \mathit{FairSend}(l)) \Rightarrow \varphi_3(es)$ is true, where ψ_4 is

$$\begin{array}{l}(\exists e@i \wedge \forall e@i \in es.\ \exists e' \succeq_i e.\ I(\neg\varphi)(\mathit{state\ after\ e'})) \vee \\ (\neg(\exists e@i \in es) \wedge I(\neg\varphi)(\mathit{initstate}_i)) \vee \\ (\exists e@i \in es \wedge \forall e@i \in es.\ \exists e' \succeq_i e.\ \mathit{kind}(e') = \mathit{local}(a)).\end{array}$$

It follows that

$$(\forall es.(\psi_1(es) \wedge \psi_2(es) \wedge \psi_3(es) \wedge \psi_4(es)) \Rightarrow (\mathit{FairSend}(l)(es) \Rightarrow \mathit{Fair}_I(\varphi, t, l)(es))).$$

Step 2. By $\mathbf{Ax\text{-}sends}$

$$@i\ \mathbf{only}\ [a]\ \mathbf{affects}\ \mathbf{msg}(l) \approx_I \psi_1.$$

By $\mathbf{Ax\text{-}if}$,

$$@i\ \mathit{kind} = \mathit{local}(a)\ \mathbf{only\ if}\ \varphi \approx_I \psi_2.$$

By $\mathbf{Ax\text{-}cause}$,

$$@i\ \mathbf{if}\ \mathit{kind} = \mathit{local}(a)\ \mathbf{then}\ \mathbf{msg}(l){:=}t \approx_I \psi_3;$$

and by $\mathbf{Ax\text{-}fair}$

$$@i\ \mathbf{if\ necessarily}\ \varphi\ \mathbf{then\ i.o.}\ \mathit{kind} = \mathit{local}(a) \approx_I \psi_4.$$

By the soundness of $\mathcal{GS}$ (Proposition 2.7), $\mathit{Fair\text{-}Pg}(\varphi, t, l, a)$ satisfies $\lambda es.\mathit{FairSend}(l)(es) \Rightarrow Fair_I(\varphi, t, l)(es)$ with respect to I. ∎

Lemma 2.9: *For all interpretations I such that φ is an i-formula and t is an i-term in I, if φ satisfies the principle of excluded middle with respect to I, then $Fair\text{-}Pg(\varphi,t,l,a)$ is consistent with respect to I.*

Proof: This argument is almost identical to that showing that fair programs are realizable with respect to interpretations where the precondition satisfies the principle of excluded middle. Since φ satisfies the principle of excluded middle with respect to I, either there exists an I-local state s_i^* for agent i such that $I(\neg\varphi)(s_i^*)$ holds, or $I(\varphi)(s_i)$ holds for all I-local states s_i for i. In the former case, let es be an empty event structure such that $i, j \in AG$, $l \in \mathit{Links}$, $a \in \mathit{Act}$, and $\mathit{initstate}_i = s_i^*$. In the latter case, choose es with AG and Links as above, let $\mathit{Act} = \{a, b\}$, and where i and j alternate sending and receiving the message t on link l, where these events have kind a and b, respectively. ∎

Corollary 2.10: *For all interpretations I such that if φ is an i-formula and t is an i-term in I, if φ satisfies the principle of excluded middle with respect to I, then the specification $Fair_I(\varphi, t, l)$ is realizable with respect to I.*

Proof: This is immediate from Lemmas 2.8 and 2.9, and from the fact that the event structure constructed in Lemma 2.8 satisfies $\mathit{FairSend}(l)$. ∎

The notion of strong communication fairness is essential for the results above: $\mathit{Fair}_I(\varphi, t, l)$ may not be realizable if we assume that communication satisfies only a weak notion of fairness that says that if a message is sent after some point on, then it is eventually received. This is so essentially because our programming language is replacing standard "if condition then take action" programs with weaker variants that ensure that, if after some point a condition holds, then eventually some action is taken.

We now show that the composition of $\mathit{Fair}\text{-}\mathit{Pg}(\varphi, t, l, a)$ and $\mathit{Fair}\text{-}\mathit{Pg}(\varphi, t, l', a)$ for different links l and l' satisfies the corresponding fairness assumptions.

Lemma 2.11: *For all distinct actions a and a', and all distinct links l and l', $Fair\text{-}Pg(\varphi, t, l, a) \oplus Fair\text{-}Pg(\varphi', t', l', a')$ satisfies*

$$\begin{array}{c}\lambda es.(\mathit{FairSend}(l)(es) \wedge \mathit{FairSend}(l')(es)) \Rightarrow \\ (\mathit{Fair}_I(\varphi, t, l)(es) \wedge \mathit{Fair}_I(\varphi', t', l')(es))\end{array}$$

with respect to all interpretations I such that φ is an i-formula, t is an i-term, φ' is an i'-formula, and t' is an i'-term in I.

Proof: Suppose $a \neq a'$. We again use scheme $\mathcal{GS}$.

Step 1. Clearly, we can take φ_1 to be $\lambda es.\ \mathit{FairSend}(l)(es) \Rightarrow \mathit{Fair}_I(\varphi, t, l)(es)$ and φ_2 to be $\lambda es.\ \mathit{FairSend}(l')(es) \Rightarrow \mathit{Fair}_I(\varphi', t', l')(es)$.

Step 2. By Lemma 2.8, $Fair\text{-}Pg(\varphi, t, l, a) \approx_I \varphi_1$ and $Fair\text{-}Pg(\varphi', t', l', a') \approx_I \varphi_2$. ∎

Finally, we can show that $Fair\text{-}Pg(\varphi, t, l, a) \oplus Fair\text{-}Pg(\varphi', t', l', a')$ is consistent, where l is a link from i to j, l' is a link from i' to j', and $l \neq l'$ (so that we may have $i = i'$ or $j = j'$, but not both), and thus the specification $\lambda es.(\mathit{FairSend}(l)(\ es) \wedge \mathit{FairSend}(l')(es)) \Rightarrow (\mathit{Fair}_I(\varphi, t, l)(es) \wedge \mathit{Fair}_I(\varphi', t', l')(es))$ is realizable with respect to I. if both φ and φ' satisfy the principle of excluded middle with respect to I.

Lemma 2.12: *For all interpretations I such that φ is an i-formula, t is an i-term, φ' is an i'-formula, and t' is an i-term in I, if both φ and φ' satisfy the principle of excluded middle with respect to I, then, for all distinct actions a and a' and all distinct links l and l', $Fair\text{-}Pg(\varphi, t, l, a) \oplus Fair\text{-}Pg(\varphi', t', l', a')$ is consistent with respect to I.*

Proof: If $I(\neg\varphi \wedge \neg\varphi)(s)$ holds for some global state s, then let es be the empty event structure such that $\mathit{initstate}_i = s_i$ and $\mathit{initstate}_{i'} = s_{i'}$. Clearly es is consistent with $Fair\text{-}Pg(\varphi, t, l, a) \oplus Fair\text{-}Pg(\varphi', t', l', a')$. Otherwise, let es be an event structure with domain Val_I, $i, j, i', j' \in AG$, and $l, l' \in \mathit{Links}$, consisting of an infinite sequence of states such that if $I(\varphi)$ holds for infinitely many states, then i sends t on link l infinitely often; if $I(\varphi')$ holds for infinitely many states, then i' sends t' on link l' infinitely often; if t is sent on l infinitely often, then j receives it on link l infinitely often; and if t' is sent on l' infinitely often, then j' receives it on l' infinitely often. It is straightforward to construct such an event structure es. Again, it should be clear that es is consistent with $Fair\text{-}Pg(\varphi, t, l, a) \oplus Fair\text{-}Pg(\varphi', t', l', a')$. ∎

3 Adding knowledge to Nuprl

We now show how knowledge-based programs can be introduced into Nuprl.

3.1 Consistent cut semantics for knowledge

We want to extend basic programs to allow for tests that involve knowledge. For simplicity, we take $AG = \{1, 2, \ldots, n\}$. As before, we start with a finite set $\mathcal{P} \cup \mathcal{F}$ of predicates and functions, and close off under conjunction, negation, and quantification over non-local variables; but now, in addition, we also close off under application of the temporal operators $\Box$ and $\Diamond$, and the epistemic operators K_i, $i = 1, \ldots, n$, one for each process i.

We again want to define a consistency relation in Nuprl for each program. To do that, we first need to review the semantics of knowledge. Typically, semantics for knowledge is 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 has access to [Fagin, Halpern, Moses, and Vardi 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. We just apply their definition in our framework.

The truth of formulas is defined relative to a pair (Sys, c), consisting of a system Sys (i.e., a a set of event structures) and a *consistent cut* c of some event structure $es \in$ *Sys*, where a *consistent cut* c in es is a set of events in es closed under the causality relation. Recall from Section 2.2 that this amounts to c satisfying the constraint that, if e' is an event in c and e is an event in es that precedes e' (i.e., $e \prec e'$), then e is also in c. We write $c \in$ *Sys* if c is a consistent cut in some event structure in Sys.

Traditionally, a knowledge formula $K_i\varphi$ is interpreted as true at a point (r, m) if φ is true regardless of i's uncertainty about the whole system at (r, m). Since we interpret formulas relative to a pair (Sys, c), we need to make precise i's uncertainty at such a pair. For the purposes of this paper, we assume that each agent keeps track of all the events that have occurred and involved him (which corresponds to the assumption that agents have *perfect recall*); we formalize this assumption below. Even in this setting, agents can be uncertain about what events have occurred in the system, and about their relative order. Consider, for example, the scenario in the left panel of Figure 1: agent i receives a message from agent j (event e_2), then sends a message to agent k (e_3), then receives a second message from agent j (e_6), and then performs an internal action (e_7). Agent i knows that $send(e_2)$ occurred prior to e_2 and that $send(e_6)$ occurred prior to e_6. However, i considers possible that after receiving his message, agent k sent a message to j which was received by j before e_7 (see the right panel of Figure 1).

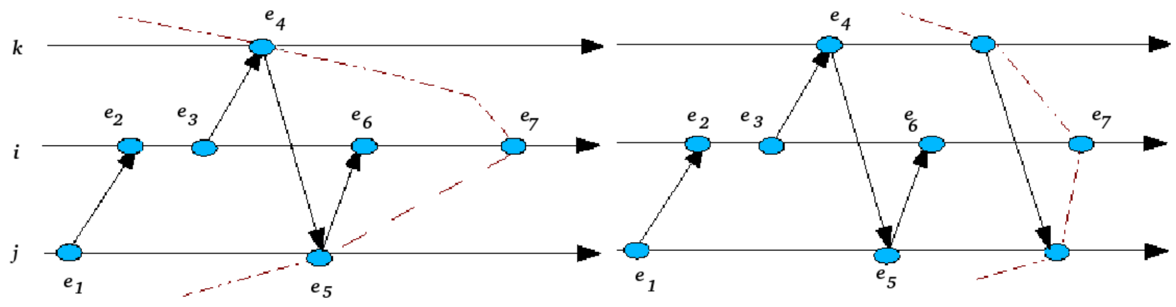

Figure 1: Two consistent cuts that cannot be distinguished by agent i.

In general, as argued by Panangaden and Taylor, agent i considers possible any consistent cut in which he has recorded the same sequence of events. To formalize this intuition, we define equivalence relations $\sim_i$, $i = 1, \ldots, n$, on consistent cuts by taking $c \sim_i c'$ if i's history is the same in c and c'. 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'. Relative to (Sys, c), agent i considers possible any consistent cut $c' \in Sys$ such that $c' \sim_i c$.

Since the semantics of knowledge given here implicitly assumes that agents have perfect recall, we restrict to event structures that also satisfy this assumption. So, for the remainder of this paper, we restrict to systems where *local states encode histories*, that is, we restrict to systems Sys such that, for all event structures $es, es' \in Sys$, if e is an event in es, e' is an event in es', $agent(e) = agent(e') = i$, and $state\ before\ e = state\ before\ e'$, then i has the same history in both es and es'. For simplicity, we guarantee this by assuming that each agent i has a local variable $history_i \in X_i$ that encodes its history. Thus, we take $initstate_i(history_i) = \perp$ and for all events e associated with agent i, we have $(s\ after\ e)(history_i) = (s\ before\ e)(history_i) \cdot e$. It immediately follows that in two global states where i has the same local state, i must have the same history. Let $System$ be the set of all such systems.

Recall that events associated with the same agent are totally ordered. This means that we can associate with every consistent cut c a global state s^c: for each agent i, s_i^c is i's local state after the last event e_i associated with i in c occurs. Since local states encode histories, it follows that if $s_i^c = s_i^{c'}$, then $c \sim_i c'$. It is not difficult to see that the converse is also true; that is, if $c \sim_i c'$, then $s_i^c = s_i^{c'}$. We also write $s^c \prec s^{c'}$ if $c \prec c'$. In the following, we assume that all global states in a system Sys have the form s^c for some consistent cut c.

Nuprl is rich enough that epistemic and modal operators can be defined within Nuprl. Thus, to interpret formulas with epistemic operators and temporal operators, we just translate them to formulas that do not mention them. Since the truth of an epistemic formula depends not just on a global state, but on a pair (Sys, c), where the consistent cut c can be identified with a global state in some event structure in Sys, the translated formulas will need to include variables that, intuitively, range over systems and global states. To make this precise, we expand the language so that it includes rigid binary predicates $\mathbf{CC}$ and $\succeq$, a rigid binary function $\mathbf{ls}$, and rigid constants s and $\mathbf{Sys}$. Intuitively, s represents a global state, $\mathbf{Sys}$ represents a system, $\mathbf{CC}(x, y)$ holds if y is a consistent cut (i.e., global state) in system x, $\mathbf{ls}(x, i)$ is i's local state in global state x, and $\succeq$ represents the ordering on consistent cuts defined above.

For every formula that does not mention modal operators, we take $\varphi^t = \varphi$. We define

$$(K_i\varphi)^t =_{\text{def}} \forall \mathrm{s}'((\mathbf{CC}(\mathbf{Sys}, \mathrm{s}') \land \mathbf{ls}(\mathrm{s}', i) = \mathbf{ls}(\mathrm{s}, i)) \Rightarrow \varphi^t[\mathrm{s}/\mathrm{s}'])),$$

$$(\Box\varphi)^t =_{\text{def}} \forall \mathrm{s}'((\mathbf{CC}(\mathbf{Sys}, \mathrm{s}') \land \mathrm{s}' \succeq \mathrm{s} \Rightarrow \varphi^t[\mathrm{s}/\mathrm{s}']),$$

and

$$(\Diamond\varphi)^t =_{\text{def}} \exists \mathrm{s}'((\mathbf{CC}(\mathbf{Sys}, \mathrm{s}') \land \mathrm{s}' \succeq \mathrm{s} \land \varphi^t[\mathrm{s}/\mathrm{s}']).$$

Given an interpretation I, let I' be the interpretation that extends I by adding to φ_I formulas characterizing $\mathbf{Sys}$, s, $\mathbf{CC}$, $\mathbf{ls}$, and $\succeq$ appropriately. That is, the formulas force $\mathbf{Sys}$ to represent a set of event structures, s to be a consistent cut in one of these event structures, and so on. These formulas are all expressible in Nuprl. More specifically, we restrict here to constructive systems, that is, systems that can be defined in Nuprl. A constructive system Sys can be characterized by a formula φ_{Sys} in Nuprl. φ_{Sys} has a free variable $\mathbf{Sys}$ ranging over systems such that φ_{Sys} holds under interpretation I' and valuation V iff $V(\mathbf{Sys}) = Sys$. We now define a predicate $I'_V(\varphi)$ on systems and global states by simply taking

$I'_V(\varphi)(Sys, s)$ to hold iff $\varphi_{I'}$ together with the conjunction of atomic formulas of the form $x = V(x)$ for all non-local variables x that appear in φ, $x = s_i(x)$ for variables $x \in X_i$, $i \in AG$, that appear in φ, $\mathbf{s} = s$, and φ_{Sys}, imply $(\varphi^t)^+$ (where, in going from φ^t to $(\varphi^t)^+$, we continue to use the $\mathbf{s}$). Thus, we basically reduce a modal formula to a non-modal formula, and evaluate it in system Sys using I_V.

Just as in the case of non-epistemic formulas, the valuation V is not needed to interpret formulas whose only free variables are in $\cup_{i \in AG} X_i$. For such formulas, we typically write $I'(\varphi)(Sys, s)$ instead of $I'_V(\varphi)(Sys, s)$. We can also define i-formulas and i-terms, but now whether a formula is an i-formula or a term is an i-term depends, not only on the interpretation, but on the system. A formula φ is an *i-formula in interpretation I' and system Sys* if, for all states s, s' in Sys, $I'_V(\varphi)(Sys, s) = I'_V(\varphi)(Sys, s')$ if $s_i = s'_i$; similarly, t is an *i-term in interpretation I' and system Sys* if $I'_V(t)(Sys, s) = I'_V(t)(Sys, s')$ if $s_i = s'_i$. We write this as *i-formula*(φ, I, Sys) and *i-term*(t, I, Sys), respectively. If φ is an i-formula and t is an i-term in I and Sys for all systems Sys, then we simply write *i-formula*(φ, I) and *i-formula*(t, I). For an i-formula, we often write $I'_V(\varphi)(Sys, s_i)$ rather than $I'_V(\varphi)(Sys, s)$. Note that a Boolean combination of epistemic formulas whose outermost knowledge operators are K_i is guaranteed to be an i-formula in every interpretation, as is a formula that has no nonrigid functions or predicates and does not mention K_j for $j \neq i$. The former claim is immediate from the following lemma.

Lemma 3.1: *For all formulas φ, systems Sys, and global states s and s', if $s_i = s'_i$, then $I'(K_i\varphi)(Sys, s)$ holds iff $I'(K_i\varphi)(Sys, s')$ does.*

Proof: Follows from the observation that if we have a proof in Nuprl that an i-formula holds given I', Sys, and $s \in Sys$, then we can rewrite the proof so that it mentions only s_i rather than s. Thus, we actually have a proof that the i-formula holds in all stats $s' \in Sys$ such that $s'_i = s_i$. ∎

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.

3.2.1 Syntax and semantics Define *knowledge-based message automata* just as we defined message automata in Section 2.3, except that we now allow arbitrary epistemic formulas in tests. If we want to emphasize that the tests can involve knowledge, we talk about *knowledge-based* initialization, precondition, effect, and fairness programs. For the purposes of this paper, we take knowledge-based programs to be knowledge-based message automata.

We give semantics to knowledge-based programs by first associating with each knowledge-based program a function from systems to systems. Let $(Pg^{kb})^t$ be the result of replacing every formula φ in Pg^{kb} by φ^t. Note that $(Pg^{kb})^t$ is a standard program, with no modal formulas. Given an interpretation I and a system Sys let $I(Sys)$ be the result of adding to φ_I the formula φ_{Sys}.

Now we can apply the semantics of Section 2.3.2 and get the system $S_{I(Sys)}((Pg^{kb})^t)$. In general, the system $S_{I(Sys)}((Pg^{kb})^t)$ will be different from the system Sys. A system Sys *represents* a knowledge-based program Pg^{kb} (with respect to interpretation I) if it is a fixed point of this mapping;

that is, if $S_{I(Sys)}((Pg^{kb})^t) = Sys$. Following Fagin et al. [1995, 1997], we take the semantics of a knowledge-based program Pg^{kb} to be the set of systems that represent it.

Definition 3.2: A *knowledge-based program semantics* is a function associating with a knowledge-based program Pg^{kb} and an interpretation I the systems that represent Pg^{kb} with respect to I; that is, $S_I^{kb}(Pg^{kb}) = \{Sys \in System \mid S_{I(Sys)}((Pg^{kb})^t) = Sys\}$. ∎

As observed by Fagin et al. [1995, 1997], it is possible to construct knowledge-based programs that are represented by no systems, exactly one system, or more than one system. However, there exist conditions (which are often satisfied in practice) that guarantee that a knowledge-based 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, $S_I^{kb}(Pg) = \{S_I(Pg)\}$. Thus, we can view S_I^{kb} as extending S_I.

A (standard) program Pg *implements* the knowledge-based program Pg^{kb} with respect to interpretation I if $S_I(Pg)$ represents Pg^{kb} with respect to I, that is, if $S_{I(S_I(Pg))}((Pg^{kb})^t) = S_I(Pg)$. In other words, by interpreting the tests in Pg^{kb} with respect to the system generated by Pg, we get back the program Pg.

3.2.2 Knowledge-based specifications Recall that a standard specification is a predicate on event structures. Following [Fagin, Halpern, Moses, and Vardi 1997], we take a knowledge-based specification (*kb specification* from now on) to be a predicate on systems.

Definition 3.3: A *knowledge-based specification* is a predicate on $System$. A knowledge-based program Pg^{kb} satisfies a knowledge-based specification Y^{kb} with respect to I, written $Pg^{kb} \approx\!\!|_I\ Y^{kb}$, if all the systems representing Pg^{kb} with respect to I satisfy Y^{kb}, that is, if the following formula holds: $\forall Sys \in S_I^{kb}(Pg^{kb}).\ Y^{kb}(Sys)$. The knowledge-based specification Y^{kb} is *realizable* with respect to I if there exists a (standard) program Pg such that $S_I(Pg) \neq \emptyset$ and $Pg \approx\!\!|_I\ Y^{kb}$ (i.e., $Y^{kb}(S_I(Pg))$ is true). ∎

As for standard basic programs, it is not difficult to show that knowledge-based precondition, effect, and frame programs are trivially consistent: we simply take Sys to consist of only one event structure es with no events. A knowledge-based initialization program is realizable iff $\varphi_I \wedge \psi^t$ is satisfiable. Finding sufficient conditions for fair knowledge-based programs to be realizable is nontrivial. We cannot directly translate the constructions sketched for the standard case to the knowledge-based case because, at each step in the construction (when an event structure has been only partially constructed), we would have to argue that a certain knowledge-based fact holds when interpreted with respect to an entire system and an entire event structure. However, in the next section, the knowledge-based programs used in the argument for STP (which do include fairness requirements) are shown to be realizable.

3.2.3 Axioms We now consider the extent to which we can generalize the axioms characterizing (standard) programs presented in Section 2.3 to knowledge-based programs.

Basic knowledge-based message automata other than knowledge-based precondition and fairness requirement programs satisfy analogous axioms to their standard counterparts. The only difference is that now we view the specifications as functions on systems, not on event structures. For example, the axiom corresponding to **Ax-init** is

$$\textbf{Ax-initK}: \quad @i \textbf{ initially } \psi \models\!\!\approx_I \lambda Sys.\ \textit{i-formula}(\psi, I, Sys) \wedge \forall es \in Sys.\ I(\psi)(Sys, initstate_i).$$

(Note that here, just as in the definition of **Ax-init**, for simplicity, we write $initstate_i$ instead of $es.initstate_i$. Since ψ is constrained to be an i-formula in makes sense to talk about $I(\psi)(Sys, initstate_i)$ instead of $I(\psi)(Sys, s)$ for a global state s with $s_i = initstate_i$.) The knowledge-based analogues of axioms **Ax-cause**, **Ax-affects**, and **Ax-sends** are denoted **Ax-causeK**, **Ax-affectsK**, and **Ax-sendsK**, respectively, and are identical to the standard versions of these axioms. The knowledge-based counterparts of **Ax-if** and **Ax-fair** now involve epistemic preconditions, which are interpreted with respect to a system:

$$\begin{array}{ll}\textbf{Ax-ifK}: & @i\ kind = local(a) \textbf{ only if } \varphi \models\!\!\approx_I \ \lambda Sys.\ \textit{i-formula}(\varphi, I, Sys) \wedge \\ & \forall es \in Sys.\ \forall e@i \in es.\ (kind(e) = local(a)) \Rightarrow I(\varphi)(Sys, state\ before\ e)\end{array}$$

$$\begin{array}{ll}\textbf{Ax-fairK}: & @i \textbf{ if necessarily } \varphi \textbf{ then i.o. } kind = local(a) \models\!\!\approx_I \ \lambda Sys.\ \textit{i-formula}(\varphi, I, Sys) \wedge \\ & \forall es \in Sys. \quad ((\exists e@i \in es \wedge \forall e@i \in es.\ \exists e' \succeq_i e. \\ & \qquad I(\neg\varphi)(Sys, state\ after\ e') \vee kind(e') = local(a)) \vee \\ & \qquad (\neg(\exists e@i \in es) \wedge I(\neg\varphi)(Sys, initstate_i(es)))).\end{array}$$

There are also obvious analogues axioms **Ax-ref** and **Ax-**$\oplus$, which we denote **Ax-refK** and **Ax-** $\oplus$ **K** respectively.

Lemma 3.4: *Axioms* **Ax-initK**, **Ax-causeK**, **Ax-affectsK**, **Ax-sendsK**, **Ax-ifK**, **Ax-fairK**, *and* **Ax-refK** *hold for all interpretations I.*

Proof: Since the proofs for all axioms are similar in spirit, we prove only that **Ax-ifK** holds for all interpretations I'. Fix an interpretation I. Let Pg^{kb} be the program $@i\ kind = local(a)$ **only if** φ, where φ is an i-formula. Let Y^{kb} be an instance of **Ax-ifK**:

$$\lambda Sys.\ \textit{i-formula}(\varphi, I, Sys) \wedge \forall es \in Sys.\ \forall e@i \in es.\ (kind(e) = local(a)) \Rightarrow I(\varphi)(Sys, state\ before\ e).$$

By Definition 3.3, $Pg^{kb} \models\!\!\approx_I \ Y^{kb}$ is true if and only if, for all systems $Sys \in S^{kb}_I(Pg^{kb})$, $Y^{kb}(Sys)$ holds. That is, for all systems Sys such that $S_{I(Sys)}((Pg^{kb})^t) = Sys$, the following holds:

$$\forall es \in Sys.\ \textit{i-formula}(\varphi, I, Sys) \wedge \forall e@i \in es.\ (kind(e) = local(a)) \Rightarrow I(\varphi)(Sys, state\ before\ e).$$

Let Sys be a system such that $S_{I(Sys)}((Pg^{kb})^t) = Sys$. By Definition 2.4, all event structures in Sys are consistent with the program $(Pg^{kb})^t$ with respect to interpretation $I(Sys)$. Recall that $(Pg^{kb})^t$ is the (standard) program $@i\ kind = local(a)$ **only if** $\varphi^{\mathbf{t}}$, where $I(Sys)(\varphi^t)(s) = I(\varphi)(Sys, s)$. We can thus apply axiom **Ax-if** and conclude that the following holds for all event structures es consistent with $I(Sys)((Pg^{kb})^t)$ with respect to $I(Sys)$ (i.e., for all $es \in Sys$):

$$\textit{i-formula}(\varphi^t, I(Sys)) \wedge \forall e@i \in es.\ (kind(e) = local(a)) \Rightarrow I(Sys)(\varphi^t)(state\ before\ e).$$

The first conjunct says that, for all global states s and s' in Sys, if $s_i = s'_i$ then $I(Sys)(\varphi^t)(s) = I(Sys)(\varphi^t)(s')$, which is equivalent to saying that $I(\varphi)(Sys, s) = I(\varphi)(Sys, s')$, that is, $\textit{i-formula}(\varphi, I, Sys)$ holds. The second conjunct is equivalent to

$$\forall e@ies.\ (kind(e) = local(a)) \Rightarrow I(\varphi)(Sys, state\ before\ e),$$

by the definition of φ^t and $I(Sys)$. Thus, $Y^{kb}(Sys)$ holds under interpretation I. ∎

The proof of Lemma 3.4 involves only unwinding the definition of satisfiability for knowledge-based specifications and the application of simple refinement rules, already implemented in Nuprl. In general, proofs of epistemic formulas will also involve reasoning in the logic of knowledge. Sound and complete axiomatizations of (nonintuitionistic) first-order logic of knowledge are well-known (see [Fagin, Halpern, Moses, and Vardi 1995] for an overview) and can be formalized in Nuprl in a straightforward way. This is encouraging, since it supports the hope that Nuprl's inference mechanism is powerful enough to deal with knowledge specifications, without further essential additions.

Note that **Ax-**$\oplus$**K** is not included in Lemma 3.4. That is because it does not always hold, as the following example shows.

Example 3.5: Let $Y_i^{kb} =_{\rm def} \Box(\neg K_{2-i}(x_i = i))$ for $i = 1, 2$, where $x_i \in X_i$, and let $I = \emptyset$. Let Pg_i, $i = 1, 2$ be the standard program for agent i such that $S_I(Pg_i)$ consists of all the event structures such that $x_i = i$ at all times; that is, Pg_i is the program

$$@i \textbf{ initially } x_i = i \ \oplus \ @i \textbf{ only } \emptyset \textbf{ affects } x_i.$$

Since Pg_i places no constraints on x_{2-i}, is straightforward to prove that $Pg_i \approx\!\!\!>_I \ Y_{2-i}^{kb}$, for $i = 1, 2$. On the other hand, $S_I(Pg_1 \oplus Pg_2)$ consists of all the event structures where $x_i = i$ at all times, for $i = 1, 2$, so $Pg_1 \oplus Pg_2 \approx\!\!\!>_I \ \neg Y_1^{kb} \wedge \neg Y_2^{kb}$. ∎

3.3 Example

Recall from Section 2.4 that the specification $\mathit{FairSend}(l) \Rightarrow \mathit{Fair}_I(\varphi, t, l)$ is satisfied by the program $\mathit{Fair}\text{-}\mathit{Pg}(\varphi, t, l, a)$, for all actions a. We now consider a knowledge-based version of this specification. If φ is an i-knowledge-based formula and t is an i-term in I, define

$$\mathit{Fair}_I^{kb}(\varphi, t, l) =_{\rm def} \lambda Sys.\, \forall es \in Sys.\, \mathit{Fair}_{I(Sys)}(\varphi^t, t, l)(es),$$

that is

$$\begin{array}{l}
\mathit{Fair}_I^{kb}(\varphi, t, l) =_{\rm def} \\
\quad \lambda Sys.\mathit{i\text{-}formula}(\varphi, I, Sys) \wedge \mathit{i\text{-}term}(t, I, Sys) \wedge \\
\quad\quad \forall es \in Sys.((\forall e' \in es.\ (\mathit{kind}(e') = \mathit{rcv}(l)) \Rightarrow \\
\quad\quad\quad I(\varphi)(Sys, \mathit{state\ before\ send}(e')) \wedge \mathit{val}(e') = I(t)(Sys, \mathit{state\ before\ send}(e'))) \\
\quad\quad\quad \wedge((\exists e@i \in es \wedge \forall e@i \in es.\ \exists e' \succeq_i e.\ I(\neg\varphi)(Sys, \mathit{state\ after\ } e')) \vee \\
\quad\quad\quad\quad (\exists e@i \in es \wedge \forall e@i \in es.\ \exists e'.\ \mathit{kind}(e') = \mathit{rcv}(l) \wedge \mathit{send}(e') \succeq_i e) \vee \\
\quad\quad\quad\quad (\neg(\exists e@i \in es) \wedge I(\neg\varphi)(Sys, \mathit{initstate}_i))).
\end{array}$$

For example, $\mathit{Fair}_I^{kb}(K_i\varphi, t, l)$ says that every message received on l is given by the term t interpreted at the local state of the sender i, and that i must have known fact φ when it sent this message on l; furthermore, if from some point on i knows that φ holds, then eventually a message is received on l.

As in Section 2.4, we assume that message communication satisfies a strong fairness condition. The knowledge-based version of the condition $\mathit{FairSend}(l)$ simply associates with each system Sys the specification $\mathit{FairSend}(l)$; that is, $\mathit{FairSend}^{kb}(l)$ is just $\lambda Sys.\ \forall es \in Sys.\mathit{FairSend}(l)(es)$.

Lemma 3.6: *For all interpretations I such that φ is an i-formula and t is an i-term in I, and all actions a, we have that*

$$Fair\text{-}Pg(\varphi,t,l,a) \approx\!\!\!|_I \; FairSend^{kb}(l) \Rightarrow Fair_I^{kb}(\varphi,t,l).$$

The proof is similar in spirit to that of Lemma 3.4; by supplying a system Sys as an argument to the specification, we essentially reduce to the situation in Lemma 2.8. We leave details to the reader.

We can also prove the following analogue of Lemma 2.11.

Lemma 3.7: *For all interpretations I such that φ is an i-formula, φ' is a j-formula, t is an i-term, and t' is a j-term in I, all distinct links l and l', and all distinct actions a and a', we have that*

$$\begin{array}{l} Fair\text{-}Pg(\varphi,t,l,a) \oplus Fair\text{-}Pg(\varphi',t',l',a') \approx\!\!\!|_I \\ \quad (FairSend^{kb}(l) \wedge FairSend^{kb}(l')) \Rightarrow (Fair_I^{kb}(\varphi,t,l) \wedge Fair_I^{kb}(\varphi',t',l')). \end{array}$$

4 The sequence transmission problem (STP)

In this section, we give a more detailed example of how a program satisfying a knowledge-based specification X can be extracted from X 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., a 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}. This two-stage process has several advantages:

- A proof carried out to derive Pg^{kb} does not rely on particular assumptions about how knowledge is gained. Thus, it is potentially more intuitive and elegant than a proof based on certain implementation assumptions.
- By definition, if Pg^{kb} satisfies a specification, then so do all its implementations.
- This methodology gives us a general technique for deriving standard programs that implement the knowledge-based program, by finding weaker (non-knowledge-based) predicates that imply the knowledge preconditions in 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).

4.1 Synthesizing a knowledge-based program for STP

The STP involves a sender S that 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. (Here we assume that $X(n)$ is a bit only for simplicity; our analysis of the STP does not essentially change once we allow $X(n)$ to be an element of an arbitrary constructive domain.) A solution to the STP 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 $X(n)$ is eventually written by R on the output tape.

Halpern and Zuck [1992] give two knowledge-based programs that solve the STP, and show that a number of standard programs in the literature, like Stenning's [1976] protocol, the alternating bit protocol [1969], and Aho, Ullman and Yannakakis's algorithms [1982], are all particular instances of these programs.

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. It is easy to see that if undetectable corruption is allowed, then the STP is not solvable. Neither is it solvable if all messages can be lost. Thus, following [Halpern and Zuck 1992], we assume (a) that all corruptions are detectable and (b) a strong fairness condition: for any given link l, if infinitely often a message is sent on l, then infinitely often some message is delivered on l. We formalize strong fairness by restricting to systems where $\mathit{FairSend}(l)$ holds for all links l.

The safety and liveness conditions for STP are run-based specifications. As argued by Fagin et al. [1997], 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 $K_R(X(n))$ as an abbreviation for $K_R(X(n)=0) \vee K_R(X(n)=1)$, we really want to satisfy the knowledge-based specification

$$\varphi_{stp}^{kb} =_{\mathrm{def}} \forall n \, \Diamond K_R(X(n)).$$

This is the specification we now synthesize.

Since we are assuming fairness, S can ensure that R learns the nth bit by sending it sufficiently often. Thus, S can ensure that R learns the n^{th} bit if, infinitely often, either S sends $X(n)$ or S knows that R knows $X(n)$. (Note that once S knows that R knows $X(n)$, S will continue to know this, since local states encode histories.) We can enforce this by using an appropriate instantiation of Fair^{kb}.

Let $\mathbf{c}_S$ be a (nonrigid) constant that, intuitively, represents the smallest n such that S does not know that R knows $X(n)$, if such an n exists. That is, we want the following formula to be true:

$$\forall n.\, \forall k < n.\, K_S K_R(X(k)) \wedge \neg K_S K_R(X(n)) \Rightarrow n = \mathbf{c}_S.$$

Let φ_S be the knowledge-based formula that holds at a consistent cut c if and only if there exists a smallest n such that, at c, S does not know that R knows $X(n)$:

$$\varphi_S =_{\mathrm{def}} \exists n.\, \forall k < n.\, K_S K_R(X(k)) \wedge \neg K_S K_R(X(n)).$$

Let t_S be the term $\langle \mathbf{c}_S, X(\mathbf{c}_S)\rangle$.[4] Let l_{SR} denote the communication link from S to R. Now consider the knowledge-based specification $\mathit{Fair}_I^{kb}(\varphi_S, t_S, l_{SR})$. $\mathit{Fair}_I^{kb}(\varphi_S, t_S, l_{SR})$ holds in a system *Sys* if, (1) whenever R receives a message from S, the message is a pair of the form $\langle n, X(n)\rangle$; (2) at the time S sent this message to R, S knew that R knew the first n elements in the sequence X, but S did not know whether R knew $X(n)$; and (3) R is guaranteed to either eventually receive the message $\langle n, X(n)\rangle$ or eventually know $X(n)$.

How does the sender learn which bits the receiver knows? One possibility is for S to receive from R a request to send $X(n)$. This can be taken by S to be a signal that R knows all the preceding bits. We can ensure that S gets this information by again using an appropriate instantiation of Fair^{kb}. Define $\mathbf{c}_R$

[4] We are implicitly assuming here that the pairing function that maps x and y to $\langle x, y\rangle$ is in the language.

be a (nonrigid) constant that, intuitively, represents the smallest n such that R does not know $X(n)$, if such an n exists. In other words, we want the following formula to be true:

$$\forall n.\, \forall k < n.\, K_R(X(k)) \land \neg K_R(X(n)) \Rightarrow n = \mathbf{c}_R.$$

We take φ_R to be the knowledge-based formula

$$\varphi_R =_{\text{def}} \exists n.\, \forall k < n.\, K_R(X(k)) \land \neg K_R(X(n)),$$

which says that there exists a smallest n such that R does not know $X(n)$ (or, equivalently, such that $\mathbf{c}_R = n$ holds). Finally, let l_{RS} denote the communication link from R to S. $\mathit{Fair}^{kb}_I(\varphi_R, t_R, l_{RS})$ implies that whenever S receives a message n from R, it is the case that, at the time R sent this message, R knew the first n elements of X, but not $X(n)$. Note that, for all n, S is guaranteed to eventually receive a message n unless R eventually knows $X(n)$.

We can now use the system to verify our informal claim that we have refined the initial specification φ^{kb}_{stp}. That is, the system can prove

$$\begin{array}{l}(\mathit{Fair}^{kb}_I(\varphi_S, t_S, l_{SR}) \land \mathit{Fair}^{kb}_I(\varphi_R, \mathbf{c}_R, l_{RS}) \land \\ \quad (\forall n.\, \forall k < n.\, K_S K_R(X(k)) \land \neg K_S K_R(X(n)) \Rightarrow n = \mathbf{c}_S) \land \\ \quad (\forall n.\, \forall k < n.\, K_R(X(k)) \land \neg K_R(X(n)) \Rightarrow n = \mathbf{c}_R)) \Rightarrow \varphi^{kb}_{stp}.\end{array}$$

No new techniques are needed for this proof: we simply unwind the definitions of the semantics of knowledge formulas and of the fairness specifications, and proceed with a standard proof by induction on the smallest n such that R does not know $X(n)$.

It follows from Lemma 3.7 that $\mathit{Fair}^{kb}_I(\varphi_S, t_S, l_{SR}) \land \mathit{Fair}^{kb}_I(\varphi_R, \mathbf{c}_R, l_{RS})$ is satisfied by the combination of two simple knowledge-based programs, assuming that message communication on links l_{SR} and l_{RS} satisfies the strong fairness conditions $\mathit{FairSend}^{kb}(l_{SR})$ and $\mathit{FairSend}^{kb}(l_{RS})$. That is, for any two distinct actions a_S and a_R, the following is true:

$$\begin{array}{l}\mathit{Fair\text{-}Pg}(\varphi_S, t_S, l_{SR}, a_S) \oplus \mathit{Fair\text{-}Pg}(\varphi_R, \mathbf{c}_R, l_{RS}, a_R) \approx\!\!\mid_I \\ \quad (\mathit{FairSend}^{kb}(l_{SR}) \land \mathit{FairSend}^{kb}(l_{RS})) \Rightarrow (\mathit{Fair}^{kb}_I(\varphi_S, t_S, l_{SR}) \land \mathit{Fair}^{kb}_I(\varphi_R, \mathbf{c}_R, l_{RS})).\end{array}$$

As explained in Section 2.4, $\mathit{FairSend}^{kb}(l_{SR}) \land \mathit{FairSend}^{kb}(l_{RS})$ says that if infinitely often a message is sent on l_{SR} then infinitely often a message is received on l_{SR}, and, similarly, if infinitely often a message is sent on l_{RS} then infinitely often a message is received on l_{RS}; as mentioned at the beginning of this section, we restrict to systems where these conditions are met. Furthermore, it is not difficult to show that we can use simple initialization clauses to guarantee that the constraints on the interpretation of $\mathbf{c}_S$ and $\mathbf{c}_R$ are satisfied:

$$\begin{array}{l}@S\ \mathbf{initially}\ \Box\,(\forall n.\, \forall k < n.\, K_S K_R(X(k)) \land \neg K_S K_R(X(n)) \Rightarrow n = \mathbf{c}_S) \approx\!\!\mid_I \\ \quad \forall n.\, \forall k < n.\, K_S K_R(X(k)) \land \neg K_S K_R(X(n)) \Rightarrow n = \mathbf{c}_S,\end{array}$$

$$\begin{array}{l}@R\ \mathbf{initially}\ \Box\,(\forall n.\, \forall k < n.\, K_R(X(k)) \land \neg K_R(X(n)) \Rightarrow n = \mathbf{c}_R) \approx\!\!\mid_I \\ \quad \forall n.\, \forall k < n.\, K_R(X(k)) \land \neg K_R(X(n)) \Rightarrow n = \mathbf{c}_R.\end{array}$$

Thus, $Pg_S^{kb}(\varphi_S, t_S, l_{SR}, a_S) \oplus Pg_R^{kb}(\varphi_R, \mathbf{c}_R, l_{RS}, a_R)) \approx_I \varphi_{stp}^{kb}$, where

$$
\begin{aligned}
&Pg_S^{kb}(\varphi_S, t_S, l_{SR}, a_S) =_{\text{def}} \mathit{Fair}\text{-}\mathit{Pg}(\varphi_S, t_S, l_{SR}, a_S)\oplus \\
&\quad @S \textbf{ initially } \Box\, (\forall n.\, \forall k < n.\ K_S K_R(X(k)) \wedge \neg K_S K_R(X(n)) \Rightarrow n = \mathbf{c}_S),
\end{aligned}
$$

$$
\begin{aligned}
&Pg_R^{kb}(\varphi_R, \mathbf{c}_R, l_{RS}, a_R)) =_{\text{def}} \mathit{Fair}\text{-}\mathit{Pg}(\varphi_R, \mathbf{c}_R, l_{RS}, a_R)\oplus \\
&\quad @R \textbf{ initially } \Box\, (\forall n.\, \forall k < n.\ K_R(X(k)) \wedge \neg K_R(X(n)) \Rightarrow n = \mathbf{c}_R).
\end{aligned}
$$

From the definition of $\mathit{Fair}\text{-}\mathit{Pg}(\varphi_R, \mathbf{c}_R, l_{RS}, a_R)$ in Section 3.3, it follows that $Pg_S^{kb}(\varphi_S, t_S, l_{SR}, a_S)$ is the following composition:

$$
\begin{aligned}
&@S \textbf{ initially } \Box\, (\forall n.\, \forall k < n.\ K_S K_R(X(k)) \wedge \neg K_S K_R(X(n)) \Rightarrow n = \mathbf{c}_S)\oplus \\
&@S\ \mathit{kind} = \mathit{local}(a_S) \textbf{ only if } \exists n.\ (\forall k < n.\ K_S K_R(X(k))) \wedge \neg K_S K_R(X(n)) \oplus \\
&@S \textbf{ if } \mathit{kind} = \mathit{local}(a_S) \textbf{ then } \mathbf{msg}(l_{SR}) := t_S \oplus \\
&@S \textbf{ only events in } [a_S] \textbf{ affects } \mathbf{msg}(l_{SR})\oplus \\
&@S \textbf{ if necessarily } \exists n.\ (\forall k < n.\ K_S K_R(X(k))) \wedge \neg K_S K_R(X(n)) \textbf{ then i.o. } \mathit{kind} = \mathit{local}(a_S).
\end{aligned}
$$

Using the program notation of Fagin et al. [1995], $Pg_S^{kb}(\varphi_S, t_S, l_{SR}, a_S)$ is essentially semantically equivalent to the following collection of programs, one for each value n:

$$\textbf{if } K_S(K_R X(0) \wedge \ldots \wedge K_R X(n-1)) \wedge \neg K_S K_R X(n) \textbf{ then } \mathit{send}_{l_{SR}}(\langle n, X(n)\rangle) \textbf{ else } \mathit{skip}.$$

In both of these programs, S takes the same action under the same circumstances, and with the same effects on its local state. That is, given a run r (i.e., a sequence of global states) consistent with the collection of knowledge-based programs, we can construct an event structure es consistent with $Pg_S^{kb}(\varphi_S, t_S, l_{SR}, a_S)$ such that the sequence of local states of S in es, with stuttering eliminated, is the same as in r. The converse is also true. More precisely, in a run r consistent with the collection of knoweldge-based programs, at each point of time, either S knows that R knows the value of $X(n)$ for all n, or there exists a smallest n such that $\neg K_S K_R(X(n))$ holds. In the first case, S does nothing, while in the second case S sends $\langle n, X(n)\rangle$ on l_{SR}. Similarly, in an event structure es consistent with $Pg_S^{kb}(\varphi_S, t_S, l_{SR}, a_S)$, if S knows that R knows $X(n)$ for all n, then S does nothing; if not, then it is impossible for S to know that R knows the first n bits, but never know that R knows $X(n)$, without eventually S taking an a_S action with value $\langle n, X(n)\rangle$. This means that for each run r consistent with the collection of knowledge-based programs, the event structure es in which S starts from the same initial state as in r and performs action a_S as soon as it is enabled has the same sequence of local states of S as r. For each event structure es consistent with $Pg_S^{kb}(\varphi_S, t_S, l_{SR}, a_S)$, in the run r of global states in es with stuttering eliminated, S takes action a_S as soon as enabled; subsequently, r is consistent with the collection of knowledge-based programs.

Similarly, $Pg_R^{kb}(\varphi_R, \mathbf{c}_R, l_{RS}, a_R)$ is essentially semantically equivalent to the following collection of programs, one for each value n:

$$\textbf{if } K_R X(0) \wedge \ldots\ldots \wedge K_R X(n-1) \wedge \neg K_R X(n) \textbf{ then } \mathit{send}_{l_{RS}}(n) \textbf{ else } \mathit{skip}.$$

Thus, the derived program is essentially one of the knowledge-based programs considered by Halpern and Zuck [1992]. 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 knowledge-based program and then verify that it satisfied the specification. Rather, we derived the knowledge-based programs for the sender and receiver from the proof that the specification was satisfiable. And, while Nuprl required "hints" in terms of what to prove, the key ingredients of the proof, namely, the specification $Fair_I^{kb}(\varphi, t, l)$ and the proof that $Fair\text{-}Pg(\varphi, t, l, a)$ 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.

4.2 Synthesis of standard programs for STP

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 [1992], the exact standard program that we use depends on the underlying assumptions about the communications systems. Here we sketch an approach to finding such a standard program.

The first step is to identify the exact properties of knowledge that are needed for the proof. This can be done by inspecting the proof to see which properties of the knowledge operators K_S and K_R are used. The idea is then to replace formulas involving the knowledge operators by standard (non-epistemic formulas) which have the relevant properties.

Suppose that $\tilde{\varphi}_S^{kb}$ is a formula that mentions the function X, has a free variable m, and is guaranteed to be an S-formula in all interpretations I and systems Sys. (Recall that, as noted just before Lemma 3.1, there are simple syntactic conditions that guarantee that a formula is an i-formula for all I and Sys.) Roughly speaking, we can think of $\tilde{\varphi}_S^{kb}$ as corresponding to $K_S K_R(X(m))$. Let φ_S^{kb} be an abbreviation of

$$\exists n.\ ((\forall k < n.\ \tilde{\varphi}_S^{kb}[m/k]) \wedge \neg\tilde{\varphi}_S^{kb}[m/n]).$$

Similarly, suppose that $\tilde{\varphi}_R^{kb}$ is a formula that mentions X, has a free variable m, and is guaranteed to be an R-formula in all interpretations I; let φ_R^{kb} be an abbreviation of

$$\exists n.\ ((\forall k < n.\ \tilde{\varphi}_R^{kb}[m/k]) \wedge \neg\tilde{\varphi}_R^{kb}[m/n]).$$

Thus, φ_S^{kb} and φ_R^{kb} are the analogues of φ_S and φ_R in Section 4.1. While φ_S is a formula that says that there is a least n such that $K_S K_R X(n)$ does not hold, φ_S^{kb} says that there is a least n such that $\tilde{\varphi}_S^{kb}(n)$ does not hold. Similarly, while φ_R^{kb} says that there is a least n such that $K_R X(n)$ does not hold, φ_R^{kb} says that there is a least n such that $\tilde{\varphi}_R^{kb}(n)$ does not hold.

We also use we use constants $\tilde{\mathbf{c}}_S$, and $\tilde{\mathbf{c}}_R$ that are analogues to $\mathbf{c}_S$, $\mathbf{c}_R$; $\tilde{\varphi}_S^{kb}$ plays the same role in the definition of $\tilde{\mathbf{c}}_S$ as $K_S K_R(X(m))$ played in the definition of $\mathbf{c}_S$, and $\tilde{\varphi}_R^{kb}$ plays the same role in the definition of $\tilde{\mathbf{c}}_R$ as $K_R(X(m))$ played in the definition of $\mathbf{c}_R$. Thus, we take $\tilde{\mathbf{c}}_S$ to be a constant that represents the least n such that $\tilde{\varphi}_S^{kb}[m/n]$ does not hold (that is, we want $\exists n.\ \forall k < n.\ \tilde{\varphi}_S^{kb}[m/k] \wedge \neg\tilde{\varphi}_S^{kb}[m/n] \Rightarrow (n = \tilde{\mathbf{c}}_S)$ to be true), and define $\tilde{t}_S$ as the pair $\langle \tilde{\mathbf{c}_S}, X(\tilde{\mathbf{c}_S})\rangle$, Similarly, we take $\tilde{\mathbf{c}}_R$ to be a constant that represents the least n such that $\tilde{\varphi}_R^{kb}[m/n]$ does not hold (that is, we want $\exists n.\ \forall k < n.\ \tilde{\varphi}_R^{kb}[m/k] \wedge \neg\tilde{\varphi}_R^{kb}[m/n] \Rightarrow (n = \tilde{\mathbf{c}}_R)$ to be true).

Let $\varphi_{stp}^{kb}(\tilde{\varphi}_R^{kb})$ be the specification that results by using $\tilde{\varphi}_R^{kb}$ instead of K_R in φ_{stp}^{kb}:

$$\varphi_{stp}^{kb}(\tilde{\varphi}_R^{kb}) =_{\rm def} \forall n.\ \Diamond\tilde{\varphi}_R^{kb}[m/n].$$

We prove the goal $\varphi_{stp}^{kb}(\tilde{\varphi}_R^{kb})$ by refinement: at each step, a rule (or tactic) of Nuprl is applied, and a number of subgoals (typically easier to prove) are generated; the rule gives a mechanism of constructing a proof of the goal from proofs of the subgoals. Some of the subgoals cannot be further refined in an obvious manner; this is the case, for example, for the simple conditions on $\tilde{\varphi}_S^{kb}$ or $\tilde{\varphi}_R^{kb}$. The new theorem states that, under suitable conditions on $\tilde{\varphi}_S^{kb}$ and $\tilde{\varphi}_R^{kb}$, $\varphi_{stp}^{kb}(\tilde{\varphi}_R^{kb})$ is satisfiable if both $Fair_I^{kb}(\varphi_S^{kb}, \tilde{t}_S, l_{SR})$ and $Fair_I^{kb}(\varphi_R^{kb}, \tilde{\mathbf{c}}_R, l_{RS})$ are satisfiable.

We now explain the conditions placed on the predicates $\tilde{\varphi}_S^{kb}$ and $\tilde{\varphi}_R^{kb}$. One condition is that $\tilde{\varphi}_R^{kb}$ be *stable*, that is, once true, it stays true:

$$\begin{aligned} Stable(\tilde{\varphi}_R^{kb}) =_{\text{def}} \quad & \lambda Sys.\, \forall es \in Sys.\, \forall e_R@R \in es.\, \forall n.\, I(\tilde{\varphi}_R^{kb}[m/n])(Sys, state\ before\ e_R) \Rightarrow \\ & I(\tilde{\varphi}_R^{kb}[m/n])(Sys, state\ after\ e_R). \end{aligned}$$

Assuming $Stable(\tilde{\varphi}_R^{kb})$ allows us to prove φ_R^{kb} by induction on the least index n such that $\neg\tilde{\varphi}_R^{kb}[m/n]$ holds.

To allow us to carry out a case analysis on whether $\tilde{\varphi}_R^{kb}$ holds, we also assume that $\tilde{\varphi}_R^{kb}$ satisfies the principle of excluded middle; that is, we assume that $Determinate(\tilde{\varphi}_R^{kb}) =_{\text{def}} Determinate(\forall n.\, (\tilde{\varphi}_R^{kb}[m/n])^t)$. For similar reasons, we also restrict $\tilde{\varphi}_S^{kb}$ to being stable and determinate; that is, we require that $Stable(\tilde{\varphi}_S^{kb})$ and $Determinate(\tilde{\varphi}_S^{kb})$ both hold.

The third condition we impose establishes a connection between $\tilde{\varphi}_S^{kb}$ and $\tilde{\varphi}_R^{kb}$, and ensures that, for all values n, if $\tilde{\varphi}_S^{kb}[m/n]$ holds, then eventually $\tilde{\varphi}_R^{kb}[m/n]$ will also hold:

$$\begin{aligned} Implies(\tilde{\varphi}_S^{kb}, \tilde{\varphi}_R^{kb}) =_{\text{def}} \quad & \lambda Sys.\, \forall es \in Sys.\, \forall n.\, \forall e_S@S \in es.\, I(\tilde{\varphi}_S^{kb}[m/n])(Sys, state\ before\ e_S) \Rightarrow \\ & \exists e_R \succ e_S@R \in es.\, I(\tilde{\varphi}_R^{kb}[m/n])(Sys, state\ after\ e_R). \end{aligned}$$

To explain the next condition, recall that $\tilde{\varphi}_R$ is meant to represent $K_R(X(m))$. With this interpretation, $I(\forall k \leq n.\ \tilde{\varphi}_R^{kb}[m/k])(Sys, state\ before\ send(e_S))$ says that R knows the first n bits before it sends a message to S. We would like it to be the case that, just as with the knowledge-based derivation, when S receives R's message, S knows that R knows the n^{th} bit. Since we think of $\tilde{\varphi}_S^{kb}$ as saying that $K_S K_R(X(m))$ holds, we expect $I(\tilde{\varphi}_S^{kb}[m/n])(Sys, state\ after\ e_S)$ to be true. Define $Rcv(\tilde{\varphi}_S^{kb}, \tilde{\varphi}_R^{kb}, l_{RS})$ to be an abbreviation of

$$\begin{aligned} & \lambda Sys. \forall es \in Sys.\, \forall e_S@S \in es.\, (kind(e_S) = rcv(l_{RS})) \Rightarrow \\ & \quad \forall n.\, (\forall k \leq n.\, I(\tilde{\varphi}_R^{kb}[m/n])(Sys, state\ after\ send(e_S))) \Rightarrow I(\tilde{\varphi}_S^{kb}[m/n])(Sys, state\ after\ e_S). \end{aligned}$$

With this background, we can describe the last condition. Intuitively, it says that if n is the least value for which $\tilde{\varphi}_S^{kb}$ fails when S sends a message to R, then $\tilde{\varphi}_R^{kb}$ holds for n upon message delivery:

$$\begin{aligned} & Rcv(\tilde{\varphi}_R^{kb}, \tilde{\varphi}_S^{kb}, l_{SR}) \equiv \\ & \quad \lambda Sys.\, \forall es \in Sys.\ \forall e_R@R \in es.\, (kind(e_R) = rcv(l_{SR})) \Rightarrow \\ & \quad \forall n.\, (I(\varphi_S^{kb}[m/n])(Sys, state\ before\ send(e_R)) \Rightarrow I(\tilde{\varphi}_R^{kb}[m/n])(Sys, state\ after\ e_R)). \end{aligned}$$

We denote the conjunction of these conditions as $\psi^{kb}(\tilde{\varphi}_S^{kb}, \tilde{\varphi}_R^{kb}, \tilde{t}_S, \tilde{\mathbf{c}}_R, l_{SR}, l_{RS})$. The new theorem says

$$\begin{aligned} & \psi^{kb}(\tilde{\varphi}_S^{kb}, \tilde{\varphi}_R^{kb}, \tilde{t}_S, \tilde{\mathbf{c}}_R, l_{SR}, l_{RS}) \wedge \\ & Fair_I^{kb}(\varphi_S^{kb}, \tilde{t}_S, l_{SR}) \wedge Fair_I^{kb}(\varphi_R^{kb}, \tilde{\mathbf{c}}_R, l_{RS}) \wedge \\ & \forall n.\, \forall k < n.\, \tilde{\varphi}_S^{kb}[m/k] \wedge \neg\tilde{\varphi}_S^{kb}[m/n] \Rightarrow (n = \tilde{\mathbf{c}}_S) \wedge \\ & \forall n.\, \forall k < n.\, \tilde{\varphi}_R^{kb}[m/k] \wedge \neg\tilde{\varphi}_R^{kb}[m/n] \Rightarrow (n = \tilde{\mathbf{c}}_R) \\ & \quad \Rightarrow \varphi_{stp}^{kb}(\tilde{\varphi}_R^{kb}). \end{aligned}$$

We can prove that the following is true for any two distinct actions a_S and a_R:

$$\begin{aligned}&Pg^{kb}_S(\varphi^{kb}_S, \tilde{t}_S, l_{SR}, a_S) \oplus Pg^{kb}_R(\varphi^{kb}_R, \tilde{\mathbf{c}}_R, l_{RS}, a_R) \approx_I \\ &\quad \psi^{kb}(\tilde{\varphi}^{kb}_S, \tilde{\varphi}^{kb}_R, \tilde{t}_S, \tilde{\mathbf{c}}_R, l_{SR}, l_{RS}) \land \mathit{FairSend}^{kb}(l_{RS}) \land \mathit{FairSend}^{kb}(l_{SR}) \Rightarrow \varphi^{kb}_{stp}(\tilde{\varphi}^{kb}_R),\end{aligned}$$

where

$$\begin{aligned}&Pg^{kb}_S(\varphi^{kb}_S, \tilde{t}_S, l_{SR}, a_S) =_{\mathrm{def}} \\ &\quad \mathit{Fair\text{-}Pg}(\varphi^{kb}_S, \tilde{t}_S, l_{SR}, a_S) \oplus \\ &\quad @S\ \mathbf{initially}\ \Box\,(\forall n.\ \forall k < n.\ \tilde{\varphi}^{kb}_S[m/k] \land \neg\tilde{\varphi}^{kb}_S[m/n] \Rightarrow n = \tilde{\mathbf{c}}_S),\end{aligned}$$

$$\begin{aligned}&Pg^{kb}_R(\varphi^{kb}_R, \tilde{\mathbf{c}}_R, l_{RS}, a_R)) =_{\mathrm{def}} \\ &\quad \mathit{Fair\text{-}Pg}(\varphi^{kb}_R, \tilde{\mathbf{c}}_R, l_{RS}, a_R) \oplus \\ &\quad @R\ \mathbf{initially}\ \Box\,(\forall n.\ \forall k < n.\ \tilde{\varphi}^{kb}_R[m/k] \land \neg\tilde{\varphi}^{kb}_R[m/n] \Rightarrow n = \tilde{\mathbf{c}}_R).\end{aligned}$$

In particular, for the terms t_S and $\mathbf{c}_R$ defined in the previous section, we can show that $\psi^{kb}(K_SK_RX(m), K_RX(m), t_S, \mathbf{c}_R, l_{SR}, l_{RS})$ is true. Thus, the new theorem is indeed a generalization of the previous results.

The formulas $K_SK_RX(m)$ and $K_RX(m)$ are not the only ones that satisfy these conditions. Most importantly for the purpose of extracting standard programs, the conditions are satisfied by non-epistemic formulas, that is, formulas whose interpretations do not depend on the entire system, just on the local states of the sender or the receiver agents, respectively. Note that Lemma 2.12 guarantees that the extracted program is consistent.

For example, we can take x_S to be an S-local variable that stores the largest n such that S has received requests from R for all of the first n bits in X; that is, initially x_S is set to to -1, and if S receives a request for $X(n)$ and $x_S = n-1$, then x_S is set to n. We similarly take x_R to be an R-local variable that stores the largest n such that R has received the first $n-1$ bits of X from S . That is, initially, x_R is set to 0; if R receives a message of the form $\langle n, msg\rangle$ from S and $x_R = n-1$, then x_R is set to n. We have in mind a setting in which the receiver requests bits from the sender in order, that is, R starts by requesting $X(0)$, and does not request $X(n)$ before receiving $X(n-1)$. Similarly, the aim is to have the sender send bits in order; that is, S does not not send $X(n+1)$ before S knows that $X(n)$ has been received. For the generalized knowledge-based formulation of the STP problem, we use a formula $\tilde{\varphi}^{kb}_S$ that is meant to correspond to $K_SK_RX(m)$. Given the intuition above, at all times, S knows that R has received bits $0, \ldots, x_S - 1$; that is, $\tilde{\varphi}^{kb}_S(m)$ holds here iff $x_S > m$. Since φ^{kb}_S holds iff there is a least m such that $\neg\tilde{\varphi}(m)$ holds, φ^{kb}_S is vacuously *true* here. Moreover, the term $\tilde{\mathbf{c}}_s$ that represents the least such m is just x_S. Similarly, the formula $\tilde{\varphi}^{kb}_R$ is meant to correspond to $K_RX(m)$; here we can take $\tilde{\varphi}^{kb}_R$ to be $x_R > m$. Again, φ^{kb}_R becomes vacuously *true*, and the corresponding term $\tilde{c}_R$, which represents the least m such that $\tilde{\varphi}^{kb}_R$ does not hold, is just x_R.

It is not hard to show that $\psi^{kb}(x_S > m x_R > m, \langle x_S, X(x_S)\rangle, x_R, l_{SR}, l_{RS})$ holds in the system generated by $Pg(\mathit{true}, \langle x_S, X(x_S)\rangle, l_{SR}, a_S) \oplus Pg(\mathit{true}, x_R, l_{RS}, a_R)$, for any distinct actions a_S and a_R; this specification is not knowledge-based. Recall that $\varphi^{kb}_{stp}(\tilde{\varphi}^{kb}_R)$ is $\forall n.\Diamond\tilde{\varphi}^{kb}_R[m/n]$. In this context, it is the formula $\forall n.\Diamond(x_R = n)$. In addition, $\varphi^{kb}_{stp}(x_R > m)$ implies φ^{kb}_{stp} (in the system generated by $Pg(\mathit{true}, \langle x_S, X(x_S)\rangle, l_{SR}, a_S) \oplus Pg(\mathit{true}, x_R, l_{RS}, a_R)$), so if message communication is fair,

$$Pg(\mathit{true}, \langle x_S, X(x_S)\rangle, l_{SR}, a_S) \oplus Pg(\mathit{true}, x_R, l_{RS}, a_R)$$

satisfies the STP specification, as long as a_S and a_R are distinct actions. We can easily give a justification for this result: If R follows $Pg(\mathit{true}, x_R, l_{RS}, a_R)$ and S follows $Pg(\mathit{true}, \langle x_S, X_S(x_S)\rangle, l_{SR}, a_S)$, then

R starts by sending message 0 to S; since communication is fair, eventually S receives this message, and x_S is set to 0; S starts sending $\langle 0, X(0)\rangle$ to R; since communication is fair, eventually R receives this message, so R sets x_R to 1, stops sending message 0 to S, and starts sending message 1 to S. It is not difficult to show that, for all values n, there is a time when x_R is set to n, which triggers R to send message n to S. S eventually receives this message and starts sending $\langle n, X(n)\rangle$ to R. This, in turn, ensures that R eventually receives this message and thus learns $X(n)$. Note that the program $Pg(true, \langle x_S, X_S(x_S)\rangle, l_{SR}, a_S) \oplus Pg(true, x_R, l_{RS}, a_R)$ is realizable. We have thus extracted a standard program that realizes the STP specification. In fact, the program turns out to be essentially equivalent to Stenning's [1976] 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.

5 Conclusion and Future Work

We have shown that the mechanism for synthesizing programs from specifications in Nuprl can be extended to knowledge-based programs and specifications, Moreover, we have shown that axioms much in the spirit of those used for standard programs can be used to synthesize kb programs as well. We applied this methodology to the analysis of the sequence transmission problem and showed that the kb programs of proposed by Halpern and Zuck for solving the STP problem can be synthesized in Nuprl. We also sketched an approach for deriving standard programs that implement the kb programs that solve the STP. A feature of our approach is that the extracted standard programs are close to the type of pseudocode designers write their programs in, and can be translated into running code.

There has been work on synthesizing both standard programs and kb programs from kb specifications. In the case of synchronous systems with only one process, Van der Meyden and Vardi [1998] provide a necessary and sufficient condition for a certain type of kb specification to be realizable, and show that, when it holds, a program can be extracted that satisfies the specification. Still assuming a synchronous setting, but this time allowing multiple agents, Engelhardt, van der Meyden, and Moses [1998, 2001] propose a refinement calculus in which one can start with an epistemic and temporal specification and use refinement rules that eventually lead to standard formulas. The refinement rules annotate formulas with preconditions and postconditions, which allow programs to be synthesized from the leaf formulas in a straightforward way. A search up the tree generated in the refinement process suffices to build a program that satisfies the specification. The extracted programs are objects of a programming language that allows concurrent and sequential executions, variable assignments, loops and conditional statements.

We see our method for synthesizing programs from kb specifications as an alternative to this approach. As in the Engelhart et al. approach, the programs extracted in Nuprl are close to realistic programming languages. Arguably, distributed I/O message automata are general enough to express most of the distributed programs of interest when communication is done by message passing. Our approach has the additional advantage of working in asynchronous settings.

A number of questions, both theoretical and more applicative, still remain open. While synthesis of distributed programs from epistemic and temporal specifications is undecidable in general, recent results [Meyden and Wilke 2005] show that, under certain assumptions about the setting in which agents communicate, the problem is decidable. It would be worth understanding the extent to which these assumptions apply to our setting. Arguably, to prove a result of this type, we need a better understanding of how properties of a number of kb programs relate to the properties of their composition; this would also allow us to prove stronger composition rules than the one presented in Section 3.2. As we said, we believe that the approach that we sketched for extracting a standard program from the kb specification for the STP problem can be extended into a general methodology. As pointed out by Engelhart et al., the key difficulty in extracting standard programs from abstract specifications is in coming up with good standard tests to replace the abstract tests in a program. However, it is likely that, by reducing the complexity of the problem and focusing only on certain classes of kb specifications, "good" standard tests can be more easily identified. We plan to investigate heuristics for finding such tests and to implement them as tactics in Nuprl.

References

Aho, A. V., J. D. Ullman, A. D. Wyner, and M. Yannakakis (1982). Bounds on the size and transmission rate of communication protocols. *Computers and Mathematics with Applications 8*(3), 205–214. This is a later version of [Aho, Ullman, and Yannakakis 1979].

Aho, A. V., J. D. Ullman, and M. Yannakakis (1979). Modeling communication protocols by automata. In *Proc. 20th IEEE Symposium on Foundations of Computer Science*, pp. 267–273.

Bartlett, K. A., R. A. Scantlebury, and P. T. Wilkinson (1969). A note on reliable full-duplex transmission over half-duplex links. *Communications of the ACM 12*, 260–261.

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

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

Bickford, M. and R. L. Constable (2005). A causal logic of events in formalized computational type theory. Report, Cornell University.

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

Brouwer, L. E. J. (1923). On the significance of the principle of excluded middle in mathematics, especially in function theory. *J. für die Reine und Angewandte Mathematik 154*, 1–7.

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

Constable, R. L. Naive computational type theory. In U. Berger and H. Schwichtenberg (Eds.), *Proof and System-Reliability, Proceedings of International Summer School, vol. 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. van der 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. van der Meyden, and Y. Moses (2001). A refinement theory that supports reasoning about knowledge and time for synchronous agents. In *Proc. International Conference 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. A slightly revised paperback version was published in 2003.

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

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.

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

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.

Meyden, R. van der and M. Y. Vardi (1998). Synthesis from knowledge-based specifications. In *Proc. Ninth International Conference on Concurrency Theory (CONCUR'98)*, pp. 34–49.

Meyden, R. van der and T. Wilke (2005). Synthesis of distributed systems from knowledge-based specifications. Technical Report UNSW-CSE-TR-0504, University of New South Wales.

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

Stenning, M. V. (1976). A data transfer protocol. *Comput. Networks 1*, 99–110.

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