Knowledge-Based Synthesis of Distributed Systems Using Event Structures*

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

Robert 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 [20] 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. 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 (see, e.g., [5; 10; 12; 13; 14; 26]. Here programs are constructed from *proofs* that the specifications are satisfiable. That is, a constructive proof that a specification is satisfiable gives a program that satisfies the specification. This

*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.

method has been successfully used by several research groups and companies to construct large complex *sequential* programs; and it has been used to synthesize distributed protocols such as Paxos [22], and various authentication protocols (see www.nuprl.org).

The Cornell Nuprl proof development system was among the first tools used to create correct-by-construction functional and sequential programs [12]. Nuprl has also been used extensively to optimize distributed protocols [23], and to formalize them in the language of I/O Automata [7]. Recent work by two of the authors [11] 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 [11]. Moreover, the current version of the Nuprl prover is itself a distributed system [3].

However, as has long been recognized [18], 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* [18; 19]. 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. [18, 19]. 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 [15; 20; 28].

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 (STP)*, 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 [20] provide knowledge-based programs for the sequence-transmission problem, 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. This paper relies heavily on prior work on knowledge-based programs of Halpern et al. [18, 19, 20]; the novelty lies in offering a proof of concept that knowledge-based specifications and programs can be formulated in a constructive logic, and that knowledge-based programs can be synthesized in a semi-automatic system like Nuprl.

Engelhardt, van der Meyden, and Moses [16, 17] 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 formalize 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 a discussion of related work and 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 [3] 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 the product space and function constructors $\times$ 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 (also called *computational* type theory), on which Nuprl is based, was developed to provide a foundation for computer science and 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 [8], 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. These definitions rely on a formalization of the notion of executions *consistent* with a program, that is, executions that could have been generated by running the program. As will be clear in the next sections, we can formally define in Nuprl a notion of consistency for the programs and executions considered in this paper.

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 $Sem = 2^{Sem'}$, where 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.

Theoremhood in constructive type theory 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* [24] and are similar to *UNITY* programs [9] (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 meet criterion (b), we formalize runs as structures that we call *event structures*, much in the spirit of Lamport's [21] model of events in distributed systems. Event structures are explained in more detail in the next section. We have shown [11] 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. Intuitively, sending a message m will be identified with setting the variable $msg(l)$ to $m \neq \perp$.

We assume that communication is asynchronous, so there is no global notion of time. Following Lamport [21], 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 non-deterministically). 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 [6].)

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 the 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 [21], 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.[1] 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 $agent(send(e)) = source(l)$ if $kind(e) = rcv(l)$. 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 $first(e)$ holds exactly when e is the first event in the history associated with $agent(e)$ in es; if $first(e)$ does not hold, then $pred(e)$ is the unique predecessor of e in es. Following Lamport [21], 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 $initstate_i$. (Note that one way to ensure this is to assume that each local variable has an initial value; the initial state is the state that assigns each local variable its initial value.) 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 local state 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.

[1] We use the term *sequence* as a simplification. As explained in the remainder of the section, just as for Lamport, executions are technically partial orders on events respecting local orders and causality.

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 m 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 $x\ after\ e = x\ before\ e + v = v'$.

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 $agent(send(e)) = source(l)$:

$$\forall e \in es. \forall l.\ (kind(e) = rcv(l)) \Rightarrow (val(e) = msg(l)\ after\ send(e)) \wedge (agent(e) = dest(l)) \wedge (agent(send(e)) = source(l)).$$ [2]

- 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:

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

[2] For simplicity, in the remainder of this paper, we abuse notation and write $e \in es$ to indicate that e is an event that occurs in es.

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

$$\forall e \in es.\, \forall i.\, ((agent(e) = i) \wedge \neg \mathit{first}(e)) \Rightarrow \\ ((pred(e) \prec_i e) \wedge (\forall e' \in es.\, \neg(pred(e) \prec_i e' \prec_i e))).$$

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

$$\forall e \in es.\, \forall i.\, (agent(e) = i \wedge \neg \mathit{first}(e)) \Rightarrow \\ \forall x \in X_i.\, (x \ \mathit{after}\ pred(e) = x \ \mathit{before}\ e).$$

- 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 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 [11] 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:

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

The set of event structures is definable in Nuprl (see [11]). 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 $x \notin \cup_{i \in AG} X_i$. Allowing non-local variables is not an artificial generalization; just by looking at a few classic distributed problems, we

can see that non-local variables are ubiquitous. For example, in a problem where each agent has an input variable and the goal is for agents to compute an aggregate of the local inputs, the aggregate is a non-local variable.

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.[3] 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 $x \in X_i$;
- $@i$ $kind = local(a)$ **only if** φ;
- $@i$ **if necessarily** φ **then i.o.** $kind = local(a)$; and
- $@i$ **only events in** L **affect** x, where L is a list of kinds in $Act \cup Links$ and $x \in X_i$.

Note that all basic programs 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 or same agents. (Note that, since message automata associated with same agent can be composed in our language, we are not relying on a standard notion of parallel composition.) 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 $Pg_1 \oplus Pg_2$.[4]

Readers familiar with UNITY [9] will see some obvious similarities. In UNITY, a program consists of an initial condition on a global state, a set of guarded assignment statements that update this state non-deterministically as if running an unbounded loop, and a set of allowed actions. As we said earlier, communication occurs through reading and writing shared variables (rather than by message passing, as in Nuprl). States in Nuprl are also considerably more expressive than those used in UNITY.

2.3.2 Semantics We give semantics by associating with each program the set of event structures consistent with it. Intuitively, a set of event structures is consistent with a distributed message automaton if each event structure in the set is generated from 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, that is, an interpretation consists of a domain and an assignment of each predicate and function symbol to a predicate and function, respectively, over that domain. 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 function 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

[3] 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.

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

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 $P \in \mathcal{P}$ and function symbol $f \in \mathcal{F}$, 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 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 interpreted as a global state.[5] 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 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 [12]. 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 $\neg\varphi^+$.

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).

[5] 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.

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 $@i$ **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,

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

(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 $@i$ **if** $kind = k$ **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

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

where we write $\forall e@i \in es.\ \varphi$ as an abbreviation for $\forall e \in es.agent(e) = i \Rightarrow \varphi$. 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; $Consistent_I(@i$ **if** $kind = k$ **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,

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

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, $@i\ kind = local(a)$ **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:

$$\begin{aligned} &Consistent_I(@i\ kind = local(a) \textbf{ only if } \varphi, es) =_{\text{def}} \\ &\quad i\text{-}\textit{formula}(\varphi, I)\ \wedge \forall e@i \in es.\ (kind(e) = local(a) \Rightarrow I(\varphi)(state\ before\ e)). \end{aligned}$$

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

Standard formalizations of input-output automata (see [24]) 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.** $kind = local(a)$, which we call a *fairness program*. Intuitively, it says that if φ holds from some point on, then an event with kind $local(a)$ 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

$$\begin{aligned} &Consistent_I(@i \textbf{ if necessarily } \varphi \textbf{ then i.o. } kind = local(a), es) =_{\text{def}} \\ &\quad i\text{-}\textit{formula}(\varphi, I) \wedge \\ &\quad [((\exists e@i \in es) \wedge \forall e@i \in es.\ \exists e' \succeq_i e.\ I(\neg\varphi)(state\ after\ e') \vee (kind(e') = local(a))) \\ &\quad \vee (\neg(\exists e@i \in es) \wedge I(\neg\varphi)(initstate_i))]. \end{aligned}$$

The last type of basic program, $@\mathbf{i}$ **only events in L affect** 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 $msg(l)$. If x does not have the form $msg(l)$, then

$$Consistent_I(@i \textbf{ only events in L affect } x, es) =_{\text{def}} \\ \forall e@i \in es.\ ((x\ after\ e) \neq (x\ before\ e) \Rightarrow (kind(e) \in L)).$$

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

$$Consistent_I(@i \textbf{ only events in L affect msg}(l), es) =_{\text{def}} \\ \forall e@i \in es.\ ((msg(l)\ after\ e \neq \perp) \Rightarrow (kind(e) \in L)).$$

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:

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

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 by 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 $@i$ **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 $@i$ **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 $@i$ **if necessarily** φ **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 $@i$ **if necessarily** φ **then i.o**. $kind = local(a)$.

If φ does not satisfy the principle of excluded middle in I, then $@i$ **if necessarily** φ **then i.o**. $kind = local(a)$ may not be realizable with respect to I. 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 $@i$ **initially** ψ and $@i$ **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 Constable and Bickford [11] 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 is 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:

$$@i \textbf{ initially } \psi \models\!\approx_I \lambda es.\ i\text{-}\textit{formula}(\psi, I) \wedge I(\psi)(initstate_i).$$

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

Ax-cause:

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

Ax-if:

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

Ax-fair:

$$@i \textbf{ if necessarily } \varphi \textbf{ then i.o.}\ kind = local(a) \mid\!\approx_I$$
$$\lambda es.\ \ i\text{-}formula(\varphi, I) \wedge$$
$$[((\exists e @ i \in es) \wedge \forall e @ i \in es.\ \exists e' \succeq_i e.\ I(\neg\varphi)(state\ after\ e') \vee (kind(e') = local(a)))$$
$$\vee(\neg(\exists e @ i \in es) \wedge I(\neg\varphi)(initstate_i))].$$

Ax-affect:

$$@i \textbf{ only events in L affect } x \mid\!\approx_I\ \lambda es.\ \forall e @ i \in es.\ (x\ after\ e \neq x\ before\ e) \Rightarrow (kind(e) \in L).$$

Ax-sends:

$$@i \textbf{ only events in L affect msg}(l) \mid\!\approx_I\ \lambda es.\ \forall e @ i \in es.\ (msg(l)\ after\ e \neq \bot) \Rightarrow (kind(e) \in L).$$

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

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

Lemma 2.6: *Axioms* **Ax-init**, **Ax-cause**, **Ax-if**, **Ax-fair**, **Ax-affect**, **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 $Consistent_I$. ∎

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 $\exists Pg.\ (Pg \mid\!\approx_I\ \varphi)$ holds. We now provide a general scheme for doing this. Consider the following scheme, which we call $\mathcal{GS}$:

1. Find specifications φ_1, φ_2, ..., φ_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, ..., Pg_n such that $Pg_i \mid\!\approx_I\ \varphi_i$ holds for all $i \in \{1, \ldots n\}$.
3. Conclude that $Pg \mid\!\approx_I\ \varphi$, 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 $Fair_I(\varphi, t, l)$, where $i \neq j$, l is a link with $source(l) = i$ and $dest(l) = j$, φ is an i-formula, and t is an i-term. $Fair_I(\varphi, t, l)$ 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. (More precisely, φ holds when evaluated with respect to the state of agent i before e occurs, that is, in *state before* e.) 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:

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

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:

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

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).

In this section, we show that, assuming that the communication on link l satisfies a strong fairness requirement, the specification above is satisfiable, and that a program that satisfies it can be formulated in our language. Furthermore, we show that there are simple conditions on the formulas involved in this program that ensure the existence of at least one execution of the program. Though the specification above and the program that satisfies it refer to a single agent, we show that it is not difficult to extend our results to a system with many agents.

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

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

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, Fair-Pg*(φ,t,l,a) *satisfies* $\lambda es.FairSend(l)(es) \Rightarrow Fair_I(\varphi,t,l)(es)$ *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 *i-formula*(φ, I) and *i-term*(t, I) both hold.

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

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

We want to find formulas $\psi_1(es), \ldots, \psi_4(es)$ that follow from the four basic programs that make up *Fair-Pg*(φ,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 $kind(e) = local(a)$. This assumption is expressed by:

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

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

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

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 $msg(l)$ after $send(e)$, it is easy to check that $(\psi_1(es) \wedge \psi_3(es)) \Rightarrow \varphi_2(es))$ is true, where $\psi_3(es)$ is

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

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

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

It follows that

$$(\forall es.(\psi_1(es) \wedge \psi_2(es) \wedge \psi_3(es) \wedge \psi_4(es)) \Rightarrow (FairSend(l)(es) \Rightarrow Fair_I(\varphi,t,l)(es))).$$

Step 2. By **Ax-sends**,

$$@i\ \textbf{only events in}\ [a]\ \textbf{affect}\ \textbf{msg}(l) \approx_I \psi_1.$$

By **Ax-if**,

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

By **Ax-cause**,

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

and by **Ax-fair**

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

By the soundness of $\mathcal{GS}$ (Proposition 2.7), $Fair\text{-}Pg(\varphi, t, l, a)$ satisfies $\lambda es.FairSend(l)(es) \Rightarrow Fair_I(\varphi, t, l)(es)$ 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 $i, j \in AG$, $l \in Links$, $a \in Act$, and $initstate_i = s_i^*$. In the latter case, choose es with AG and $Links$ as above, let $Act = \{a, b\}$, 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 $FairSend(l)$. ∎

The notion of strong communication fairness is essential for the results above: $Fair_I(\varphi, t, l)$ 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 $Fair\text{-}Pg(\varphi, t, l, a)$ and $Fair\text{-}Pg(\varphi, t, l', a)$ 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*

$$\lambda es.(FairSend(l)(es) \wedge FairSend(l')(es)) \Rightarrow (Fair_I(\varphi,t,l)(es) \wedge Fair_I(\varphi',t',l')(es))$$

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 $\lambda es.\ FairSend(l)(es) \Rightarrow Fair_I(\varphi, t, l)(es)$ and φ_2 to be $\lambda es.\ FairSend(l')(es) \Rightarrow Fair_I(\varphi', t', l')(es)$.

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

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 $\lambda es.(FairSend(l)(\ es) \wedge FairSend(l')(es)) \Rightarrow (Fair_I(\varphi, t, l)(es) \wedge Fair_I(\varphi', t', l')(es))$ 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 $initstate_i = s_i$ and $initstate_{i'} = s_{i'}$. 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 $l, l' \in Links$, 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 finite sets $\mathcal{P}$ of predicate symbols and $\mathcal{F}$ of function symbols, 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 [18]). In event structures, there is no external notion of time. Fortunately, Panangaden and Taylor [25] give a variant of the standard definition with respect to what they call *asynchronous runs*, which are essentially identical to event structures. We can simply

apply their definition in our framework, replacing using "event structure" instead of "asynchronous run", as we do in the following account.

The truth of formulas is defined relative to a pair (Sys, c), consisting of a system Sys (i.e., 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 $initstate_i(history_i) = \bot$ 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 sufficiently expressive that epistemic and modal operators can be defined within it. 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 only 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 $\mathbf{s}$ and $\mathbf{Sys}$. Intuitively, $\mathbf{s}$ 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

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

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

and

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

Given an interpretation I, let I' be the interpretation that extends I by adding (i.e., conjoining) to φ_I formulas characterizing $\mathbf{Sys}$, $\mathbf{s}$, $\mathbf{CC}$, $\mathbf{ls}$, and $\succeq$ appropriately. That is, the formulas force $\mathbf{Sys}$ to represent a set of event structures, $\mathbf{s}$ to be a consistent cut in one of these event structures, and so on. These formulas are all expressible in Nuprl. 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 $\mathbf{Sys} = Sys$, 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 in an interpretation 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.

Proposition 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 states $s' \in Sys$ 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. Formally, there are five basic knowledge-based clauses for agent i:

- $@i$ **initially** ψ;
- $@i$ **if** $kind = k$ **then** $x := t$;
- $@i$ $kind = local(a)$ **only if** φ;
- $@i$ **if necessarily** φ **then i.o.** $kind = local(a)$; and
- $@i$ **only events in** L **affect** x,

where ψ and φ are i-knowledge-based formulas, $k \in Act \cup Links$, $x \in X_i$, t is an i-term, and L is a list of kinds in $Act \cup Links$.

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 interpretation that is characterized by the formula that results from adding (i.e., conjoining) to φ_I the formula $\mathbf{Sys} = Sys$.[6] Now we can apply the semantics of Section 2.3.2 to get the system $S_{I(Sys)}((Pg^{kb})^t)$. In more detail, since $(Pg^{kb})^t$ is a standard program, we can apply Definition 2.4, which says that the semantics of $(Pg^{kb})^t$ with respect to the interpretation $I(Sys)$ is the set of all event structures in Sys that are consistent with $(Pg^{kb})^t$ with respect to $I(Sys)$; that is,

$$S_{I(Sys)}((Pg^{kb})^t) = \{es \mid Consistent_{I(Sys)}((Pg^{kb})^t, es)\}.$$

[6]The notation $I(Sys)$ may seem somewhat awkward for a formula, but in this case it is a formula that characterizes a system, so it is perhaps not so unreasonable. In any case, since this formula will appear in subscripts (e.g., in Definition 3.2), it seems a better choice than, say, I_{Sys}.

(Note that, technically, $(Pg^{kb})^t$ does take a system as an argument, which was not the case of the type of programs defined in Section 2.; however, as $(Pg^{kb})^t$ is independent of the system argument, its semantics is also independent of any system, which is why we treat it as a standard program.) Since $(\mathbf{Sys} = Sys)$ is a conjunct of $I(Sys)$, all the event structures es in $S_{I(Sys)}((Pg^{kb})^t)$ must be in the set Sys; in other words, $S_{I(Sys)}((Pg^{kb})^t) \subseteq Sys$, for all systems Sys.

In general, the system $S_{I(Sys)}((Pg^{kb})^t)$ will be a strict subset of the system Sys. Indeed, $S_{I(Sys)}((Pg^{kb})^t)$ may even be empty (if there exists no event structure in Sys consistent with $(Pg^{kb})^t$ when interpreted with respect to $I(Sys)$). For example, consider a system with two agents, i and j, where $x_j \in X_j$ is a local variable of agent j. Let i follow the simple program $Pg^{kb} = @i$ **initially** $K_i(x_j = 0)$ that says that initially i knows that j's variable x_j has value 0. Clearly, Pg^{kb} is a knowledge-based program, an instance of the knowledge-based initialization clause $@i$ **initially** ψ for $\psi = K_i(x_j = 0)$. By definition, $\psi^t = \forall \mathbf{s}'.\ (\mathbf{CC}(\mathbf{Sys}, \mathbf{s}') \wedge \mathbf{ls}(\mathbf{s}', i) = \mathbf{ls}(\mathbf{s}, i)) \Rightarrow (\mathbf{s}'_j(x) = 0)$. That is, $I(Sys)(\psi^t)(initstate_i) = (\mathbf{Sys} = Sys) \wedge \forall \mathbf{s}'.\ (\mathbf{CC}(\mathbf{Sys}, \mathbf{s}') \wedge \mathbf{ls}(\mathbf{s}', i) = \mathbf{ls}(\mathbf{s}, i) = initstate_i \Rightarrow (\mathbf{s}'_j(x) = 0)$. This means that an event structure es in Sys is consistent with the clause $@i$ **initially** ψ^t (i.e., with $(Pg^{kb})^t$) if and only if, for all consistent cuts (i.e., global states) s in Sys such that i's local state in s is same as the initial state of i in es, x_j has value 0 in j's local state in s. Consider now a system Sys such that $s_j(x_j) \neq 0$ for all s in Sys. Clearly, no event structure in Sys satisfies this condition, which means that $S_{I(Sys)}((Pg^{kb})^t) = \emptyset$. On the other hand, if Sys is a system such that $s_j(x_j) = 0$ for all s in Sys, then $S_{I(Sys)}((Pg^{kb})^t) = Sys$.

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 $S_{I(Sys)}((Pg^{kb})^t) = Sys$. Intuitively, if Sys is a fixed point, then when interpreted with respect to Sys, the program is acting the way it should. Following Fagin et al. [18, 19], 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, $S^{kb}_I(Pg^{kb}) = \{Sys \in System \mid S_{I(Sys)}((Pg^{kb})^t) = Sys\}$. ∎

As observed by Fagin et al. [18, 19], 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, $S^{kb}_I(Pg) = \{S_I(Pg)\}$. Thus, we can view S^{kb}_I 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 [19], we take a knowledge-based specification 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 $Pg^{kb} \models\!\approx_I Y^{kb}$, 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 $Pg \mathrel{\approx\!\!\!\rightarrow}_I Y^{kb}$ (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

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

(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 I, it 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-affect**, and **Ax-sends** are denoted **Ax-causeK**, **Ax-affectK**, 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:

$$\begin{aligned}\textbf{Ax-ifK}: \quad & @i\ kind = local(a) \textbf{ only if } \varphi \mathrel{\approx\!\!\!\rightarrow}_I \lambda Sys.\ i\textit{-formula}(\varphi, I) \wedge \\ & \forall es \in Sys.\ \forall e @ i \in es.\ (kind(e) = local(a)) \Rightarrow I(\varphi)(Sys, state\ before\ e)\end{aligned}$$

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

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-affectK**, **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**:

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

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

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

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 $es \in Sys$):

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

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, $i\text{-}\mathit{formula}(\varphi, I)$ holds. The second conjunct is equivalent to

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

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 [18] 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 $Y_i^{kb} =_{\text{def}} \Box(\neg K_{1-i}(x_i = i))$ for $i = 0, 1$, where $x_i \in X_i$, and let $I = \emptyset$. Let Pg_i, $i = 0, 1$ 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

$$@i\ \mathbf{initially}\ x_i = i\ \oplus\ @i\ \mathbf{only\ events\ in}\ \emptyset\ \mathbf{affect}\ x_i.$$

Since Pg_i places no constraints on x_{1-i}, is straightforward to prove that $Pg_i \models\!\!\approx_I\ Y_{1-i}^{kb}$, for $i = 0, 1$. 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 $i = 0, 1$, so $Pg_0 \oplus Pg_1 \models\!\!\approx_I\ \neg Y_0^{kb} \wedge \neg Y_1^{kb}$. ∎

3.3 Examples

In this section, we give some examples of programs in our framework. A few simple programs are given in Section 3.3.1, while a more complex program is discussed in Section 3.3.2.

3.3.1 Simple examples Suppose that a sender S wants to send the value of a bit to a receiver R, and that this value does not change over time. This can be easily modeled in our framework by requiring the sender S to follow this program:

$$@S \textbf{ initially } (x_S = 0) \vee (x_S = 1) \oplus$$
$$@S \textbf{ only events in } [\,] \textbf{ affect } x_S,$$

where x_S is a variable local to S. The first clause says that the initial value of x_S is either 0 or 1, while the second clause says that the value of x_S does not change.

Call this program Pg^0_S. The goal is for the receiver R to eventually know the (value of the) bit x_S. We write this specification simply as $Y^{kb} =_{\text{def}} \Diamond K_R(x_S)$, where $K_R(x_S)$ is an abbreviation for $K_R(x_S = 0) \vee K_R(x_S = 1)$. Intuitively, whether this specification is satisfiable or not depends on the assumptions made regarding the communication between S and R, that is, regarding the links l_{SR} and l_{RS}, and on whether agents forget facts they once knew. For simplicity, we assume that agents have perfect recall. Among other things, this implies that if R knows the bit at some point in time, since the bit does not change its value, R will know the the value of the bit at all later times. Suppose we further assume that communication on l_{SR} is reliable: all messages sent on l_{SR} are guaranteed to be eventually received by R. It is then not difficult to see that Y^{kb} is achieved if the sender continues to send the bit to R as long as he does not know that R knows the bit. For if at some point in time S knows that R knows the bit, then R knows the bit, and will subsequently always know it; and if S does not know that R knows the bit, then S will send the value of the bit and R will eventually receive it. We can model this in the framework by assuming that S follows the program

$$\begin{array}{lll} Pg^0_S & \oplus & @S\ kind = local(a_S) \textbf{ only if } \neg K_S(K_R(x_S)) \\ & \oplus & @S \textbf{ if } kind = local(a_S) \textbf{ then } msg(l_{SR}) := x_S. \end{array}$$

The role of R so far has been passive. If the communication on l_{RS} is also reliable, we can ensure that S sends fewer messages by having R sending some token to S as soon as he receives the bit. To reason at a more abstract level, we can ensure that R sends a token to S as soon as he knows the bit. This is modeled by having R follow the program:

$$@R\ kind = local(a_R) \textbf{ only if } K_R(x_S) \oplus$$
$$@S \textbf{ if } kind = local(a_R) \textbf{ then } msg(l_{RS}) := token,$$

where $token$ is an arbitrary constant. In this program, a_R is the action of R sending a token to S.

With this program, R continues to send the token once he learns the bit. We can minimize communication further by having R send the token only if he does not know that S knows that he knows the bit:

$$@R\ kind = local(a_R) \textbf{ only if } K_R(x_S) \wedge \neg K_R(K_S K_R(x_S)) \oplus$$
$$@S \textbf{ if } kind = local(a_R) \textbf{ then } msg(l_{RS}) := token.$$

3.3.2 A knowledge-based specification and program for fairness Recall from Section 2.4 that the specification $FairSend(l) \Rightarrow Fair_I(\varphi, t, l)$ is satisfied by the program $Fair\text{-}Pg(\varphi,t,l,a)$, 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

$$Fair_I^{kb}(\varphi, t, l) =_{\text{def}} \lambda Sys.\, \forall es \in Sys.\, Fair_{I(Sys)}(\varphi^t, t, l)(es),$$

that is

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

For example, $Fair_I^{kb}(K_i\varphi, t, l)$ 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 $FairSend(l)$ simply associates with each system Sys the specification $FairSend(l)$; that is, $FairSend^{kb}(l)$ is just $\lambda Sys.\, \forall es \in Sys.FairSend(l)(es)$.

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*

$$Fair\text{-}Pg(\varphi,t,l,a) \models\!\!\approx_I\ FairSend^{kb}(l) \Rightarrow Fair_I^{kb}(\varphi, t, l).$$

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*

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

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 stronger (non-knowledge-based) predicates that imply the knowledge preconditions in Pg^{kb}.

We illustrate this methodology by applying it to a problem 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 [20] give two knowledge-based programs that solve the STP, and show that a number of standard programs in the literature, like Stenning's [27] protocol, the alternating-bit protocol [4], and Aho, Ullman and Yannakakis's algorithms [1], 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 [20], 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 $FairSend(l)$ holds for all links l.

The safety and liveness conditions for STP are run-based specifications. As argued by Fagin et al. [19], 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

$$\varphi^{kb}_{stp} =_{\text{def}} \forall n \Diamond K_R(X(n)).$$

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:

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

We abbreviate the formula $\forall k < n.\ K_S K_R(X(k)) \wedge \neg K_S K_R(X(n))$ as $K_S K_R(X[0..n))$.

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 $X(n)$:

$$\varphi_S =_{\text{def}} \exists n.\ K_S K_R(X[0..n)).$$

Let t_S be the term $\langle \mathbf{c}_S, X(\mathbf{c}_S)\rangle$.[7] Let l_{SR} denote the communication link from S to R. Now consider the knowledge-based specification $Fair_I^{kb}(\varphi_S, t_S, l_{SR})$. $Fair_I^{kb}(\varphi_S, t_S, l_{SR})$ 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 $\mathbf{c}_R$ 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:

$$\exists n.\ \neg K_R X(n) \Rightarrow (\neg K_R X(\mathbf{c}_R) \wedge \forall k < \mathbf{c}_R.\ K_R X(k)).$$

We abbreviate $\forall k < n.\ K_R(X(k)) \wedge \neg K_R(X(n))$ simply as $K_R(X[0..n))$. We take φ_R to be the knowledge-based formula

$$\varphi_R =_{\text{def}} \exists n.\ K_R(X[0..n)),$$

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. $Fair_I^{kb}(\varphi_R, t_R, l_{RS})$ 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 Nuprl system to verify our informal claim that we have refined the initial specification φ_{stp}^{kb}. That is, the Nuprl system can prove

$$\begin{array}{l}(Fair_I^{kb}(\varphi_S, t_S, l_{SR}) \wedge Fair_I^{kb}(\varphi_R, \mathbf{c}_R, l_{RS}) \wedge \\ \quad ((\exists n.\ \neg K_S K_R X(n)) \Rightarrow K_S K_R X[0..\mathbf{c}_S)) \wedge \\ \quad ((\exists n.\ \neg K_R X(n)) \Rightarrow K_R X[0..\mathbf{c}_R))) \Rightarrow \varphi_{stp}^{kb}.\end{array}$$

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 $Fair_I^{kb}(\varphi_S, t_S, l_{SR}) \wedge Fair_I^{kb}(\varphi_R, \mathbf{c}_R, l_{RS})$ is satisfied by the combination of two simple knowledge-based programs, assuming that message communication on links l_{SR}

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

and l_{RS} satisfies the strong fairness conditions $FairSend^{kb}(l_{SR})$ and $FairSend^{kb}(l_{RS})$. That is, for any two distinct actions a_S and a_R, the following is true:

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

As explained in Section 2.4, $\mathit{FairSend}^{kb}(l_{SR}) \wedge \mathit{FairSend}^{kb}(l_{RS})$ 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:

$$\begin{array}{l} @S\ \mathbf{initially}\ \Box\,((\exists n.\ \neg K_S K_R X(n)) \Rightarrow K_S K_R X[0..\mathbf{c}_S)) \models\!\!\!\sim_I \\ \quad (\exists n.\ \neg K_S K_R X(n)) \Rightarrow K_S K_R X[0..\mathbf{c}_S), \end{array}$$

$$\begin{array}{l} @R\ \mathbf{initially}\ \Box\,((\exists n.\ \neg K_R X(n)) \Rightarrow K_R X[0..\mathbf{c}_R)) \models\!\!\!\sim_I \\ \quad (\exists n.\ \neg K_R X(n)) \Rightarrow K_R X[0..\mathbf{c}_R). \end{array}$$

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

$$\begin{array}{l} Pg_S^{kb}(\varphi_S, t_S, l_{SR}, a_S) =_{\mathrm{def}} \mathit{Fair\text{-}Pg}(\varphi_S, t_S, l_{SR}, a_S) \oplus \\ \qquad @S\ \mathbf{initially}\ \Box\,((\exists n.\ \neg K_S K_R X(n)) \Rightarrow K_S K_R X[0..\mathbf{c}_S)), \end{array}$$

$$\begin{array}{l} Pg_R^{kb}(\varphi_R, \mathbf{c}_R, l_{RS}, a_R)) =_{\mathrm{def}} \mathit{Fair\text{-}Pg}(\varphi_R, \mathbf{c}_R, l_{RS}, a_R) \oplus \\ \qquad @R\ \mathbf{initially}\ \Box\,((\exists n.\ \neg K_R X(n)) \Rightarrow K_R X[0..\mathbf{c}_R)). \end{array}$$

From the definition of $\mathit{Fair\text{-}Pg}(\varphi_R, \mathbf{c}_R, l_{RS}, a_R)$ in Section 2.4, it follows that $Pg_S^{kb}(\varphi_S, t_S, l_{SR}, a_S)$ is the following composition:

$$\begin{array}{l} @S\ \mathbf{initially}\ \Box\,((\exists n.\ \neg K_S K_R X(n)) \Rightarrow K_S K_R X[0..\mathbf{c}_S)) \oplus \\ @S\ \mathit{kind} = \mathit{local}(a_S)\ \mathbf{only\ if}\ \exists n.\ K_S K_R(X[0..n)) \oplus \\ @S\ \mathbf{if}\ \mathit{kind} = \mathit{local}(a_S)\ \mathbf{then\ msg}(l_{SR})\ := t_S \oplus \\ @S\ \mathbf{only\ events\ in}\ [a_S]\ \mathbf{affect\ msg}(l_{SR}) \oplus \\ @S\ \mathbf{if\ necessarily}\ \exists n.\ K_S K_R(X[0..n))\ \mathbf{then\ i.o.}\ \mathit{kind} = \mathit{local}(a_S). \end{array}$$

Using the program notation of Fagin et al. [18], $Pg_S^{kb}(\varphi_S, t_S, l_{SR}, a_S)$ is essentially semantically equivalent to the following collection of programs, one for each value n:

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

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

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

Thus, the derived program is essentially one of the knowledge-based programs considered by Halpern and Zuck [20]. 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 $Fair^{kb}_I(\varphi, t, l)$ 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 [20], 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 $\tilde{\varphi}^{kb}_S$ is a formula that has a free variable m, and is guaranteed to be an S-formula in all interpretations I. Roughly speaking, we can think of $\tilde{\varphi}^{kb}_S$ as corresponding to $K_S K_R(X(m))$.

Let φ^{kb}_S be an abbreviation of

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

Thus, φ^{kb}_S is the analogue of φ_S in Section 4.1. Similarly, suppose that $\tilde{\varphi}^{kb}_R$ is a formula that has a free variable m, and is guaranteed to be an R-formula in all interpretations I; let φ^{kb}_R be an abbreviation of

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

We can think of $\tilde{\varphi}_R^{kb}$ as corresponding to $K_R(X(m))$.

We also use constants $\tilde{\mathbf{c}}_S$ and $\tilde{\mathbf{c}}_R$ that are analogues of $\mathbf{c}_S$, $\mathbf{c}_R$; $\tilde{\varphi}_S^{kb}$ 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 $(\exists n.\ \neg\tilde{\varphi}_S^{kb}[m/n]) \Rightarrow (\forall k < \mathbf{c}_S.\ \tilde{\varphi}_S^{kb}[m/k] \wedge \neg\tilde{\varphi}_S^{kb}[m/\mathbf{c}_S])$ 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 $(\exists n.\ \neg\tilde{\varphi}_R^{kb}[m/n]) \Rightarrow (\forall k < \mathbf{c}_R.\ \tilde{\varphi}_R^{kb}[m/k] \wedge \neg\tilde{\varphi}_R^{kb}[m/\mathbf{c}_R])$ to be true).

Let $\tilde{\varphi}_{stp}^{kb}(\tilde{\varphi}_R^{kb})$ be the specification that results by using $\tilde{\varphi}_R^{kb}[m/n]$ instead of $K_R(X(n))$ in φ_{stp}^{kb}:

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

We prove the goal $\tilde{\varphi}_{stp}^{kb}(\tilde{\varphi}_R^{kb})$ 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 $\tilde{\varphi}_R^{kb}$, $\varphi^{kb}(\tilde{\varphi}_R^{kb})$ 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.[8]

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, $I(\forall k \le n.\ \tilde{\varphi}_R^{kb[m/k]})(Sys, state\ before\ send(e_S))$ 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 $\tilde{\varphi}_S^{kb}$

[8] The Nuprl lemma that corresponds to this result can be viewed at http://www.cs.cornell.edu/info/projects/nuprl/fdlcontent/p0_963683_/send-minimal-realizable.html. For ease of exposition, we have simplified and modified some Nuprl notation in our presentation in this paper. The differences between the Nuprl lemma and the result of the paper are discussed at http://www.cs.cornell.home/halpern/papers/synthesis-appendix.pdf.

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

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

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:

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

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

$$\psi^{kb}(\tilde{\varphi}_S^{kb}, \tilde{\varphi}_R^{kb}, \tilde{t}_S, \tilde{\mathbf{c}}_R, l_{SR}, l_{RS}) \wedge \\ Fair_I^{kb}(\varphi_S^{kb}, \tilde{t}_S, l_{SR}) \wedge Fair_I^{kb}(\varphi_R^{kb}, \tilde{\mathbf{c}}_R, l_{RS}) \wedge \\ ((\exists n.\ \neg\tilde{\varphi}_S^{kb}[m/n]) \Rightarrow (\forall k < n.\ \tilde{\varphi}_S^{kb}[m/k] \wedge \neg\tilde{\varphi}_S^{kb}[m/\mathbf{c}_S])) \wedge \\ ((\exists n.\ \neg\tilde{\varphi}_R^{kb}[m/n]) \Rightarrow (\forall k < n.\ \tilde{\varphi}_R^{kb}[m/k] \wedge \neg\tilde{\varphi}_R^{kb}[m/\mathbf{c}_R])) \\ \Rightarrow \varphi^{kb}(\tilde{\varphi}_R^{kb}).$$

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

$$Pg_S^{kb}(\varphi_S^{kb}, \tilde{t}_S, l_{SR}, a_S) \oplus Pg_R^{kb}(\varphi_R^{kb}, \tilde{\mathbf{c}}_R, l_{RS}, a_R) \mid\!\approx_I \\ \psi^{kb}(\tilde{\varphi}_S^{kb}, \tilde{\varphi}_R^{kb}, \tilde{t}_S, \tilde{\mathbf{c}}_R, l_{SR}, l_{RS}) \wedge FairSend(l_{RS}) \Rightarrow \varphi_{stp}^{kb}(\tilde{\varphi}_R^{kb}),$$

where

$$Pg_S^{kb}(\varphi_S^{kb}, \tilde{t}_S, l_{SR}, a_S) =_{\text{def}} \\ Fair\text{-}Pg(\varphi_S^{kb}, \tilde{t}_S, l_{SR}, a_S) \oplus \\ @S\ \mathbf{initially}\ \Box((\exists n.\ \neg\tilde{\varphi}_S^{kb}[m/n]) \Rightarrow (\forall k < \mathbf{c}_S.\ \tilde{\varphi}_S^{kb}[m/k] \wedge \neg\tilde{\varphi}_S^{kb}[m/\mathbf{c}_S])),$$

$$Pg_R^{kb}(\varphi_R^{kb}, \tilde{t}_R, l_{RS}, a_R)) =_{\text{def}} \\ Fair\text{-}Pg(\varphi_R^{kb}, \tilde{t}_R, l_{RS}, a_R) \oplus \\ @R\ \mathbf{initially}\ \Box((\exists n.\ \neg\tilde{\varphi}_R^{kb}[m/n]) \Rightarrow (\forall k < \mathbf{c}_R.\ \tilde{\varphi}_R^{kb}[m/k] \wedge \neg\tilde{\varphi}_R^{kb}[m/\mathbf{c}_R])).$$

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

The formulas φ_S and φ_R 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.

4.2.1 Stenning's protocol In the next two sections, we show that by making relatively straightforward choices for the formulas $\tilde{\varphi}_S^{kb}$ and $\tilde{\varphi}_R^{kb}$ and terms $\tilde{t}_S$ and $\tilde{t}_R$, we can derive two well-known solutions for STP, Stenning's protocol [27] and an infinite-state variant of the alternating-bit protocol [4]. We start with Stenning's protocol.

In Stenning's protocol, the sender transmit the bits on the tape in order to the receiver. The sender S keeps track of the position i_S of the bit in the sequence that he will next send to R, while the receiver R keeps track of the first position i_R in the sequence for which he has not received the corresponding bit. Initially, both i_S and i_R are set to 0. S always sends R message of the form $\langle X(i_S), i_S \rangle$. When R receives a message from S whose second component is i_R, then R increments i_R and acknowledges the messages by sending S the message i_R; R disregards other messages. If S receives i_R and $i_R > i_S$ (it is easy to see that this can happen only if $i_R = i_S + 1$), then S increments i_S; S disregards all other messages. Note that it is straightforward to write clauses that ensure that i_S and i_R indeed have these properties. The clauses should say that initially both i_S and i_R are set to 0, that i_S only changes when S receives from R a message larger than i_S, and that, if infinitely often this is the case, then infinitely often i_S is incremented; similarly, the clauses should say that i_R only changes when R receives from S a message whose last component is i_R, and if infinitely often this is the case, then infinitely often i_R is increased. As apparent from this short description, all such clauses can be expressed in the message automata framework.

We can choose $\tilde{\varphi}_R^{kb}$ such that $\tilde{\varphi}_R^{kb}(m)$ holds in R's local state s_R exactly when s_R records that R has received a message containing index m (that is, $\tilde{\varphi}_R^{kb}(m) =_{\text{def}} i_R > m$), and choose $\tilde{\varphi}_S^{kb}$ such that $\tilde{\varphi}_S^{kb}(m)$ holds in S's local state s_S exactly when s_S records that S has received an index strictly greater than m (that is, $\tilde{\varphi}_S^{kb} =_{\text{def}} i_S > m$) . It is not difficult to show that $\psi^{kb}(\tilde{\varphi}_S^{kb}, \tilde{\varphi}_R^{kb}, \tilde{t}_S, \tilde{\mathbf{c}}_R, l_{SR}, l_{RS})$ holds, except that now this specification is not knowledge-based. Note that $\varphi_S^{kb}(= \exists n.\ (\forall k < n.\ \tilde{\varphi}_S^{kb}[m/k]) \wedge \neg\tilde{\varphi}_S^{kb}[m/n]) = true$ and, similarly, $\varphi_R^{kb} = true$. In addition, $\varphi_{stp}^{kb}(\tilde{\varphi}_R^{kb})$ implies φ_{stp}^{kb}, which means that, assuming message communication is fair,

$$Pg_S(\varphi_S^{kb}, \tilde{t}_S, l_{SR}, a_S) \oplus Pg_R(\varphi_R^{kb}, \tilde{t}_R, l_{RS}, a_R)$$

(together with the basic clauses ensuring that the variables i_S and i_R behave appropriately) satisfies the STP specification, as long as a_S and a_R are distinct actions. Note that the program $Pg_S(\varphi_S^{kb}, \tilde{t}_S, l_{SR}, a_S) \oplus Pg_R(\varphi_R^{kb}, \tilde{t}_R, l_{RS}, a_R)$ is realizable. We have thus extracted a program that realizes the STP specification. Moreover, we can show that this program is essentially semantically equivalent to Stenning's protocol.

The Nuprl system is semi-automatic, in the sense that the programmer indicates at each step which refinement rule to apply. Users can group a sequence of rules together into what is called a tactic. In the discussion above, we did not apply any Nurpl tactics in the derivation. However, the reader can easily check that each refinement step in the proof outlined above is either a basic refinement rule (i.e., induction, case analysis for a formula satisfying the principle of excluded middle), or an instance of the fairness specification from Section 2.4.

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

4.2.2 The alternating-bit protocol Stenning's protocol works even if messages can be dropped or duplicated, and messages can be reordered. All that is required is that communication is fair, in the sense that a message sent infinitely often is eventually received. The alternating-bit protocol also works in an environment where messages can be dropped or duplicated, but it does require that messages are received in the order in which they are sent. The advantage of making this extra assumption is that now a finite-state protocol can be used. Instead of using counters i_S and i_R to keep track of which prefix of the sequence has been received, it suffices to use a bit that alternates in value to do this.

In more detail, the sender starts by reading the first value, stores it in the variable x_S, and sends (x_S, i_S) to the receiver, where i_S is a bit initially set to 0. The receiver maintains a bit i_R, initialized to λ (a null value). Upon receiving a message (x_S, i_S) from the sender, if $i_S \neq i_R$, then the receiver sets $i_R = i_S$, writes x_S, and acknowledges (x_S, i_S) (by sending i_R to the sender); if $i_S = i_R$, then the receiver ignores the message. When the sender receives a message i_R from the receiver with $i_R = i_S$, then the sender reads the next bit in the sequence into x_S and sets i_S to $1 - i_S$; otherwise, the sender ignores the message. (Note that the values of i_S alternates between 0 and 1, hence the name of the protocol.)

Let cnt_S be a variable representing how many times the bit i_S has been flipped; similarly, let cnt_R be a variable representing how many times the bit i_R has been flipped. Let $\tilde{\varphi}_S^{kb} =_{\text{def}} (cnt_S \geq m+1)$, $\tilde{\varphi}_R^{kb} =_{\text{def}} (cnt_R \geq m+1)$, $\tilde{t}_S =_{\text{def}} \langle x_S, i_S \rangle$, and $\tilde{t}_R =_{\text{def}} i_R$. Intuitively, whenever $cnt_R \geq m+1$ holds, that is, whenever R has flipped his bit at least $m+1$ times, R knows the first $m+1$ bits in the sequence, that is, $X(0), X(1) \ldots X(m)$; similarly, whenever $cnt_S \geq m+1$ holds, that is, whenever S has flipped his bit $m+1$ times, it must be that S has received acknowledgments that R has received bits $X(0), X(1) \ldots X(m)$. (For a formal proof of this claim, see [20]. Note that the proof in [20] relies essentially on the fact that messages cannot be reordered.)

As in Stenning's protocol, we need to add some basic clauses that ensure that the variables i_S and i_R have the right properties. These clauses should say that initially i_S is set to 0, that i_S is flipped only if S receives the message i_S from R, and that, if infinitely often R receives a message from R equal to i_S, then i_S is flipped infinitely often; similarly, the clauses should say that initially i_R is set to a null value, that i_R is changed only when R receives a message from S (either equal to i_R, if i_R is not null, or not null), and that, if infinitely often this is the case, then infinitely often i_R is changed. Note that all these clauses can be easily expressed using the message automata language.

We now show that $\tilde{\varphi}_S^{kb}$, $\tilde{\varphi}_R^{kb}$, $\tilde{t}_S$ and $\tilde{t}_R$ chosen as above satisfy all the conditions identified during the derivation at the beginning of this section. We do not give a formal proof here; rather, we present enough details for the reader to have an understanding of how the proof works. It is not difficult to see that both $Stable(\tilde{\varphi}_R^{kb})$ and $Stable(\tilde{\varphi}_S^{kb})$ hold, as both cnt_S and cnt_R can never decrease, and that $Determinate(\tilde{\varphi}_R^{kb})$ and $Determinate(\tilde{\varphi}_S^{kb})$ also hold. To see that $Fair_I^{kb}(\varphi_R^{kb}, \tilde{t}_R, l_{RS})$ also holds, recall that φ_R^{kb} is defined as $\exists n.\ \forall k < n.\ \tilde{\varphi}_R^{kb}[m/k] \wedge \neg\tilde{\varphi}_R^{kb}[m/n]$, which is equivalent to $\exists n. \forall k < n.\ (cnt_R \geq k+1) \wedge (cnt_R < n+1)$, that is, $\exists n.\ cnt_R = n$, and so φ_R^{kb} is always true. By inspecting the definition of $Fair_I^{kb}(\varphi_R^{kb}, \tilde{t}_R, l_{RS})$, this implies that $Fair_I^{kb}(\varphi_R^{kb}, \tilde{t}_R, l_{RS})$ is reduced to showing that the following holds in all runs es of the alternating-bit protocol: $\exists e_R @ R \in es \wedge \forall e @ R \in es.\ \exists e' \in es.kind(e') = rcv(l_{RS}) \wedge send(e') \succeq e @ R$. In other words, we need to show that, in all runs of the alternating-bit protocol, some event occurs associated with R, and for all events associated with R, such as R receiving a message from S, there will be a subsequent message sent by R to S and received by S. This is clearly true for the alternating-bit protocol. Similarly, $Fair_I^{kb}(\varphi_S^{kb}, \tilde{t}_S, l_{SR})$ is reduced to the condition $\exists e @ S \in es \wedge \forall e @ S \in es.\ \exists e'.\ kind(e') = rcv(l_{SR}) \wedge send(e') \succeq e @ S \in es$, which basically

says that some event associated with S occurs and that, whenever S receives a message from R, there will be a subsequent message sent by S to R and received by R. Again, this is true for all runs of the alternating-bit protocol; that is, φ_S^{kb}, like φ_S^{kb}, is equivalent to the formula $true$ in all runs of the system corresponding to the alternating-bit protocol.

The formula $Implies(\tilde{\varphi}_S^{kb}, \tilde{\varphi}_R^{kb})$ is equivalent in this case to the following formula:

$$\forall n.\, \forall e_S@S \in es.\, ((cnt_S \geq n+1)\ \mathit{before}\ e_S) \Rightarrow \exists e_R \succ e_S@R \in es.\, ((cnt_R \geq n+1)\ \mathit{after}\ e_R).$$

This says that if the sender has flipped his bit at least $n+1$ times before he sends a message to R, upon receiving that message R will have flipped his bit at least $n+1$ times, as well. In fact, we can see that with the alternating-bit protocol (that is, with the enforced semantics for i_S and i_R), if S has flipped his bit exactly k times before he sends a message to R, and if that message is received by R, then, when R receives this message, either R has already flipped his bit exactly $k+1$ times and discards this message, or R has flipped his bit k times, R does not discard this message and flips his bit one more time, ensuring R will have been flipped his bit $k+1$ times after receiving this message.

The formula $Rcv(\tilde{\varphi}_S^{kb}, \tilde{\varphi}_R^{kb}, l_{RS})$ is equivalent to

$$\begin{array}{l}\forall e_S@S.\ (kind(e_S) = rcv(l_{RS})) \Rightarrow \\ \quad \forall n.((\forall k \leq n.\ ((cnt_R \geq k+1)\ \mathit{before}\ send(e_S))) \Rightarrow ((cnt_S \geq n+1)\ \mathit{after}\ e_S)).\end{array}$$

This formula basically says that if R has flipped his bit at least $n+1$ times before sending a message to S, and S receives this message, then S will have flipped his bit at least $n+1$ after seeing this message. We leave it to the reader to check that this is true for the runs of the alternating-bit protocol (again, based on the enforced semantics for i_S and i_R).

Finally, the formula $Rcv(\tilde{\varphi}_R^{kb}, \tilde{\varphi}_S^{kb}, l_{SR})$ is equivalent to

$$\begin{array}{l}\forall e_R@R.\ (kind(e_R) = rcv(l_{SR})) \Rightarrow \forall n.\ (((\forall k < n.\ cnt_S \geq k+1 \wedge cnt_S < n+1)\ \mathit{before}\ send(e_R)) \\ \quad \Rightarrow ((cnt_R \geq n+1)\ \mathit{after}\ e_R)).\end{array}$$

This formula says that if S has flipped his bit exactly n times before he sends a message to R, and R receives this message, then after receiving this message R will have flipped his bit at least $n+1$ times. This easily follows from the argument made above that $Implies(\tilde{\varphi}_S^{kb}, \tilde{\varphi}_R^{kb})$ holds. It follows that all the conditions that we identified for deriving a standard program from a knowledge-based program are satisfied. Thus, if messages are not reordered, the specification for the sequence-transmission problem is satisfied by the standard program

$$Fair\text{-}Pg^{kb}(\varphi_S^{kb}, \tilde{t}_S, l_{SR}, a_S) \oplus Fair\text{-}Pg_I^{kb}(\varphi_R^{kb}, \tilde{t}_R, l_{RS}, a_R).$$

Since, as we showed above, both φ_S^{kb} and φ_R^{kb} are always true for our particular choices of $\tilde{\varphi}_S^{kb}$, $\tilde{\varphi}_R^{kb}$, $\tilde{t}_S$, and $\tilde{t}_R$, this becomes

$$\begin{array}{l}@S\ kind = local(a_S)\ \textbf{only if}\ true \oplus \\ @S\ \textbf{if}\ kind = local(a_S)\ \textbf{then}\ msg(l_{SR}) := \langle x_S, i_S \rangle \oplus \\ @S\ \textbf{only events in}\ [a_S]\ \textbf{affect}\ msg(l_{SR}) \oplus \\ @S\ \textbf{if neccessarily}\ true\ \textbf{then i.o.}\ kind = local(a_S) \oplus \\ @R\ kind = local(a_R)\ \textbf{only if}\ true \oplus \\ @R\ \textbf{if}\ kind = local(a_R)\ \textbf{then}\ msg(l_{RS}) := i_R \oplus \\ @R\ \textbf{only events in}\ [a_R]\ \textbf{affect}\ msg(l_{RS}) \oplus \\ @R\ \textbf{if neccessarily}\ true\ \textbf{then i.o.}\ kind = local(a_R).\end{array}$$

Note that this program indeed corresponds to the alternating-bit protocol.

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 knowledge-based programs as well. We applied this methodology to the analysis of the sequence-transmission problem, and showed that the knowledge-based programs 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 knowledge-based programs that solve the STP. A feature of our approach is that the extracted standard programs are closer to the pseudocode that designers write, and can be translated into running code.

There has been work on synthesizing both standard programs and knowledge-based programs from knowledge-based specifications. In the case of synchronous systems with only one process, Van der Meyden and Vardi [29] provide a necessary and sufficient condition for a certain type of knowledge-based 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 [16, 17] 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 view our method for synthesizing programs from knowledge-based specifications as an alternative to this approach. As in the Engelhart et al. approach, the Nuprl programs that we extract are close to programs in standard 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 not computable in general, recent results [30] show that, under certain assumptions about the setting in which agents communicate, the problem is computable. 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 knowledge-based 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 knowledge-based 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 knowledge-based specifications, "good" standard tests can be more easily identified.

Acknowledgements

We would like to thank Richard Eaton from the Nuprl group for making the Nuprl lemma corresponding to our proof of the sequence-transmission problem available online.

References

[1] A. V. Aho, J. D. Ullman, A. D. Wyner, and M. Yannakakis. Bounds on the size and transmission rate of communication protocols. *Computers and Mathematics with Applications*, 8(3):205–214, 1982. This is a later version of [2].

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

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

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

[5] S. Berghofer. Program extraction in simply-typed higher-order logic. In H. Geuvers and F. Wiedijk, editors, *Types for Proofs and Programs, International Workshop, (TYPES 2002)*, LNCS, Volume 2646, pages 21–38. Springer-Verlag, 2002.

[6] M. Bickford and R. L. Constable. A causal logic of events in formalized computational type theory. Technical Report 1893, Cornell University, 2003.

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

[8] L. E. J. Brouwer. 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, 1923.

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

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

[11] R. L. Constable and M. Bickford. Formal foundations of computer security. In *NATO Science for Peace and Security Series D: Information and Communication Security*, volume 14, pages 29–52. 2008.

[12] R. L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, NJ, 1986. Available at www.nuprl.org.

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

[14] C. Cornes, J. Courant, J.-C. Filliâtre, G. P. Huet, P. Manoury, C. Paulin-Mohring, C. Muñoz, C. Murthy, C. Parent, A. Saïbi, and B. Werner. The Coq proof assistant reference manual. Technical report, INRIA-Rocquencourt, CNRS, and ENS Lyon, 1996.

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

[16] K. Engelhardt, R. van der Meyden, and Y. Moses. A program refinement framework supporting reasoning about knowledge and time. In J. Tiuryn, editor, *Proc. Foundations of Software Science and Computation Structures (FOSSACS 2000)*, pages 114–129. Springer-Verlag, Berlin/New York, 1998.

[17] K. Engelhardt, R. van der Meyden, and Y. Moses. 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*, pages 125–141. Springer-Verlag, Berlin/New York, 2001.

[18] R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. *Reasoning About Knowledge*. MIT Press, Cambridge, Mass., 1995. A slightly revised paperback version was published in 2003.

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

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

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

[22] L. Lamport. The part-time parliament. *ACM Trans. on Computer Systems*, 16(2):133–169, 1998.

[23] X. Liu, C. Kreitz, R. van Renesse, J.Hickey, M. Hayden, K. Birman, and R. Constable. Building reliable, high-performance communication systems from components. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 80–92, 1999.

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

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

[26] C. Paulin-Mohring and B. Werner. Synthesis of ML programs in the system Coq. *Journal of Symbolic Computation*, 15:607–640, 1993.

[27] M. V. Stenning. A data transfer protocol. *Comput. Networks*, 1:99–110, 1976.

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

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

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