A predicate transformer approach to knowledge and knowledge-based protocols

RC. 92. 181

Eidgenössische
Technische Hochschule
Zürich

Departement Informatik
Institut für
Computersysteme

Beverly Sanders

A Predicate Transformer Approach to Knowledge and Knowledge-based Protocols

September 1992

181

92. 10. 21

7596

ETH Zürich
Departement Informatik
Institut für Computersysteme

Authors' address:

Prof. Beverly Sanders
Institut für Computersysteme
ETH Zentrum, CH-8092 Zurich, Switzerland

e-mail: sanders@inf.ethz.ch

Abstract

It has been argued that reasoning about what processes "know" is a good way to think about distribute programs. In this paper, a predicate transformer representing knowledge is defined and used to extend a simple and modern programming theory to allow knowledge-based protocols to be expressed. This allows the mathematics of predicate transformers to be brought to bear in the theoretical study of knowledge, provides a formal notation and proof method for knowledge-based protocols, and facilitates understanding the pragmatic aspects of algorithm development using knowledge-based protocols.

1 Introduction

Most of the work on knowledge in distributed systems has been done in the context of the basic semantic model which associates a program with the set of runs that it can generate. Although this approach is very general, it is not usually very convenient for working with specific programs since there is a large gap between the description of the program in a convenient notation and the set of runs that it generates. Programming theories bridge this gap and provide an alternative to operational reasoning via a proof theory that allows one to reason about the texts of programs and draw conclusions about the set of runs.

In this paper, knowledge is defined as a predicate transformer. Process i "knows" p, denoted K_ip, is a function from a predicate p to predicates which depends only on variables accessible to process i, and holds in exactly those states where process i "knows" p. Knowledge-based protocols are an extension to normal protocols where where knowledge predicates are allowed to appear in guards of statements in programs. The goal is to formally capture the intuition of protocol designers who often describe the action of a distributed program with remarks such as "when process i knows that process j has unlocked some resource, it sends a grant message to the next waiting process."

Using the predicate transformer, knowledge and knowledge-based protocols are defined in the context of a slightly modified version of Chandy and Misra's programming theory, UNITY [CM88]. The logic has been slightly modified [San90, San91] and is extended here with a minimal notion of a process. There are several benefits from this approach. First of all, one can reason formally about the knowledge obtained by processes in standard programs and also knowledge-based protocols (the terms protocols, algorithms and programs will be used interchangeably) without having to construct the set of runs generated by the program. (Techniques for reasoning about knowledge obtained in CSP programs, but not knowledge-based protocols, were given in [KT86]) Second, the theory of predicate transformers [DS90] suggests questions which yield new insights about knowledge and knowledge-based protocols. In particular, it will be shown that knowledge-based protocols do not enjoy certain monotonicity properties that are usually taken for granted for programs. For example, properties which hold for some knowledge-based protocol may not hold for the same protocol with stronger initial conditions. Finally, since our method is an extended version of a complete programming theory, carrying out algorithm development utilizing knowledge-based protocols in this framework allows a careful comparison between the formal derivation with and without knowledge. This is done for an example that has previously appeared in the literature [HZ87]. The most important lesson from the exercise is that knowledge-based protocols, as a high level description, may be more restrictive than necessary, and thus make the job of constructing a formal proof harder instead of easier.

2 Predicate transformers and the strongest invariant

In this section, a very brief introduction to the style of predicate calculus used in the paper and to the theory of predicate transformers is given. More details are found in [DS90]. The strongest invariant is also defined.

A *predicate* is a Boolean valued total function on the state space of a program. Predicates are semantic objects; we are not concerned with whether or not they can be expressed in some particular language. The operators $\equiv, \Rightarrow, \Leftarrow, \wedge, \vee$, and $\neg$ are applied pointwise and yield another predicate. This is usual for $\wedge, \vee$, and $\neg$ but unusual for $\equiv, \Rightarrow$, and $\Leftarrow$; for example, for predicates p and q, $p \equiv q$ is a predicate which is true at points where p and q have the same value, and false where they differ. The *everywhere* operator, denoted by surrounding predicates with square brackets, denotes universal quantification over the state-space. Thus $[p]$ is true if p is true at every point. p and q are equal if $[p \equiv q]$. We say that q is *weaker* than p, and p is *stronger* than q if $[p \Rightarrow q]$. Function application is denoted with a ".", as in $f.p$

A *predicate transformer* is a function from predicates to predicates. An important property of predicate transformers that will appear frequently in the sequel is monotonicity. A predicate transformer f is *monotonic* if for all predicates p and q: $[p \Rightarrow q] \Rightarrow [f.p \Rightarrow f.q]$. Other junctivity properties of interest are or-continuity, universal conjunctivity, and finite disjunctivity.

A predicate transformer f is *or-continuous* if for all non-empty bags of predicates W which can be arranged in a monotonic sequence, $[(\exists v : v \in W : f.v) \equiv f.(\exists v : v \in W : v)]$. f is ***universally conjunctive*** if for all bags of predicates W, $[(\forall v : v \in W : f.v) \equiv f.(\forall v : v \in W : v)$. f is ***finitely disjunctive*** if for all predicates p and $q, [f.p \vee f.q \equiv f.(p \vee q)]$ And-continuity, finite conjunctivity, and universal disjunctivity are defined analogously.

The predicate transformers most often used in program semantics the weakest precondition, wp, and weakest liberal precondition, wlp. Here, $wp.F.p$ is the weakest predicate so that program F starting in a state satisfying $wp.F.p$ will terminate in a state satisfying p, while $wlp.F.p$ is the weakest predicate so that program F starting in a state satisfying $wlp.F.p$ will either terminate in a state satisfying p, or fail to terminate. wp will be used in the UNITY proof theory which will be discussed later in section 5.

Here, the basic model is a non-terminating state transition system with interleaving semantics rather than a termination sequential program. For a given program, $SP.p$ (strongest postcondition) is defined to be the strongest predicate such that if the program is in a state satisfying p, then the next transition will result in a state satisfying $SP.p$. We assume that SP is total, monotonic, and or-continuous. These properties hold for most programs of interest, and in particular, they hold for all programs in UNITY, which will be discussed in section 5. If $SP.p \Rightarrow p$, then p is stable, i.e. once satisfied, it will remain satisfied. $sst.p$, the strongest stable predicate weaker than p is defined using SP as follows:

$$sst.p \overset{def}{\equiv} \text{strongest } x : [SP.x \Rightarrow x] \wedge [p \Rightarrow x] \qquad (1)$$

For all constant programs, with monotonic, and or-continuous SP, and all predicates p and q, the following hold:

$sst.p$ exists and is unique. (2)

$$sst.p = (\exists i : 0 \leq i : f^i.\text{false}) \quad \text{where} \quad f.x = SP.x \vee p \qquad (3)$$

sst is monotonic (4)

Proofs are given in [San91].

A program has the property **invariant** p if p holds initially, and remains true during the entire program execution. The *strongest invariant*, abbreviated SI, is given by $sst.init$ where $init$ is the predicate describing allowed initial states. Thus SI is a predicate characterizing the reachable states of the program. Invariant properties are defined using SI as follows:

$$[SI \Rightarrow p] \overset{def}{\equiv} \textbf{invariant } p \tag{5}$$

3 Knowledge

In this section, we use SI to define a predicate transformer for knowledge. Since a process can only access its own variables, any predicate which holds exactly when process i "knows" p, should be independent of the variables not accessible to process i. (A predicate is independent of a variable if it has the same value in any two states that differ only in the value of that variable.) As a preliminary step, we define a predicate transformer $wcyl$ (weakest cylinder) where $wcyl.V.p$ is the weakest predicate as strong as p which depends only on the variables in V.

$$wcyl.V.p \overset{def}{\equiv} (\forall \overline{V} :: p) \tag{6}$$

where $\overline{V}$ is the compliment of V in the set of program variables.

Several properties of $wcyl$ are given in (7–12) below:

$$[wcyl.V.p \Rightarrow p]. \tag{7}$$

$wcyl$ exists and is monotonic in both arguments. (8)

If p depends only on variables in V, then

$$p \equiv wcyl.V.p. \tag{9}$$

If $[q \Rightarrow p]$ and q depends only on variables in V, then

$$[q \Rightarrow wcyl.V.p]. \tag{10}$$

$wcyl$ is universally conjunctive. (11)

$wcyl$ is not disjunctive. (12)

The proofs of (8–12) follow easily from the definition and properties of universal quantification. A counterexample to demonstrate (12) is nevertheless illustrative. Consider a state space constructed out of two integer variables x and y. Then $wcyl.x.(x > 0 \wedge y > 0) =$ false and $wcyl.x.(x > 0 \wedge y \leq 0) =$ false while $wcyl.x.(x > 0) = x > 0$.

Now we are in a position to give a predicate transformer $K_i p$ which holds whenever process i "knows" p. Informally, one says that a process knows a fact in some state if that fact is true in all possible global states which are indistinguishable to the process. The possible global states are given by SI. Thus we obtain:

$$K_ip \overset{def}{\equiv} wcyl.vars_i.(SI \Rightarrow p)$$

With this definition, K_ip has the correct value on all reachable states. However, we have found it technically convenient to define K_ip so that K_ip always has the value p on unreachable states. This slightly more complicated definition is given below:

$$K_ip \overset{def}{\equiv} p \wedge (wcyl.vars_i.(SI \Rightarrow p) \vee \neg SI) \tag{13}$$

Equations (14–18) hold for K, and imply the axioms and inference rule of the modal logic S5:

$$[K_ip \Rightarrow p] \tag{14}$$

$$[(K_ip \wedge K_i(p \Rightarrow q)) \Rightarrow K_iq] \tag{15}$$

$$[K_ip \equiv K_iK_ip] \tag{16}$$

$$[\neg K_ip \equiv K_i\neg K_ip] \tag{17}$$

$$[p] \Rightarrow [K_ip] \tag{18}$$

Several properties of K follow from similar properties of $wcyl$:

K_ip is monotonic with respect to p (19)

K_ip is anti-monotonic with respect to SI (20)

K_i is universally conjunctive (21)

K_i is not disjunctive (22)

The following two results illustrate the close relationship between K and invariants.

$$\textbf{invariant } p \equiv \textbf{invariant } K_ip \tag{23}$$

If q depends only on variable accessible to process i, then

$$\textbf{invariant } (q \Rightarrow p) \equiv \textbf{invariant } (q \Rightarrow K_ip) \tag{24}$$

This result, together with 14 give a characterization of K_ip as the weakest predicate, depending only on i's variables, which implies p.

Proof of $\Rightarrow$:

$$
\begin{array}{ll}
 & \textbf{invariant } (q \Rightarrow p) \\
= & \{(5)\} \\
 & SI \Rightarrow (q \Rightarrow p) \\
= & \\
 & q \Rightarrow (SI \Rightarrow p) \wedge (SI \wedge q \Rightarrow p) \\
\Rightarrow & \{\text{from } (10)\} \\
 & q \Rightarrow wcyl.vars_i.(SI \Rightarrow p) \wedge (SI \wedge q \Rightarrow p) \\
\Rightarrow & \\
 & SI \wedge q \Rightarrow (wcyl.vars_i.(SI \Rightarrow p) \wedge p) \\
\Rightarrow & \{\text{weaken r.h.s and use } (13)\} \\
 & q \wedge SI \Rightarrow K_i p \\
= & \{\text{from } (5)\} \\
 & \textbf{invariant } (q \Rightarrow K_i p)
\end{array}
$$

Proof of $\Leftarrow$:

$$
\begin{array}{ll}
 & \textbf{invariant } (q \Rightarrow K_i p) \\
= & \{\text{from } (5)\} \\
 & SI \Rightarrow (q \Rightarrow K_i p) \\
\Rightarrow & \{\text{from } (14)\} \\
 & SI \Rightarrow (q \Rightarrow p) \\
= & \{\text{from } (5)\} \\
 & \textbf{invariant } (q \Rightarrow p)
\end{array}
$$

The above proof[1] illustrates an important advantage of a predicate transformer approach to program semantics. The proof of this metatheorem was carried out entirely with straightforward calculations in the predicate calculus.

In [HM90], a model of distributed systems is given along with semantic definitions for knowledge. In that paper, the semantics of a distributed program is given by a set of runs where a run is a sequence of actions, including the time each action occurred. A point is a pair consisting of a run and a time t. A process's local history at time t is the subsequence of events that the process has observed up until t. Under a view-based knowledge interpretation, knowledge is ascribed to processes by defining a view, which is a function of the local histories, and saying that a processor "knows"

[1]This result is apparently not as obvious as it seems. An expert reviewer of an earlier version of this paper (without the proof) claimed it was incorrect

a fact at some point if the fact holds (according to some assignment π which associates a truth value for all "ground facts" with every point in the set of runs) in all possible points that the processor cannot distinguish from the given one, i.e. in which the process has the same view. The notion of a view function is quite general, ranging from allowing processes to use their entire local histories to distinguish between points, to not being able to distinguish between points at all.

In this paper, the semantics of a distributed program is also given by a set of runs, however, a run is a sequence of global states. The processes view is always the projection of the current global state onto the variables accessible to the process. There is considerable freedom in determining what is included in the state and which variables are accessible to a process; time may or may not be included, and any function of the processes history may be included in the state by explicitly including appropriate history variables. The ground facts are the predicates on the state space. As in [HM90], a process "knows" a fact at some point if the fact holds in all possible points that the processor cannot distinguish from the given one.

Although the definitions of knowledge given here are slightly less general than those in [HM90], they are adequate for many useful problems and because certain aspects are fixed (e.g. the particular view is fixed and the set of ground facts is always the set of predicates on the state space) they are simpler and have useful mathematical properties. Also, knowledge can be incorporated into a proof theory which allows non-operational reasoning. The approach can easily be extended to include other variants of knowledge, such as common knowledge [HM90].

4 Knowledge-based protocols

The previous discussion was concerned with knowledge obtained by processes executing standard programs. A seemingly natural extension is to allow processes to explicitly test for knowledge. In [HF89] and the several references cited therein, such programs, called knowledge-based protocols, may include statements of the form "if process i knows ϕ_1 then $action_1$ elsif process i knows ϕ_2 then $action_2$ ...". In this section, we will introduce this generalization into our framework. First, we examine the consequences on the theoretical results of the previous section. Then, we will introduce UNITY and the extensions necessary to allow knowledge-based protocols to be expressed. As a result, we will obtain a well-defined notation in which knowledge-based protocols can be formally specified and expressed. The UNITY proof theory is inherited, plus additional results developed in the previous section. The use of extended UNITY will be illustrated with an example which will allow us to gain some insights into the pragmatic aspects of using knowledge-based protocols as a tool for the derivation of distributed algorithms.

Eventually, we will arrive at a programming notation that allows statements with knowledge predicates to appear in guards. In section 2, (constant) programs were assumed, for which SP could be calculated. As a simpleminded and degenerate example, which will nevertheless illustrate the point, consider a program with a single statement as follows:

$$x := x + 1 \text{ if } b$$

Then

$$SP.p \;=\; (\neg b \wedge p) \vee (x := x - 1).(b \wedge p)$$

where $(x := E).p$ means substitute E for each occurrence of x in p.[2] If b is a constant predicate, then this can be computed. However, if b is, say $K_i p$, then we see that b depends on SI, which depends on SP, which depends on b. Thus SP is now a function of SI. The new definitions of SI is obtained by explicitly including the dependence of SP on SI in (1).

$$sst.p \overset{def}{\equiv} \text{ strongest } x : [(SP.SI).x \Rightarrow x] \wedge [p \Rightarrow x]$$

Alternatively, restricting ourselves to the case where $p = init$, this can be rewritten as

$$SI \overset{def}{\equiv} \text{ strongest } x : [\widehat{SP}.x \Rightarrow x] \wedge [init \Rightarrow x] \qquad (25)$$

where $widehatSP$ is obtained by partially evaluating $SP.SI.x$ with $SI = x$. Given a solution to (25), the knowledge-based protocol can be converted to a standard protocol by replacing all the knowledge predicates with the corresponding standard predicate obtained

[2]Properties and definitions of the strongest postcondition are discussed in detail in [DS90].

```
       var $shared, x : boolean$
 processes $V_0 = \{shared\}, V_1 = \{shared, x\}$
      init $\neg shared \wedge \neg x$
    assign
           $shared :=$ true if $K_0 \neg x$
         ▯
           $x, shared :=$ true, false if $shared$
```

Figure 1: Knowledge-based protocol with no solution.

using SI. Equation (25) now has the same form as equation(1). Unfortunately, (2–4) are no longer guatanteed, since $\widehat{SP}.x$ is not monotonic.

A first consequence of this is that SI need not exist. As a simple example, consider the knowledge-based protocol shown in figure 1. The notation will be explained in more detail in section 5. Briefly, process 0, which can access *shared*, executes the first statement, process 1, which can access both *shared* and x, executes the second. At any point, a non-deterministic choice is made between the statements. If the guard of the chosen statement does not hold, the execution of the statement has no effect. There is no possible choice for SI for which the resulting $K_i \neg x$ will result in a standard protocol which actually yields this strongest invariant. The possibility that there is no standard protocol consistent with a knowledge-based protocol is well known. The predicate transformer approach makes it clear that, technically, lack of monotonicity of $\widehat{SP}$ is the culprit.

In the previous discussion, we have seen that because $\widehat{SP}$ is not monotonic, the knowledge-based protocol may not be well-posed. However, non-monotonicity also shows up in another guise. Even when it exists, the strongest invariant of a knowledge-based protocol need not be monotonic with respect to the initial conditions. A counter-example, shown in figure 2 suffices to establish the claim.

When the $init = \neg y$, then the strongest invariant is $\neg y$. When $init = \neg y \wedge x$, then the strongest invariant is x. In addition to the invariant properties, in the first case, the program satisfies the property true $\mapsto z$ (i.e. eventually z will hold), in the second, with a stronger initial condition, the program does not.

```
      var $x, y, z$ : $boolean$
processes $V_0 = \{y\}, V_1 = \{z\}$
   assign
            $y :=$ true if $K_0.x$
          $\Box$
            $z :=$ true if $K_1.(\neg y)$
```

Figure 2: *SI* is not monotonic with respect to initial conditions.

The ramifications of this result are that neither safety nor liveness properties of knowledge-based protocols are necessarily preserved when the initial conditions are strengthened, thus violating one of the most intuitive and fundamental properties of standard programs. This important result does not seem to have been previously recognized.

The semantics of knowledge-based protocols as defined above is not identical to that commonly found in the literature. We consider the knowledge-based protocol to correspond to the set of runs generated by a standard protocol with the knowledge predicates replaced with corresponding standard predicates. In, for example, [HF89, HZ87, Hal91], a knowledge-based protocol is formally considered to be a function from local states and an interpreted system (set of runs) to actions. Thus, a knowledge-based protocol corresponds to many different systems. The advantage of this is that one can use the same knowledge-based protocol in different environments. For example, a knowledge-based protocol can describe the behavior of a protocol for reliable communication over communication channels with various types of failures. The runs of one interpreted system would only include actions that involve a particular type of failure, the runs of another, a different type of failure, etc. In our approach, we take the "maximal" system where no actions are arbitrarily ruled out and where initial state allowed by the initial condition is considered possible. This corresponds to the *canonical* system described in [HF89]. It is interesting to note that often, a knowledge based protocol can be specified for different environments, with the "selected" behavior encoded in the initial condition. Then strengthening the initial condition corresponds to execution of the protocol in a more predictable environment. From the nonmonotonicity result above, properties that are satisfied in one environment may not be preserved if the environment becomes more predictable.

5 Knowledge-based protocols in UNITY

In this section, we give a brief introduction to UNITY. Then we will extend the programming notation to allow knowledge-based protocols to be expressed. The necessary semantic foundations have been developed in the previous section. The result is that we will have a rich proof theory to use when reasoning about knowledge-based protocols. This is in contrast to previous work in the literature where the psuedocode definitions of knowledge-based protocols are only informal descriptions and both formal semantics and correctness proofs are constructed in an *ad hoc* manner.

UNITY is a programming theory which comprises a specification language with which certain safety and liveness properties can be expressed, a notation for programs, and a proof theory with which one can prove that a program satisfies the specification.

The basic specification language includes 4 properties: **invariant**, **unless**, **ensures**, and leads to ($\mapsto$). **invariant** p means that p holds initially and continues to hold. p **unless** q means that if at some point p holds, then it continues to hold at least as long as q does not. p **ensures** q requires both p **unless** q and that there is a single statement in the program who's execution in a state satisfying p, will establish q. $p \mapsto q$ states that if at some point p holds, then eventually q will hold. Additionally, we have **stable** p, which means that if p holds at some point, it will continue to hold thereafter.

The programming notation allows a program to be denoted by a set of variable declarations, a predicate *init* characterizing allowed initial states, and a non-empty set of guarded, multiple, deterministic, terminating assignment statements. Assignments are written, for example as follows:

$$x, y := f(x, y), g(x, y, z) \text{ if } b$$

This would be executed by first evaluating b, f, and g, then if b holds assigning the computed results to x and y. An equivalent notation for multiple assignment is

$(x := f(x, y) \,||\, y := (x, y, z))$ if b

Assignment statements in a program are separated by a bar $\square$. An execution of a program begins in a state satisfying *init*, then repeatedly executes, atomically, statements of the program. The choice of the statement to execute at each step is non-deterministic with a fairness constraint that each statement must be attempted infinitely often. There is no flow of control determined by a hidden program counter. All control information is explicitly coded in the guards. UNITY programs do not terminate—the analogy to termination is reaching a fixed point where no statement changes the state.

When convenient, several statements may be generated by quantifications. For example,

$$\langle \square i : i \leq 0 < n : x[i], x[i+1] := x[i+1], x[i] \text{ if } x[i] > x[i+1] \rangle$$

generates n statements separated by a $\square$. The quantified program is a nondeterministic bubble sort which reaches a fixed point when the array is sorted.

The basic proof theory comprises rules for proving that the properties hold for a given program. The logic presented here has been slightly modified and is described in more detail in [San91]. In the rules below,

$$SP.p \equiv (\exists s : s \text{ is a statement in the program} : sp.s.p) \tag{26}$$

where sp is the usual strongest postcondition of an individual statement. For standard UNITY programs, sp, and thus SP is monotonic and or-continuous. The definitions for **unless** and **ensures** use wp. For UNITY programs, all statements are required to terminate, thus $wp = wlp$. In the proof rules below, the range of s is the set of statements in the program.

$$\begin{aligned} &p \,\textbf{unless}\, q \equiv \\ &\quad (\forall s :: [SI \Rightarrow ((p \wedge \neg q) \Rightarrow wp.s.(p \vee q))]) \end{aligned} \tag{27}$$

$$\begin{aligned} &p \,\textbf{ensures}\, q \equiv \\ &\quad p \,\textbf{unless}\, q \wedge \\ &\quad (\exists s :: [SI \Rightarrow ((p \wedge \neg q) \Rightarrow wp.s.q)]) \end{aligned} \tag{28}$$

$$\frac{p \,\textbf{ensures}\, q}{p \mapsto q} \tag{29}$$

$$\frac{p \mapsto r, r \mapsto q}{p \mapsto q} \tag{30}$$

For any set W: $$\frac{(\forall m : m \in W : p.m \mapsto q)}{(\exists m : m \in W : p.m) \mapsto q} \tag{31}$$

invariant was defined by (5). Since actually determining SI is usually not practical, a useful proof rule is as follows:

$$(\textbf{invariant } I \land (\forall s :: [(p \land I)) \Rightarrow wp.s.p]) \Rightarrow \textbf{invariant } p \tag{32}$$

Since **invariant** true holds for every program, true is often a convenient choice for I in (32). **stable** is a special case of **unless**:

$$\textbf{stable}\, p \equiv p\, \textbf{unless}\, \text{false} \tag{33}$$

In addition, there are a large number of useful metatheorems which contribute a great deal to making the theory practical. The ones used in the example are listed in the appendix.

Mixed specifications [San90] are UNITY programs augmenting with a set of safety and liveness properties. Legal execution sequences are those generated by the state transitions (statements) plus stuttering steps which also satisfy the properties. If all the runs generated by the statements are legal, then the mixed specification is said to be implementable.

Since a main goal in formalizing knowledge is to have a method to understand the behavior of processes with incomplete information, we must extend the above model with some notion of a process. Since there is no "flow of control", a process is determined by its address space. Thus a process in our framework is simply a subset of program variables. UNITY is extended to allow expression of knowledge-based protocols by allowing knowledge predicates to appear as guards of the assignment statements. We then have the UNITY proof system, plus known properties of knowledge (14–24). Results are valid for any solution to the knowledge-based protocol.

UNITY/mixed specifications can be used to represent most interesting computation models. Shared memory is trivial to represent, the shared variables simply belong to more than one process. Message communication can be modeled by sequence variables to which the sender appends messages and where the receiver removes the head of the sequence. The process's memory, if any, must be explicitly included using history variables. Thus it is possible, in the same framework, to reason about programs where processes must remember part or all of their history (by including appropriate history variables) and where they do not.

6 Algorithm development using knowledge-based protocols

In this section, an example is sketched in order to clarify some issues involved in using knowledge-based protocols in algorithm design. The example chosen is the sequence transmission problem described in [HZ87, HZar]. The problem is to transmit a sequence data items over a possibly faulty communication channel: all messages must be delivered in the order sent, and all messages sent must eventually be delivered. There are several different algorithms for this problem depending on what sort of faulty behavior the channel may exhibit, whether or not the sender and receiver are synchronized, etc. Well known examples include protocols in the AUY model [AUY79, AUWY82] (the sender and receiver communicate synchronously over a channel that allows only one bit messages), the alternating bit protocol [BSW69], and Stenning's protocol [Ste82]. Halpern and Zuck give a high-level, intuitively correct, knowledge-based protocol which is relatively independent of the particular assumptions about the communication channel and *a priori* knowledge, and prove its correctness. They then propose an infinite state standard protocol that is shown to be an implementation (in a precise sense) of the knowledge-based protocol. The standard protocol is then further refined to obtain several known protocols which are implementations of the standard protocol. Following this approach, we state and prove correctness of a high-level knowledge-based protocol and an infinite state standard protocol using the formalism and proof theory described in the previous sections. The protocols are essentially the same as those given in [HZar], allowing the reader to easily compare the approaches. Refinements from the infinite state standard protocol to various finite-state protocols are interesting in their own right and can be done using techniques described in [San90] but are independent of knowledge, thus are not discussed here.

6.1 Specification First, we define x to be the infinite sequence of values, taken from a finite alphabet A, which are to be sent by the Sender. w is the initially empty sequence of values that have been delivered by the Receiver. We capture the requirement that messages are delivered in the order sent by giving an invariant property that w is always a prefix of x. To capture the requirement that messages are eventually delivered, we give a leads-to property requiring that the number of delivered messages will eventually increase.

$$\text{Safety: } \textbf{invariant } w \sqsubseteq x \tag{34}$$

$$\text{Liveness: } |w| = k \mapsto |w| > k \tag{35}$$

In the leads-to property (35), k is a free variable which is implicitly universally quantified. Thus we actually have a leads to property corresponding to every non-negative value of k.

6.2 A knowledge-based protocol The basic idea behind the protocol is very simple. At each step, the Sender either transmits the value of the *ith* element of x together with i or it increments i and gets the next element of x. In both cases, an attempt is made to receive a message from the communication channel. The first choice is made when the Sender does not know that the Receiver knows the value of the *ith* element of x, the second, when the Sender knows that the Receiver knows the value of x. At each step, the Receiver either delivers (writes) the *jth* element of x, and increments j, or transmits the value of j. In both cases, an attempt is also made to receive a value from the communication channel. The first alternative is performed when the Receiver knows the value of the *jth* element of x, the second when the receiver does not know the value.

The knowledge-based protocol is formally given in figure 3. We use the notation $(K_R(x_k = \alpha))_{@k=j}$ to indicate $K_R(x_k = \alpha)$ with k free, but evaluated at the value of j. $K_R x_k$ is an abbreviation for $(\exists \alpha : \alpha \in A : K_R x_k = \alpha)$. The command *transmit* need not, at this point, be completely defined, but intuitively means transmit the value of the parameter on the communication channel. The command *receive*(var) attempts to receive a value from the communication channel and assign it to var. If there is no message to be received, or the message has been corrupted, then var receives the value denoted $\perp$, which is different from any legal value. The vari-

```
declare $x$ :  seq of $A$
        $y : A$
        $z$ : nat$\cup \perp$
        $i$ : nat
        $w$ :  seq of $A$
        $z'$ : (nat, $A$)$\cup \perp$
        $j$ : nat

processes Sender $= \{x, y, i, z\}$, Receiver $= \{w, z', j\}$

init $(i = 0 \wedge y = x_0 \wedge z = \perp) \wedge (j = 0 \wedge w = \phi \wedge z' = \perp)$

assign

        Sender

        $transmit((i, y)) \, || \, receive(z)$ if $\neg(K_S K_R x_k)_{@k=i}$
      ▯
        $y := x_{i+1} \, || \, i := i + 1 \, || \, receive(z)$ if $(K_S K_R x_k)_{@k=i}$


      ▯ Receiver

        $\langle \alpha : \alpha \in A : w := w; \alpha$
           $|| \, j := j + 1 \, || \, receive(z')$ if $(K_R(x_k = \alpha))_{@k=j} \rangle$
      ▯
        $transmit(j) \, || \, receive(z')$ if $\neg(K_R x_k)_{@k=j}$
properties

   (Kbp-1) $(i = k \wedge y = \alpha \wedge \neg K_S K_R x_k) \mapsto$
              $K_R(x_k = \alpha) \vee \neg(i = k \wedge y = \alpha \wedge \neg K_S K_R x_k)$
   (Kbp-2) $(j = k \wedge \neg K_R x_k) \mapsto (K_S(j \geq k) \vee \neg(j = k \wedge \neg K_R x_k))$
   (Kbp-3) stable $K_R(x_k = \alpha)$
   (Kbp-4) stable $K_S K_R x_k$
```

Figure 3: Knowledge-based protocol

ables and commands belonging to the environment have not been specified. However, in order to prove the liveness property, it is necessary to state some property of the channel. This is done by giving two $\mapsto$ properties along with the protocol which say that once the conditions at the Sender (or Receiver) for transmitting a particular message have been satisfied, then either certain "knowledge" contained in the message will be attained at the Receiver (Sender) or the conditions for sending that message will be cancelled by the Sender. A consequence of this rule is that if, for example, the Sender follows the rule of repeatedly sending a message until it knows that the knowledge has been attained by the Receiver, then eventually the knowledge *will* be attained by the receiver. This can be guaranteed by a communication channel that will eventually correctly deliver any message that is sent repeatedly. The **stable** properties indicate that certain knowledge is not forgotten once it is attained. These assumptions are necessary for the correctness of the protocol but are not guaranteed by the definition of knowledge, thus they are listed in the properties section. It is not necessary at this point to discuss *how* this property is enforced, however, these properties will need to be verified for any instantiation of the knowledge-based protocol with a standard protocol.

Now we prove that any instantiation of the knowledge-based protocol satisfies the specification. Where details are straightforward, they will be omitted for the sake of brevity. An attempt is made, however, to give enough details to provide the reader unfamiliar with this type of formal proof an idea of what they look like. Much use is made of standard UNITY metatheorems. These are listed in the appendix. When the note "from program text" is mentioned, that means that the property mentioned can be proved directly by using the basic proof rules. For example, for an **invariant** property, say p, we would verify that $[Init \Rightarrow p]$ and that for each statement s in the program, $p \wedge I \Rightarrow wp.s.p$. Where I is an invariant, possibly true. As a practical matter, if p doesn't mention any variable changed by a particular statement s, then $wp.s.p$ holds immediately and no calculation is required.

The following invariant property can be proved directly from the program text.

$$\textbf{invariant}\ |w| = j \tag{36}$$

We can easily prove the safety property (34). First show **invariant** $(|w| = j \wedge w \sqsubseteq x)$ from the program text, which together with (36) allows us to conclude the desired result. $|w| = j \wedge (w \sqsubseteq x)$ holds initially since $|\phi| = 0$ and $\phi \sqsubseteq x$ for any sequence x. Now, we verify $|w| = j \wedge w \sqsubseteq x \Rightarrow wp.s.|w| = j \wedge w \sqsubseteq x$ for the statements of the form $w := w; \alpha \,||\, j := j + 1 \,||\, receive(z')$ if $(K_R(x_k = \alpha))_{@k=j}$. These are the only statements necessary to consider since the other statements do not modify w, x, or j.

$$
\begin{array}{ll}
 & |w| = j \wedge w \sqsubseteq x \Rightarrow wp.s.|w| = j \wedge w \sqsubseteq x \\
= & \{\text{calculate } wp \text{ and simplify}\} \\
 & |w| = j \wedge w \sqsubseteq x \wedge (K_R(x_k = \alpha))_{@k=j} \Rightarrow \\
 & \qquad |w; \alpha| = j + 1 \wedge w; \alpha \sqsubseteq x \\
\Leftarrow & \{(14)\} \\
 & |w| = j \wedge w \sqsubseteq x \wedge x_j = \alpha \Rightarrow |w; \alpha| = j + 1 \wedge (w; \alpha) \sqsubseteq x \\
= & \{\text{using } |w| = j \wedge w \sqsubseteq x \Rightarrow w = (x_0, x_1, \ldots, x_{j-1})\} \\
 & \text{true}
\end{array}
$$

The following two invariants will be used in the liveness proofs and depend on the assumptions that $K_R x_k$ and $K_S K_R x_k$ are stable.

$$\textbf{invariant } (\forall l : 0 \leq l < j : K_R x_l) \qquad (37)$$

$$\textbf{invariant } (\forall l : 0 \leq l < i : K_S K_R x_l) \qquad (38)$$

The proofs of (37) and (38) are very similar. We give the proof of (37).

Proof of (37):
Let $P.k$ be an abbreviation for $(\forall l : 0 \leq l < k : K_R x_l)$. $init \Rightarrow P.j$ trivially since $j = 0$.

$j = k$ **unless** $j = k + 1$	{from text}
$K_R x_k$ **unless** false	{(Kbp-3)}
$j = k \wedge K_R x_k$ **unless** $j = k + 1 \wedge K_R x_k$	{conjunction}
$j = k$ **unless** $j = k \wedge K_R x_k$	{from text}
$j = k$ **unless** $j = k + 1 \wedge K_R x_k$	{cancellation}
stable $P.k$	{conj. with (Kbp-3)}
$j = k \wedge P.k$ **unless** $j = k + 1 \wedge P.(k + 1)$	{conj. with above}
$(\forall l : 0 \leq l < j : K_R x_l)$ **unless** false	{gen. disj}

Because of the invariant (36), the following property is equivalent to the original liveness property (35).

$$j = k \mapsto j > k \tag{39}$$

Property (39) follows from properties (40) and (41) (using transitivity and disjunction).

$$j = k \wedge K_R x_k \mapsto j > k \tag{40}$$

$$j = k \wedge \neg K_R x_k \mapsto j = k \wedge K_R x_k \tag{41}$$

Property (40) indicates that if the Receiver knows the value of the next element to write, then it will eventually deliver it. Property (41) indicates that if the Receiver does not know the value of the next element, then it will eventually come to know the value. One would expect the proof of (40) to be straightforward and indeed it is. It is given below as an example of a simple liveness proof, and to demonstrate the usefulness of the metatheory in constructing proofs of knowledge-based protocols.

Proof of (40):
Let $s =$ "$w := w; \alpha \,||\, j := j + 1 || receive(z')$ if $(K_R(x_k = \alpha))_{@k=j}$".

$j = k$ **unless** $j > k$	{from text}
$K_R(x_k = \alpha)$ **unless** false	{(Kbp-3)}
$j = k \wedge K_R(x_k = \alpha)$ **unless** $j > k$	{simple conjunction}
$j = k \wedge K_R(x_k = \alpha) \Rightarrow$	
$wp.s.j > k$	{from text}
$j = k \wedge K_R(x_k = \alpha) \mapsto j > k$	{(28, 29)}
$j = k \wedge K_R x_k \mapsto j > k$	{(31)}

In order to prove $j = k \wedge K_R(x_k = \alpha)$ **unless** $j > k$, we used a metatheorem with the assumptions that $K_R(x_k = \alpha)$ is stable, instead of proving the **unless** property directly from the text. This is because the necessarily wp cannot be computed for the term with $K_R(x_k = \alpha)$ until an instantiation for this is given. The metatheory allows us to stay at the desired level of abstraction while still giving a formal proof.

The proof of (41) is clearly going to be more involved than the above since guaranteeing this will require actions by the Sender, Receiver, and communication channels.

Proof of (41):

Use transitivity on (44) and (45), disjunction with $K_R x_k \mapsto K_R x_k$, transitivity with 43, and PSP with (42).

$$j = k \land \neg K_R x_k \textbf{ unless } j = k \land K_R x_k \tag{42}$$

$$j = k \land \neg K_R x_k \mapsto K_S(j \geq k) \lor K_R x_k \tag{43}$$

$$K_S(j \geq k) \mapsto i \geq k \tag{44}$$

$$i \geq k \mapsto K_R x_k \tag{45}$$

Proof of (42):

From text.

Proof of (43):

PSP on (Kbp-2) and 42, simplify and weaken right hand side.

Proof of (44):

Leads-to implication on (46) and transitivity with (47)

$$K_S(j \geq k) \Rightarrow (\forall l : 0 \leq l < k : K_S K_R x_l) \tag{46}$$

$$(\forall l : 0 \leq l < k : K_S K_R x_l) \mapsto i \geq k \tag{47}$$

Proof of (46):

invariant $K_S(j \geq k) \Rightarrow j \geq k$ {(14)}

invariant $K_S(j \geq k) \Rightarrow K_S(\forall l : 0 \leq l < k : K_R x_l)$ {(15),(37)}

invariant$K_S(j \geq k) \Rightarrow (\forall l : 0 \leq l < k : K_S K_R x_l)$ {(21)}

Proof of (47):

$i = m \land K_S K_R x_k \mapsto i > m$ {similar to (40)}

$i = m \land m < k \land (\forall l : 0 \leq l < k : K_S K_R x_l) \mapsto$
$i > m$ {strengthen ant.}

$i = m \land m < k \land (\forall l : 0 \leq l < k : K_S K_R x_l) \mapsto$
$i \geq k$ {induction}

$(\forall l : 0 \leq l < k : K_S K_R x_l) \mapsto i \geq k$ {(31)}

Proof of (45):

Use leads-to implication on (48) then disjunction with (49) where (48) is obtained from (38) and (14).

$$\textbf{invariant } (i > k) \lor (i = k \land K_S K_R x_k) \Rightarrow K_R x_k \tag{48}$$

$$i = k \wedge \neg K_S K_R x_k \mapsto K_R x_k \tag{49}$$

Proof of (49):

$i = k \wedge y = \alpha \wedge \neg K_S K_R x_k$ **unless** $K_S K_R x_k$ {from text}

$i = k \wedge y = \alpha \wedge \neg K_S K_R x_k \mapsto$

$(K_R(x_k = \alpha) \wedge i = k \wedge y = \alpha) \vee K_S K_R x_k$ {PSP, (Kbp-1)}

$i = k \wedge y = \alpha \wedge \neg K_S K_R x_k \mapsto K_R x_k$ {weaken cons., (14)}

$i = k \wedge \neg K_S K_R x_k \mapsto K_R x_k$ {(31)}

6.3 Standard protocol The preceding proof shows that the knowledge-based protocol, provided it exists, is correct. Any standard protocol obtained by specifying the communication channel so that the channel liveness assumptions are satisfied, replacing $(K_R(x_k = \alpha))_{@k=j}$, $(K_S K_R x_k)_{@k=i}$, with normal predicates, and verifying that the liveness and stability assumptions are satisfied, is a correct protocol.

Now, we will consider a standard protocol. In this case, we want the communication channel to allow loss, duplication, and detectable corruption of messages. This implies that if a received message contains a legal value, then the message was, at some previous time, actually sent. A minimal amount of extra state is introduced in order to specify this, namely history variables for the Sender and Receiver which record messages sent. History variables are not accessible to any process.

Proposed values for the knowledge predicates $K_S K_R x_k$ and $K_R(x_k = \alpha)$ are as follows:

$$K_R(x_k = \alpha) : (j = k \wedge z' = (k, \alpha)) \vee (j > k \wedge w_k = \alpha) \tag{50}$$

$$K_S K_R x_k : (i = k \wedge z = k + 1) \vee i > k \tag{51}$$

$K_S(j \geq k)$ did not appear in the protocol statements, but only in the channel liveness property ((Kbp-2). Thus we do not actually need the value, but only need prove the channel liveness properties. Because of (24), it suffices to show:

$$\textbf{invariant } z \geq k \Rightarrow K_S(j \geq k) \tag{52}$$

$$(j = k \wedge \neg K_R x_k) \mapsto ((z \geq k) \vee \neg(j = k \wedge \neg K_R x_k)) \tag{53}$$

By (24), (54) suffices to show (52).

$$\textbf{invariant } z \geq k \Rightarrow j \geq k \tag{54}$$

The stable properties corresponding to (Kbp-3) and (Kbp-4) are:

$$\textbf{stable}\ (i = k \land z = k+1) \lor i > k \tag{55}$$

$$\textbf{stable}\ z' = (k, \alpha) \lor (j > k \land w_k = \alpha) \tag{56}$$

It is easy to show that the following two properties imply the channel liveness properties (Kbp-1) and (53):

$$\begin{aligned} & i = l \land y = \alpha \land z \leq i \mapsto \\ & \quad z' = (l, \alpha) \lor \neg(i = l \land y = \alpha \land z \leq i) \end{aligned} \tag{57}$$

$$\begin{aligned} & j = l \land \neg(\exists \alpha :: z' = (l, \alpha)) \mapsto \\ & \quad z = l \lor \neg(j = l \land \neg(\exists \alpha :: z' = (l, \alpha))) \end{aligned} \tag{58}$$

These liveness properties require that a message repeatedly transmitted will eventually be received (unless the transmitter gives up) and will be satisfied by a communication channel that does not lose or corrupt all messages sent. In order to capture the property of the communication channel that allows lost, duplicated, or detectably corrupted messages, we add history variables which record all messages sent, and add invariant assumptions stating that any message received with a legal value must have been sent.

$$\textbf{invariant}\ z = k \Rightarrow k \in \overline{ch_R} \tag{59}$$

$$\textbf{invariant}\ z' = (k, \alpha) \Rightarrow (k, \alpha) \in \overline{ch_S} \tag{60}$$

The resulting standard protocol is given in figure 4.

The properties (St-1), (St-2), (St-3), and (St-4) represent assumptions about the communication channel. We may use these in proofs and the protocols so proved will be correct provided the communication channel does indeed satisfy the properties. The required stable properties, 55 and 56 are easily verified from the program text and can therefore be omitted from the protocol description.

The remaining task is to prove that the proposed values for the knowledge predicates are indeed valid knowledge predicates. Recall that $K_R p \Rightarrow p$ is invariant, and that it is the weakest such predicate which only depends on R's view. This gives the following two additional invariant properties:

$$\begin{aligned} & \textbf{invariant}\ (j = k \land z' = (k, \alpha)) \lor (j > k \land w_k = \alpha) \Rightarrow \\ & \quad (x_k = \alpha) \end{aligned} \tag{61}$$

$$\begin{aligned} & \textbf{invariant}\ (i = k \land z = k+1) \lor i > k \Rightarrow \\ & \quad (\exists \alpha :: (j > k \land w_k = \alpha) \Rightarrow (x_k = \alpha)) \end{aligned} \tag{62}$$

```
declare  $x :$ seq of $A$
         $y : A$
         $z :$ nat $\cup \perp; \overline{ch_R} :$ seq of nat
         $i :$ nat
         $w :$ seq of $A$
         $z' :$ (nat, $A$)$\cup \perp; \overline{ch_S} :$ seq of (nat, $A$)
         $j :$ nat

processes  Sender $= \{x, y, i, z\}$, Receiver $= \{w, z', j\}$

init  $(i = 0 \wedge y = x_0 \wedge z = \perp) \wedge (j = 0 \wedge w = \phi \wedge z' = \perp)$

assign

         Sender

         $transmit((i, y)) \,||\, \overline{ch_S} := \overline{ch_S}; (i, y)$
            $||\, receive(z)$ if $\neg(z = i + 1)$
       ▯
         $y := x_{i+1} \,||\, i := i + 1 \,||\, receive(z)$ if $z = i + 1$

       ▯ Receiver

         $\langle \alpha : \alpha \in A : w := w; \alpha \,||\, j := j + 1 \,||\, receive(z')$
            if $z' = (j, \alpha)\rangle$
       ▯
         $transmit(j) \,||\, \overline{ch_R} := \overline{ch_S}; j$
            $||\, receive(z')$ if $\neg(\exists \alpha :: z' = (j, \alpha))$

properties

    (St-1) invariant $z = k \Rightarrow k \in \overline{ch_R}$
    (St-2) invariant $z' = (k, \alpha) \Rightarrow (k, \alpha) \in \overline{ch_S}$
    (St-3) $i = l \wedge y = \alpha \wedge z \leq i \mapsto$
              $z' = (l, \alpha) \vee \neg(i = l \wedge y = \alpha \wedge z \leq i)$
    (St-4) $j = l \wedge \neg(\exists \alpha :: z' = (l, \alpha)) \mapsto$
              $z = l \vee \neg(j = l \wedge \neg(\exists \alpha :: z' = (l, \alpha)))$
```

Figure 4: Standard protocol

Proof of (54):

invariant $k \in \overline{ch_R} \Rightarrow j \geq k$ {from text, $\overline{ch_R}$ initially empty}

invariant $z \geq k \Rightarrow j \geq k$ {initially $z = \perp$, with (St-1)}

Proof of (61):

$(k, \alpha) \notin \overline{ch_S}$ **unless** $i = k \wedge y = \alpha$ {from text}

invariant $i = k \wedge y = \alpha \Rightarrow x_k = \alpha$ {from text}

$(k, \alpha) \notin \overline{ch_S}$ **unless** $x_k = \alpha$ {sub., cons. weakening}

$x_k = \alpha$ **unless** false {x constant}*

$(k, \alpha) \notin \overline{ch_S} \vee x_k = \alpha$ **unless** false {cancellation}

invariant $(k, \alpha) \in \overline{ch_S} \Rightarrow x_k = \alpha$ {$\overline{ch_S}$ initially empty}

invariant $z' = (k, \alpha) \Rightarrow x_k = \alpha$ {(St-2) } **

$\neg(w_k = \alpha)$ **unless** $z' = (k, \alpha) \wedge j = k$ {from text}

$\neg(w_k = \alpha)$ **unless** {sub., (**)}
$z' = (k, \alpha) \wedge j = k \wedge x_k = \alpha$

$\neg(w_k = \alpha)$ **unless** $x_k = \alpha$ {cons. weakening}

$\neg(w_k = \alpha) \vee x_k = \alpha$ **unless** false {cancellation, (*)}

invariant $w_k = \alpha \Rightarrow x_k = \alpha$ {w initially empty}

invariant $((z' = (k, \alpha) \wedge j = k)$
$\vee(j > k \wedge w_k = \alpha)) \Rightarrow$ {(**)}
$x_k = \alpha$

Proof of (62):

$i \leq k$ **unless** $z \geq k$ {from text}

$i \leq k$ **unless** $(z \geq k) \wedge (j > k)$ {sub., (62)}

$i \leq k$ **unless** $j > k$ {cons. weakening}

$i \leq k \vee j > k$ **unless**false {canc., **stable** (j ¿ k)}

invariant $i > k \Rightarrow j > k$ {$init \Rightarrow \neg(i > k)$}

invariant $i = k \wedge z = k + 1 \Rightarrow j > k$ {(62)}

invariant $(i = k \wedge z = k + 1) \vee (i > k) \Rightarrow$
$j > k$

invariant $j > k \Rightarrow (\exists \alpha :: w_k = \alpha)$ {(36)}

invariant $(i = k \wedge z = k + 1) \vee (i > k) \Rightarrow$
$j > k \wedge (\exists \alpha :: w_k = \alpha)$

invariant $(i = k \wedge z = k + 1) \vee (i > k) \Rightarrow$
$\langle \exists \alpha :: (j = k \wedge z\prime = (k, \alpha)) \vee$
$(j > k \wedge w_k = \alpha) \rangle$

The next step to show that the standard protocol instantiates the knowledge-based protocol is to show that the predicates we have given in (51) and (50) are indeed the weakest ones which will work. However, this is difficult since it requires determining SI and there is actually little point in doing this. We have *already* proven enough to establish correctness of the standard protocol since we have directly verified all properties of the knowledge predicates that were used in the proofs. Further, although the standard protocol does in fact instantiate the knowledge-based protocol provided that there is no *a priori* information about x other than A, and A has at least two elements, if some of the values are known *a priori* or can be computed from previous values, then the standard protocol is no longer an instantiation of the knowledge-based protocol, even though it will still satisfy the original specification.[3]

6.4 Conclusions from the example

The discussion above suggests that the interpretation of the program in figure 3 as a knowledge-based protocol is an overspecification of the problem. Requiring the standard protocol to be consistent with the knowledge-based protocol rules out correct standard protocols, as would be the case in our example if there is *a priori* information about the values of the messages. It is important to stress that the standard protocol is still *correct*, however, it no longer instantiates the knowledge-based protocol.

It has been argued [Hal91] that the correct standard protocols which do not satisfy a knowledge-based protocol are non-optimal in the sense that more messages are sent than strictly necessary.

[3]In [HZar], what is proved is that the runs of the standard protocol are mapped, in a way that preserves the order of reading and writing, to a set of runs consistent with the knowledge-based protocol. In the proof, they give a result analogous to showing that (51–50) are the weakest predicates that will work. Namely, in Proposition 4.5, they state: Let $r \in \mathcal{R}(A^{st})$. Then for every $m \geq 0$:

1. $z^r(m) \neq i^{r(m)}$ *iff* $(\mathcal{I}, \Psi^{st}(r), m) \models \neg K_R K_S(x_{@_i})$.
2. $proj_1(z'^{r(m)}) \neq j^{r(m)}$ *iff* $(\mathcal{I}, \Psi^{st}(r), m) \models \neg K_R(x_j)$.

A weaker version, namely with the "iff" in both cases replaced by "follows from" suffices for correctness of the standard protocol. In addition, the stronger proposition does not hold if there is *a priori* information, an assumption which was never stated but was implicitly used in the proof.

For example, suppose the value of the first element of x is known *a priori*. The standard protocol above would still result in the value being sent and acknowledged, while a standard protocol consistent with the knowledge-based protocol would have the receiver deliver the value immediately, and the sender would begin with the second element, thus saving one message. In one sense, this is a better protocol, fewer messages are sent, but on the other hand, it makes the protocol more complicated. Whether or not the extra complication is better would depend on the particular circumstances and on the particular implementation. Another way of looking at the optimality resulting from the use of knowledge-predicates is that a message is sent as soon (within the constraints imposed by the structure of the protocol) as it can safely be sent. If x_0 is already known, then a message containing the value of x_1 can be sent as soon as it is read. For the sequence-transmission problem, this is probably good. There are examples, however, where it is not. There are examples where strengthening the initial condition, and thus giving more "knowledge" to the processes, may result in an action being taken "too soon" with the effect that there is no longer an instantiation of the knowledge-based protocol which satisfies the original specification, even though a standard protocol which instantiates the knowledge-based protocol with weaker initial conditions will work. Knowledge-based protocols exhibit a sort of process-by-process optimality which may or may not translate into global optimality for the protocol. One might also argue that the difficulty of showing that a standard protocol is consistent with a knowledge-based protocol is an artifact of the particular formalism used here. It is, in the sense that other approaches will not specifically require SI. However, other approaches still require specifically delineating the entire set of runs of the protocol. And no matter what the approach, it is certainly to be expected that proving that something is optimal will be more difficult than proving that it satisfies a set of constraints. Given the difficulty of proving that a standard protocol is an instantiation of a knowledge-based protocol, and the questionable benefit of that the type of optimality provided by knowledge-based protocols in general, there seems little reason to doubt the wisdom of maintaining a *separation of concerns* between correctness and efficiency in algorithm development.

Another weakness of knowledge-based protocols as a general tool for protocol design which can also be seen from this example is that knowledge-based protocols are not necessarily expressive enough. In the sequence transmission problem, we required that certain types of knowledge be stable. In [HZar], this was done by incorporating this property into the actual semantics. As a result, the nice looking, intuitive protocols were only informal descriptions–while the actual semantics were "tedious to write down and left for the reader". In our framework, we simply stated the stable properties with the specification language.

In light of the above discussion, a weaker interpretation of the protocol in figure 3 suggests itself. Instead of considering $\bar{K}$ to be a knowledge predicate, it is considered as an unspecified predicate which is assumed to satisfy a set of explicitly stated properties. In this case, the invariant and stable properties which were used in the proofs of correctness. The explicitly stated properties can be used in proofs of the high level, abstract protocols. In a refinement, they are explicitly verified or continue to be part of the property section. In an implementable version of the protocol, all unspecified predicates have been given values, and all assumptions hold. This interpretation is a generalization of mixed specifications described in [San90]. This approach allows algorithm development to be carried out at a similar level of abstraction as knowledge-based protocols. It is, however, more flexible, and eliminates excess baggage which comes with the definition of knowledge. The cost is that one must write down the properties that are needed explicitly. The only difference between this approach and what we have done is that we would include the properties such as **invariant**$\bar{K}_R(x_k = \alpha) \Rightarrow (x_k = \alpha)$ in the properties section.

In this particular example, proving correctness of the standard protocol in figure 4 by starting with the abstract version is more work than to show directly that this protocol satisfies the original specification. The reason for this is the increased generality necessary to take into account the various types of behaviors when there is *a priori* information about the contents of the messages. If, for example, all of the messages are known *a priori*, then there will be no communication or synchronization at all between the partners in an instantiation of the knowledge-based protocol. In the standard protocol, the values of i and j are synchronized in order to maintain **invariant** $i \leq j \leq i+1$. Whether or not the extra work imposed by the added generality pays is problem dependent. For this example, it probably doesn't. In [HZar], the standard protocol, which is infinite state, is refined to obtain several interesting finite state protocols. This work is interesting and important, however, the generality of the knowledge-based protocol was not exploited.

7 Conclusion

Analysis of knowledge in distributed systems has proved to be valuable in understanding fundamental issues in distributed systems [CM86, HM90, MDH86] and has helped to design algorithms that are of both theoretical and practical [MT88, DM90, Ric92] interest. However, when working with particular algorithms, the lack of well-defined method has meant that the semantics and proof method must be defined in an *ad hoc* way for each problem. In this paper, a predicate transformer for knowledge was defined which provides the basis for reasoning about knowledge attained by processes in a slightly modified and extended version of UNITY, and for extensions to represent knowledge-based protocols. The properties of the predicate transformer yielded new insights into knowledge and knowledge-based protocols, and the extended version of UNITY provides a well defined specification language, notation for writing algorithms, and a proof theory for proving that algorithms satisfy their specification. In addition, an exercise in algorithm development using our method also revealed, in a clear way, several fundamental weaknesses of knowledge-based protocols which will prevent this approach from finding application as an "all-purpose" tool for designing algorithms.

Acknowledgement I would like to thank Yoram Moses, Joe Halpern, and Lenore Zuck for their comments on an earlier version of the paper, and Joe for his efforts to explain their previous work to me.

References

[AUWY82] A.V. Aho, J.D. Ullman, A.D Wyner, and M. Yannakakis. Bounds on the size and transmission rate of communication protocols. *Comp. and Maths. with Appls.*, 8(3):205–214, 1982. This is a later version of [AUY79].

[AUY79] A.V. Aho, J.D. Ullman, and M. Yannakakis. Modeling communication protocols by automata. In *Proc. 20th IEEE Symp. on Foundations of Computer Science*, pages 267–273, 1979.

[BSW69] 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.

[CM86] K. Mani Chandy and Jayadev Misra. How processes learn. *Distributed Computing*, 1:40–52, 1986.

[CM88] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.

[DM90] Cynthia Dwork and Yoram Moses. Knowledge and common knowledge in a Byzantine environment 1: Crash failures. *Information and Computation*, 88(2):156–186, October 1990.

[DS90] Edsger W. Dijkstra and Carel S. Scholten. *Predicate Calculus and Program Semantics.* Springer-Verlag, 1990.

[Hal91] Joseph Y. Halpern. A note on knowledge-based protocols and specifications. Technical Report RJ 8454 (76462), IBM Almaden Research Center, 1991.

[HF89] Joseph Y. Halpern and Ronald Fagin. Modelling knowledge and action in distributed systems. *Distributed Computing*, 3:159–177, 1989.

[HM90] Joseph Y. Halpern and Yoram Moses. Knowledge and common knowledge in a distributed environment. *Journal of the ACM*, 37(3):549–587, July 1990.

[HZ87] Joseph Y. Halpern and Lenore Zuck. A little knowledge goes a long way: Simple knowledge-based derivations and correctness proofs for a family of protocols. In *Proceeding of the 6th Annual ACM Symposium on Principles of Distributed Computing*, 1987.

[HZar] Joseph Y. Halpern and Lenore Zuck. A little knowledge goes a long way: Simple knowledge-based derivations and correctness proofs for a family of protocols. *Journal of the ACM*, to appear.

[KT86] Shmuel Katz and Gadi Taubenfeld. What processes know: Definitions and proof methods. In *Proceeding of the 5th ACM Symposium on Principles of Distributed Computing*, 1986.

[MDH86] Yoram Moses, D. Dolev, and Joseph Y. Halpern. Cheating husbands and other stories: A case study of knowledge, action, and communication. *Distributed Computing*, 1:167–176, 1986.

[MT88] Yoram Moses and Mark Tuttle. Programming simultaneous actions using common knowledge. *Algorithmica*, 3:121–169, 1988.

[Ric92] Aleta M. Ricciardi. Practical utility of knowledge-based analyses. In Yoram Moses, editor, *Proceedings of the 4th Conference on Theoretical Aspects of Reasoning About Knowledge*, pages 15–28, 1992.

[San90] Beverly Sanders. Stepwise refinement of mixed specifications of concurrent programs. In M. Broy and C.B. Jones, editors, *Proceedings of the IFIP Working Conference on Programming Concepts and Methods*, Israel, 1990. Elsiever Science Publishers.

[San91] Beverly Sanders. Eliminating the substitution axiom from UNITY logic. *Formal Aspects of Computing*, 3(2):189–205, 1991.

[Ste82] M.V. Stenning. A data transfer protocol. *Computer Networks*, 1:99–110, 1982.

8 UNITY metatheorems

8.1 Substitution Any invariant may be replaced by true and truemay be replaced by any invariant in any property.

8.2 Consequence weakening

$$\frac{p \textbf{ unless } q, q \Rightarrow r}{p \textbf{ unless } r}$$

$$\frac{p \mapsto q, q \Rightarrow r}{p \mapsto r}$$

8.3 Conjunction

$$\frac{p \textbf{ unless } q, p\prime \textbf{ unless } q\prime}{(p \wedge p\prime) \textbf{ unless } (q \vee q\prime)}$$

$$\frac{p \textbf{ unless } q, p\prime \textbf{ unless } q\prime}{(p \wedge p\prime) \textbf{ unless } (p \wedge q\prime) \vee (p\prime \wedge q) \vee (q \wedge q\prime)}$$

8.4 Cancellation $\dfrac{p \textbf{ unless } q, q \textbf{ unless } r}{(p \vee q) \textbf{ unless } r}$

8.5 Generalized disjunction

$$\frac{(\forall i :: p.i \textbf{ unless } q.i)}{(\exists i :: p.i) \textbf{ unless } (\forall i :: \neg p.i \vee q.i) \wedge (\exists i :: q.i)}$$

8.6 PSP (progress-safety-progress)

$$\frac{p \mapsto q, r \textbf{ unless } b}{(p \wedge r) \mapsto ((q \wedge r) \vee b)}$$