What Processes Know: Definitions and Proof Methods

(Preliminary Version)

by

Shmuel Katz and Gadi Taubenfeld

Computer Science Department

Technion, Haifa 32000

Israel

ABSTRACT

The importance of the notion of knowledge in reasoning about distributed systems has been recently pointed out by several works. It has been argued that a distributed computation can be understood and analyzed by considering how it affects the state of knowledge of the system. We show that there are a variety of definitions which can reasonably be applied to what a process can *know* about the global state. We also move beyond the semantic definitions, and present the first proof methods for proving *knowledge assertions*. Both shared memory and message passing models are considered.

1. INTRODUCTION

In this paper we will show how to prove knowledge assertions about both distributed and shared memory parallel programs. Before describing the proof methods we consider the fundamental question of "what is the meaning of knowledge in concurrent programs?" Put another way, what does it mean for (an observer of) a process to know something? Other tempting questions immediately arise: What does a process always know? What does a process know whenever it terminates? What does it know at a certain moment during a given execution?

A number of successful attempts have been made in order to formally capture various notions of knowledge [HM1,HM2,FHV,Le]. Some other recent works are directly connected to a concurrent programming environment. In [CM] a characterization is described of the minimum information flow necessary for a process to gain or lose knowledge about the system. The results are applied for proving lower bounds on the number of messages required to solve certain problems. In [HF] a formal model that relates the concepts of knowledge, action and communication in distributed systems is presented. An impossibility proof for the well known coordinated-attack problem is presented. In [PR] a connection between distributed systems and the logic of knowledge is established. A set of axioms which this knowledge must obey is presented.

We first motivate our approach to knowledge. Consider a concurrent program which is frozen at some point during its execution. Let b stand for a predicate defined on the global state of that program. Now, suppose you are interested in knowing whether predicate b is true when the program is suspended. For this task, you are allowed to use certain *global information* which is assumed to be known a priori. Such information might be, for example, an invariant of the program, certain properties of the processes, or properties of the model. In addition it is possible that you have access to

the following *local information* about the behavior of process i.

case one: no additional information.

case two: the control position of process i.

case three: the complete visible state of process i.

case four: the complete history of process i.

For a distributed model, a process' visible state is identical to its local state. In the shared variable model, a process' visible state refers to all variables it can test. The history of a process means the sequence of all events in which it has participated.

We say that *process i knows b* when the program is suspended, if using the above information it is possible to prove that b holds. Thus a process' knowledge is always defined with respect to some given information. Note that the global information is static, in the sense that it does not change during an execution, while the local information is dynamic. We call the available global and local information, the *knowledge assumption*.

In this paper we assume the (entire) text of the program and its input specification as the a priori global information of a process. The opposite extreme would be to assume no global information. This case is of little interest since extremely limited knowledge is possible when even the properties of the model are not known a priori. A less extreme case, where nothing is assumed about *some* of the processes, is of greater interest. A possible application for such a case is the Byzantine Generals Problem, where the protocol which a faulty Byzantine process is following is not available. However, it follows from [LSP] that in order for the processes to *know* the agreed value, it must be part of the global information that over two thirds of the processes are following a protocol which is known. An intermediate level is also possible, in which only certain properties of the program are assumed to be known a priori. This level would be appropriate during program development. In [HF] it is not assumed that the text is given.

Given our assumption of maximal global information, the above four cases of *local* information are considered to define different types of knowledge. Cases one and four represent opposite extremes from process i's point of view. In case one, called invariant knowledge, no local information at all is available, and thus process i may know only 'general things' like 'the program terminates' and all safety properties. In case four, called history knowledge, all possible local information is available, which of course, leads to the strongest type of knowledge. Numerous intermediate levels, where only a partial history is given, can be defined between those extremes. Examples are cases two and three, called location and state knowledge, respectively. Previous papers [CM,HF,PR] refer to (variants of) the concept of history knowledge. They all mention state knowledge as an additional option. Location knowledge refers to the knowledge of a process at a certain point for all possible computations. We establish in the sequel a connection between fundamental verification concepts such as proof outline, interference freedom and cooperation [AFR,OG] and the concepts of knowledge. A method for proving knowledge assertions of all types is presented.

2. THE MODEL

A concurrent program is a collection of a finite set of processes operating asynchronously, which may communicate with each other in one of the following ways, (a)asynchronously using send and receive, (b)synchronously, as in CSP, also using send and receive commands, or (c) by setting shared variables.

An event is the basic unit in the behavior of a process. It describes a change in the environment during the process' execution. An event occurs on a process, by executing an atomic command of that process. An execution of an atomic command includes executing a skip or assignment command, a loop exit, evaluation of a boolean expression, or execution of a com-

munication.[1] The same atomic commands executed in different environments correspond to different events. A behavior of a process can be described as a finite sequence of events on that process. This is known as a trace [BHR,M]. A process is characterized by a (prefix-closed) set of all its possible traces. By generalizing the above, a concurrent program P is defined. Assuming an interleaving semantics, we are only interested in the relative order in which events occur. No clock is assumed. A trace t of P is a finite sequence of events on processes of P satisfying the conditions:

(1) Any subsequence of t which consists of all events on a process which belongs to P, is a trace of that process.

(2) *Asynchronous communication*: for every event of 'receiving message m' there is a corresponding 'send message m' event which occurs earlier.

(2') *Synchronous communication*: for every event of 'send message m' there is a corresponding 'receive message m' event which occurs immediately after, i.e, no other event occurs between those events (those two events can be considered to be a single one).

A concurrent program is characterized as the set of all possible traces of the program.

A trace can also be seen as a transition from the initial state of the program into a unique state of the program, which in addition records all the *'history'* of how it reached that state. Each such state contains two kinds of information: the exact values of the program variables and the control position of each process (the value of the program counter).

To be able to express the control position, we assume that all commands are uniquely labeled. By saying that (the control of) a process is *after* c, we mean that the command labeled c was the last one to be executed in that process. In addition the labels $init_1, \ldots, init_n$ are used to label dummy commands placed at the beginning of each process. If (control of) process i is after $init_i$, execution of process i has not yet started.[2]

With each trace t we associate a (trace) predicate s_t which expresses the values of all the variables (including the control) after a computation described by t took place. This predicate depends on a given initial state. For the rest of the paper let l_i stand for a label of a command from process p_i $(i=1..n)$. Let C be a function from traces into n-tuples of labels.

Definition: $C(t) = (l_1, \ldots, l_n)$ iff after a computation described by t the control of processes $p_1, \ldots, p_n$ is after $l_1, \ldots, l_n$ respectively.

A *location set* $T_{(l_1,\ldots,l_n)}$ is defined to be a set of traces such that: $T_{(l_1,\ldots,l_n)} \equiv \{ t \mid C(t)=(l_1, \ldots, l_n) \}$.

Thus location set $T_{(l_1,\ldots,l_n)}$ describes the set of traces which leave the control of the program after $(l_1, \ldots, l_n)$. It easy to see that the number of such location sets is exactly the product of the number of labels in each process. They correspond to all possible positions of the control of the program during an execution.

For the rest of the paper, let lc denote an n-tuple of labels $(l_1, \ldots, l_n)$. With each location set T_{lc} we associate a *location predicate* S_{lc}. S_{lc} is defined to be the disjunction of all s_t such that t belongs to T_{lc}. If T_{lc} is an empty set (i.e for no execution is the control after lc), then S_{lc} is *false*. Assume that the control of a program is after lc. The state of a specific execution at this point refers to the exact values of all program (and environment) variables. The location predicate S_{lc} at this point refers to all possible values of the variables, without committing to a certain execution. S_{lc} is actually the strongest predicate which is true when the control is after lc.

[1] The language may also contain selection statments with local nondeterminism, but this will not be atomic.

[2] In case that c and d denote two different labels of distinct commands belonging to the same process, then the assertion $after(c) \wedge after(d)$ equals *false*.

Example 1: Consider the (shared variable) program $P :: [P_1 \parallel P_2]$ where:

$P_1 :: end_1: x := x+1$
$P_2 :: end_2: x := 2*x$

Assume that the initial state is $x=2$.

The possible events are: (1) end_1 executes first, (2) end_2 executes first, (3) end_1 executes second, (4) end_2 executes second.

The predicates associated with the traces $<>, <1>, <2>, <2,3>, <1,4>$ are:

$s_{<>} \equiv x=2 \wedge after(init_1, init_2)$,
$s_{<1>} \equiv x=3 \wedge after(end_1, init_2)$,
$s_{<2>} \equiv x=4 \wedge after(init_1, end_2)$,
$s_{<2,3>} \equiv x=5 \wedge after(end_1, end_2)$,
$s_{<1,4>} \equiv x=6 \wedge after(end_1, end_2)$, respectively.

The possible location sets are:

$T_{(init_1, init_2)} \equiv \{<>\}$, $T_{(end_1, init_2)} \equiv \{<1>\}$,

$T_{(init_1, end_2)} \equiv \{<2>\}$, $T_{(end_1, end_2)} \equiv \{<2,3>, <1,4>\}$.

The possible location predicates are:

$S_{(init_1, init_2)} \equiv x=2 \wedge after(init_1, init_2)$,
$S_{(end_1, init_2)} \equiv x=3 \wedge after(end_1, init_2)$,
$S_{(init_1, end_2)} \equiv x=4 \wedge after(init_1, end_2)$,
$S_{(end_1, end_2)} \equiv ((x=5) \vee (x=6)) \wedge after(end_1, end_2)$.

We will also assume that each process i has a local history named h_i. The history will presumably be accumulated in a variable which records the sequence of all events in which process i has participated. As mentioned in [HF] "This is certainly not a reasonable assumption", because it assumes that a process has unbounded storage space to record its history. A trace t *matches* h_i ($match(t, h_i)$) iff t_i, the subsequence of t which consists of all events on process i, equals h_i. Similarly h_i *matches* location set T_{lc} iff it *matches* with some trace belonging to T_{lc}.

A *history location predicate* – Sh_{lc} – associated with location set T_{lc} and history h, is defined as the disjunction of all (trace) predicates s_t such that t belongs to T_{lc} and $match(t, h)$. If no such traces exist then $Sh_{lc} \equiv false$. The predicate Sh_{lc} is, by definition, the *strongest* predicate known to be true after lc when only h is known.

Example: If the local history h of P_1 was empty (no events have yet occurred), then that history matched the first and third traces given above. The possible history location predicates in that case are:
$Sh_{(init_1, init_2)} \equiv x=2 \wedge after(init_1, init_2)$,
$Sh_{(end_1, init_2)} \equiv false$,
$Sh_{(init_1, end_2)} \equiv x=4 \wedge after(init_1, end_2)$,
$Sh_{(end_1, end_2)} \equiv false$.

3. DEFINITION OF KNOWLEDGE

The (global) trace which corresponds to the execution up to any moment, belongs to exactly one location set (called the *actual* location set). Informally, we will say that *process i knows b* iff b is true in all location sets that process i considers possible candidates to be the actual one.

In the first case where no additional local information is given, process i must consider all location sets which are possible under the global a priori information. When, in case two, process i knows the exact position of its control, the possible location sets reduce to those for which l_i corresponds to i's actual position. With the additional assertion giving the exact visible state of process i, in case three, the above possible location sets reduce to those where the location predicate of the location sets does not contradict the assertion about the actual visible state. Finally, with the history of process i, only those location sets which *match* that local history are possible.

For now, we restrict the language so that no knowledge operator appears within the scope of another. Let c be a label of a command in process i, s be an assertion describing a visible state of process i (including the control of process i), and h be a history of process i.

P::[$P_1 \| P_2$] where:

P_1 ::	P_2 ::
a: [$a1$: *true* $\rightarrow x:=1$ [] $a2$: *true* $\rightarrow x:=2$];	d: [*true* $\rightarrow y:=1$ [] *true* $\rightarrow y:=2$ [] *true* $\rightarrow y:=3$];
$b: P_2!x$;	$e: P_2?z$;
$c: P_2?x$	$f: P_2!(y+z)$

Figure 1. Simple CSP Program.

The notations $KI_i(b)$, $KL_i^c(b)$, $KS_i^s(b)$, $KH_i^h(b)$ are used for: *process i knows b*, *process i knows b after c*, *process i knows b with s*, *process i knows b with h*, respectively. Recall that lc is an n-tuple of labels.

Definition:

(1) Invariant knowledge:

$KI_i(b)$ iff $\forall\, lc\ .\ S_{lc} \rightarrow b$.

(2) Location knowledge:

$KL_i^c(b)$ iff $\forall\, lc\ .\ (S_{lc} \wedge after(c)) \rightarrow b$.

(3) State knowledge:

$KS_i^s(b)$ iff $\forall\, lc\ .\ (S_{lc} \wedge s) \rightarrow b$.

(4) History knowledge:

$KH_i^h(b)$ iff $\forall\, lc\ .\ Sh_{lc} \rightarrow b$.[3]

Let K_i denote all knowledge types. The following properties are over all labels, visible states and histories.

Properties:

(1) $(K_i(b) \wedge (b \rightarrow b')) \rightarrow K_i(b')$.

(2) $K_i(b) \wedge K_i(b')$ *iff* $K_i(b \wedge b')$.

(3) $(K_i(b) \vee K_i(b')) \rightarrow K_i(b \vee b')$.

(4) $(K_i(b \rightarrow b') \wedge K_i(b)) \rightarrow K_i(b'))$.

(5) $KL_i^c(after(c))$.

(6) $KS_i^s(s)$.

(7) $(KL_i^c(b) \wedge (s \rightarrow after(c))) \rightarrow KS_i^s(b \wedge s)$.

We give a proof of property (7):

(1) $KL_i^c(b)$. [Assumption]

(2) $s \rightarrow after(c)$. [Assumption]

(3) $\forall\, lc\ .\ (S_{lc} \wedge after(c)) \rightarrow b$. [(1), Def. of location knowledge]

(4) $\forall\, lc\ .\ (S_{lc} \wedge s) \rightarrow b$. [(2), (3)]

(5) $KS_i^s(b)$. [(4), Def. of state knowledge]

(6) $KS_i^s(b \wedge s)$. [(5), properties 2 & 6]

[]

Example 2: Consider the (CSP) program, shown in figure 1. Let s describe a local state of P_1, $s \equiv \{x=3 \wedge after(c)\}$, Let h be a history of P_1. Assume that it implies that the current local state is the one described by s, and that the first event on P_1 occurred by executing command $a1$. The following knowledge assertions are then true:

(1) $KI_1(y=1 \vee 2 \vee 3 \vee Undefined)$, (2) $KL_1^c(y=1 \vee 2 \vee 3)$, (3) $KS_1^s(y=1 \vee 2)$. (4) $KH_1^h(y=2)$.

In the next section, we formally prove all these knowledge assertions.

4. THE PROOF METHOD

In this section a method for proving knowledge assertions about a program is presented. We initially establish a connection between *location knowledge* and various verification methods. The method for

[3] An alternative equivalent definition is, $KH_i^h(b)$ *iff* $\forall\, t\ .\ match(t,h)\ .\ s_i \rightarrow b$.

proving location knowledge assertions is then developed based on existing verification methods. For proving history knowledge assertions, an original method is developed, based on the new notion of *history outline*. For now, we restrict a process to know only predicates which do not contain temporal operators. In the future we intend to relax that restriction.

From the variety of verification methods available [B], we choose two proof systems [AFR,OG] which we consider representative. Both systems are designed to prove partial correctness, one for CSP and the other for a shared variable model. A stronger system, such as [OL], which can be used for proving liveness properties, should be considered when temporal operators are incorporated.

In each of the two systems, a partial correctness proof of a program is done in two stages: (1) separate proofs are constructed in isolation for each component process, (2) the separate sequential proofs are combined by showing that they are *interference-free* [OG] or *cooperate* [AFR].

The separate proofs, in the first stage, are presented by a *proof outline* in which each statement of a process is preceded and followed by a corresponding assertion. Next, in the second stage it is proved that the possible interactions do not invalidate the sequential proofs. In the shared variable model, it is necessary to show that the assertions used in the proof outline of each process are left invariantly true under parallel execution of the other processes. This is done by proving *interference-freedom*. In the distributed model, when establishing the separate proofs, a process 'guesses' the value its parameters will receive upon communication. When the proofs are combined, these guesses have to be checked for consistency using the *cooperation* test. In order to combine the separate sequential proof outlines, a *parallel composition* (meta) rule is used.

Both proof systems use additional *auxiliary variables* for the correctness proof. In [AFR] the notion of brackets is introduced, in order to delimit the sections within which an invariant need not necessarily hold. Here we assume that an interleaving can occur between every two atomic statements, so that all possible states after an atomic action must be considered in deciding what is known. For this reason, a bracketed section is restricted to contain exactly one basic command (i.e a command which does not refer to auxiliary variables). This does not affect the completeness as is shown in [Ap]. An await statement (for the shared-memory model) or a bracketed section is considered as an atomic action. For later reference, we denote a program P with additional auxiliary variables by P'.

In the proof outline for process i, $post(l_i)$ stands for the *post assertion* of a command labeled by l_i. A global invariant which may be used for the proof is denoted by I, while $post(l_1,\ldots,l_n)$ is an abbreviation for $\bigwedge_{i=1}^{n} post(l_i) \wedge I \wedge after(l_1,\ldots,l_n)$.

4.1 LOCATION KNOWLEDGE

We now present a proof method for proving location knowledge assertions. First, a theorem which links the definition of location knowledge and that of interference free (cooperating) proof outlines is formulated. Then we give the proof method itself.

Lemma 1: Assume a program with a locally correct sequential proof outline for each process. If the proof outlines of the individual processes are interference-free (cooperate) then for every label l_i of process i (i=1..n), $KL_i^{l_i}(post(l_i))$, i.e, process i *knows* (location knowledge) $post(l_i)$ after l_i.[4]

Proof: Assume to the contrary that for some l_i, $KL_i^{l_i}(post(l_i))$ does not hold. This means that there exists a trace t such that $C(t)=(l_1,\ldots,l_n)$ and its associated predicate s_t does not imply $post(l_i)$, which immediately contradicts the soundness [Ap,Ow] of the proof systems. []

[4] Here and afterwards only executions which start in an initial state satisfying $post(init_1,\ldots,init_n)$ are considered.

Remark: Let us define the assertions of (annotated) process i as a correct sequential *location outline* iff for every label l_i of process i, $KL_i^{l_i}(post(l_i))$. We claim that in order to prove partial correctness of a program, it is sufficient to establish such a location outline for each process, and then apply to them a (modified) parallel composition (meta) rule without further use of interference-freedom, cooperation or any other test. From the above lemma and claim the interference-freedom test and the cooperation test are viewed as providing sufficient (but not necessary!) conditions for a correct sequential *proof* outline to be also a location outline.

For the rest of the section, we refer to a program with correct interference-free (cooperating) proof outlines. In fact, it is sufficient to have correct location outlines.

Corollary 1.1: For any location set T_{lc} its associated predicate S_{lc} implies $post(lc)$.

Proof: The location predicate S_{lc} is the *strongest* predicate true after lc, while it follows from Lemma 1 that $post(lc)$ is a true predicate after lc. []

Theorem 1: If c is a label of process i, and $\Phi(c)$ is the *disjunction* of all assertions $post(l_1, \ldots, l_n)$ such that $l_i \equiv c$, then $KL_i^c(\Phi(c))$.

Proof:

(1) $\forall\, lc\ .\ (S_{lc} \wedge after(c)) \rightarrow (post(lc) \wedge post(c))$. [From corollary 1.1]

(2) $\forall\, lc\ .(post(lc) \wedge after(c)) \rightarrow \Phi(c)$. [Def. of $\Phi(c)$]

(3) $\forall\, lc\ .\ (S_{lc} \wedge post(c)) \rightarrow \Phi(c)$. [(1), (2)]

(4) $KL_i^c(\Phi(c))$. [(3), Def. of location knowledge]

[]

The preceding theorems refer to a given program which, as mentioned, may contain auxiliary variables. A rule similar to the usual rule [OG] for removing such variables is used, to refer back to the original program (i.e without auxiliary variables).

Auxiliary Variables rule:

Let AV be a set of variables such that $x \in$ AV implies x appears in P' only in assignments $y := t$, where $y \in$ AV. Then if Ψ does not contain free any variables from AV, P is obtained from P' by deleting all assignments to variables in AV, and processes i and i' are corresponding processes belonging to P and P' respectively,

$$\frac{KL_{i'}^{c'}(\Psi)}{KL_i^c(\Psi)}$$

Following is a method for proving location knowledge assertions. Suppose it is to be proved that $KL_i^c(b)$ is true for a given program P. The proof goes as follows:

(1) Using one of the mentioned proof systems [AFR,OG] a "strong enough" correctness proof is established for P'.

(2) Assertion $\Phi(c)$, as defined above, is constructed. [by theorem 1 $KL_{i'}^{c'}(\Phi(c))$].

(3) It is proved that $\Phi(c)$ implies b. [by property 1 $KL_{i'}^{c'}(b)$].

(4) By applying the *auxiliary variables rule*, $KL_i^c(b)$ is proved.

Theorem 2 (completeness): If $KL_i^c(b)$ is true then it can be proved to be true.

Proof: Recall that b is asserted to be non-temporal, and not to include knowledge assertions. It must be shown that if $KL_i^c(b)$ is true, then (in step(1)) a "strong enough" correctness proof can indeed be established. From the fact that the proof systems are complete [Ap,Ow], it follows that it is possible (although not practical) to establish a correctness proof in which for any n-tuple $(l_1, \ldots, l_n)$, $post(l_1, \ldots, l_n)$ is the *strongest post assertion* possible. Such a correctness proof is always "strong enough" [].

Example 3: Recall example 2. We prove $KL_1^c(y=1 \vee 2 \vee 3)$

P_1' ::

$\{i=0\}$

$a:[true \rightarrow x:=1 \;[]\; true \rightarrow x:=2];$

$\{(x=1 \vee 2) \wedge i=0\}$

$b:<P_2!x;i:=1>$

$\{i=1\}$

$c:<P_1?x;i:=2>$

$\{(x=2 \vee 3 \vee 4 \vee 5) \wedge i=2\}$

P_2' ::

$\{j=0\}$

$d:[true \rightarrow y:=1 \;[]\; true \rightarrow y:=2 \;[]\; true \rightarrow y:=3];$

$\{(y=1 \vee 2 \vee 3) \wedge j=0\}$

$e:<P_2?z;j:=1>$

$\{(y=1 \vee 2 \vee 3) \wedge (z=1 \vee 2) \wedge j=1\}$

$f:<P_2!(y+z);j:=2>$

$\{(y=1 \vee 2 \vee 3) \wedge (z=1 \vee 2) \wedge j=2\}$

$I \equiv i=j \wedge (i=2 \rightarrow x=y+z)$

Figure 2. Proof Outline.

step 1: To verify the program, two auxiliary variables i,j are used. The *proof outline* for the bracketed program P' is shown in figure 2.

step 2: $\Phi(c) \equiv post(c,init_2) \vee post(c,d) \vee post(c,e) \vee post(c,f)$
$\equiv false \vee false \vee false \vee$
$((x=2 \vee 3 \vee 4 \vee 5) \wedge i=2 \wedge (y=1 \vee 2 \vee 3)$
$\wedge (z=1 \vee 2) \wedge j=2 \wedge I \wedge after(c,f)).$

step 3: $\Phi(c)$ *implies* $y=1 \vee 2 \vee 3$, by property 1 $KL_1^c(y=1 \vee 2 \vee 3)$.

step 4: By applying the *auxiliary variables rule*, $KL_1^c(y=1 \vee 2 \vee 3)$ [].

It is easy to see how the method is applied for proving *state knowledge assertions* as well. Let Ψ be an assertion describing some location knowledge (of process i) after c (proved using the above method), and for some computation let s_i be an assertion describing the *visible state* of process i after c. By property (7), $\Psi \wedge s_i$ is *state knowledge* of process i after c for that computation. This method is not strong enough and in the next section we show a better way for proving state knowledge assertions.

Example: Recall example 2. We prove $KS_1^s(y=1 \vee 2)$. Let $\Psi \equiv (z=1 \vee 2) \wedge x=y+z$.

- $\Phi(c)$ *implies* Ψ. By property 1 $KL_1^c(\Psi)$.
- By applying the *auxiliary variable rule*, $KL_1^c(\Psi)$.
- By property 4, $KS_1^s(\Psi \wedge s)$. $(s \equiv \{x=3 \wedge after(c)\})$
- $\Psi \wedge s$ *implies* $y=1 \vee 2$; By property 1 $KS_1^s(y=1 \vee 2)$. [].

4.2 HISTORY KNOWLEDGE

We now present a proof method for proving history knowledge assertions as well. First the notion of *history outline* is defined. It generalize the notion of *cooperating proof outline* (i.e a collection of sequential proof outlines which pass the cooperation test), and that of a location outline. Once this is done, a theorem which links the definition of history knowledge and that of a history outline is formulated. We conclude by giving the proof method itself. We restrict ourselves to the distributed model. It should be possible to apply the ideas presented in the sequel also to the shared variables model.

A history h_i of process i is expressed as a finite sequence of assertions describing i's local states $<s_0, \cdots, s_j, s_{j+1}, \ldots, s_n>$, where s_0 describes i's initial state and s_{j+1} describes a successive state to s_j,. We say that history h_i implies *after*(c) iff s_n implies *after*(c).

Example 4: Consider example 2. A possible history for P_1 is $h_1 \equiv <s_0, s_1, s_2, s_3, s_4>$, where:

$s_0 \equiv \{after(init_1)\}$, $s_1 \equiv \{x=1 \wedge after(a\,1)\}$,

$s_2 \equiv \{x=1 \wedge after(a)\}$, $s_3 \equiv \{x=1 \wedge after(b)\}$,

$s_4 \equiv \{x=3 \wedge after(c)\}$.

We introduce the notion of a **history outline** (*h.o.*), which is a central concept to the proof method. Given history $h_i \equiv <s_0, \cdots, s_n>$, a *h.o.* is intuitively the assertions of an annotated program in which the information known from the given history, and its implications on the behavior of the rest of the processes, may be captured.

Assume an annotated program where each command $c:S$ of a process is preceded and followed by corresponding assertions $pre(c)$ and $post(c)$ respectively. We will define when those assertions form a correct *h.o*, with respect to a given history h. The definition is given using *marking rules*. Initially all assertions are assumed unmarked. Each assertion is expressed as a conjunction of conjuncts. A rule may allow marking a conjunct, and an assertion is marked when all its conjuncts are marked. The assertions form a correct *h.o.* with respect to history h iff it is possible to mark them all.

The marking rules

Let $post(c)$ denote the post assertion of a command labeled by c, and let ξ denote one of its conjuncts. ξ can be marked if it satisfies *at least one* of the following marking rules,

\# *First rule*: Let PO stand for a possible cooperating proof outline established for the above bracketed program using the usual proof system ([AFR]). Let $po(c)$ denote the post assertion belonging to PO, of the command labeled c.

* $po(c) \rightarrow \xi$. (From the previous section, this is equivalent to: * $KL_i^c(\xi)$)

Remark: The various conjuncts marked by this rule do not necessarily refer to the same PO.

The rule assures that information expressed within an ordinary cooperating proof outline, can also be expressed within a *h.o.*. It follows immediately that a correct cooperating proof outline is also a correct *h.o.* with respect to *any* given history.

An Invariant is proven in the usual way, and can then be marked.

\# *Second rule*: Let S be a command labeled by c.

* The weakest liberal pre-condition of ξ and S is implied by an already marked conjunct of $pre(c)$.

This means that for any sequential Hoare rule, if all premises are marked, the consequent can be marked. Through the rule a marked conjunct can affect its local environment.

\# *Third rule*: Let c be a label of the process i to which the history h is known ($h \equiv <s_0, \cdots, s_n>$), and let $\Theta_h(c)$ denote the disjunction of all s_j's such that $s_j \rightarrow after(c)$. (If for no s_j, $s_j \rightarrow after(c)$ then $\Theta_h(c) \equiv false$.)

* $\Theta_h(c) \rightarrow \xi$.

Through the rule, information from the history of a process is 'transferred' directly into the assertions of that process, with no further proof obligations.

\# *Fourth rule* (output): Let c be a label of an output command $c:P_j!expr$. Let Inp be the set of all labels of input commands which semantically match [5] with that output command. For each input command $d:P_k?y_d$ where $d \in Inp$, define ν_d as an assertion for which there are marked conjuncts of $post(d)$ which imply ν_d then,

* $\bigvee_{d \in Inp} \nu_d\,[expr/y_d] \rightarrow \xi$.

[5] The input command $P_k?y$ and the output command $P_j!expr$, taken from process j and process k respectively, are called a syntactic matching pair. A syntactic matching pair is also a *semantic* matching pair if there exists an execution in which a communication between the matching communication commands actually takes place. Using the ordinary proof system it is possible to distinguish which are the semantic matching pairs.

P_1' ::

$\{i=0\}$

$a:[true \rightarrow x:=1 \;[]\; true \rightarrow x:=2];$

$\{x=1 \wedge i=0\}$

$b:<P_2!x;i:=1>$

$\{i=1\}$

$c:<P_1?x;i:=2>$

$\{x=3 \wedge i=2\}$

P_2' ::

$\{j=0\}$

$d:[true \rightarrow y:=1 \;[]\; true \rightarrow y:=2 \;[]\; true \rightarrow y:=3];$

$\{j=0\}$

$e:<P_2?z;j:=1>$

$\{z=1 \wedge j=1\}$

$f:<P_2!(y+z);j:=2>$

$\{z=1 \wedge j=2\}$

$I \equiv i=j \wedge (i=2 \rightarrow x=y+z)$

Figure 3. History Outline.

It follows from the rule that the possible values of an output expression $-expr-$ **after** the communication must include all possible values of the (matching) input variables $-\vec{y_d}-$ after communication. The reason for referring to the value of the output expression after communication is due to the fact that it may not take place.

Example: Assume $P_j!expr$ has exactly two semantic matching input commands, $e:P_k?v$ and $f:P_k?w$. Let $post(e) \equiv \{(v=3 \vee 4) \wedge (w=7)\}$ and $post(f) \equiv \{(v=1 \vee 2) \wedge (w=5 \vee 6)\}$, and assume the first conjunct of $post(e)$ and the second conjunct of $post(f)$ are marked, then $\nu_e \equiv \{v=3 \vee 4\}$, $\nu_f \equiv \{w=5 \vee 6\}$ and $\nu_e[expr/v] \vee \nu_f[expr/w] \equiv \{expr=3 \vee 4 \vee 5 \vee 6\}$.

\# *Fifth rule* (input): Let c be a label of an input command $c:P_k?y$. Let *Out* be the set of all labels of output commands which semantically match with that input command. For each output command $d:P_j!expr_d$ where $d \in Out$, define ν_d as an assertion for which there are marked conjuncts of $post(d)$ which imply ν_d then,

* $\bigvee_{d \in Out} \nu_d[y/expr_d] \rightarrow \xi$.

It follows from the rule that the possible values of an input variable $-y-$ after communication include all possible values of the (matching) output expressions $-\vec{expr}_d-$ **after** communication.

Example: Assume $c:P_k?y$ has exactly two semantic matching outputs commands, $e:P_j!3$ and $f:P_j!(v+w)$. Let $post(e) \equiv \{true\}$ and $post(f) \equiv \{(v=1 \vee 2) \wedge (w=3 \vee 5)\}$ and assume it is marked, then $\nu_e \equiv \{3=3\}$, $\nu_f \equiv \{v+w=4 \vee 5 \vee 6 \vee 7\}$ and $\nu_e[y/3] \vee \nu_f[y/v+w] \equiv \{y=3 \vee 4 \vee 5 \vee 6 \vee 7\}$.

The *history outline* concept as described above is not complete, in the sense that it is not possible to capture all possible information induced by a given history. First $\Theta_h(c)$ does not reflect the relative order in which the local states appear. Second, the pre assertion of an output command may not be as strong as possible. This may happen when an input command semantically matches with several output commands. In this case, we might want to capture the fact that at least one of those outputs has to send a specific value. Such a fact can not be expressed in a *h.o.* as defined above. It seems that by wider use of auxiliary variables, this information might also be expressed.

Example 5: A correct *h.o.* for the bracketed program P' and history h_1 from example 4 is shown in figure 3.

Explanation: All conjuncts which refer to auxiliary variables are marked using the first rule, where as *PO* we use the proof outline presented in example 3.

This PO provides also a proof of the invariant I. Marking the rest of the conjuncts of assertions from P_1' are done by using the third rule (with history h_1). The conjunct $(z=1)$ of $post(e)$ is marked by using the fifth rule while the conjunct $(z=1)$ of $post(f)$ is marked using the second rule.

Remark: In the above example, the history h_1 of program P (example 2) is only a partial history of P', in that it does not refer to the auxiliary variables (*a.v.*). This should not cause any problem because any history can be uniquely extended to refer also to the exact values of the *a.v.* in each state of the sequence of states which form that history. This follows immediately from the role of the *a.v.* as history variables. In practice, as in the above example, it is not necessary to establish the extended history, because the conjuncts which refer to the *a.v.* can be marked using the first rule. To avoid cumbersome presentation we use h for both the history without references to *a.v.* and its unique extension. The actual use can be understood from the context.

For the rest of the section we refer to an annotated program, where the assertions form a correct history outline with respect to a given history h of process i. The previous notations $post(l_i)$ and $post(lc)$ refer now to that annotated program.

Lemma 2: For a history outline as defined above and for any n–tuple of labels lc, $post(lc)$ is a true predicate after lc for all traces which match with h.

Proof: omitted.

The following observation follows directly from the definition of a history location predicate $-Sh_{lc}-$ (see the end of section two).

Observation: Let h be the history of process i. If $h \rightarrow after(c)$ and $l_i \neq c$ then $Sh_{(l_1,\ldots,l_n)} \equiv false$.

Lemma 3: For any location set T_{lc} and history h, the associated history location predicate Sh_{lc} implies $post(lc)$.

proof: Sh_{lc} is, by definition, the *strongest* predicate known to be true after lc when only h is known, while from lemma 2, $post(lc)$ is a true predicate after lc for all traces which match with h. []

Theorem 3: If c is a label of process i, $h \rightarrow after(c)$, and $\Phi(c)$ is the *disjunction* of all assertions $post(l_1,\ldots,l_n)$ such that $l_i \equiv c$, then $KH_i^h(\Phi(c))$.

Proof:

(1) $h \rightarrow after(c)$. [Assumption]

(2) $\forall\, lc\ .\ Sh_{lc} \rightarrow post(lc)$. [Lemma 2]

(3) $\forall\, lc\ .\ (post(lc) \wedge after(c)) \rightarrow \Phi(c)$. [Def. of $\Phi(c)$]

(4) $\forall\, lc\ .\ Sh_{lc} \rightarrow (post(lc) \wedge after(c))$. [(1),(2),Obs.]

(5) $\forall\, lc\ .\ Sh_{lc} \rightarrow \Phi(c)$. [(3), (4)]

(6) $KH_i^h(\Phi(c))$. [(5), Def. of history knowledge]

[]

Now suppose it is to be proved that $KH_i^h(b)$ is true for a given program P and that $h \rightarrow after(c)$. The proof involves four steps similar to those used for location knowledge.

(1) A "strong enough" *h.o.* (with respect to h) is established for P'.

(2) Assertion $\Phi(c)$, as defined above, is constructed. [by Theorem 3 $KH_i^h(\Phi(c))$].

(3) It is proved that $\Phi(c)$ implies b. [by property 1 $KH_i^h(b)$].

(4) By applying the *auxiliary variables rule*,[6] $KH_i^h(b)$ is proved.

Example 6: Consider examples 2 & 4. We prove $KH_1^{h_1}(y=2)$.

step 1: See *h.o.* presented in example 5.

step 2: $\Phi(c) \equiv post(c,init_2) \vee post(c,d) \vee post(c,e) \vee post(c,f)$
$\equiv false \vee false \vee false \vee (x=3 \wedge z=1 \wedge i=j=2 \wedge I \wedge after(c,f))$.

[6] An auxiliary variable rule similar to the one presented in the previous section is used.

step 3: $\Phi(c)$ *implies* $y=2$, by property 1 $KH_1^{h_1}(y=2)$.

step 4: By applying the *auxiliary variable rule*, $KH_1^{h_1}(y=2)$ □.

It is easy to modify the proof method just presented to prove knowledge assertions when only a partial history is given. The modification which is needed is in how a correct (partial) *h.o.* is established in such cases. This will involve redefining the third marking rule, which showed how to satisfy proof obligations directly from the history. Now only a partial history can be used.

As an example let us modify the proof method, so as to prove state knowledge assertions. Let s denote the given state. The modified third marking rule takes the following form:

\# *Third rule*: * $s \rightarrow (\xi \wedge after(c))$.

Defining the (usual) four step proof method is now obvious.

In general if the current location is not given as part of the partial history then process i can know the disjunction of $\Phi(c)$ over all its possible locations.

Note that now it will make sense to mark preconditions of statements on the basis of the marking of its postcondition (in the process whose partial history is given), at least back to an input/output statement. Previously there was no need to do such 'backward marking' since the entire history was available.

Modifying the proof method for proving location knowledge assertions, involves omitting the third marking rule completely. Note that in that case, as expected, the resulting marking rules[7] define what has previously been defined as a location outline.

5. NESTED KNOWLEDGE

An extension of the knowledge definition and the proof method to deal with knowledge about knowledge is natural. In order to prove an assertion like: process i knows after c that process j knows p, without further information about the exact location of process j ($KL_i^c(KL_j(p))$), one must prove that process j knows p at all locations possible when process i is after c. That is for every label a of process j either $KL_i^c(\neg after(a))$ or $KL_j^a(p)$. Again, considering example 2, such an assertion is: $KL_2^s(KL_1(x=z))$.

More generally, we consider a claim of the form $KH_i^h(b)$ where b *may* include other knowledge operators (but is still non-temporal). The assertion b often (but not necessarily) will include the form $KH_j(p)$, without a specific history as a parameter. This means that for every local history h' of process j which matches with one of the possible traces of the system, $KH_j^{h'}(p)$. It is also possible to consider history knowledge with only partial histories. For example, $KH_j^s(p)$ (s - for state) means that process j will have available a local history h' which can be used to show p true, but h' is not given as a parameter. Thus, h' may be any history of process j which matches with a trace of the system and also is consistent with the state s. Note that, at least when appearing in the scope of an outer knowledge operator, this differs from $KS_j^s(p)$, where process j does not have a local history available.

Let us concentrate, as an example, on assertions of the form $KH_i^h(KH_j(p))$, where p does not contain knowledge operators. This means that, given local history h, process i knows that process j (history) knows p, without further information about the actual history of process j. Thus this assertion is true only if process j knows p with any of its possible histories. Although process i cannot in general know the entire actual local history of process j, it can know (by using h) various assertions which will be true of any local history of process j which is consistent both with the possible traces of the system and with h. Such assertions can be proven by devising a history outline for h, and properly marking all of the conjuncts as described previously. This will be in fact the first stage in proving

[7] actually the first marking rule is enough.

$KH_i^h(KH_j(p))$. The following claim is then crucial to the rest of the proof method: informally, the actual local history of process j must satisfy all marked assertions annotated in process j, from (any) history outline established with respect to the above h. The following lemma express this idea formally.

Let h and h' denote for the rest of the section, local histories of processes i and j respectively. Let $post_h(c)$ denote a post assertion of a command labeled by c, which can appear in some history outline established with respect to history h, and recall the definition of $\Theta_h(c)$ from the previous section.

Definition: Two histories h and h' as above, are consistent *iff* there exists a trace t such that both $match(t,h)$ and $match(t,h')$ hold.

Lemma 4: If two histories h and h' as above, are consistent then for each label c of process j, $\Theta_{h'}(c) \rightarrow post_h(c)$.

Proof: Omitted.

Next we describe how to establish the second stage in proving a nested knowledge assertion, again using marking rules. We require a (single) history outline which will be correct with respect to *every* history of process j which is consistent with the given history h of process i. The aim of this is to be able to show $KH_j(p)$ in context. Its correctness will follow from lemma 4. The intuition is to allow marking the assertions annotated in process j if they can appear in the history outline (with respect to the above h) from the first stage. Formally, the assertions of the annotated program in the new proof form a correct (multi) history outline as described above, if they can be marked using the marking rules from the previous section, where the third rule is replaced by:

\# *Third rule*: Let c be a label of process j, and let $post_h(c)$ denote, an assertion from a history outline previously established with respect to h.

* $post_h(c) \rightarrow \xi$.

Now, once such a (multi) history outline is established, we can deduce from it the truth of the needed assertion in the same manner as described in the beginning of this section for location knowledge.

More formally, let $\Phi(a)$ be defined as before but with respect to the history outline just described. From theorem 3 and lemma 4, it follows that for *every* history h' (of process j) consistent with the above h such that $h \rightarrow after(a)$, $KH_j^{h'}(\Phi(a))$. Thus, in order to prove $KH_i^h(KH_j(p))$, one must prove that for every label a of process j, either $KH_i^h(\neg after(a))$ or $\Phi(a) \rightarrow p$.

In general, for $KH_i^h(b)$, the proof of b will be as previously described, except that certain conjuncts can be marked immediately if they define assertions true of any history, state, or location which is available in b and is consistent with h. This process can be continued recursively, for an arbitrarily deep nesting of knowledge operators.

Example 7: Consider, yet again, the program from the previous examples. Using the technique described above, we can easily show that $KH_1^{h_1}(KH_2(x=3))$.

From the history outline established in example 5, it follows that $KH_1^{h_1}(\neg after(init_1))$, $KH_1^{h_1}(\neg after(d))$, and $KH_1^{h_1}(\neg after(e))$. Next a correct history outline with respect to all histories of process j consistent with h_1 is established. This is done by adding $y=2$ as a conjunct of $post(f)$ and retaining only those conjuncts in process P_1 which refer to auxiliary variables. With respect to that history outline, $\Phi(f) \rightarrow (x=3)$. []

6. CONCLUSIONS

In this paper we have shown how to prove knowledge assertions under a variety of knowledge assumptions about the available local information, assuming the text of the program as global information. We have generalized existing proof techniques, by relaxing the proof obligations for assertions which follow from the given knowledge assumption, and shown

what knowledge follows from such a proof.

The motivation for this work is in the specification and design of distributed programs, and in future work we plan to demonstrate the utility of knowledge for a variety of tasks.

Acknowledgement: We would like to thank Nissim Francez for helpful discussions on the subject.

REFERENCES

[AFR] Apt,K.R., Francez, N., and de Roever, W.P. A proof system for communicating sequential processes, *ACM–TOPLAS*, 2,3 1980, 359-385.

[Ap] Apt,K.R., Formal justification of proof system for communicating sequential processes. *JACM* 30,1 1983, 197-216.

[B] Barringer, H. A survey of verification techniques for parallel programs, *LNCS* 191 1985.

[BHR] Brookes, S.D., Hoare C.A.R., Roscoe A.W. A theory of communicating sequential processes, *JACM* 31,3 1984, 560-599.

[CM] Chandy, M., and Misra, J. How processes learn, *ACM–PODC* 1985, 204-214.

[FHV] Fagin, R., Halpern, J., and Vardi, M. A model theoretic analysis of knowledge, IEEE-FOCS 1984, 268-278.

[HF] Halpern, J., and Fagin, R. A formal model of knowledge, action, and communication in distributed systems: preliminary report, *ACM–PODC* 1985, 224-236.

[HM1] Halpern, J., and Moses, Y. Knowledge and common knowledge in a distributed environment, *ACM–PODC* 1984, 50-61.

[HM2] Halpern, J., and Moses, Y. A guide to the modal logic of knowledge and belief, *IJCAI* 1985.

[Le] Lehmann, D. Knowledge, Common Knowledge and related puzzles, *ACM–PODC* 1984, 62-67.

[Ow] Owicki, S. Axiomatic proof technique for parallel programs, Computer Science Dept., Cornell University, Ph.D. thesis, 1975.

[OG] Owicki, S., and Gries, D. An axiomatic proof technique for parallel programs, *I. Acta Inf.* 6, 1976, 319-340.

[OL] Owicki, S., Lamport, L. Proving liveness properties of concurrent programs, *ACM–TOPLAS* 4,3 1982, 455-495.

[LSP] Lamport, L., Shostak, R., Pease, M. The Byzantine General Problem, *ACM–TOPLAS* 4,3 1982, 382-401.

[M] Misra, J. Reasoning about network of communicating processes, *Proc. Advanced NATO Institute on Logic* and *Models* for *Verification* and *Specification of Concurrent Systems*, Oct. 1984.

[PR] Parikh, R., and Ramanujam, R. Distributed processes and the logic of knowledge: preliminary report, *LNCS* 193, 256-268.