Layers as Knowledge Transitions in the Design of Distributed Systems

Wil Janssen*†‡

ABSTRACT Knowledge based logics allow to give *generic* specifications of classes of network protocols. This genericity is combined with methods to derive sequentially structured or *layered* implementations of distributed algorithms. Knowledge based logic is used to specify layers in such algorithms as *knowledge transitions*. The resulting layered implementations are transformed to *distributed* algorithms by means a transformation rule based on the principle of *communication closed layers*.
In this way a class of solutions to a problem for different architectures can be derived along the same lines simultaneously. This design technique for distributed algorithms is applied to a number of examples including different versions of the Two-Phase Commit protocol.

1 Introduction

The design and analysis of distributed systems is a complicated task. Many different processes can be active simultaneously and communicate in a seemingly unstructured way, communication protocols are intertwined with the basic program, and different system architectures can result in completely different algorithms. Over the last few years there have been a number of attempts to solve these problems concerning the specification and design of distributed systems. One of the possible approaches is to remove all architectural decisions from the specification language, in order to be able to concentrate on the algorithmic aspects. This approach has been taken in, for example, *action systems* or *IO-automata* [5, 17, 3, 20, 19].
A second approach is the use of *knowledge-based* or *epistemic* logics and language constructs [11, 12, 21, 10]. The use of knowledge-based logics allows to express properties of systems and actions in a more global way,

*This work has partially been supported by Esprit/BRA Project 6021 (REACT).
†Most of the work reported on was done when the author was working at the University of Twente.
‡University of Oldenburg, Fachbereich Informatik, Postfach 2503, D-26111 Oldenburg, Germany. Phone: + 49 441 7982362; E-mail: Wil.Janssen@informatik.uni-oldenburg.de.

abstracting away from communication structures and architectural decisions.
Finally, there has been a considerable amount of attention to the use of ***layered methods*** in the design of distributed systems [8, 27, 28, 6, 18, 15, 31, 13]. It has been observed that in many protocols in distributed systems the logical structure of the system is basically a sequential one, whereas the actual structure is distributed and depends very much on the details of the implementation architecture. By viewing the algorithm as a sequentially structured system, analysis becomes much simpler and is more or less the same for larger classes of protocols, instead of being applicable to a single algorithm only.
In this paper we combine the above observations. We use the fact that many systems can be designed and analyzed in a layered fashion, plus the fact that knowledge-based logics allow for a specification of such layers at an appropriate level of abstraction, that is, as *knowledge transitions.*

Knowledge concerns facts that we associate a *location* or *distribution* with. Facts can be known to a certain process or set of processes. Knowledge can exist in different ways: *distributed knowledge* is knowledge of the group of processes as a whole. It concerns facts that would be known if all processes would combine their information.
The strongest level of knowledge is *common knowledge,* which informally corresponds to facts that are "publicly known." For example, in systems with reliable communication it is common knowledge that no messages are lost. States of knowledge are expressed using a set of modalities, K, D, S, E, and C. Let G be a group of processes, or *agents* as they are usually called in this context. The expression $K_i\varphi$ states that process i *knows* the proposition φ. $S_G\varphi$ states that *somebody* in the group G knows φ, and $E_G\varphi$ gives that *everybody* in G knows φ. Finally, $D_G\varphi$ states that it is *distributed* knowledge in G that φ holds, which means that if we combine the knowledge of every process $i \in G$ we can derive φ, and $C_G\varphi$ that it is *common* knowledge in G that φ holds. In this paper common knowledge will not play an important role and is not discussed further.

Protocols, distributed algorithms, and conceptual layers in them can often be described as transitions from one state of knowledge to another. A transition

$$\Phi \leadsto \Psi$$

states that if we start in a state satisfying Φ, on termination we will be in a state satisfying Ψ. Therefore, knowledge transitions can be viewed as a generalization of Hoare style preconditions and postconditions to knowledge based assertions. For example, broadcasting protocols can be specified as a transition from a state of knowledge where one process i (the broadcaster) knows a fact φ to a state where all processes in the set G of participating processes know the same fact. So it is a transition of the form

$$K_i\varphi \leadsto E_G\varphi.$$

If we do not know the identity of the broadcaster this would result in

$$S_G\varphi \leadsto E_G\varphi.$$

(In fact, this is a simplification. There must also be some common knowledge in the system for this to hold but this is beyond the scope of this paper. See Halpern and Moses [11] for details.) Often parts of protocols are used to gather information of all processes to a single coordinating process. This means that from a state where every process i knows some fact φ_i, the system evolves to a state where a single process c knows all these facts:

$$\wedge_{i\in G} K_i\varphi_i \quad \leadsto \quad K_c(\wedge_{i\in G}\varphi_i),$$

or stated differently:

$$D_G(\wedge_{i\in G}\varphi_i) \quad \leadsto \quad K_c(\wedge_{i\in G}\varphi_i).$$

Larger protocols can often also be specified in such a manner. Take for example *atomic commit* protocols for distributed databases (see Bernstein, Hadzilacos and Goodman [4] for an overview of this field). Informally speaking, the protocol has to make a decision for a set of participating processes, based on the internal state of those processes. Every process P_i can decide locally whether or not it can make the changes made in a transaction permanent. The protocol decides to commit iff all processes can do so. If one or more processes cannot, it should decide to abort in order to keep the data at the different processes consistent. The decision should be made known to all processes which will then take the appropriate actions. The internal state is reflected in a vote YES or NO for every process i, such that $vote_i = \text{YES}$ iff changes can be made permanent. Such a protocol can be specified as the following knowledge transition. Let $total\_vote \coloneqq \text{YES}$ iff $\wedge_{i\in G} vote_i = \text{YES}$.

$$\bigwedge_{i\in G} K_i vote_i \leadsto \bigwedge_{i\in G} K_i(total\_vote \wedge (dec_i = \text{COMMIT} \Leftrightarrow total\_vote = \text{YES})).$$

Here, $K_i vote_i$ means that i knows the value of $vote_i$. This in fact abbreviates $K_i(vote_i = \text{YES}) \vee K_i(vote_i = \text{NO})$.

The approach we introduce in this paper is the following. Given a specification of a problem (using knowledge modalities), we refine this specification to a *sequence of knowledge transitions*. For example, the above transition can be split into three simpler transitions are follows:

$$\begin{aligned}\bigwedge_{i\in G} K_i vote_i \quad &\leadsto \quad K_c total\_vote\\ &\leadsto \quad E_G total\_vote\\ &\leadsto \quad \bigwedge_{i\in G} K_i total\_vote \wedge (dec_i = \text{COMMIT} \Leftrightarrow total\_vote = \text{YES}).\end{aligned}$$

These knowledge transitions are then *instantiated* with protocol layers that are suited for the architecture under consideration and implement the knowledge transitions specified. The result of this is an algorithm that consists of a *sequence of layers*.
Such an algorithm can then be *transformed* to a parallel or distributed algorithm, using the techniques developed by Janssen, Poel and Zwiers as discussed in, for example, [16, 25, 31, 13]. This transformation is based on the principle of *communication closed layers* as introduced by Elrad and Francez [8], translated to an algebraic setting. After some optimizations the transformed system results in an algorithm that solves the original problem and is tailored to a certain implementation architecture.
In order to be able to take this approach, we give a *classification* of knowledge transitions, and of the ways such transitions can be implemented for different architectures. As such, the knowledge transitions serve as a vehicle for the abstract specification of protocol layers. By taking different refinements of the problem specification using different transitions, different implementations of the problem can be obtained along the same lines, thus emphasizing the similarities and characteristics of the implementations.
The applicability of such layered approaches in general (not particularly using knowledge transitions) has been shown by numerous examples, such as distributed minimum weight spanning tree algorithms, parallel parsing, parts of a caching algorithm, pipelining, real-time mutual exclusion, and minimal distance algorithms.

The outline of this paper is as follows. We first discuss our process language and knowledge based logic with their semantics, and a transformation principle for programs. Thereafter we introduce knowledge transitions and classify well-known communication structures for different networks as knowledge transitions. Finally we explain how to derive algorithms as sequences of knowledge transitions and apply this to different versions of the Two-Phase Commit protocol and to so-called waves.

2 Programs, communication closedness and knowledge

Many protocols and distributed algorithms are given in a setting of asynchronous message passing. In this paper we restrict ourselves to this form of communication, in order to simplify technical details that would divert the attention from the main issues of this paper. We use a normal process language with choice, sequential composition and parallel composition, and asynchronous message passing. This language is extended with the notions of *layers* and *layer composition*.

Systems consist of a number of components with local variables that communicate using *send*(c, e) and *receive*(c, x) actions, where c is a chan-

nel, e is an expression and x is a variable. Channels connect two unique processes and are unidirectional. Often a channel is therefore represented as a pair (v, v') of nodes or processes. Channels are viewed as *single place* buffers.
Besides communication actions and assignments $x := e$ to local variables, our programming language includes conditionals of the form **if** b **then** S **else** T **fi**. If T is omitted it is assumed to be **skip** (do nothing). Actions can be composed by means of parallel composition "$\|$" and sequential composition " ; ".

Any process that is composed out of the constructs above is called a *layer*. Layers can be composed by means of *layer composition* "$\bullet$". Informally speaking, when we compose two layers S and T by means of layer composition the resulting process $S \bullet T$ executes actions a of S before actions b of T iff a and b are *dependent*. Two actions are dependent, denoted by $a \leftrightsquigarrow b$, iff they access the same (local) variables, or access the same channel. So layer composition can be seen as an intermediate between sequential composition, where full ordering between S and T would be specified, and parallel composition, where ordering between dependent actions of S and T can be in an arbitrary direction, not necessarily from S to T. As such, layer composition cannot directly be translated into well-known program constructs, but it serves as a specification construct in the initial and intermediate design stages. Moreover, layer composition has nice algebraic properties that make it well-suited for a transformational style of program derivation. Please refer to the work by Janssen, Poel and Zwiers, for example [15, 31, 13], for detailed discussions thereof.

Layered programs and communication closed layers

One of the most important algebraic properties that relies on the use of layer composition is the so-called *communication closed layers law* CCL. It is based on the principle of communication closed layers as introduced by Elrad and Francez [8]. This law states that under a certain side condition, a *layered* or sequentially structured system $(P \parallel R) \bullet (Q \parallel S)$ behave the same (has the same semantics) as the *parallel* system $(P \bullet Q) \parallel (R \bullet S)$. The side condition is that there exist no "cross-dependencies" between components in different layers. Formally, assume for processes P, Q, R, and S, that P and S are independent, and that Q and R are independent ($P \not\leftrightsquigarrow S$ and $Q \not\leftrightsquigarrow R$). Under this assumption we have

$$(P \parallel R) \bullet (Q \parallel S) \quad = \quad (P \bullet Q) \parallel (R \bullet S) \qquad\qquad (\textsc{ccl})$$

This law can be generalized to more processes and more layers of course.

The idea is to derive layered implementations that satisfy this side condition, and to transform these to distributed implementations. In general this side condition does not hold for systems consisting of a number of layers, as

different layers can have common channels leading to cross-dependencies. In order to circumvent these problems, we temporarily introduce *virtual channels* per layer, for example by replacing every channel C in layer l by a channel C_l. The resulting process is equivalent to the original one. Thereafter the CCL law trivially applies, as all dependencies are either within a single layer or between different layers but within the same process.
After transforming the renamed system to a parallel system, we can replace the layer composition by sequential composition and replace the virtual channels by again a single channel per edge by means of *multiplexing techniques* (see [13, 31]). These multiplexing techniques do not always apply. In this setting a sufficient condition is to ensure that in every layer every *send* is matched by a *receive* action for the same channel for every possible evaluation of conditionals. Informally speaking this implies that channels are empty at the end of a layer, and therefore *receive* actions in other layers will read the values sent in the layer they belong to. Multiplexing and replacing layer composition by sequential composition do not preserve semantic equality. They do however preserve the input/output behavior of systems, that is, if viewed as pairs of initial and final states the systems are the same. This is called *IO-equivalence.*

The combined result of the above steps is summarized by the following transformation principle.

Let S be a system consisting of a number of layers

$$S \triangleq L(0) \bullet L(1) \bullet \cdots \bullet L(n),$$

where every layer is of the form

$$L(l) \triangleq \textbf{for } i \in G \textbf{ par } P(i,l) \textbf{ rof},$$

with every *send* action matched by a *receive* action, for all possible evaluations of the conditionals. Assume all components communicate by means of asynchronous message passing only. Then S is IO-equivalent to the system S'

$$S' \triangleq \textbf{for } i \in G \textbf{ par } P(i) \textbf{ rof},$$

where

$$P(i) \triangleq P(i,0) \,;\, P(i,1) \,;\, \cdots \,;\, P(i,n).$$

Knowledge based logic

Knowledge based or epistemic logic [11, 10, 21] is a class of modal logics that allow to add some notion of locality to formulae. We cannot only say that φ holds, but also that φ holds for a process or agent i, or is a fact that holds for the combined states of a group G of processes, so-called *distributed* or *group* knowledge.

The basic modality is K, which stands for knowledge. The formula $K_i\varphi$ states that process i knows φ. $K_i\varphi$ holds in a states s such that the local state part s_i of s for process i satisfies φ. Knowledge of different processes can be combined. We say that φ is distributed knowledge for a group of processes G iff φ holds for the combined states of all processes $i \in G$. For example, if $K_i x = 1$ and $K_j y = 2$, then $D_{\{i,j\}} y = x + 1$.
Formally speaking, we use the following logic. Assume a given non-empty set P of propositional constants and let A be a finite set of agents or processes. The set $L_A(P)$ of epistemic formulae $\varphi, \psi, \ldots$ is the smallest set closed under

- If $p \in P$ then $p \in L_A(P)$;
- If $\varphi, \psi \in L_A(P)$, then $\varphi \wedge \psi \in L_A(P)$ and $\neg\varphi \in L_A(P)$;
- If $G \subseteq A, i \in A, \varphi \in L_A(P)$, then $K_i\varphi \in L_A(P), D_G\varphi \in L_A(P)$.

As usual, we define implication "$\Rightarrow$" and disjunction "$\vee$" as abbreviations. Also, *true* and *false* abbreviate $p_0 \vee \neg p_0$ and $p_0 \wedge \neg p_0$ for some constant $p_0 \in P$ respectively. Finally the modalities "E" and "S" are defined as abbreviations as well. The modality E_G states that *everybody* in G knows a certain proposition, and the modality S_G states that *somebody* in G knows a certain proposition. They are defined as

$$\begin{aligned} E_G\varphi &\equiv \bigwedge_{i\in G} K_i\varphi, \\ S_G\varphi &\equiv \bigvee_{i\in G} K_i\varphi. \end{aligned}$$

The basic modalities are characterized by a number of axioms and rules. (See, for example, Meyer, van der Hoek and Vreeswijk [21] or Fagin et al. [10] for detailed discussions.)

$K_i\varphi \Rightarrow \varphi$ $D_G\varphi \Rightarrow \varphi$	knowledge axioms
$(K_i\varphi \wedge K_i(\varphi \Rightarrow \psi)) \Rightarrow K_i\psi$ $(D_G\varphi \wedge D_G(\varphi \Rightarrow \psi)) \Rightarrow D_G\psi$	consequence closure
$K_i\varphi \Rightarrow K_iK_i\varphi$ $D_G\varphi \Rightarrow D_GD_G\varphi$	positive introspection
$\neg K_i\varphi \Rightarrow K_i\neg K_i\varphi$ $\neg D_G\varphi \Rightarrow D_G\neg D_G\varphi$	negative introspection

$$\frac{\varphi}{K_i\varphi, \quad D_G\varphi} \qquad \text{knowledge generalization}$$

In the following we use a number of properties of the logic. Let $i \in G$, and let G be a subset of G'. Then

$$\begin{aligned}
K_i\varphi &\Rightarrow D_G\varphi, \\
K_i\varphi &\Rightarrow S_G\varphi, \\
D_G\varphi &\Rightarrow D_{G'}\varphi, \\
E_{G'}\varphi &\Rightarrow E_G\varphi, \\
E_G\varphi &\Rightarrow S_G\varphi, \\
K_i(\varphi \wedge \psi) &\Leftrightarrow K_i\varphi \wedge K_i\psi, \\
D_G(\varphi \wedge \psi) &\Leftrightarrow D_G\varphi \wedge D_G\psi, \\
K_i(\varphi \vee \psi) &\Leftarrow K_i\varphi \vee K_i\psi.
\end{aligned}$$

Note that the latter implication is not an equivalence. As a counter example that K_i*true*, which obviously holds. However, $K_i\varphi \vee K_i\neg\varphi$ is *not* a tautology. Process i need not know whether φ holds or whether its negation holds.

We use K_ix to state that i knows the value of x, which abbreviates $\bigvee_{v \in Val} K_ix = v$, if x takes its value from Val.

In the next two paragraphs we define the semantics of processes and the semantics of the logic. Knowledge thereof is not needed to understand what follows and they can be skipped on first reading.

Semantics of processes

Any run h of the system can be represented as a partially ordered set of events $h = (V, \rightarrow)$, where events are different occurrences of actions. Every event e has as an attribute its process identity $Id(e)$. Furthermore, events have a *read set* $R(e)$ and a write set $W(e)$, consisting of the variables read and written plus their values respectively. Two events e, e' are *dependent*, denoted by $e \leftrightsquigarrow e'$, iff one event writes a variable the other accesses as well.

Every run should be *dependency closed*, that is, for events $e, e' \in V$, if $e \leftrightsquigarrow e'$ then either $e \rightarrow e'$ or $e' \rightarrow e$. Cycles in the ordering are not allowed, as $(V, \rightarrow)$ is an (irreflexive) partial order.

Channels also fit into this framework: A channel c is modeled by a pair $(c.\mathit{flag}, c.\mathit{val})$, where $c.\mathit{flag}$ is a boolean variable, and $c.\mathit{val}$ is a variable of the same type as the messages to be sent. A send $\mathit{send}(c, e)$ of message e along channel c and a $\mathit{receive}(c, x)$ can be defined as abbreviations of the following (guarded) assignments.

$$\begin{aligned}
\mathit{send}(c, f) &\triangleq \textbf{await } \neg c.\mathit{flag} \textbf{ do } c.\mathit{flag}, c.\mathit{val} := \mathit{true}, f, \\
\mathit{receive}(c, x) &\triangleq \textbf{await } c.\mathit{flag} \textbf{ do } c.\mathit{flag}, x := \mathit{false}, c.\mathit{val}.
\end{aligned}$$

In this paper we do not take deadlock into account. Moreover we give only a short overview of the semantics. For a detailed account see [13, 31].

The semantics of processes is given by a function $[\![\cdot]\!] : S \longrightarrow \mathcal{H}$, where $\mathcal{H}$ is the domain of sets of partially ordered event sets. The **skip** action results in the empty run $(\emptyset, \emptyset)$, also denoted as "$\oslash$". Therefore

$$[\![\textbf{ skip }]\!] \overset{\text{def}}{=} \{\oslash\}.$$

The semantics of (guarded) assignments is the set of runs $\{(\{e\}, \emptyset)\}$ where e is an event that reads the variables in the guards and the expression such that the guard evaluates to *true*, and writes the variables in the assignment. As an example, events e corresponding to $receive(c, x)$ have

$$R(e) \;=\; \{(c.\mathit{flag}, \mathit{true}), (c.\mathit{val}, n)\},$$

and

$$W(e) \;=\; \{(c.\mathit{flag}, \mathit{false}), (x, n)\},$$

for some value of n. Obviously, events accessing the same channel are dependent. Conditionals **if** b **then** S **else** T **fi** result in a choice between either b ; S or $\neg b$; T, where b is an empty assignment guarded by b.

Sequential composition of two runs is obtained by taking the disjoint union of the sets of events, and augmenting the order correspondingly.

$$(V_0, \rightarrow_0) \,;\, (V_1, \rightarrow_1) \overset{\text{def}}{=} (V_0 \uplus V_1, \rightarrow_0 \uplus \rightarrow_1 \uplus (V_0 \times V_1)).$$

The sequential composition of two sets of runs is obtained by pointwise extension. Therefore the semantics of a sequentially composed term is obtained by the sequential composition of the semantics of the components. Note that we do not require that the states of the corresponding runs match in some sense. This in order to allow events from other components to "interfere".

For layer composition only *minimal* order to obtain dependency closedness is added.

$$(V_0, \rightarrow_0) \bullet (V_1, \rightarrow_1) \overset{\text{def}}{=} (V_0 \uplus V_1, \rightarrow_0 \uplus \rightarrow_1 \uplus ((V_0 \times V_1) \cap \leftrightsquigarrow)).$$

Parallel composition is similar to layer composition in that is requires minimal extension of ordering only, but ordering can be chosen arbitrarily. Therefore it results in a *set of runs.*

For *closed systems,* that is, systems without any processes running in its environment, we require that they are *state consistent.* This means that any event should read the last value that was written before it for any variable x, and that all initial reads should be consistent as well. Note that events writing into the same variable are dependent and therefore ordered. Thus the last written value is well-defined. Let

$$[\![S]\!]_S \overset{\text{def}}{=} \{h \in [\![S]\!] \mid h \text{ is state consistent }\}.$$

With any process i we associate a set of variables Var_i consisting of the local variables of i and the channels it is connected to. Let Var be the set of all variables and channels. A state s is a mapping from Var to values. It can be represented as a tuple $(s_1, s_2, \ldots, s_n)$ of local states of processes, where we require that if $c \in Var_i \cap Var_j$, then $s_i(c) = s_j(c)$, that is, the local states are consistent for shared variables. (In this case, the only variables shared are the channels.)
With any state-consistent run h we can associate a final state, given an initial global state s that is consistent with h. This is represented by $state(s, h)$. Informally speaking, this states gives the final values read or written in h, or the value a variable has in the initial state s if it is not accessed in h.

Semantics of the logic

The semantics we give in this section is a straightforward adaptation of the semantics given by, for example, Fagin et al. in [10], to our partial order framework.

As basic assertions Φ we use are first order formulae over Var. We could extend this with propositions of the form "i received a value from j" etcetera, but in the context of this paper this is not needed. The only special assertions we use are $full(c)$ and $val(c)$, denoting that channel c stores a message and the value of that message, respectively. These can be viewed as abbreviations of basic assertions, as channels are implemented as pairs of shared variables. We assume a given interpretation π mapping states plus assertions to $\{true, false\}$.

Let S be a system with $[\![S]\!]_S = H$. Given H and π we define an equivalence relation "$\sim_i$" on states for every process i as follows

$$s \sim_i s' \text{ iff } s_i = s'_i,$$

where we view s and s' as tuples. This equivalence partitions the states of the processes in classes such that two states are in the same class iff process i cannot distinguish between them.
We now define the semantics of the logic inductively. Formally speaking, the equivalence relations "$\sim_i$" will act as the accessibility relations $\mathcal{K}_i$ in a Kripke Structure

$$(\mathcal{S}, \pi, \mathcal{K}_1, \ldots, \mathcal{K}_n),$$

where $\mathcal{S}$ is the set of states of the system. Let $p \in \Phi$, and let s be a global state.

$$((H,\pi), s) \models p \overset{\text{def}}{=} \pi(s)(p),$$

$$((H,\pi), s) \models K_i\varphi \overset{\text{def}}{=} \forall h \in H, s' \text{ consistent with } h.$$
$$state(s', h) \sim_i s \Rightarrow ((H,\pi), state(s', h)) \models \varphi,$$

$$((H,\pi),s) \models D_G\varphi \overset{\text{def}}{=} \forall h \in H, s' \text{ consistent with } h. \; (\forall i \in G.\; state(s',h) \sim_i s) \Rightarrow ((H,\pi), state(s',h)) \models \varphi.$$

As usual, let $(H,\pi) \models \varphi$ be defined as

$$\forall s.\; ((H,\pi),s) \models \varphi,$$

and $\models \varphi \overset{\text{def}}{=} \forall (H,\pi).\; (H,\pi) \models \varphi$.

3 Knowledge transitions and communication structures

We have discussed our process language and knowledge based logic. In the introduction we have argued informally that protocols or protocol layers can sometimes be viewed as transitions from one state of knowledge to another. In this section we give an overview of possible knowledge transitions and classify well-known communication structures as knowledge transitions.

Not all knowledge transitions make equally much sense. The transition $K_i\varphi \rightsquigarrow E_G\varphi$ intuitively corresponds a broadcast-like protocol where φ is sent to all processes. A transition such as $K_i\varphi \rightsquigarrow D_G\varphi$ however makes less sense: if $i \in G$ this is immediately fulfilled without any communication.
In table .1 the transitions are summarized. Every entry in the table gives the relation between φ and ψ for which that knowledge transition makes sense for different relations between i, j, G and G'. In the general case, protocol layers can also lead to an increase in knowledge due to the fact that, for example, a process knows from whom it has received messages. This increase in knowledge is not reflected in this table, only the way φ directly relates to ψ is given. Transitions that have a non-trivial implementation are named in this table. The entry "skip" means that the transition is trivially satisfied by doing nothing.

The roles of $\wedge K_i$ and of E_G are often similar, due to the similarities in their definitions. Furthermore, there is a correspondence between D_G and $\wedge K_i$, as we can observe in the table. We can roughly distinguish four different types of transitions:

- *Broadcast, distribute* or *notify* transitions. These distribute information that is known for a certain process or set of processes to a larger set of processes.
- *Centralize* or *search* transitions. In this case information from different processes is gathered to a single process or different set of processes.

$\rightsquigarrow$	$K_j\psi$	$S_{G'}\psi$	$\bigwedge_{G'} K_i\psi_i$	$E_{G'}\psi$	$D_{G'}\psi$
$K_i\varphi$	$i = j : \varphi \Rightarrow \psi$, skip $i \neq j : \varphi \Rightarrow \psi$, notify	$i \in G'$: $\varphi \Rightarrow \psi$, skip	$\varphi \Rightarrow \psi_i$, broadcast	$\varphi \Rightarrow \psi$, broadcast	$i \in G'$: $\varphi \Rightarrow \psi$, skip
$S_G\varphi$	$\varphi \Rightarrow \psi$, search	$G \subseteq G'$: $\varphi \Rightarrow \psi$, skip	$\varphi \Rightarrow \psi_i$, broadcast	$G \subseteq G'$: $\varphi \Rightarrow \psi$, broadcast	$G \subseteq G'$: $\varphi \Rightarrow \psi$, skip
$\bigwedge_G K_i\varphi_i$	$(\wedge\varphi_i) \Rightarrow \psi$, centralize	$(\wedge\varphi_i) \Rightarrow \psi$, elect	$G \subseteq G'$: $(\wedge\varphi_i) \Rightarrow \psi_i$, confer	$G \subseteq G'$: $(\wedge\varphi_i) \Rightarrow \psi$, confer	$G \subseteq G'$: $(\wedge\varphi_i) \Rightarrow \psi$, skip
$E_G\varphi$	$j \in G$: $\varphi \Rightarrow \psi$, skip $j \notin G$: $\varphi \Rightarrow \psi$, notify	$G \cap G' \neq \emptyset$: $\varphi \Rightarrow \psi$, skip $G \cap G' = \emptyset$: $\varphi \Rightarrow \psi$, notify	$G' \subseteq G$: $\varphi \Rightarrow \psi_i$, skip $G' \not\subseteq G$: $\varphi \Rightarrow \psi_i$, distribute	$G' \subseteq G$: $\varphi \Rightarrow \psi$, skip $G' \not\subseteq G$: $\varphi \Rightarrow \psi$, distribute	$G \cap G' \neq \emptyset$: $\varphi \Rightarrow \psi$, skip
$D_G\varphi$	$\varphi \Rightarrow \psi$, centralize	$\varphi \Rightarrow \psi$, elect	$G \subseteq G'$: $\varphi \Rightarrow \psi_i$, confer	$\varphi \Rightarrow \psi$, confer	$G \subseteq G'$: $\varphi \Rightarrow \psi$, skip $G \not\subseteq G'$: $\varphi \Rightarrow \psi$, centralize

TABLE .1. Knowledge transitions

- *Elect* transitions. In this case again distributed information is gathered, but resulting not towards a certain process or set of processes, but leading to an in general unknown "winner."
- *Confer* transitions. For confer transitions distributed information is made known to all or to a larger set of processes.

We would like to give instantiations of all non-trivial knowledge transitions with layers that implement that transition for a certain architecture. Some transitions are more difficult to implement than others for certain architectures. Take, for example, the transition $K_i\varphi \rightsquigarrow E_G\varphi$. In a fully connected network a single round of send actions suffices. In an arbitrary network one needs message diffusion or other more complicated algorithms. Also, some knowledge transitions can be built up from other transitions. In order to confer one can, for example, combine a centralizing phase with a distribution phase. Here we restrict ourselves to a few characteristic transitions, needed in the examples.

From the literature many communication structures are known. Broadcasts, waves, phases, heartbeats, logic pulsing, rooted tree communication, message diffusion all correspond to certain types of protocols for different network architectures. (See Raynal and Helary [26] and Andrews [1] for overviews.) Such protocols or protocol layers correspond to (sequences of) knowledge transitions. We give a classification of such layers for different architectures. This classification is by no means complete; not all communication structures are discussed. It should however be possible to classify other communication structures along the same lines. A proof rules for doing so is given at the end of this section.

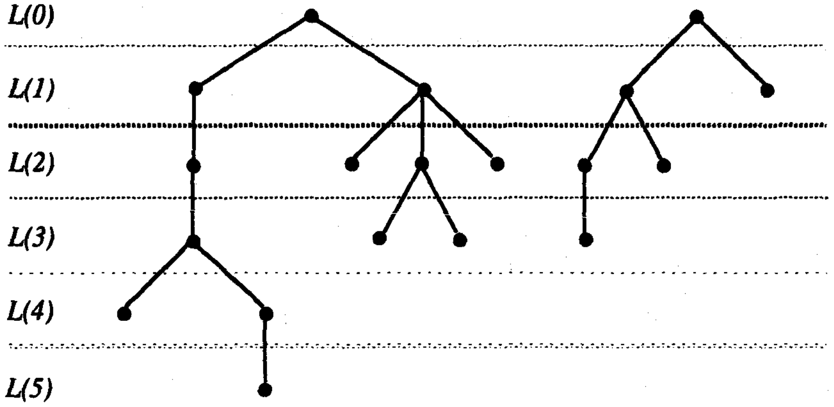

FIGURE 1. Levels of nodes in a tree structured network

We discuss two different types of network architectures: rooted trees or sets of rooted trees, and connected graphs. Other architectures, such as linear lists or fully connected graphs, are special cases of these two.
Assume we have a finite set of nodes V, and a subset $Root \subseteq V$ of root nodes. Let $root(v) \equiv v \in Root$. Every node v has a set of directed downward edges $down(v)$, and a set of successor nodes $S(v)$, and every non-root node v has an upward edge $up(v)$ pointing towards its root node. Every node in a tree is at a certain level, that is, it has a certain distance to its root. The set $L(l)$ is the set of nodes at level l (see figure 1).
A graph $\mathcal{G}$ is represented as a pair $\mathcal{G} = (V, E)$, where V is the set of nodes and E is a set of undirected edges. For every edge $j \in E$ we assume two directed unidirectional channels. For any node $v \in V$ let $out(v)$ denote the set of edges incident to v, that is,

$$out(v) \overset{\text{def}}{=} \{ (v,u) \in E \}.$$

Furthermore, let $adj(v)$ be the set of nodes adjacent to v. We have the following generic instantiations of knowledge transitions, going from top-left to bottom-right in the table.

$K_i\varphi \rightsquigarrow K_j\varphi$. This is the most elementary transition. A process i notifies some process j of a certain fact known to i. If i and j are adjacent, it can be implemented by a pair of communication actions. If not, some kind of relaying or forwarding protocol is to be used. We omit details.
In general, this transition can be implemented abstractly by a so-called notify action as proposed by Moses and Kislev [22]. An action notify(j, φ) by process i ensures that eventually j knows φ.

$K_v\varphi \rightsquigarrow E_V\varphi$. This coincides with $K_v\varphi \rightsquigarrow \wedge K_v\psi_v$ for $\varphi \Rightarrow \psi_v$. Under this category fall *broadcast* protocols for arbitrary graphs, and simple direct ***distribution layers*** for fully connected graphs and two-level trees with a single root. Broadcast algorithms in general are more complicated. See Cristian et al. [7] or Mullender [23] for more details.

$S_V\varphi \leadsto K_v\varphi$. As an example of this category we have search algorithms such as, for example, employed in the termination protocol of the Two-Phase Commit algorithm. A node that recovers or times out asks all other nodes for the answer, and (at least) one of them can give the answer.

$S_G\varphi \leadsto E_G\varphi$. This transition can be implemented as a combination of the above two: first collect the information to a single node and thereafter broadcast the result.
A simpler solution can be given is the case of a fully connected network: the node that knows φ sends the value to all other nodes. Other nodes can either send a default message or do nothing, depending on the context of the algorithm.

$\bigwedge_G K_i\varphi_i \leadsto K_j\psi$. Centralizers captured by the transition above occur frequently in protocols. Information that is located at different nodes is to be gathered to a single node. For tree-structured networks this can be done by means of the so-called *wave concept.* (See Raynal and Helary, [26].) The root node initiates a request wave down the tree, which is returned from the leaves upwards, gathering the information. If all nodes know that the information is to be sent, the request part can be omitted. The downward part, upward part, and the full wave are given in figure 2. We assume that $\wedge\varphi_v \Rightarrow \psi$. Fully connected networks can handled similarly, as if they were two-level trees.

$\bigwedge_G K_i\varphi_i \leadsto E_G\psi$. This transition can be implemented by adding the downward part of a wave to a full wave for tree-structured networks. For fully connected graphs this can be implemented much simpler: every node sends its information to every other node. This is, for example, used in decentralized Two-Phase Commit algorithms.

$E_G\varphi \leadsto E_{G'}\varphi$. For $G \subseteq G'$ this transition models that knowledge is distributed to a larger set of nodes. The transition $K_i\varphi \leadsto E_{G'}\varphi$ is a special case. Such transitions (conceptually) occur in tree-structured networks where the information is distributed from nodes at one level to nodes at the next level, or in networks that use ***message diffusion*** to broadcast messages over an arbitrary connected network (see, for example, Cristian et al. [7]).
In fact, message diffusion and downward wave parts can be viewed as sequences of transitions of this type themselves. We come back to this later.

```
$Downward \triangleq$
  for $v \in V$ par
    if $\neg root(v)$ then $receive(up(v), req_v)$ fi ;
    for $j \in down(v)$ par $send(j, req_v)$ rof
  rof,
$Upward \triangleq$
  for $v \in V$ par
    for $j = (v, v') \in down(v)$ par $receive(j, \varphi_{v'})$ rof ;
    "Compute $\psi_v$ from $\bigwedge_{v' \in S(v)} \varphi_{v'} \wedge \varphi_v$" ;
    if $\neg root(v)$ then $send(up(v), \psi_v)$ fi
  rof,
$Wave \triangleq$
  for $v \in V$ par
    if $\neg root(v)$ then $receive(up(v), req_v)$ fi ;
    for $j \in down(v)$ par $send(j, req_v)$ rof ;
    for $j = (v, v') \in down(v)$ par $receive(j, \varphi_{v'})$ rof ;
    "Compute $\psi_v$ from $\bigwedge_{v' \in S(v)} \varphi_{v'} \wedge \varphi_v$" ;
    if $\neg root(v)$ then $send(up(v), \psi_v)$ fi
  rof.
```

FIGURE 2. Waves in tree-structured networks

$D_G\varphi \rightsquigarrow K_j\varphi$. This transition is a special case of the centralizer transition $\wedge_G K_i\varphi_i \rightsquigarrow K_j\psi$ if φ_i is the information of node i to compute φ. It should therefore be known in what way the information to compute φ is distributed over the nodes.

$D_G\varphi \rightsquigarrow E_G\varphi$. Again this case is similar to the case for $\wedge_G K_i\varphi_i \rightsquigarrow E_G\psi$.

$D_G\varphi \rightsquigarrow D_{G'}\varphi$. For $G' \subseteq G$ this can correspond to a level transition in the upward part of a wave. When the information from one level is gather to a higher level, only the information of the higher levels is needed to compute the over all result. So again, an upward wave can be viewed as a sequence of these transitions.

This list is by no means complete. It simply presents a number of generic implementations of knowledge transitions for two types of networks.

How to classify layers?

Above we have given a classification of certain layers as knowledge transitions. An intuitive explanation has been given of why those layers belong to that transition class. In principle we have to *prove* that an algorithm satisfies a certain specification or knowledge transition.

To give such a prove we would like stay as much as possible within the limits of well-known proof systems for parallel algorithms, such as Owicki-Gries style proofs [24, 2]. The programs we use in this paper can be treated as programs with an **await** construct to implement *send* and *receive* actions, and channels plus disjoint sets of variables. Thus we can give (non-knowledge based) proof outlines for programs in the usual way (see Apt and Olderog [2] for a extensive overview). In order to be able to prove knowledge based properties of programs we add the following rule, based on proof outlines for parallel programs, the rule for knowledge generalization, and the definition of $K_i\varphi$. Let φ_i and ψ_i be basic assertions, not using the knowledge modalities, and let $S \vdash \Phi \rightsquigarrow \Psi$ denote that program S satisfies the knowledge transition $\Phi \rightsquigarrow \Psi$. We have the following proof rule.

$$\frac{\begin{array}{l}\{\varphi_i\}S_i\{\psi_i\},\ \text{for all } 1 \leq i \leq n,\\ \text{There exist valid proof outlines } \{\varphi_i\}S_i^\dagger\{\psi_i\} \text{ that}\\ \text{are interference free,}\end{array}}{S_1 \parallel \cdots \parallel S_n \quad \vdash \quad \bigwedge_{1\leq i\leq n} K_i\varphi_i \quad \rightsquigarrow \quad \bigwedge_{1\leq i\leq n} K_i\psi_i}$$

Using rules for disjunction and conjunction, and the properties of our modalities, we can give derived rules for the other modalities, such as D_G and S_G. Soundness of the knowledge based rule follows in a rather straightforward way from the soundness of the Owicki/Gries rule for parallelism and the definition of validity of K_i (see [14] for a proof).

4 Deriving distributed algorithms

In this section we give a number of examples of derivations using the approach sketched above. We start with a class of Two-Phase Commit implementations and end with waves. The same derivation style can be applied to other systems that have an underlying logical structure that is layered. In [14] some other examples are discussed as well, such as a distributed algorithm for computing minimal distances in networks.

4.1 Two-Phase Commit

The Two-Phase Commit protocol is an example of *atomic commit protocols* that are used in distributed databases to guarantee *consistency* of the database. A distributed database consists of a number of sites connected by some network, where every site has a local database. Data are distributed over a number of sites. In such a distributed database system *transactions,* consisting of a series of read and write actions, are executed. Reading and writing database items is be done by forwarding the action to the site where the item is stored. Terminating the transaction however involves *all* sites accessed in the transaction, as all sites must agree on the decision

to be taken—which is either to *commit* or to *abort*—in order to guarantee consistency. In the case of an abort all changes made by the transaction are discarded, in the case of a commit they are made permanent. A protocol that guarantees such consistency is called an atomic commit protocol (ACP). We refer to Bernstein, Hadzilacos and Goodman [4] for more details.
In an ACP every participating process has one vote: YES or NO, and every process can reach one out of two decisions: COMMIT or ABORT. Here we do not take into account the possibility of communication failures or site failures, that is, we assume that every message sent is eventually delivered and that sites are working correctly.

First of all we should give a specification of the atomic commit problem as a knowledge transition. Thereafter we refine this transition to a sequence of (simpler) transitions. As we do not take failures into account the requirements can be phrased as follows: *Given the votes of every participating process, each process should decide to* COMMIT *iff every process has voted* YES. This is represented by the following knowledge transition. Let G be the set of participating processes and define $\mathit{total\_vote} = \text{YES}$ iff $\bigwedge_{i \in G} \mathit{vote}_i = \text{YES}$. So *total_vote* is not a variable but represents the combined values of all local variables vote_i.

$$\bigwedge_{i\in G} K_i \mathit{vote}_i \rightsquigarrow \bigwedge_{i\in G} K_i(\mathit{total\_vote} \wedge (\mathit{dec}_i = \text{COMMIT} \Leftrightarrow \mathit{total\_vote} = \text{YES})).$$

Using the definitions of D_G and *total_vote* (given the distribution of the variables over the processes) this can be rewritten to

$$D_G \mathit{total\_vote} \rightsquigarrow \bigwedge_{i\in G} K_i(\mathit{total\_vote} \wedge (\mathit{dec}_i = \text{COMMIT} \Leftrightarrow \mathit{total\_vote} = \text{YES})).$$

Deriving layered implementations

To derive implementations for knowledge transitions the following strategy is employed. We first check whether the transition under consideration has an immediate implementation for a certain network architecture. If this is the case, we're done. If not so we split the transition into two or more smaller transitions and continue with them. This "transition splitting" is in fact a real design step which can have consequences for the eventual implementation. The resulting *layered* algorithm is thereafter transformed to a *distributed* system using the transformation principle discussed in section 2.

In order to simplify matters, we first split of a transition "$\overset{2}{\rightsquigarrow}$," to be implemented by a final layer TPC_2 from the transition specified, where the decision is "executed," from the rest. In this final layer only local changes need to be performed, so its implementation is straightforward. This results in the following two transitions.

$$D_G \mathit{total\_vote} \overset{1}{\leadsto} E_G \mathit{total\_vote} \overset{2}{\leadsto}$$
$$\bigwedge_{i \in G} K_i(\mathit{total\_vote} \wedge (dec_i = \text{COMMIT} \Leftrightarrow \mathit{total\_vote} = \text{YES})).$$

The first transition is a *confer transition* (see table .1), and can immediately be implemented for fully connected networks by means of sending the votes to all other nodes, as was mentioned in the previous section. This would result in the following layer implementing "$\overset{1}{\leadsto}$."

```
$TPC_1 \triangleq$
  $\{D_G \mathit{total\_vote}\}$
  for $i \in G$ par
    for $i' \in G - \{i\}$ par $send((i,i'), vote_i)$ rof ;
    for $i' \in G - \{i\}$ par $receive((i,i'), vote_{ii'})$ rof
  rof
  $\{E_G \mathit{total\_vote}\}$
```

A second possibility is that we do *not* have a fully connected network, but some kind of tree structured network. In that case there is no apparent immediate solution to the above transition. So we split the transition again, and do so in the following way. Let $c \in G$ be some participating process.

$$D_G \mathit{total\_vote} \overset{3}{\leadsto} K_c \mathit{total\_vote} \overset{4}{\leadsto} E_G \mathit{total\_vote}.$$

The question now is what a sensible choice for c would be. Under the given assumption that we have a tree structured network an obvious choice is to take for c the root of the tree. There is however—under additional conditions—a second possibility. For a *linear tree* or chain, that is, a tree where every node has at most one downward edge, we can also take the (unique) leaf of the tree! We assume that the tree has at least two nodes. We first discuss the former possibility.
To obtain a $D_G \mathit{total\_vote} \leadsto K_c \mathit{total\_vote}$ transition, which is a *centralize transition*, we can use a full wave as discussed in the previous section. This would result in the following implementation.

```
$TPC_3 \triangleq$
  $\{D_G \mathit{total\_vote}\}$
  for $i \in G$ par
    if $\neg root(i)$ then $receive(up(i), req_i)$ fi ;
    for $j \in down(i)$ par $send(j, req_i)$ rof ;
    for $j = (i,i') \in down(i)$ par $receive(j, vote_{ii'})$ rof ;
    if $(\forall i' \neq i \in S(i).\ vote_{ii'} = \text{YES}) \wedge vote_i = \text{YES}$ then $rep_i := \text{YES}$
                                                          else $rep_i := \text{NO}$ fi ;
    if $\neg root(i)$ then $send(up(i), rep_i)$ fi
  rof
  $\{K_c \mathit{total\_vote}\}$
```

The value of *total_vote* is stored in rep_c.

In the linear case we have the following implementation. Let $down(i)$ denote the unique edge downward for every node i in this case. For the leaf of the linear tree this is nil, and $send(nil, e) \overset{\text{def}}{=}$ **skip** .

```
$TPC_{3'} \triangleq$
  $\{D_G\, total\_vote\}$
  for $i \in G$ par
    if $\neg root(i)$ then $receive(up(i), v_i)$ fi ;
    if $(root(i) \wedge vote_i = \text{YES}) \vee (v_i = \text{YES} \wedge vote_i = \text{YES})$
      then $send(down(i), \text{YES})$ else $send(down(i), \text{NO})$ fi
  rof
  $\{K_c\, total\_vote\}$
```

The *total_vote* follows in this case from the values of v_c and $vote_c$.

To implement the second transition in this layer, "$\overset{4}{\rightsquigarrow}$", we can use a downward wave for the first case, and an upward wave in the linear case, as it is a *broadcast transition,* leading to the following implementations:

```
$TPC_4 \triangleq$
  $\{K_c\, total\_vote\}$
  for $i \in G$ par
    if $\neg root(i)$ then $receive(up(i), rep_i)$ fi ;
    for $j \in down(i)$ par $send(j, rep_i)$ rof
  rof,
  $\{E_G\, total\_vote\}$,
$TPC_{4'} \triangleq$
  $\{K_c\, total\_vote\}$
  (if $v_c = \text{YES} \wedge vote_c = \text{YES}$ then $rep_c := \text{YES}$ else $rep_c := \text{NO}$ fi ;
   $send(up(c), rep_c)$ )   ||
  for $i \in G - \{c\}$ par
    $receive(down(i), rep_i)$ ;
    $send(up(i), rep_i)$
  rof
  $\{E_G\, total\_vote\}$.
```

The first two lines of $TPC_{4'}$ correspond to the process for the leaf node. A third possible network configuration is a special case of general networks: the ring. In this case we could again take a similar approach as in the previous case by appointing one node to gather all votes, and send the result through the ring (see [14]).

Transforming sequences of layers to parallel processes

We have derived a number of layered implementations for the Two-Phase Commit protocol consisting of two or three layers, where every layer is a parallel composition over all participants. The actual implementation we should arrive at must be of the form **for** $i \in G$ **par** $P(i)$**rof**, that is, a single (sequential) process for every participant. The transformation from the layered to the distributed structure can be carried out using the CCL law, or more precisely, the transformation principle discussed in section 2. Using this principle we transform the layered implementations given above. As an example take the layered implementation for tree-structured networks. This layered implementation is

$$TPC_l \triangleq TPC_3 \bullet TPC_4 \bullet TPC_2.$$

Transforming this system immediately results in the distributed process *TPC* given below, which is IO-equivalent to the layered implementation. Therefore it satisfies the same initial knowledge transition specification.

```
$TPC \triangleq$
  for $i \in G$ par
    if $\neg root(i)$ then $receive(up(i), req_i)$ fi ;
    for $j \in down(i)$ par $send(j, req_i)$ rof ;
    for $j = (i, i') \in down(i)$ par $receive(j, vote_{ii'})$ rof ;
    if $(\forall i' \neq i \in S(i).\ vote_{ii'} = \text{YES}) \wedge vote_i = \text{YES}$ then $rep_i := \text{YES}$
                                                    else $rep_i := \text{NO}$ fi ;
    if $\neg root(i)$ then $send(up(i), rep_i)$ fi ;
    if $\neg root(i)$ then $receive(up(i), rep_i)$ fi ;
    for $j \in down(i)$ par $send(j, rep_i)$ rof ;
    if $rep_i = \text{YES}$ then $dec_i := \text{COMMIT}$ else $dec_i := \text{ABORT}$ fi
  rof.
```

This algorithm can be optimized by combining the two conditionals in the sixth and seventh line, but the basic structure remains the above.

Similarly, we can transform the layered implementation of the linear algorithm TPC'_l using the transformation rule to the following distributable algorithm TPC'.

```
$TPC' \triangleq C \parallel$ for $i \in G - \{c\}$ par $P(i)$ rof,
$C \triangleq$
  if $\neg root(c)$ then $receive(up(c), v_c)$ fi ;
  if $v_c = \text{YES} \wedge vote_c = \text{YES}$ then $rep_c := \text{YES}$ else $rep_c := \text{NO}$ fi ;
  $send(up(c), rep_c)$ ;
  if $rep_c = \text{YES}$ then $dec_i := \text{COMMIT}$ else $dec_i := \text{ABORT}$ fi,
$P(i) \triangleq$
  if $\neg root(i)$ then $receive(up(i), v_i)$ fi ;
  if $(root(i) \wedge vote_i = \text{YES}) \vee (v_i = \text{YES} \wedge vote_i = \text{YES})$
    then $send(down(i), \text{YES})$ else $send(down(i), \text{NO})$ fi ;
  $receive(down(i), rep_i)$ ;
  $send(up(i), rep_i)$ ;
  if $rep_i = \text{YES}$ then $dec_i := \text{COMMIT}$ else $dec_i := \text{ABORT}$ fi.
```

Note that for the networks under consideration, which have at least two participants, the first guard ($\neg root(c)$) always evaluates to *true* and can therefore be removed.

These protocols correspond to a generalization of the *decentralized Two-Phase Commit* and the *linear Two-Phase Commit* as they are known from the literature. The result for the fully connected network is known as *centralized Two-Phase Commit.*

Finally, we have the case of arbitrary network structures. To obtain the knowledge transition we can do two things: either we use some broadcast algorithm for networks, for example based on message diffusion, or we first construct a spanning tree in one layer and then apply tree based communication protocols. In the former case this is similar to the fully connected network case but with broadcast instead of send, and in the latter it is the same as for tree-based networks with an additional layer. We do not discuss these cases in any detail.

4.2 Waves as sequences of layers

The same principles as applied above can be used to derive certain knowledge transitions from more elementary ones. We can, for example, derive the protocol for (upward or downward) waves as a sequence of layers corresponding to levels in the tree. In a similar way we can derive the protocol for rings.

Take for example the downward part of a wave. It is characterized by the knowledge transition

$$\bigwedge_{v \in Root} K_v \varphi \quad \rightsquigarrow \quad \bigwedge_{v \in V} K_v \varphi.$$

With every node $v \in V$ a level number l is associated, giving the distance to the root it belongs to. Recall that $L(l)$ is the set of nodes at level l (see section 3, figure 1). Let $Upto(l) \overset{\text{def}}{=} \cup\{L(l') \mid 0 \leq l' \leq l\}$, and let n be the maximal level number of any node. We can now split the above transition in n transitions "$\overset{l}{\leadsto}$" for $0 \leq l < n$, where

$$E_{Upto(l)}\varphi \quad \overset{l}{\leadsto} \quad E_{Upto(l+1)}\varphi.$$

As $Upto(1) = Root$ and $Upto(n+1) = V$ this sequence refines the original transition. Every transition $E_{Upto(l)} \overset{l}{\leadsto} E_{Upto(l+1)}$ can be implemented as follows:

```
$Down(l) \triangleq$
  $\{E_{Upto(l)}\varphi\}$
  for $i \in L(l)$ par
    for $j \in down(i)$ par $send(j, \varphi)$ rof
  rof ||
  for $i \in L(l+1)$ par $receive(up(i), \varphi)$ rof
  $\{E_{Upto(l+1)}\varphi\}$
```

Note that for $down(i) = \emptyset$ the parallel statement

for $j \in down(i)$ **par** ... **rof**

reduces to **skip**.

This layered system we would like to transform to a distributed system. At first it seems however that the transformation principle we applied above does not apply here: the layers are not of the form **for** $v \in V$ **par** $P(v)$ **rof**. But fortunately, we can add **skip** components for any process that does not occur in the parallel composition in a layer. Moreover, after transformation these **skip** components can be removed, as $(P$; **skip** $)$ = $($ **skip** ; $P)$ = P.

The result is the following algorithm.

```
$Down \triangleq$
  for $i \in Root$ par
    for $j \in down(i)$ par $send(j, \varphi)$ rof
  rof ||
  for $i \in V - Root$ par
    $receive(up(i), \varphi)$ ;
    for $j \in down(i)$ par $send(j, \varphi)$ rof
  rof.
```

Merging the code for the root nodes and the non-root nodes using a conditional statement results in the *Downward* layer given in section 3.

5 Concluding remarks

In this paper we have discussed how to use knowledge based logics in the layered design of distributed systems. The contribution of this paper is twofold. First of all we have given a classification scheme for protocol layers as knowledge transitions. Secondly we have shown how such knowledge transitions can be used to derive layered implementations of protocols. Thus we have used knowledge based logics to give generic specifications of program layers and protocols.

We have shown that this design principle applies to a number of algorithms. In principle, any algorithm that can be viewed as a layered system should fit in this framework, which concerns a substantial class of algorithms. There exist however algorithms that cannot be written as layered systems, for example, highly interactive systems such as memories, or so-called retroactive systems (see Janssen [13] for a discussion of these problems).

The role of knowledge based logic has been limited to the specification of knowledge transitions. It would be interesting to use that logic to prove the layers correct themselves, possible in the style of van Hulst and Meyer [29]. Possibly such ideas would allow the approach presented to be extended to non-layered systems as well. The advantage of the approach presented here however is that the extensions to well-known techniques for program verification needed in this approach are rather limited.

We have used epistemic logic primarily as a logic to express locality of information. In [30] Wieczorek proposes a logic with modalities that directly express location. This logic however is weaker in the sense that it does not allow to combine information of different locations using the properties of the knowledge modalities.

Another interesting approach using knowledge based logics is to use program constructs that allow for the use of knowledge based expressions. The notify construct as introduced by Moses and Kislev [22] is an example thereof. Furthermore one can use actions guarded by knowledge based expressions instead of normal boolean guards. Such programs are discussed by Fagin et al. in [10, 9]. One of the difficulties with these programs is however that the transition of a knowledge based program to an ordinary program has not yet been formalized. Possibly classification schemes as introduced here combined with layered derivation can be of help to formalize this transition.

Acknowledgements. The author would like to thank Mannes Poel for detailed reading of the manuscript, and Yoram Moses, Wim Koole and John-Jules Meyer for useful comments on this work.

6 References

[1] G. Andrews. *Concurrent Programming — Principles and Practice.* The Benjamin/Cummings Publishing Company, 1991.

[2] K. Apt and E.-R. Olderog. *Verification of sequential and concurrent programs.* Springer-Verlag, 1991.

[3] R. Back and K. Sere. Stepwise refinement of action systems. *Structured Programming,* 12:17–30, 1991.

[4] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[5] R. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.

[6] C. Chou and E. Gafni. Understanding and verifying distributed algorithms using stratified decomposition. In *Proceeding 7th ACM Symposium on Principles of Distributed Computing,* 1988.

[7] F. Cristian, H. Aghili, R. Strong, and D. Dolev. Atomic broadcast: From simple message diffusion to byzantine agreement. In *Proceedings 15th International Symposium on Fault-Tolerant Computing,* 1985.

[8] T. Elrad and N. Francez. Decomposition of distributed programs into communication closed layers. *Science of Computer Programming,* 2:155–173, 1982.

[9] R. Fagin, J. Halpern, Y. Moses, and M. Vardi. Knowledge-based programs. In *Proceedings ACM Symposium on Principles of Distributed Computing.* ACM, 1995.

[10] R. Fagin, J. Halpern, Y. Moses, and M. Vardi. *Reasoning About Knowledge.* MIT Press, 1995. To appear.

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

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

[13] W. Janssen. *Layered Design of Parallel Systems.* PhD thesis, University of Twente, 1994.

[14] W. Janssen. Layers as knowledge transitions in the design of distributed systems. Technical Report 94-71, University of Twente, 1994.

[15] W. Janssen, M. Poel, and J. Zwiers. Action systems and action refinement in the development of parallel systems. In *Proceedings of CONCUR '91, LNCS 527,* pages 298–316. Springer-Verlag, 1991.

[16] W. Janssen and J. Zwiers. Protocol design by layered decomposition, a compositional approach. In J. Vytopil, editor, *Proceedings Formal Techniques in Real-Time and Fault-Tolerant Systems, LNCS 571*, pages 307–326. Springer-Verlag, 1992.

[17] B. Jonsson. Modular verification of asynchronous networks. In *Proceedings 6th ACM Symposium on Principles of Distributed Computing*, pages 152–166, 1987.

[18] S. Katz and D. Peled. Verification of distributed programs using representative interleaving sequences. *Distributed Computing*, 6(2), 1992.

[19] N. Lynch, M. Merritt, W. Weihl, and A. Fekete. *Atomic Transactions*. Morgan Kaufman Publishers, 1994.

[20] N. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings 6th ACM Symposium on Principles of Distributed Computing*, pages 137–151, 1987.

[21] J.-J. Meyer, W. van der Hoek, and G. Vreeswijk. Epistemic logic for computer science: A tutorial. *Bulletin of the EATCS, numbers 44 and 45*, 1991.

[22] Y. Moses and O. Kislev. Knowledge-oriented programming, (extended abstract). In *Proceedings 12th ACM Symposium on Principles of Distributed Computing*, pages 261–270. ACM, 1993.

[23] S. Mullender, editor. *Distributed Systems*. Addison-Wesley, second edition, 1993.

[24] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6:319–340, 1976.

[25] M. Poel and J. Zwiers. Layering techniques for development of parallel systems. In G. v. Bochmann and D. Probst, editors, *Proceedings Computer Aided Verification, LNCS 663*, pages 16–29. Springer–Verlag, 1992.

[26] M. Raynal and J.-M. Helary. *Synchronization and control of distributed systems and programs*. John Wiley & Sons, 1990.

[27] F. Stomp and W.-P. de Roever. A correctness proof of a distributed minimum-weight spanning tree algorithm (extended abstract). In *Proceedings of the 7th ICDCS*, 1987.

[28] F. Stomp and W.-P. de Roever. A principle for sequential reasoning about distributed systems. *Formal Aspects of Computing*, 6(6):716–737, 1994.

[29] M. van Hulst and J.-J. Meyer. An epistemic proof system for parallel processes. In R. Fagin, editor, *Proceedings 5th TARK*, pages 243–254. Morgan Kaufmann, 1994.

[30] M. Wieczorek. *Locative Temporal Logic and Distributed Real-Time Systems*. PhD thesis, Catholic University of Nijmegen, 1994.

[31] J. Zwiers and W. Janssen. Partial order based design of concurrent systems. In J. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Proceedings of the REX School/Symposium "A Decade of Concurreny", Noordwijkerhout, 1993, LNCS 803*, pages 622–684. Springer-Verlag, 1994.