A Proof Technique for Rely/Guarantee Properties

Eugene W. Stark*
Department of Computer Science
State University of New York at Stony Brook
Stony Brook, New York 11794-4400/USA

June 30, 1986

Abstract

A *rely/guarantee* specification for a program P is a specification of the form $R \supset G$ (R implies G), where R is a *rely condition* and G is a *guarantee condition*. A rely condition expresses the conditions that P relies on its environment to provide, and a guarantee condition expresses what P guarantees to provide in return. This paper presents a proof technique that permits us to infer that a program P satisfies a rely/guarantee specification $R \supset G$, given that we know P satisfies a finite collection of rely/guarantee specifications $R_i \supset G_i, (i \in I)$. The utility of the proof technique is illustrated by using it to derive global liveness properties of a system of concurrent processes from a collection of local liveness properties satisfied by the component processes. The use of the proof rule as a design principle, and the possibility of its incorporation into a formal logic of rely/guarantee assertions, is also discussed.

1 Introduction

A *rely/guarantee* specification for a program P is a specification of the form $R \supset G$ (R implies G), where R is a *rely condition* and G is a *guarantee condition*. A rely condition expresses the conditions that P relies on its environment to provide, and a guarantee

*This research was supported in part by ARO grant DAAG29-84-K-0058, NSF grant DCR-83-02391, and DARPA grant N00014-82-K-0125.

condition expresses what P guarantees to provide in return. This paper presents a proof technique that permits us to infer that a program P satisfies a rely/guarantee specification $R \supset G$, given that we know P satisfies a finite collection of rely/guarantee specifications $R_i \supset G_i, (i \in I)$. In a typical application, $R \supset G$ will be a global property of a large program P, whereas each $R_i \supset G_i$ will be a locally verifiable property of a smaller component P_i of P. In a top-down design methodology based on successive decomposition [Lis79] [Wir71], the proof technique can be used as a decomposition principle for determining specifications $R_i \supset G_i$ for component modules, when these component modules are used to implement a "higher-level module" that must satisfy the specification $R \supset G$.

Two examples are given to illustrate the utility of the proof technique: a *distributed synchronization algorithm*, in which a collection of processes communicate in a ring-like pattern to synchronize access to critical sections, and a *distributed resource allocation algorithm*, in which processes communicate in a tree-like pattern to distribute a finite collection of resources among themselves. Although the statement of the proof technique does not depend on the choice of a particular specification or programming language, in the examples we use as a programming language a concurrent version of Dijkstra's guarded command language [Dij76], and as a specification language a version of *temporal logic* [Pnu77] [Lam80] [Lam83] [MP83].

In the examples, we are concerned with the proof of *liveness properties* of systems of concurrent processes. In particular, we are interested in deriving *global* liveness properties satisfied by a system from a collection of *local* liveness properties satisfied by the component processes. The fact that the technique applies readily to the proof of general liveness properties is interesting, since not many useful techniques for performing such proofs have been developed.

1.1 Related Work

The proof rule and examples presented in this paper are adapted from [Sta84].

The idea that program specifications are conveniently formulated and manipulated in the form of rely/guarantee conditions is not new. Pre/postcondition specifications for sequential programs are examples of rely/guarantee specifications, in which the precondition expresses the conditions on the program variables the program relies on when control enters it, and the postcondition expresses the conditions the program guarantees when and if control leaves it. In fact, the Floyd/Hoare techniques for proving partial correctness of sequential programs [Flo67] [Hoa69] can be viewed as a special case of the proof technique presented here (see Section 2). However, our technique extends the Floyd/Hoare approach, since the former can be applied to the proof of liveness properties, whereas the applicability of the latter (in the usual formulation) is limited to safety, or invariance properties.

For concurrent or distributed programs, a kind of rely/guarantee specification and associated proof technique was introduced in [MC81]. In that paper, a process h is specified by an assertion of the form $r|h|s$, where r and s are predicates on finite sequences (called *traces*) of *communication events*. Such an assertion is interpreted as: "The predicate s holds of the empty trace, and for all traces t that can be produced by process h, if r holds for all *proper* prefixes of t, then s holds for all prefixes (both proper and improper) of t.

Misra and Chandy's proof technique is expressed as a "Theorem of Hierarchy," which gives conditions under which specifications that are satisfied by a collection of component processes can be used to infer a specification that holds for the network formed by interconnecting the components. Their proof technique can be stated as follows: To show that the specification $R_0|H|S_0$ for the network H is a consequence of the specifications $r_i|h_i|s_i, (i \in I)$ for the components, it suffices to show that:

1. S implies S_0,
2. (R_0 and S) implies R,

where R and S denote the conjunction of the r_i and s_i, respectively. These conditions are closely related to the *cut set* conditions presented below.

In [MCS82], the techniques of [MC81] are extended to encompass a weak form of liveness specification in which an additional predicate q is used to state conditions under which a process trace is guaranteed to be extended. The Theorem of Hierarchy is augmented with additional conditions to permit its application to these more general specifications. The additional conditions do not appear to relate in a simple way to the proof technique presented here.

The use of rely and guarantee conditions has also been proposed for safety specifications by Jones [Jon81] [Jon83]. Barringer and Kuiper [BK83] (see also [BKP84]) have proposed the use of liveness specifications that are partitioned into an "environment part," which captures assumptions made about the environment, and a "component part," which captures committments made by the module being specified. Jones, as well as Barringer and Kuiper, exploit the rely/guarantee condition structure of specifications by defining inference rules for process composition.

Hailpern and Owicki [HO80] have performed some example proofs in which liveness properties (expressed in temporal logic) for network protocols are derived from more primitive liveness properties satisfied by each of the constituent processes. Although they are successful at constructing proofs for examples of reasonable complexity, it is difficult to discern much in the way of general principles that might be used to systematize the construction of proofs for different examples. In contrast, the proof rule presented here suggests a way of thinking about process interaction that can systematize and simplify the construction of correctness proofs.

2 The Proof Rule

We assume a programming language, a meaning function that assigns to each program the set of its computations, a specification language, and a binary relation $\models$ between computations and specifications, where if x is a computation and S is a specification, then $x \models S$ means that computation x *satisfies* specification S.

We assume that the specification language is closed under the formation rules for the logical connectives $\neg$ and $\supset$:

($\neg$) If S is a specification, then $\neg S$ is a specification,

($\supset$) If S_1 and S_2 are specifications, then $S_1 \supset S_2$ is a specification,

and that $\neg$ and $\supset$ are endowed with their usual meanings:

($\neg$) $x \models \neg S$ iff $x \not\models S$,

($\supset$) $x \models S_1 \supset S_2$ iff $x \models S_1$ implies $x \models S_2$.

The other standard logical connectives can be treated as definitional extensions in the usual way.

We are interested in establishing statements of the form "$P \models S$," which we define to mean "$x \models S$ for all computations x of program P."

To state our proof rule we do not need to make any other assumptions about the precise form of computations or the programming or specification languages. Later, in demonstrating the application of the rule to examples, we will assume that computations are sequences of states and that specifications are sentences in a language of temporal logic. Although the proof rule is a logical truth that has nothing specific to do with the structure of programs, specifications, or computations, it derives power from the fact that the rely/guarantee paradigm is a useful way to think about interaction between program modules.

The proof rule described in this section permits us to derive a statement of the form:

$$P \models R \supset G$$

from a finite collection of statements of the form:

$$P \models R_i \supset G_i, \qquad i \in I$$

under certain conditions on the specifications R, G, R_i, and G_i.

Intuitively, $R \supset G$ should be thought of as an "abstract" or "high-level" statement that we wish to prove about the program P, and each $R_i \supset G_i$ should be thought of as a "concrete" or "low-level" statement that we have already shown to hold for P. In the examples given later on in the paper, P will be a parallel program composed of a finite set of component processes $\{P_i : i \in I\}$, and each $R_i \supset G_i$ will express a property of the component process P_i that we assume has already been shown to hold by arguments involving P_i alone.

The proof rule presented below is based on the following intuition: If we know, for each $i \in I$, that component program P_i guarantees condition G_i under assumption R_i, then we can prove that P guarantees condition G under assumption R by showing the existence of a set of specifications that "cuts," in a certain sense, the dependence between each pair of component programs, and between each component program and the external environment. The sense in which dependence is cut is highly analogous to the way in which a loop invariant is used to isolate reasoning about one iteration of the loop from reasoning about the preceding and succeeding iterations.

Formally, we say that the collection of specifications $\{RG_{i,j} : i, j \in I \cup \{\mathrm{ext}\}\}$ is a *cut set* for the program P and specifications $R, G, \{R_i, G_i : i \in I\}$ if:

$$\begin{aligned} P &\models R \supset (\textstyle\bigwedge_{j \in I} RG_{\mathrm{ext},j}) && (1)\\ P &\models (\textstyle\bigwedge_{i \in I} RG_{i,\mathrm{ext}}) \supset G && (2)\\ P &\models (\textstyle\bigwedge_{i \in I \cup \{\mathrm{ext}\}} RG_{i,j}) \supset R_j, \qquad \text{for all } j \in I && (3)\\ P &\models G_i \supset (\textstyle\bigwedge_{j \in I \cup \{\mathrm{ext}\}} RG_{i,j}), \qquad \text{for all } i \in I. && (4) \end{aligned}$$

Here "ext" is a special symbol that does not appear in I.

If i, j are both in I, then the specification $RG_{i,j}$ should be thought of as expressing both what component i guarantees to component j, and dually, what component j relies on component i to provide. The specification $RG_{\mathrm{ext},j}$ expresses what the external environment of the entire program guarantees to component j, and also what component j

relies on the external environment to provide. Similarly, the specification $RG_{i,\mathrm{ext}}$ expresses what component i guarantees to the external environment, and also what the external environment relies on module i to provide. By convention, we define $RG_{\mathrm{ext},\mathrm{ext}} \equiv \mathsf{true}$. This specification is not used in the proof rule and has no particular intuitive significance. We include it merely for uniformity.

Conditions (1) and (2) above can be interpreted as stating, respectively, that the rely condition R implies what each component relies on the external environment to provide, and the guarantee condition G is implied by the conjunction of what each component guarantees to the external environment. Conditions (3) and (4) can be interpreted, respectively, as stating that component j's rely condition is implied by the conjunction of what the external environment and each component i guarantees to provide to j, and component i's guarantee condition implies the conjunction of what the external environment and each component j relies on i to provide.

The existence of a cut set is not sufficient to imply that $P \models R \supset G$ is a consequence of $\{P \models R_i \supset G_i : i \in I\}$. Intuitively, the reason is that even though the rely and guarantee conditions imply each other in the proper way, it might still be the case in a computation of P satisfying the rely condition R, that no component's rely condition R_i holds, hence no component's guarantee condition G_i need necessarily hold either, and hence the guarantee condition G need not hold. To avoid this kind of degeneracy, we introduce the additional condition that, in every computation of P, every possible cycle of mutual dependence between components is broken by at least one condition in RG that holds for that computation.

Formally, If I is a finite set, then define a *cycle* of I to be a nonempty finite set of pairs of the form $\{(i_0, i_1), (i_1, i_2), \ldots, (i_{n-1}, i_n)\}$ such that $i_n = i_0$. We say that the collection $\{RG_{i,j} : i, j \in I\}$ is *acyclic* if:

$$P \models \bigvee_{k=0}^{n-1} RG_{i_k, i_{k+1}}$$

for all cycles $\{(i_0, i_1), \ldots, (i_{n-1}, i_n)\}$ of I.

Note that acyclicity implies the "diagonal" elements $RG_{i,i}$ hold unconditionally:

$$P \models RG_{i,i} \qquad \text{for all } i \in I.$$

We now present our proof rule.

Theorem 1 *(Rely/Guarantee Proof Rule) – Suppose P is a program, I is a finite index set, and the collection $RG = \{RG_{i,j} : i, j \in I \cup \{\mathrm{ext}\}\}$ is an acyclic cut set for program P*

and specifications $R, G, \{R_i, G_i : i \in I\}$. Then to prove the statement

$$P \models R \supset G,$$

it suffices to show

$$P \models R_i \supset G_i,$$

for all $i \in I$.

Proof – Suppose $RG = \{RG_{i,j} : i, j \in I \cup \{\text{ext}\}\}$ is a cut set for program P and specifications $R, G, \{R_i, G_i : i \in I\}$. Suppose further that

$$P \models R_i \supset G_i$$

holds for each $i \in I$, but

$$P \not\models R \supset G.$$

This means that there is a computation x of P such that $x \models R$, but $x \not\models G$. We perform an inductive construction to obtain a cycle

$$\{(i_m, i_{m+1}), \ldots, (i_{n-1}, i_n)\}$$

of I such that $x \not\models \bigvee_{k=m}^{n-1} RG_{i_k,i_{k+1}}$. This implies that RG is not acyclic for P.

As the induction hypothesis at stage k of the construction, we assume that $i_0, i_2, \ldots, i_k$ have been constructed so that $x \not\models R_{i_k}$ and $x \not\models \bigvee_{j=1}^{k-1} RG_{i_j,i_{j+1}}$.

Basis: From property (1) of a cut set and the assumption that $x \models R$, we know that $x \models RG_{\text{ext},j}$ for all $j \in I$. Since $x \not\models G$, by property (2) of a cut set we know that $x \not\models RG_{i_0,\text{ext}}$ for some $i_0 \in I$. By property (4) of a cut set we know that $x \not\models G_{i_0}$, and from the assumption that $x \models R_{i_0} \supset G_{i_0}$, we conclude that $x \not\models R_{i_0}$.

Induction: Assume the induction hypothesis holds for some $k \geq 0$. By property (3) of a cut set we know that $x \not\models RG_{i_k,i_{k+1}}$ for some i_{k+1} in I. If $i_{k+1} = i_m$ for some m with $0 \leq m \leq k$, then we have obtained the desired cycle and the construction terminates. Otherwise, by property (4) of a cut set we know that $x \not\models G_{i_{k+1}}$, and from the assumption that $x \models R_{i_{k+1}} \supset G_{i_{k+1}}$, we conclude that $x \not\models R_{i_{k+1}}$. This establishes the induction hypothesis for $k+1$.

Since the set I is finite by hypothesis, we cannot extend the sequence $i_0, i_1, \ldots, i_k$ indefinitely without obtaining a cycle. ∎

In a sense, Theorem 1 can be viewed as a generalization of the Floyd/Hoare technique [Flo67] [Hoa69] for proving partial correctness of sequential programs. In the Floyd/Hoare proof technique, a program contains a collection of *control points*, which are "tagged" or "annotated" by associating with them *assertions* about the values of the program variables. The meaning of an assertion A_p associated with control point p is the invariance property: "Whenever control is at point p, assertion A_p will be true of the program variables." If we assume (which we can, without loss of generality) that to each ordered pair (S_i, S_j) of program statements there corresponds at most one control point $p_{i,j}$, representing the point at which control leaves S_i and enters S_j, then the invariance property corresponding to control point $p_{i,j}$ can be thought of both as what statement S_i guarantees to statement S_j, and as what statement S_j relies on S_i to provide. The collection of all such invariance properties therefore corresponds directly to the set RG in the proof technique presented here.

Once an annotation for a program has been selected, proving the partial correctness of the program with respect to a *precondition* R and a *postcondition* G is reduced to showing the partial correctness of each statement S_i with respect to precondition R_i and postcondition G_i, assuming a certain relationship holds between the pre- and postconditions and the annotations associated with the control points. In Floyd's original formulation, the precondition for statement S_i is required to be exactly the conjunction of the assertions associated with points at which control enters S_i, and the postcondition is required to be exactly the conjunction of the assertions associated with points at which control leaves S_i. In Hoare's version, the pre- and postconditions need not be exactly these conjunctions, as long as they imply or are implied by them in an appropriate way.

The precise relationship that must hold between the pre- and postconditions and the annotations of the control points corresponds to the "cut set" conditions defined above. Furthermore, the acyclicity condition defined above can be shown to follow from the fact that states in a computation are reachable from an initial state in a finite number of steps, plus the requirement that enough control points be tagged to cut any program loop. The problem of annotating a program with assertions can therefore be thought of as a special case of the problem of finding an acyclic cut set.

3 Parallel Programs and Temporal Specifications

To illustrate the use of the rely/guarantee proof rule in proving properties of concurrent programs, we now make some specific assumptions about the programming and specification languages.

We assume that expressions of both the specification and programming language are built from two kinds of symbols: *fixed* symbols and *variable* symbols. The set of fixed symbols includes function and relation symbols, logical connectives, and programming language constructs. The set of variable symbols comprises *logical variables* and *program variables*. Logical variables cannot appear in programs, and although both program and logical variables can appear in specifications, only logical variables are permitted to be bound by quantifiers.

We assume that the semantics of the specification and programming languages assign to fixed symbols a single interpretation that does not change during the course of a computation. An interpretation for the variable symbols is called a *state*. A *computation* is a sequence of states. We assume that all computations are infinite; this convenient assumption results in no loss of generality because finite computations can be modeled by introducing a special "halt flag" into the state, and assuming that finite computations are made infinite by repeating the final state with the halt flag set.

For our concurrent programming language, we use a self-explanatory variant of Dijkstra's guarded command language [Dij76], augmented with a parallel construct $\|$. Communication between processes is accomplished through the use of shared variables. A multiple assignment statement of the form:

$$v_1, v_2, \ldots, v_n := t_1, t_2, \ldots, t_n,$$

where the v_i are program variables and the t_i are terms, is used to read and update a collection of variables in a single atomic step. We assume that process scheduling is *fair* in the sense that no process can be forever enabled without taking a step. It is straightforward to give a formal semantics to this programming language by defining a mapping from programs to sets of computations.

We assume that our specification language is the set of all sentences in the language of first-order temporal logic whose atomic formulas are formed from variables, function symbols, and relation symbols. In addition to the usual logical connectives and quantifiers, we assume the specification language contains the temporal operators $\Box$ (henceforth) and $\Diamond$ (eventually), which are applied to formulas to yield new formulas, and $\bigcirc$ (next), which can either be applied to a formula to yield a new formula, or to a term to yield a new term. We assume that these operators are endowed with "linear time" semantics in the usual way (see [MP83]), and we write $x \models \phi$ to indicate that the computation x satisfies the temporal sentence ϕ.

It will also be convenient to introduce the derived temporal operators $\leadsto$ (leads to), $\uparrow$

(increases), and $\downarrow$ (decreases), defined by:

$$\begin{array}{rcl} \phi \leadsto \psi & \equiv & \Box(\phi \supset \Diamond\psi) \\ t\uparrow & \equiv & t < \bigcirc t \\ t\downarrow & \equiv & t > \bigcirc t, \end{array}$$

where in the latter two definitions we assume that t is an integer-valued term and the relation symbols $>$ and $<$ denote the usual ordering relations on the integers.

4 Example 1: Distributed Synchronization

In this section we consider the problem of coordinating the accesses of N *user processes* to *critical sections*, the executions of which must be mutually exclusive. The coordination should be done in such a way as to avoid the phenomenon of *starvation*, in which one process is prevented forever from entering its critical section while other processes repeatedly enter and exit their critical sections.

Program Ring in Figure 1 is a distributed algorithm that solves the mutual exclusion problem. In program Ring, each user process, represented by the code labeled User_i, has been associated with an additional *node process* Node_i. The user process User_i communicates with the associated node process Node_i through the boolean variables $\mathsf{waiting}_i$ and $\mathsf{critical}_i$. When process User_i is ready to enter its critical section, it informs process Node_i by setting the variable $\mathsf{waiting}_i$ to true. Process User_i then waits for the variable $\mathsf{critical}_i$ to become true before entering its critical section. When process User_i finishes its critical section, it sets $\mathsf{critical}_i$ to false.

The node processes communicate with each other in a ring-like pattern; that is, process Node_i communicates with processes Node_{i-1} and Node_{i+1}, where we assume the addition and subtraction to be performed *modulo* N. Mutual exclusion is obtained through the use of a single *token*, which propagates around the ring in the forward direction (i.e., 0 to 1 to 2, ...), in response to *requests*, which propagate in the reverse direction. The process Node_i permits its user process User_i to execute in its critical section only while Node_i possesses the token. The current position of the token is recorded by the variables token_i, and requests are recorded by the variables $\mathsf{request}_i$.

The main loop of process Node_i operates as follows: If Node_i does not currently have the token, and if either User_i is waiting to enter its critical section, or Node_{i+1} wants the token, then Node_i must request the token from Node_{i-1} by setting $\mathsf{request}_i$ to true. If User_i is not waiting, and Node_{i+1} doesn't want the token, then there is nothing to do. If Node_i

has the token, and User_i is currently executing in its critical section, then there is also nothing to do. If Node_i has the token, and User_i is not in its critical section, then Node_i must examine the variables $\mathsf{waiting}_i$, $\mathsf{request}_{i+1}$, and sched_i to see what to do. If User_i is waiting, and Node_{i+1} doesn't want the token, then User_i is allowed into its critical section. If Node_{i+1} wants the token, and User_i is not waiting, then the token is passed to Node_{i+1}. If both User_i is waiting and Node_{i+1} wants the token, then the choice is resolved on the basis of the scheduling variable sched_i—if sched_i is true, then the token is passed to Node_{i+1}, and if sched_i is false, then User_i is allowed to enter its critical section. In either case, the variable sched_i is complemented to ensure that the opposite decision will be made next time.

Using standard concurrent program proof techniques (*e.g.*, [OG76] [MP83]), we can show that the program Ring satisfies the following invariants:

$$\mathsf{Ring} \models \Box \textstyle\bigwedge_{i=0}^{N-1}(\mathsf{critical}_i \supset \mathsf{token}_i) \tag{1}$$

$$\mathsf{Ring} \models \Box \left(\textstyle\sum_{i=0}^{N-1} \mathsf{token}_i = 1\right) \tag{2}$$

where the expression $\sum_{i=0}^{N-1} \mathsf{token}_i = 1$ denotes the first order formula that states that precisely one of the variables token_i is true.[1] These invariants together imply that program Ring has the mutual exclusion property

$$\mathsf{Ring} \models \Box \textstyle\bigwedge_{i \neq j}(\mathsf{critical}_i \supset \neg\mathsf{critical}_j)\ .$$

Besides the above invariants, we can show (for example, by the "proof lattice" techniques of [OL82] or by the "chain principle" of [MP83]), that program Ring satisfies the following rely/guarantee specification for all i with $0 \leq i \leq N-1$:

$$\mathsf{Ring} \models R_i \supset G_i,$$

where

$$\begin{aligned} R_i &\equiv \mathsf{critical}_i \leadsto \neg\mathsf{critical}_i \quad \wedge \quad \mathsf{request}_i \leadsto \mathsf{token}_i \\ G_i &\equiv \mathsf{request}_{i+1} \leadsto \mathsf{token}_{i+1} \quad \wedge \quad \mathsf{waiting}_i \leadsto \mathsf{critical}_i \end{aligned}$$

To prove these properties, we must make use of our fair scheduling assumption.

Our goal is to show that if critical sections always terminate, then no process waits forever to enter its critical section. That is,

$$\mathsf{Ring} \models R \supset G$$

[1] In the sequel, we shall occasionally write expressions like this, which although not themselves first-order formulas, can be regarded as denoting equivalent first-order formulas in an obvious way.

$$
\begin{array}{lll}
\mathsf{Ring} & \equiv & \mathbf{boolean}\ (\mathsf{token}_i\ \mathbf{initially}\ (\mathbf{if}\ i = 0\ \mathbf{then}\ \mathsf{true}\ \mathbf{else}\ \mathsf{false})) : (0 \le i \le N-1); \\
 & & \mathbf{boolean}\ (\mathsf{waiting}_i, \mathsf{critical}_i, \mathsf{request}_i, \mathsf{sched}_i \\
 & & \qquad\qquad \mathbf{initially}\ \mathsf{false}, \mathsf{false}, \mathsf{false}, \mathsf{false}) : (0 \le i \le N-1); \\
 & & \|_{i=0}^{N-1}\ (\mathsf{User}_i \parallel \mathsf{Node}_i); \\[1em]
\mathsf{User}_i & \equiv & \mathbf{do}\ \mathit{Noncritical\ Section}; \\
 & & \quad \mathsf{waiting}_i := \mathsf{true}; \\
 & & \quad \mathbf{do}\ \neg\mathsf{critical}_i \rightarrow \mathbf{skip}; \mathbf{od}; \\
 & & \quad \mathit{Critical\ Section}; \\
 & & \quad \mathsf{critical}_i := \mathsf{false}; \\
 & & \mathbf{od}; \\[1em]
\mathsf{Node}_i & \equiv & \mathbf{do}\ \neg\mathsf{token}_i \qquad\qquad\quad \rightarrow \mathbf{if}\ \neg\mathsf{request}_i \wedge (\mathsf{waiting}_i \vee \mathsf{request}_{i+1}) \ \rightarrow \mathsf{request}_i := \mathsf{true}; \\
 & & \qquad\qquad\qquad\qquad\qquad\ \ \square\ \mathsf{request}_i \vee (\neg\mathsf{waiting}_i \wedge \neg\mathsf{request}_{i+1}) \rightarrow \mathbf{skip}; \\
 & & \qquad\qquad\qquad\qquad\qquad\ \ \mathbf{fi}; \\
 & & \square\ \ \mathsf{token}_i \wedge \mathsf{critical}_i \quad \rightarrow \mathbf{skip}; \\
 & & \square\ \ \mathsf{token}_i \wedge \neg\mathsf{critical}_i \rightarrow \mathbf{if}\ \neg\mathsf{waiting}_i \wedge \neg\mathsf{request}_{i+1} \rightarrow \mathbf{skip}; \\
 & & \qquad\qquad\qquad\qquad\qquad\ \ \square\ \mathsf{request}_{i+1} \wedge (\neg\mathsf{waiting}_i \vee \mathsf{sched}_i) \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad \rightarrow \mathsf{token}_i, \mathsf{token}_{i+1}, \mathsf{request}_{i+1}, \mathsf{sched}_i \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad\quad := \mathsf{false}, \mathsf{true}, \mathsf{false}, \mathsf{false}; \\
 & & \qquad\qquad\qquad\qquad\qquad\ \ \square\ \mathsf{waiting}_i \wedge (\neg\mathsf{request}_{i+1} \vee \neg\mathsf{sched}_i) \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad \rightarrow \mathsf{waiting}_i, \mathsf{critical}_i, \mathsf{sched}_i := \mathsf{false}, \mathsf{true}, \mathsf{true}; \\
 & & \qquad\qquad\qquad\qquad\qquad\ \ \mathbf{fi}; \\
 & & \mathbf{od};
\end{array}
$$

Figure 1: Distributed Synchronization Algorithm

where

$$
\begin{aligned}
R &\equiv \bigwedge_{i=1}^{N}(\mathsf{critical}_i \leadsto \neg\mathsf{critical}_i) \\
G &\equiv \bigwedge_{i=1}^{N}(\mathsf{waiting}_i \leadsto \mathsf{critical}_i)
\end{aligned}
$$

Note that the property $\mathsf{Ring} \models R_i \supset G_i$ is *local* in the sense that it is stated solely in terms of variables that are referenced by the process Node_i. In contrast, the property $\mathsf{Ring} \models R \supset G$ is a *global* property that involves variables referenced by all processes. In general, we imagine that the proof rule presented in this paper will be most useful when it is used, as in this example, to reduce the proof of a global property to the proof of a collection of local properties.

To apply our rely/guarantee proof rule, we define the set of specifications

$$RG = \{RG_{i,j} : i,j \in \{0,1,\ldots N-1\} \cup \{\mathrm{ext}\}\}$$

as follows:

$$
RG_{i,j} \equiv \begin{cases}
\mathsf{waiting}_i \leadsto \mathsf{critical}_i, & 0 \le i \le N-1, j = \mathrm{ext} \\
\mathsf{critical}_j \leadsto \neg\mathsf{critical}_j, & i = \mathrm{ext}, 0 \le j \le N-1 \\
\mathsf{request}_j \leadsto \mathsf{token}_j, & 0 \le i,j \le N-1, j = i+1 \\
\mathsf{true}, & 0 \le i,j \le N-1, j \ne i+1.
\end{cases}
$$

With these definitions, the conditions required for RG to be a cut set for program Ring and specifications $R, G, \{R_i, G_i : 1 \le i \le N\}$, are tautological. To complete the proof that $\mathsf{Ring} \models R \supset G$ it therefore remains only to prove that RG is acyclic for Ring.

To prove the acyclicity condition we need consider only the cycle $\{(0,1),(1,2)...,(N-1,0)\}$, since all other cycles contain links (i,j) for which $j \ne i+1$ and hence for which $RG_{i,j} \equiv \mathsf{true}$. We show $\mathsf{Ring} \models \bigvee_{i=0}^{N-1} RG_{i,i+1}$ indirectly, by assuming the existence of a computation x of Ring such that $x \models \bigwedge_{i=0}^{N-1} \neg RG_{i,i+1}$, and deriving a contradiction.

Suppose $x \models \bigwedge_{i=0}^{N-1} \neg RG_{i,i+1}$. Then

$$x \models \bigwedge_{i=0}^{N-1} \neg(\mathsf{request}_i \leadsto \mathsf{token}_i).$$

Using the definition of $\leadsto$ and temporal reasoning, we have

$$x \models \bigwedge_{i=0}^{N-1} \Diamond(\mathsf{request}_i \wedge \Box(\neg\mathsf{token}_i)) \ .$$

Since the conjunction $\bigwedge_{i=0}^{N-1}$ is finite, it is valid (in linear-time temporal logic) to interchange it and the temporal operator $\Diamond$. Since $\bigwedge_{i=0}^{N-1}$ and $\Box$ are both of universal character, it is valid to to interchange them as well, yielding

$$x \models \Diamond\Box \bigwedge_{i=0}^{N-1} \neg\mathsf{token}_i \ .$$

This implies that

$$x \models \Diamond\Box\left(\sum_{i=0}^{N-1} \mathsf{token}_i = 0\right) ,$$

which contradicts invariant (2) above.

5 Example 2: Distributed Resource Allocation

In this section we consider the problem of allocating a fixed number of *resources* in response to requests from a collection of *user processes*. An algorithm to solve this problem should have the property that as long as the total number of requests issued by users does not exceed the number of originally available resources, a resource will eventually be issued in response to each user request.

Program Tree in Figure 2 is a distributed algorithm, based on the "dynamic match" algorithm of [FLG83], that solves the problem. As in program Ring of the previous example, each user process, labeled User_i, has been associated with a node process Node_i. The user process User_i communicates with the node process Node_i through the variable $\mathsf{pending}_i$, which represents the number of user requests that have not yet been satisfied. Process User_i starts out with an initial number of requests IREQ_i, which it issues to Node_i (by incrementing $\mathsf{pending}_i$) at unpredictable times during execution of the system. Process Node_i records the number of free resources it has in the variable free_i, which is initially set to the constant IFREE_i. Process Node_i "responds" to requests from User_i by decrementing $\mathsf{pending}_i$ and free_i – a practical algorithm would also transmit a capability for a resource to the user process as well, but we ignore this here.

In contrast to the previous example, where the communication pattern of the node processes was a ring, the communication pattern of the node processes in this example is a tree. The set T is the set of process identifiers, which we imagine to be arranged as a binary tree. For each process $i \in T$, we write $p(i), l(i), r(i)$ for the parent, left child, and right child, respectively, of process i. For uniformity, we introduce a special symbol nil, and define $p(i) = \mathsf{nil}$ when i is the root of the tree, and define $l(i) = r(i) = \mathsf{nil}$ when i is a leaf of the tree. Furthermore, we define $p(\mathsf{nil}) = l(\mathsf{nil}) = r(\mathsf{nil}) = \mathsf{nil}$. If $i \in T$, then let $D(i)$ represent the set of all $j \in T$ (including i itself, but omitting nil) that are *descendants* of i.

Certain of the steps of process Node_i, are to be omitted from the program in case i is the root or a leaf, respectively. These branches are indicated by comments in Figure 2.

If $i, j \in T$ and $i = p(j)$, then processes Node_i and Node_j communicate through the variables $\mathsf{owes}_{i,j}$ and $\mathsf{estim}_{i,j}$. Intuitively, the variable $\mathsf{owes}_{i,j}$ records the net number of

$$
\begin{array}{rcl}
\mathsf{Tree} & \equiv & \textbf{integer } (\mathsf{owes}_{i,j}, \mathsf{estim}_{i,j} \textbf{ initially } 0, \textstyle\sum_{k \in D(j)} \mathsf{IFREE}_k) : \\
& & \qquad ((j \in T \text{ and } i = p(j)) \text{ or } (i \in T \text{ and } j \in \{l(i), r(i)\})); \\
& & \textbf{integer } (\mathsf{pending}_i, \mathsf{free}_i \textbf{ initially } 0, \mathsf{IFREE}_i) : (i \in T); \\
& & \|_{i \in T} (\mathsf{User}_i \parallel \mathsf{Node}_i); \\
\\
\mathsf{User}_i & \equiv & \textbf{integer } \mathsf{request}_i \textbf{ initially } \mathsf{IREQ}_i; \\
& & \textbf{do } \mathsf{request}_i > 0 \to \mathsf{request}_i, \mathsf{pending}_i := \mathsf{request}_i - 1, \mathsf{pending}_i + 1; \\
& & \qquad \mathsf{request}_i \leq 0 \to \textbf{skip}; \\
& & \textbf{od}; \\
\\
\mathsf{Node}_i & \equiv & \textbf{do } \mathsf{pending}_i > 0 \land \mathsf{free}_i > 0 \hfill (\textit{issue resource to user}) \\
& & \qquad \to \mathsf{pending}_i, \mathsf{free}_i := \mathsf{pending}_i - 1, \mathsf{free}_i - 1; \\
& & \square \ \mathsf{owes}_{p(i),i} < 0 \land \mathsf{free}_i > 0 \quad (\textit{pay resource owed to parent} \ - \ \textit{i not root}) \\
& & \qquad \to \mathsf{owes}_{p(i),i}, \mathsf{free}_i, \mathsf{free}_{p(i)} := \mathsf{owes}_{p(i),i} + 1, \mathsf{free}_i - 1, \mathsf{free}_{p(i)} + 1; \\
& & \square \ \mathsf{owes}_{i,l(i)} > 0 \land \mathsf{free}_i > 0 \quad (\textit{pay resource owed to left child} \ - \ \textit{i not leaf}) \\
& & \qquad \to \mathsf{owes}_{i,l(i)}, \mathsf{free}_i, \mathsf{free}_{l(i)} := \mathsf{owes}_{i,l(i)} - 1, \mathsf{free}_i - 1, \mathsf{free}_{l(i)} + 1; \\
& & \square \ \mathsf{owes}_{i,r(i)} > 0 \land \mathsf{free}_i > 0 \quad (\textit{pay resource owed to right child} \ - \ \textit{i not leaf}) \\
& & \qquad \to \mathsf{owes}_{i,r(i)}, \mathsf{free}_i, \mathsf{free}_{r(i)} := \mathsf{owes}_{i,r(i)} - 1, \mathsf{free}_i - 1, \mathsf{free}_{r(i)} + 1; \\
& & \square \ \mathsf{DEFCT}_i > 0 \land \mathsf{estim}_{i,l(i)} > 0 \quad (\textit{forward request to left child}) \\
& & \qquad \to \mathsf{owes}_{i,l(i)}, \mathsf{estim}_{i,l(i)} := \mathsf{owes}_{i,l(i)} - 1, \mathsf{estim}_{i,l(i)} - 1; \\
& & \square \ \mathsf{DEFCT}_i > 0 \land \mathsf{estim}_{i,r(i)} > 0 \quad (\textit{forward request to right child}) \\
& & \qquad \to \mathsf{owes}_{i,r(i)}, \mathsf{estim}_{i,r(i)} := \mathsf{owes}_{i,r(i)} - 1, \mathsf{estim}_{i,r(i)} - 1; \\
& & \square \ \mathsf{DEFCT}_i > 0 \land \mathsf{estim}_{i,l(i)} \leq 0 \land \mathsf{estim}_{i,r(i)} \leq 0 \quad (\textit{reject request up to parent}) \\
& & \qquad \to \mathsf{owes}_{p(i),i}, \mathsf{estim}_{p(i),i} := \mathsf{owes}_{p(i),i} + 1, 0; \\
& & \square \ \mathsf{DEFCT}_i \leq 0 \land (\mathsf{free}_i \leq 0 \lor (\mathsf{pending}_i \leq 0 \quad (\textit{nothing to do, idle}) \\
& & \qquad\qquad \land \ \mathsf{owes}_{p(i),i} \geq 0 \land \mathsf{owes}_{i,l(i)} \leq 0 \land \mathsf{owes}_{i,r(i)} \leq 0)) \\
& & \qquad \to \textbf{skip}; \\
& & \textbf{od};
\end{array}
$$

where

$$
\mathsf{DEFCT}_i = (\mathsf{pending}_i + \mathsf{owes}_{i,l(i)} + \mathsf{owes}_{i,r(i)}) - (\mathsf{free}_i + \mathsf{owes}_{p(i),i})
$$

Figure 2: Distributed Resource Allocation Algorithm

resources that Node_i owes to Node_j. If $\mathsf{owes}_{i,j}$ is positive, then Node_i owes resources to Node_j. If $\mathsf{owes}_{i,j}$ is negative, then Node_j owes resources to Node_i. The variable $\mathsf{estim}_{i,j}$ contains an estimate of the number of free resources remaining in the subtree headed by j. It is initially set to the total number of free resources initially available in the subtree headed by j. The important invariant property of this estimate is that it is always *optimistic*; that is, $\mathsf{estim}_{i,j}$ is always greater than or equal to the number of free resources actually available in the subtree headed by j.

Intuitively, the steps of process Node_i serve either to satisfy a pending user request with a locally available resource, to pay a resource owed to a neighboring node, or to reduce a *projected deficit* of resources at node i. The quantity DEFCT_i in the code for process Node_i represents this projected deficit, and should be thought of as the amount by which requests exceed resources at node i, once all debts have been paid. If process Node_i projects a deficit ($\mathsf{DEFCT}_i > 0$), then to reduce this deficit, it can either *forward* a request to its left or right child, or *reject* a request to its parent. Requests are forwarded to a child only in case it is estimated that there is a surplus of resources in the subtree headed by that child. Requests are rejected to the parent only if neither of the subtrees headed by the child nodes are estimated to have a surplus of resources.

The program Tree can be shown, by standard techniques, to satisfy the following invariants:

$$\mathsf{Tree} \models \Box(\mathsf{owes}_{\mathsf{nil},\mathsf{root}} \geq 0), \tag{1}$$

$$\mathsf{Tree} \models \Box \bigwedge_{i \in T} \left(\mathsf{owes}_{p(i),i} > 0 \ \supset \ \mathsf{owes}_{p(i),i} \leq \textstyle\sum_{j \in D(i)} (\mathsf{pending}_j - \mathsf{free}_j)\right). \tag{2}$$

Invariant (2) expresses the fundamental relationship between amount owed and amount needed: If node i is owed resources by its parent, then the amount owed to i by its parent is a lower bound on the instantanous amount by which pending requests exceed available resources in the subtree rooted at i.

It can also be shown that Tree satisfies the following rely/guarantee specifications for all $i \in T$:

$$\mathsf{Tree} \models R_i \supset G_i,$$

where

$$\begin{aligned} R_i \ \equiv \ & \mathsf{owes}_{p(i),i} > 0 \leadsto \mathsf{owes}_{p(i),i} \downarrow \\ & \wedge \ \mathsf{owes}_{i,l(i)} < 0 \leadsto \mathsf{owes}_{i,l(i)} \uparrow \\ & \wedge \ \mathsf{owes}_{i,r(i)} < 0 \leadsto \mathsf{owes}_{i,r(i)} \uparrow \end{aligned}$$

$$
\begin{aligned}
G_i \quad \equiv \quad & \mathsf{pending}_i > 0 \leadsto \mathsf{pending}_i \downarrow \\
& \wedge \ \mathsf{owes}_{p(i),i} < 0 \leadsto \mathsf{owes}_{p(i),i} \uparrow \\
& \wedge \ \mathsf{owes}_{i,l(i)} > 0 \leadsto \mathsf{owes}_{i,l(i)} \downarrow \\
& \wedge \ \mathsf{owes}_{i,r(i)} > 0 \leadsto \mathsf{owes}_{i,r(i)} \downarrow
\end{aligned}
$$

The rely condition R_i states that debts owed to node i by its parent and each of its children will eventually be paid. The guarantee condition G_i states that debts owed by node i to its parent and each of its children will eventually be paid. To obtain these properties, we must assume the scheduling of the branches of the main loop in the node program is *strongly fair*, in the sense that no branch that is enabled infinitely often during the course of a computation can fail to be selected during that computation.[2]

We are interested in establishing that, assuming the total number of user requests never exceeds the total number of resources initially available, then a resource will eventually be issued for every user request. Formally, we would like to show:

$$\mathsf{Tree} \models R \supset G,$$

where

$$
\begin{aligned}
R \quad & \equiv \quad \Box \left(\textstyle\sum_{i \in T} \mathsf{pending}_i \leq \sum_{i \in T} \mathsf{free}_i\right) \\
G \quad & \equiv \quad \left(\textstyle\sum_{i \in T} \mathsf{pending}_i > 0\right) \leadsto \left(\textstyle\sum_{i \in T} \mathsf{pending}_i\right) \downarrow
\end{aligned}
$$

That this property holds is not immediately obvious. Examples of the kinds of things that might go wrong are resources being shuttled endlessly around the system without ever reaching nodes where they are needed, and nodes with surplus resources never receiving requests from nodes with deficits.

To apply our rely/guarantee proof rule, we define the set of specifications

$$RG = \{RG_{i,j} : i, j \in T \cup \{\text{ext}\}\}$$

as follows:

$$
RG_{i,j} \equiv \begin{cases}
\Box(\mathsf{owes}_{\mathsf{nil},\mathsf{root}} = 0), & i = \text{ext}, j = \mathsf{root} \\
\mathsf{true}, & i = \text{ext}, j \in T - \mathsf{root} \\
\mathsf{pending}_i > 0 \leadsto \mathsf{pending}_i \downarrow, & i \in T, j = \text{ext} \\
\mathsf{owes}_{i,j} > 0 \leadsto \mathsf{owes}_{i,j} \downarrow, & i, j \in T, i = p(j) \\
\mathsf{owes}_{j,i} < 0 \leadsto \mathsf{owes}_{j,i} \uparrow, & i, j \in T, j = p(i) \\
\mathsf{true}, & i, j \in T, j \neq p(i), i \neq p(j).
\end{cases}
$$

[2] Actually, we can make do with the weaker fairness condition used in Example 1 if we introduce scheduling variables as we did there. We have omitted scheduling variables here in the interests of simplicity.

We must first show that RG is a cut set. To prove condition (1) in the definition of a cut set, we must show that

$$\mathsf{Tree} \models R \supset (\bigwedge_{j \in T} RG_{\mathrm{ext},j}),$$

which, applying the definitions of R and $RG_{\mathrm{ext},j}$, becomes

$$\mathsf{Tree} \models \Box\,(\sum_{i \in T} \mathsf{pending}_i \leq \sum_{i \in T} \mathsf{free}_i) \supset \Box(\mathsf{owes}_{\mathsf{nil},\mathsf{root}} = 0)\ .$$

Suppose x is a computation of Tree such that

$$x \models \Box\,(\sum_{i \in T} \mathsf{pending}_i \leq \sum_{i \in T} \mathsf{free}_i)\ .$$

Then

$$x \models \Box\,(\sum_{i \in T} \mathsf{free}_i - \mathsf{pending}_i \geq 0)\ . \tag{3}$$

From the fundamental invariant (2) above, and the fact that $D(\mathsf{root}) = T$, we infer that

$$x \models \Box(\mathsf{owes}_{\mathsf{nil},\mathsf{root}} > 0 \quad \supset \quad \mathsf{owes}_{\mathsf{nil},\mathsf{root}} \leq \sum_{i \in T} \mathsf{pending}_i - \mathsf{free}_i)\ .$$

From this and (3), we conclude that

$$x \models \Box(\mathsf{owes}_{\mathsf{nil},\mathsf{root}} > 0 \supset \mathsf{owes}_{\mathsf{nil},\mathsf{root}} \leq 0)\ ,$$

which, combined with the invariant (1), implies that

$$x \models \Box(\mathsf{owes}_{\mathsf{nil},\mathsf{root}} = 0),$$

as required.

To prove condition (2) in the definition of a cut set, we must show that

$$\mathsf{Tree} \models (\bigwedge_{i \in T} RG_{i,\mathrm{ext}}) \supset G,$$

that is,

$$\begin{aligned} \mathsf{Tree} \models\ & \bigwedge_{i \in T} (\mathsf{pending}_i > 0 \leadsto \mathsf{pending}_i \downarrow) \\ & \qquad \supset ((\sum_{i \in T} \mathsf{pending}_i > 0) \leadsto (\sum_{i \in T} \mathsf{pending}_i) \downarrow) \end{aligned}$$

This is obviously true, because at most one of the $\mathsf{pending}_i$ can change in a single step of execution.

To prove condition (3), we must show that

$$\mathsf{Tree} \models (\bigwedge_{i \in T \cup \{\mathrm{ext}\}} RG_{i,j}) \supset R_j, \quad \text{for all } j \in T.$$

We split the proof into two cases, $j = \mathsf{root}$ and $j \in T - \mathsf{root}$. In case $j = \mathsf{root}$, we must show

$$\begin{aligned}
\mathsf{Tree} \models\ & (\Box(\mathsf{owes}_{\mathsf{nil},\mathsf{root}} = 0) \\
& \quad \wedge\ \mathsf{owes}_{\mathsf{root},l(\mathsf{root})} < 0 \leadsto \mathsf{owes}_{\mathsf{root},l(\mathsf{root})} \uparrow \\
& \quad \wedge\ \mathsf{owes}_{\mathsf{root},r(\mathsf{root})} < 0 \leadsto \mathsf{owes}_{\mathsf{root},r(\mathsf{root})} \uparrow) \\
& \supset \\
& (\mathsf{owes}_{\mathsf{nil},\mathsf{root}} > 0 \leadsto \mathsf{owes}_{\mathsf{nil},\mathsf{root}} \downarrow \\
& \quad \wedge\ \mathsf{owes}_{\mathsf{root},l(\mathsf{root})} < 0 \leadsto \mathsf{owes}_{\mathsf{root},l(\mathsf{root})} \uparrow \\
& \quad \wedge\ \mathsf{owes}_{\mathsf{root},r(\mathsf{root})} < 0 \leadsto \mathsf{owes}_{\mathsf{root},r(\mathsf{root})} \uparrow)
\end{aligned}$$

This is obviously true.

In case $j \in T - \mathsf{root}$, we must show

$$\begin{aligned}
\mathsf{Tree} \models\ & (\mathsf{owes}_{p(j),j} > 0 \leadsto \mathsf{owes}_{p(j),j} \downarrow \\
& \quad \wedge\ \mathsf{owes}_{j,l(j)} < 0 \leadsto \mathsf{owes}_{j,l(j)} \uparrow \\
& \quad \wedge\ \mathsf{owes}_{j,r(j)} < 0 \leadsto \mathsf{owes}_{j,r(j)} \uparrow) \\
& \supset \\
& (\mathsf{owes}_{p(j),j} > 0 \leadsto \mathsf{owes}_{p(j),j} \downarrow \\
& \quad \wedge\ \mathsf{owes}_{j,l(j)} < 0 \leadsto \mathsf{owes}_{j,l(j)} \uparrow \\
& \quad \wedge\ \mathsf{owes}_{j,r(j)} < 0 \leadsto \mathsf{owes}_{j,r(j)} \uparrow),
\end{aligned}$$

which is a tautology.

To prove condition (4), we must show that

$$\mathsf{Tree} \models G_i \supset (\textstyle\bigwedge_{j \in T \cup \{\mathrm{ext}\}} RG_{i,j}), \qquad \text{for all } i \in T.$$

Using the definitions of R_i and $RG_{i,j}$, this becomes

$$\begin{aligned}
\mathsf{Tree} \models\ & (\mathsf{pending}_i > 0 \leadsto \mathsf{pending}_i \downarrow \\
& \quad \wedge\ \mathsf{owes}_{p(i),i} < 0 \leadsto \mathsf{owes}_{p(i),i} \uparrow \\
& \quad \wedge\ \mathsf{owes}_{i,l(i)} > 0 \leadsto \mathsf{owes}_{i,l(i)} \downarrow \\
& \quad \wedge\ \mathsf{owes}_{i,r(i)} > 0 \leadsto \mathsf{owes}_{i,r(i)} \downarrow) \\
& \supset \\
& (\mathsf{pending}_i > 0 \leadsto \mathsf{pending}_i \downarrow \\
& \quad \wedge\ \mathsf{owes}_{p(i),i} < 0 \leadsto \mathsf{owes}_{p(i),i} \uparrow \\
& \quad \wedge\ \mathsf{owes}_{i,l(i)} > 0 \leadsto \mathsf{owes}_{i,l(i)} \downarrow \\
& \quad \wedge\ \mathsf{owes}_{i,r(i)} > 0 \leadsto \mathsf{owes}_{i,r(i)} \downarrow),
\end{aligned}$$

which is a tautology.

Finally, we must show that RG is acyclic for Tree. To do this, it suffices to show that $\mathsf{Tree} \models RG_{i,p(i)} \vee RG_{p(i),i}$ for all $i \in T - \mathsf{root}$. This is because every cycle

$$\{(i_0, i_1), (i_1, i_2), \ldots, (i_{n-1}, i_n)\}$$

of T either contains a link (i_k, i_{k+1}) for which $RG_{i_k,i_{k+1}} = \mathsf{true}$ by definition, or else contains both links $(i, p(i))$ and $(p(i), i)$ for some $i \in T - \mathsf{root}$.

To show that $\mathsf{Tree} \models RG_{i,p(i)} \vee RG_{p(i),i}$ for all $i \in T - \mathsf{root}$, let i be arbitrarily fixed, and suppose, to obtain a contradiction, that x is a computation of Tree such that

$$x \models \neg RG_{i,p(i)} \wedge \neg RG_{p(i),i}. \tag{4}$$

From (4) and the definition of $RG_{i,p(i)}$ we know that

$$x \models \Diamond(\mathsf{owes}_{p(i),i} < 0 \wedge \Box\neg\mathsf{owes}_{p(i),i} \uparrow),$$

which implies that

$$x \models \Diamond\Box(owes_{p(i),i} < 0).$$

Similarly, from (4) and the definition of $RG_{p(i),i}$ we have that

$$x \models \Diamond\Box(owes_{p(i),i} > 0).$$

These two statements are contradictory, and we conclude that RG is acyclic.

6 Comparison With Other Techniques

To obtain perspective on the rely/guarantee proof method presented here, it is useful to compare this method with other extant methods. In this section we consider two methods: the "proof lattice" method of Owicki and Lamport [OL82], and the "well-founded set" method originally applied by Floyd [Flo67] to termination proofs for sequential programs, and later adapted by Manna, Pnueli [MP83], and others to prove eventuality properties expressed in temporal logic. Below we sketch how alternative proofs of the property $\mathsf{Ring} \models R \supset G$ might be constructed for the distributed synchronization example. The reader is challenged to produce simple proofs, at an adequate level of rigor, along the lines sketched. The author's own inability to accomplish this is what led him to devise the rely/guarantee proof technique.

6.1 Proof Lattice Method

The proof lattice method of Owicki and Lamport is designed to permit the proof of temporal implications of the form $\phi \leadsto \psi$ from simpler implications of the same form, plus auxiliary invariance properties of the program under consideration. A proof lattice for the program $P \models \phi \leadsto \psi$ is a finite, directed, acyclic graph, whose nodes are labeled by temporal sentences, with the following properties:

1. There is a single root node, labeled by ϕ.
2. There is a single leaf node, labeled by ψ.
3. If the children of a node labeled by ρ are labeled by $\sigma_1, \sigma_2, \ldots, \sigma_n$, then
$$P \models \rho \leadsto (\sigma_1 \vee \sigma_2 \vee \ldots \vee \sigma_n).$$

A proof lattice for $P \models \phi \leadsto \psi$ represents a sufficiently rigorous proof when each node labeled ρ, with children labeled $\sigma_1, \sigma_2, \ldots, \sigma_n$, can be justified by appeal to primitive inference rules associated with the constructs of the programming language, by appeal to an auxiliary invariance property, or by appeal to a theorem of temporal logic.

To use the proof lattice technique to prove the statement $\mathsf{Ring} \models R \supset G$, we might assume R, (that is, we consider a computation x such that $x \models \bigwedge_{i=0}^{N-1} \mathsf{critical}_i \leadsto \neg\mathsf{critical}_i$), and attempt to construct a proof lattice for $\mathsf{waiting}_i \leadsto \mathsf{critical}_i$. The informal content of the argument that would be captured formally by the proof lattice is as follows: We would show that if $\mathsf{waiting}_i$ holds, then a chain of requests is generated that propagates around the ring in the reverse direction until a node is reached that has the token. The token is then forced to propagate in the forward direction around the ring until node i is reached. Once node i is reached, then depending upon the value of sched_i, either $\mathsf{critical}_i$ will become true right away, or the token will be passed to node $i+1$. In the latter case, we have to follow another chain of requests and subsequent token passes until the token again reaches node i.

In the construction of the proof lattice, we would make use of simple eventuality properties like the following, which can be verified by local reasoning about the control flow within the process Node_i:

$$\mathsf{Ring} \models \mathsf{waiting}_i \leadsto \mathsf{critical}_i \vee \mathsf{request}_i$$
$$\mathsf{Ring} \models \mathsf{request}_i \leadsto \mathsf{token}_i \vee \mathsf{request}_{i-1}$$
$$\mathsf{Ring} \models \mathsf{token}_i \wedge \mathsf{waiting}_i \leadsto \mathsf{critical}_i \vee \neg\mathsf{sched}_i$$
$$\mathsf{Ring} \models \mathsf{waiting}_i \wedge \mathsf{token}_i \wedge \neg\mathsf{sched}_i \leadsto \mathsf{critical}_i$$

In addition, we would make use of safety properties like the following:

$$\begin{aligned}
&\mathsf{Ring} \models \mathsf{waiting}_i \ \textit{latches-until}\ \mathsf{critical}_i\\
&\mathsf{Ring} \models \mathsf{request}_i \ \textit{latches-until}\ \mathsf{token}_i\\
&\mathsf{Ring} \models \mathsf{sched}_i \ \textit{latches-until}\ \mathsf{token}_{i+1}\\
&\mathsf{Ring} \models \neg\mathsf{sched}_i \ \textit{latches-until}\ \mathsf{critical}_i\\
&\mathsf{Ring} \models \mathsf{token}_i \ \textit{latches-until}\ \mathsf{token}_{i+1}\\
&\mathsf{Ring} \models \Box(\mathsf{request}_i \supset \neg\mathsf{token}_i)\\
&\mathsf{Ring} \models \Box(\mathsf{critical}_i \supset \mathsf{token}_i),
\end{aligned}$$

where ϕ *latches-until* ψ means, intuitively, "If ϕ ever holds, then ϕ remains true from then until the next instant at which ψ holds." (See [SM81] for a formal definition of this construct.)

If one actually tries to construct a proof lattice according to the preceding informal sketch, one is quickly overwhelmed by the number of branches and cases that it is necessary to consider. Problems are also caused by the fact that the depth of the lattice is dependent upon the parameter N, which is the size of the ring. This variable parameter necessitates the use of elipses in the proof lattice.

6.2 Well-Founded Set Method

Another alternative to the rely/guarantee method is to use a method based on well-founded sets. In this approach, the proof of a statement $P \models \phi \leadsto \psi$, might proceed by contradiction as follows: Assume x is a computation of P such that $x \models \Diamond(\phi \wedge \Box\neg\psi)$. Define a *variant function* f that maps the program state into a well-founded set W (typically the nonnegative integers under the usual ordering), and prove the following properties:

$$P \models \Box((\phi \wedge \Box\neg\psi) \supset \Box\neg f \uparrow)$$

$$P \models \Box((\phi \wedge \Box\neg\psi) \leadsto \Box\Diamond f \downarrow)$$

The first condition states that, assuming ϕ holds at some instant, and $\neg\psi$ holds for that instant and all future instants, then the value of the variant function f does not increase from that instant on. The second condition states that, under the same assumptions, the value of f is repeatedly decreased. If $P \models \Diamond(\phi\wedge\Box\neg\psi)$, then we would have a contradiction with the well-foundedness of W. We conclude that $P \models \Box(\phi \supset \Diamond\psi)$; that is, $P \models \phi \leadsto \psi$.

Let us consider how a well-founded set proof of $\mathsf{Ring} \models R \supset G$ might proceed. Suppose, to obtain a contradiction, that x is a computation of Ring such that $x \models R \wedge \neg G$. Then for some i with $0 \leq i \leq N-1$, we have that $x \models \Diamond(\mathsf{waiting}_i \wedge \Box\neg\mathsf{critical}_i)$. Making use of the invariant that states that there is precisely one token in the system at all times, we know that for each state in x, there is precisely one j for which token_j is true. We select a variant function f that maps each program state to a nonnegative integer according to the following intuition: The value of f on a program state measures a kind of "distance" between that state and a "desired" state (one for which $\mathsf{critical}_i$ holds). In particular, f takes into account:

1. The distance around the ring the token has to travel from j to i.

2. The distance around the ring requests have yet to propagate from i to j.

3. The values of the scheduling variables sched_k for k on the path the token must take from j to i.

A appropriate f can be defined in the form of a polynomial in N, whose coefficients depend upon the program variables token_i, $\mathsf{request}_i$, and sched_i.

Having defined f, we must prove:

$$\mathsf{Ring} \models \Box((\mathsf{waiting}_i \wedge \Box\neg\mathsf{critical}_i) \supset \Box\neg f\uparrow)$$

$$\mathsf{Ring} \models \Box((\mathsf{waiting}_i \wedge \Box\neg\mathsf{critical}_i) \leadsto \Box\Diamond f\downarrow)$$

The first condition can be proved by a case analysis on all the kinds of steps that the program Ring might take. The second condition can be proved by showing that it is invariantly the case that there is an enabled process whose steps must decrease the variant function (for example, a node that has the token and whose next step must pass it along the ring closer to node i), and therefore by the fair scheduling assumption must eventually execute.

Although it seems intuitively clear that such a proof can in principle be carried out, the problem of doing so in a sufficiently rigorous, perhaps machine-checkable fashion seems formidable.

7 Discussion

7.1 A Decomposition Principle

In the examples presented in this paper, judicious selection of the local rely and guarantee conditions R_i and G_i, resulted in tautological, or nearly tautological "cut set" conditions, leaving most of the interesting content of the proof to be captured in the "acyclicity" part. This phenomenon suggests that the rely/guarantee proof technique might be valuable as a decomposition principle to be used during top-down design. This decomposition principle can be codified as follows:

> To decompose a module M, which is to satisfy the specification $R \supset G$, into a system of submodules $\{M_i : i \in I\}$, and to determine the specifications $\{R_i \supset G_i : i \in I\}$ that the submodules must satisfy, one should:
>
> 1. By considering what each module M_i relies on and guarantees to the external environment and each other module M_j, determine a collection of specifications $RG_{i,j}$ that satisfies the acyclicity condition and cut set conditions (1) and (2).
> 2. Use cut set conditions (3) and (4) as *definitions* of the rely and guarantee conditions R_i and G_i for component module i. Since the conditions R_i and G_i should be expressed in terms of information local to module i, this step can actually be used to help determine what variables need to be accessible to module i.
> 3. Verify that the resulting component module specifications $R_i \supset G_i$ are reasonable, in the sense of being "consistent" or "implementable." For example, $R_i \supset G_i$ should not be logically equivalent to false. Consistency can be checked either by completing the top-down decomposition to the level of primitive modules, or by performing checks at the abstract level [Sta84].

7.2 A Formal Logic of Rely/Guarantee Conditions

In this paper, we have stated the rely/guarantee proof rule as a general proof-structuring technique independent of any particular choice of specification or programming language. However, an interesting question is how the proof rule might be formalized as a formal rule of inference in a logic of rely/guarantee conditions. As discussed in Section 2, Hoare-like logics represent one way to do this for safety properties. Another possibility is suggested by the method of reasoning employed in the examples presented above. In these examples, we were concerned with establishing rely/guarantee properties $R \supset G$, where R and G were conjunctions of simple eventuality assertions, each of the form $q \leadsto r$ with q and r containing no temporal operators. We used the rely/guarantee technique to derive global rely/guarantee properties of a parallel composition of programs from local rely/guarantee properties of the component programs, plus auxiliary invariants satisfied by the composite program.

As an example of how the sort of reasoning used in the examples might be formalized as a rule for parallel composition of programs, consider a shared variable programming language like the one used informally in this paper. Correctness assertions for such a language might take the form:

$$p : \langle R \rangle P \langle G \rangle,$$

where p is a predicate on states (representing an invariant), and R and G are each conjunctions of simple eventuality assertions of the form: $q \leadsto r$. Informally, validity of such a formula would mean: "For all 'environment' programs E, if each step of E preserves the truth of p, and if $E \| P \models R$ holds, then each step of $E \| P$ preserves the truth of p, and $E \| P \models G$ holds as well." (We universally quantify over environment programs in the semantics because of our desire for the validity of an assertion about P to be independent of any particular context in which P might appear.)

The rely/guarantee proof technique might be incorporated into a formal proof rule for parallel composition of the following form (for composition of two processes):

$$\frac{\begin{array}{c} p : \langle RG_{\mathrm{ext},1} \wedge RG_{2,1} \rangle P_1 \langle RG_{1,\mathrm{ext}} \wedge RG_{1,2} \rangle, \\ p : \langle RG_{\mathrm{ext},2} \wedge RG_{1,2} \rangle P_2 \langle RG_{2,\mathrm{ext}} \wedge RG_{2,1} \rangle, \\ p \supset \bigvee_k r_k \end{array}}{p : \langle RG_{\mathrm{ext},1} \wedge RG_{\mathrm{ext},2} \rangle P_1 \| P_2 \langle RG_{1,\mathrm{ext}} \wedge RG_{2,\mathrm{ext}} \rangle},$$

where $\bigvee_k r_k$ represents the conjunction of all the right-hand sides of the eventuality formulas $q_k \leadsto r_k$ occuring in the assertions $RG_{1,2}$ and $RG_{2,1}$. The first two hypotheses above correspond to the cut-set conditions of our proof rule, and the third hypothesis to the acyclicity condition. We have used the special form of the eventuality assertions to simplify the acyclicity condition to an implication between predicates on states.

There are significant issues that remain to be examined before a complete proof system can be obtained along these lines. For example, we require a suitable fairness assumption on the parallel composition of processes, since in the absence of such an assumption there will be no interesting valid eventuality properties. Also, a proof of completeness is likely require an analysis of the notions of "weakest rely-condition," "strongest guarantee-condition," and the question of their expressibility in the assertion language.

8 Conclusion

We have examined a technique by which rely/guarantee statements of the form $P \models R \supset G$ can be inferred from a finite collection of rely/guarantee statements of the form $\{P \models R_i \supset G_i : i \in I\}$. The technique involves the discovery of a collection $RG = \{RG_{i,j} : i \in I \cup \{\text{ext}\}\}$ of specifications that "cut" the interdependence between the rely-conditions R_i and R, and the guarantee-conditions G_i and G, in a fashion analogous to the way in which a loop invariant cuts the dependence of one iteration on the preceding and succeeding iterations. An "acyclicity" condition must also be proved, to ensure that there are no computations of P for which the interdependence between the rely and guarantee conditions is degenerate. The utility of the proof technique was illustrated by two examples, in which the technique was used to infer "global" liveness properties of a system of concurrent processes from "local" liveness properties of the individual processes. We expect the inference of global properties from local ones to be the typical way in which the technique will be useful in practice. An interesting feature of the proof technique is the way in which it can be applied, with equal facility, to both ring-structured and tree-structured communication patterns.

In general, the discovery of a cut set RG for a program will require the use of intuition about why the program works correctly. Since discovery of a collection of loop invariants in the Floyd/Hoare approach to sequential program correctness can be viewed as a special case of the problem of finding a cut set, it will be at least as difficult in general to discover cut sets as it is to discover loop invariants. We therefore consider it unlikely that the proof technique presented here can be fully automated. However, once a human verifier has discovered an appropriate cut set for a program, along with necessary global invariants, it seems quite possible that the checking of the cut set and acyclicity conditions is a task that is within the capability of an automated verification system.

Acknowledgement

The author wishes to thank Professor Nancy Lynch for her support and guidance during his thesis research. Gael Buckley, Jieh Hsiang, and Scott Smolka made helpful comments on drafts of this paper.

References

[BK83] H. Barringer, R. Kuiper, "A Temporal Logic Specification Method Supporting Hierarchical Development," Manuscript, University of Manchester Department of Computer Science, November, 1983.

[BKP84] H. Barringer, R. Kuiper, A. Pnueli, "Now You May Compose Temporal Logic Specifications," *Sixteenth ACM Symposium on Theory of Computing*, 1984.

[Dij76] E. W. Dijkstra, *A Discipline of Programming*, Prentice Hall, 1976.

[FLG83] M. J. Fischer, N. D. Griffeth, L. J. Guibas, N. A. Lynch, "Probabilistic Analysis of a Network Resource Allocation Algorithm," to appear in *Information and Control*.

[Flo67] R. W. Floyd, "Assigning Meanings to Programs," in *Mathematical Aspects of Computer Science*, American Math. Soc., 1967.

[HO80] B. T. Hailpern, S. S. Owicki, "Verifying Network Protocols Using Temporal Logic," Technical Report No. 192, Computer Systems Laboratory, Stanford University, June, 1980.

[Hoa69] C. A. R. Hoare, "An Axiomatic Basis for Computer Programming," *Comm. ACM*, Vol. 21, October, 1969.

[Jon81] C. B. Jones, "Development Methods for Computer Programs Including a Notion of Interference," Wolfson College, June, 1981.

[Jon83] C. B. Jones, "Specification and Design of (Parallel) Programs," *IFIP Conference*, 1983.

[Lam80] L. Lamport, "'Sometime' is Sometimes 'Not Never'," *Seventh ACM Conference on Principles of Programming Languages*, 1980.

[Lam83] L. Lamport, "Specifying Concurrent Program Modules," *ACM Transactions on Programming Languages and Systems,* 5, 2 (April, 1983), 190-222.

[Lis79] B. H. Liskov, "Modular Program Construction Using Abstractions," MIT Computation Structures Group Memo 184, September, 1979.

[MP83] Z. Manna, A. Pnueli, "Verification of Concurrent Programs: A Temporal Proof System," Stanford University Report No. STAN-CS-83-967, June, 1983.

[MC81] J. Misra, K. M. Chandy, "Proofs of Networks of Processes," *IEEE Trans. on Software Eng.*, SE-7, 4, (July, 1981).

[MCS82] J. Misra, K. M. Chandy, T. Smith, "Proving Safety and Liveness of Communicating Processes with Examples," *ACM Conf. on Principles of Distributed Computing*, 1982.

[OG76] S. S. Owicki, D. Gries, "Verifying Properties of Parallel Programs: An Axiomatic Approach," *Comm. ACM* 15, 5 (1976).

[OL82] S. S. Owicki, L. Lamport, "Proving Liveness Properties of Concurrent Programs," *ACM Transactions on Programming Languages and Systems*, 4, 3 (July 1982), 455-495.

[Pnu77] A. Pnueli, "The Temporal Logic of Programs," *IEEE Symposium on Foundations of Computer Science*, 1977.

[SM81] R. L. Schwartz, P. M. Melliar-Smith, "Temporal Logic Specification of Distributed Systems," *Second International Conference on Distributed Systems*, INRIA, France, April, 1981.

[Sta84] E. W. Stark, "Foundations of a Theory of Specification for Distributed Systems," M.I.T. Laboratory for Computer Science MIT/LCS/TR-342, August, 1984.

[Wir71] N. Wirth, "Program Development by Stepwise Refinement," *Comm. ACM* 14, 4 (April, 1971), 221-227.