Composing Processes Using Modified Rely-Guarantee Specifications

Rajit Manohar and Paolo A.G. Sivilotti *
Computer Science 256-80
California Institute of Technology
Pasadena, California 91125
{*rajit,paolo*}*@cs.caltech.edu*

June 12, 1996

Abstract

We present a specification notation for components of concurrent systems and an accompanying proof methodology for reasong about the composition of these components. The specification construct is motivated by rely-guarantee pairs and by any-component program properties. The proof technique is based on an implication ladder and on two basic properties from which more complex properties are derived. Two examples illustrate the simplicity and compositionality of the model, and demonstrate how the model can be used to create structured and reusable proofs of distributed systems.

Key words: Specifications, Composition, Distributed Systems, Rely-Guarantee.

1 Introduction

Driven by demands for price-performance, enhanced communication, reactivity, and robustness, there has been a recent proliferation of loosely-coupled distributed systems, as reflected in the emergence of standards such as CORBA [Obj95] and technologies such as Java [GJS96] and the Web [BLCGP92]. These systems exhibit a higher degree of heterogeneity than has been typical in the past. We anticipate cooperating processes to be developed by different authors, at different times, and to be bound dynamically at run-time [CR97]. For this reason, it will be important for an application developer to be able to specify and validate each component in a distributed system in isolation, and then reason about the correctness of the resulting computation as processes are composed.

Much as a procedure in a sequential program can be specified by the guarantees it makes on its behavior under assumptions about its initial state, we would like to specify a process

*This research is supported in part by AFOSR grant 91-0070, in part by NSF grants CCR-912008 and CCR-9527130, in part by an IBM Computer Science Fellowship, and in part by the Defense and Advanced Research Projects Agency under the Office of Army Research.

in a distributed system by the guarantees it makes on its behavior under assumptions about its environment. We propose such a specification construct, based on rely-guarantee [AL93] pairs and all-component and any-component properties for distributed systems [Cha94]. This construct is an extension of the former, in that the rely clause is not restricted to safety properties, at the cost of a more complicated proof methodology; it is a restriction on the latter, in that only local properties of components are used, with the benefit that component validations can be carried out in complete isolation (and hence are reusable).

This paper defines the proposed specification construct and the derives the related composition theorem. The methodology is explored by specifying and validating two synchronization problems. The first demonstrates the strengths of the approach in independent component analysis and proof reuse. The second quantifies the limits of the expressivity of our restricted specification construct.

2 Execution Model

In this section we describe an execution model for concurrent programs. This model is suitable for describing loosely-coupled distributed applications. The model serves as a basis for reasoning about the correctness of these applications and their compositions.

Processes. A *process* is a unit of protection, i.e., it does not share variables with any other process in the computation. For simplicity, a process contains a single thread of execution. All the results presented here can be extended to multi-threaded processes without significant change. The state of a process is embodied in the values of its variables, including its program counter. A process cannot directly modify the states of other processes; interaction between processes is achieved by the exchange of messages. A *program* is a finite collection of processes. The execution of a program can be described using a sequence of states, where the next state is obtained by the execution of a single step of the program.

Channels. Processes communicate by sending messages on *channels*. A channel is a first-in first-out message buffer. The process that inserts messages into the channel is known as the *sender*, and the process that removes messages from the buffer is known as the *receiver*. We restrict processes to those in which channels have only a single sender and a single receiver. Channels are reliable, in that no messages are lost or duplicated. Sends on channels never block, and receives block only if there are no pending messages.

The state of a channel c is characterized by four "ghost" variables:

1. `#sent`(c) is the number of messages inserted into the channel by the sender.
2. `#rcv`(c) is the number of messages removed from the channel by the receiver.
3. `#del`(c) is the number of messages that have been delivered by the underlying message-passing system to the receiver.
4. `hist`(c) is the sequence of messages that have been sent on the channel

Notation. A *predicate* is a set of states. Predicates can also be defined to be the characteristic function of the set of states they represent. Examples of predicates are the set of possible initial states and final states of a computation (the *preconditions* and *postconditions* of [Hoa69]).

A *property* is a set of program executions, that is, a set of sequences of states. Like predicates, properties can be defined to be the characteristic function of the corresponding set of states. Examples of program properties are the *unless* and *ensures* properties of UNITY [CM88]. The modified rely-guarantee specification statement presented here is another example of a program property.

Our proof format is from [DS90]. We denote quantification using the expression

$$\langle Q\ x : r.x :: t.x \rangle$$

where Q is a quantifier such as $\forall$, x is an unordered list of identifier names (dummies), $r.x$ (the range) is a predicate, and $t.x$ (the term) is an expression of the appropriate type. When $r.x$ is true everywhere or understood, we omit the range and write the quantification as

$$\langle Q\ x :: t.x \rangle \quad .$$

A process can be represented in various ways:

- A sequential program with sends and receives on channels.
- A set of *actions*, guarded with boolean expressions, written as $G_i \rightarrow S_i$. An action is said to be *enabled* when its guard is true. Since the set of actions represents a single thread of control, any pair of guards must be mutually exclusive.

3 Proof Methodology

We discuss compositionality of proofs and show how to use it with basic properties about safety and progress.

3.1 Compositionality

The key to compositional proofs is local reasoning: we would like to be able to prove important properties about a process without examining its environment in detail. This not only simplifies the proof of the properties in question, but it also enables us to reuse the properties of the process when it is composed with different environments.

Guarantee. A *guarantee* property, as defined in [Cha94], is an example of an any-component property used to reason about the parallel composition of a process with another. For a process P, a guarantee property (R, G) is defined as follows:

$$(R, G).P = \langle \forall\ E : E \text{ is a legal environment for } P :: R.(E \parallel P) \Rightarrow G.(E \parallel P) \rangle$$

(R, G) holds for P just when given any legal environment E for P, when R holds for $E \parallel P$ then G holds for $E \parallel P$. The problem with using such a flexible definition is that both R and G can refer to variables that are global to the computation. As a result, we have to consider all legal environments to establish such a guarantee property.

Modified Rely-Guarantee. We now make use of rely-guarantee properties with the following restriction: clauses in the rely-guarantee property can refer only to local variables in the process. The definition remains the same:

$$(R \rhd G).P = \langle \forall\ E : E \text{ is a legal environment for } P :: R.(E \parallel P) \Rightarrow G.(E \parallel P)\rangle$$

Rely-guarantee properties which conform to this restriction, are *modified rely-guarantee* properties. Observe that the implication can be established without considering all legal environments, since the properties R and G only refer to local variables of P. We will present a theorem which allows us to restate the above definition as:

$$(R \rhd G).P = R.P \Rightarrow G.P$$

Theorem 1 (Composition Theorem) *Let* $(R_1 \rhd G_1), \ldots, (R_n \rhd G_n)$ *be a collection of modified rely-guarantee properties of a program that relies on property* R. *Assume that, for all* k*,*

$$R \wedge \langle \forall\ j : j < k :: R_j \wedge G_j\rangle \Rightarrow R_k$$

holds. Then,

$$(R \rhd \langle \forall\ i :: R_i \wedge G_i\rangle)$$

is a property of the program.

Proof: The proof is obtained by induction on i. ∎

3.2 Program Properties

The initial state of a program is captured by the predicate *initially*. *initially.p* is true for a program P if the predicate p holds initially.

A *safety property* is one that specifies that the program never executes erroneously. Examples of safety properties are:

- variable x is always nonnegative;
- a message is received only if it has been sent.

In our model, safety properties are derived from a single relation, *next* .

Definition 1 (next) Given predicates p and q, the property p *next* q holds for program P just when execution of a single step of P from any state that satisfies p results in a state that satisfies q. Observe that since a program can always do nothing, if p *next* q holds for a process P, then $p \Rightarrow q$ must be true.

A *progress property* is one that specifies that the program eventually performs the computation we desire. Examples of progress properties are:

- variable x will become positive eventually;

- a message that is sent will be received eventually.

In our model, progress properties are derived from a single property, *transient*.

Definition 2 (transient) Given predicate p, $transient.p$ holds for a program P just when all executions of P which include a state where p holds also include a later state in which p is false. If p is always false, then $transient.p$ holds.

Theorem 2 (Compositionality) *Let* $(R \rhd G)$ *be a property of program* P*. Then,* $(R \rhd G)$ *is a property of* $P \parallel Q$ *for any program* Q*.*

Proof: The proof is obtained by structural induction on properties. ∎

We are simpler than TLA [Lam94] since our model can be expressed using TLA, and there are properties of TLA we cannot express.

Restricting safety properties in this way permits us to establish the following theorem:

Theorem 3 (Composing Safety Properties) *Let* $(R_1 \rhd G_1), \ldots, (R_n \rhd G_n)$ *be a collection of modified rely-guarantee properties of a program that relies on property* R*, and let all the properties be safety properties. Let* $R, R_1, \ldots, R_n$ *hold initially and assume that, for all* k*,*

$$R \wedge \langle \forall\ j : j \neq k :: G_j \rangle \Rightarrow R_k$$

holds. Then,

$$(R \rhd \langle \forall\ j :: R_j \wedge G_j \rangle)$$

is a property of the system.

Proof: The proof is obtained by induction on the sequence of states in the computation. ∎

3.3 Some more properties

We now define certain properties that are commonly used in proofs of distributed systems.

Definition 3 (stable) Safety property $stable.p$ holds for a program if once p becomes true, it remains true from that point on.

$$stable.p \triangleq p\ next\ p$$

Definition 4 (invariant) Safety property $invariant.p$ holds for a program if p holds initially, and if p remains true for the entire computation.

$$invariant.p \triangleq initially.p \wedge stable.p$$

Definition 5 (leads-to) Progress property $p \leadsto q$ (pronounced "leads-to") holds for a program if once p becomes true, q holds eventually in the computation.

$$p \leadsto q \triangleq invariant.(p \Rightarrow transient.(\neg q))$$

Definition 6 (to-always) Progress property $p \hookrightarrow q$ (pronounced "to-always") holds for a program if $p \leadsto q$, and if q is stable.

$$p \hookrightarrow q \triangleq p \leadsto q \wedge stable.q$$

3.4 Channel Properties

In addition, our definition of the model of computation gives the following properties on the ghost variables characterizing channels.

$$\#\texttt{sent(c)} \geq \#\texttt{del(c)} \quad (1)$$
$$\#\texttt{del(c)} \geq \#\texttt{rcv(c)} \quad (2)$$
$$\#\texttt{sent(c)} \geq k \leadsto \#\texttt{del(c)} \geq k \quad (3)$$

4 First Example: Mutual Exclusion in a Ring

This section describes a solution to the mutual exclusion problem using a fixed number of tokens and a ring of servers responsible for distributing tokens to clients waiting to enter their critical section. Pictorially, the topology of the problem is represented in Figure 4.

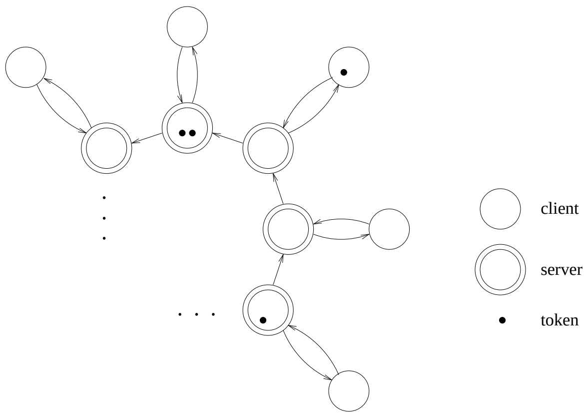

Figure 1: Clients Communicate with a Ring of Servers

The system contains N clients, N servers, and K tokens. Initially, one of the servers holds all the tokens. The tokens circulate continuously around the ring. A client is allowed to enter its critical section only if it holds a token. Because tokens can neither be created nor destroyed, this ensures that at any one time, there are at most K clients in their critical sections.

4.1 Notation

We use $\mathcal{S}$ to denote the set of servers, and s to denote a particular server process. Similarly, $\mathcal{C}$ denotes the set of clients, and c denotes a particular client process. Thus, $s \in \mathcal{S}$, $c \in \mathcal{C}$, and $|\mathcal{S}| = |\mathcal{C}| = N$. We use $p.\hat{\mathcal{E}}$ to denote the sets of processes which form p's incoming environment. That is, $p.\hat{\mathcal{E}}$ is the set of processes which can send messages to p. We use $p.\check{\mathcal{E}}$ to denote the sets of processes which form p's outgoing environment. That is, $p.\check{\mathcal{E}}$ is the set of processes to which

p can send messages. When no distinction between these two sets is needed (since they are the same), we use $p.\mathcal{E}$.

Channels are denoted by the triple $(\mathsf{msg}, p1, p2)$, where msg is the type of the messages on the channel, $p1$ is the sender, and $p2$ is the receiver.

In this algorithm there are 2 kinds of messages: tokens (denoted `tok`), and requests for tokens (denoted `req`). In addition, `tok_init` denotes the initial number of tokens in the system. The number of tokens held by a process p is given by $p.\mathsf{tok}$, and the number of tokens in a channel between processes p and q is $(p, q).\mathsf{tok}$.

4.2 Specifications

4.2.1 Safety

Conservation of Tokens. Tokens can neither be created nor destroyed. Thus, at any moment in the computation, the total number of tokens in the system is the same as it was at the beginning of the computation.

$$\sum_{s \in \mathcal{S}} (s.\mathsf{tok} + (s, s.out).\mathsf{tok}) + \sum_{c \in \mathcal{C}} (c.\mathsf{tok} + (c.s, c).\mathsf{tok} + (c, c.s).\mathsf{tok}) = \mathtt{tok\_init} \tag{4}$$

Necessity of Tokens. Clients enter their critical section only if they hold a token.

$$\langle \forall\ c : c \in \mathcal{C} :: c.\mathtt{crit} \Rightarrow (c.\mathsf{tok} > 0) \rangle \tag{5}$$

c.crit indicates whether or not the client is executing its critical section.

4.2.2 Progress

Request Satisfaction. If a client makes a request, then eventually that client receives a token.

$$\langle \forall\ c, k : c \in \mathcal{C} \wedge k \in \mathbb{N} :: \mathtt{\#sent}(\mathsf{req}, c, c.s) \geq k \leadsto \mathtt{\#rcv}(\mathsf{tok}, c.s, c) \geq k \rangle \tag{6}$$

4.3 Algorithm

We present the algorithm as an action system with pre and post conditions. For a pseudocode implementation, please refer to the appendix. We associate each action (which is considered to be atomic, since no actions causing suspension are permitted) with its postcondition. Variables not explicitly mentioned in the postcondition are not modified by an action. We also make use of the shorthand <u>inc</u> (and <u>dec</u>) to abbreviate an integer increment (and decrement) to a ghost variable.

Server (int $s.\mathtt{tok\_init}$, process_id $s.c$, process_id $s.out$, process_id $s.in$)
  initially:
    int $s.\mathsf{tok} = 0$
    $\mathtt{\#sent}(\mathsf{tok}, s, s.out) = s.\mathtt{tok\_init}$
    boolean $s.\mathtt{hungry} = \mathit{false}$
    boolean $s.\mathtt{clienttok} = \mathit{false}$
  actions:

$\neg s.\mathtt{hungry} \wedge \neg s.\mathtt{clienttok} \wedge (s.c, s).\overline{\mathsf{req}} \longrightarrow$
$\quad \underline{\mathrm{inc}}\ \#\mathtt{rcv}(\mathsf{req}, s.c, s) \wedge s.\mathtt{hungry}$
$\neg s.\mathtt{hungry} \wedge (s.in, s).\overline{\mathsf{tok}} \longrightarrow$
$\quad \underline{\mathrm{inc}}\ \#\mathtt{rcv}(\mathsf{tok}, s.in, s) \wedge \underline{\mathrm{inc}}\ \#\mathtt{sent}(\mathsf{tok}, s, s.out)$
$s.\mathtt{hungry} \wedge (s.in, s).\overline{\mathsf{tok}} \longrightarrow$
$\quad \underline{\mathrm{inc}}\ \#\mathtt{rcv}(\mathsf{tok}, s.in, s) \wedge \underline{\mathrm{inc}}\ \#\mathtt{sent}(\mathsf{tok}, s, s.c) \wedge \neg s.\mathtt{hungry} \wedge s.\mathtt{clienttok}$
$s.\mathtt{clienttok} \wedge (s.c, s).\overline{\mathsf{tok}} \longrightarrow$
$\quad \underline{\mathrm{inc}}\ \#\mathtt{rcv}(\mathsf{tok}, s.c, s) \wedge \underline{\mathrm{inc}}\ \#\mathtt{sent}(\mathsf{tok}, s, s.out) \wedge \neg s.\mathtt{clienttok}$

Client (process_id $c.s$)
initially:
$c.\mathsf{tok} = 0$
$c.\mathtt{sent\_req} = \mathit{false}$
$c.\mathtt{crit} = \mathit{false}$
$c.\mathtt{terminated} = \mathit{false}$
actions:
$\neg c.\mathtt{sent\_req} \wedge \neg c.\mathtt{crit} \longrightarrow$
$\quad c.\mathtt{terminated}$
$\neg c.\mathtt{sent\_req} \wedge \neg c.\mathtt{crit} \wedge \neg c.\mathtt{terminated} \longrightarrow$
$\quad \underline{\mathrm{inc}}\ \#\mathtt{sent}(\mathsf{req}, c, c.s) \wedge c.\mathtt{sent\_req}$
$c.\mathtt{sent\_req} \wedge (c.s, c).\overline{\mathsf{tok}} \longrightarrow$
$\quad \underline{\mathrm{inc}}\ \#\mathtt{rcv}(\mathsf{tok}, c.s, c) \wedge \underline{\mathrm{inc}}\ c.\mathsf{tok} \wedge \neg c.\mathtt{sent\_req} \wedge c.\mathtt{crit}$
$c.\mathtt{crit} \longrightarrow$
$\quad \underline{\mathrm{inc}}\ \#\mathtt{sent}(\mathsf{tok}, c, c.s) \wedge \underline{\mathrm{dec}}\ c.\mathsf{tok} \wedge \neg c.\mathtt{crit}$

Notice that the pseudocode given in Appendix A can be easily shown to be an implementation of this algorithm by annotating the program with the above assertions.

4.4 Process Properties and Invariants

4.4.1 Invariants

The following invariants can be easily established.

Server.

$$s.\mathsf{tok} = s.\mathtt{tok\_init} - \sum_{e \in s.\check{\mathcal{E}}} \#\mathtt{sent}(\mathsf{tok}, s, e) + \sum_{e \in s.\hat{\mathcal{E}}} \#\mathtt{rcv}(\mathsf{tok}, e, s) \tag{7}$$

$$\#\mathtt{rcv}(\mathsf{tok}, s.in, s) = \#\mathtt{sent}(\mathsf{tok}, s, s.out) - s.\mathtt{tok\_init} + \underline{ord}(s.\mathtt{clienttok}) \tag{8}$$

$$\#\mathtt{sent}(\mathsf{tok}, s, s.c) = \#\mathtt{rcv}(\mathsf{tok}, s.c, s) + \underline{ord}(s.\mathtt{clienttok}) \tag{9}$$

$$\#\mathtt{rcv}(\mathsf{req}, s.c, s) = \#\mathtt{sent}(\mathsf{tok}, s, s.c) + \underline{ord}(s.\mathtt{hungry}) \tag{10}$$

Client.

$$c.\mathsf{tok} = \#\mathtt{rcv}(\mathsf{tok}, c.s, c) - \#\mathtt{sent}(\mathsf{tok}, c, c.s) \tag{11}$$

$$c.\mathsf{tok} = \underline{ord}(c.\mathtt{crit}) \tag{12}$$

$$\#\mathtt{sent}(\mathsf{req}, c, c.s) = \#\mathtt{rcv}(\mathsf{tok}, c.s, c) + \underline{ord}(c.\mathtt{sent\_req}) \tag{13}$$

4.4.2 Specifications

Each process is specified by a collection of modified rely-guarantee pairs. These pairs can be seen as a process's obligation when placed in an appropriate environment.

Server. There are two processes in a server's (s) incoming environment: a client ($s.c \in \mathcal{C}$) and another server ($s.in \in \mathcal{S}$). Thus, for a server, $s.\hat{\mathcal{E}} = \{s.c, s.in\}$. There are two processes in a server's (s) outgoing environment: the same client ($s.c \in \mathcal{C}$) and a server ($s.out \in \mathcal{S}$). Thus, for a server, $s.\check{\mathcal{E}} = \{s.c, s.out\}$.

A server does not create nor destroy tokens. Also, a server does not send more tokens to its client than requested.

$$\begin{array}{ll} \llparenthesis & invariant.(true\ ) \\ \rhd & invariant.(\texttt{\#sent}(\mathsf{tok}, s, s.c) \leq \texttt{\#del}(\mathsf{req}, s.c, s)) \wedge \\ & invariant.(s.\mathsf{tok} = s\texttt{.tok\_init} - \sum_{e \in s.\check{\mathcal{E}}} \texttt{\#sent}(\mathsf{tok}, s, e) + \sum_{e \in s.\hat{\mathcal{E}}} \texttt{\#rcv}(\mathsf{tok}, e, s))\ \rrparenthesis \end{array} \tag{14}$$

If its client eventually returns all tokens it is sent, a server eventually passes all tokens it receives on to the next server.

$$\begin{array}{ll} \llparenthesis & \langle \forall\ k : k \in \mathbb{N} :: \texttt{\#sent}(\mathsf{tok}, s, s.c) \geq k \leadsto \texttt{\#del}(\mathsf{tok}, s.c, s) \geq k \rangle \\ \rhd & \langle \forall\ k : k \in \mathbb{N} :: \texttt{\#del}(\mathsf{tok}, s.in, s) \geq k \leadsto \texttt{\#sent}(\mathsf{tok}, s, s.out) \geq k + s\texttt{.tok\_init} \rangle\ \rrparenthesis \end{array} \tag{15}$$

If its client eventually returns all tokens it is sent and the tokens that are passed on to the next server eventually return, a server eventually satisfies all requests for tokens.

$$\begin{array}{ll} \llparenthesis & \langle \forall\ k : k \in \mathbb{N} :: \texttt{\#sent}(\mathsf{tok}, s, s.c) \geq k \leadsto \texttt{\#del}(\mathsf{tok}, s.c, s) \geq k \rangle \wedge \\ & \langle \forall\ k : k \in \mathbb{N} :: \texttt{\#sent}(\mathsf{tok}, s, s.out) \geq k \leadsto \texttt{\#del}(\mathsf{tok}, s.in, s) \geq k + \texttt{tok\_init} - s\texttt{.tok\_init} \rangle \\ \rhd & \langle \forall\ k : k \in \mathbb{N} :: \texttt{\#del}(\mathsf{req}, s.c, s) \geq k \leadsto \texttt{\#sent}(\mathsf{tok}, s, s.c) \geq k \rangle\ \rrparenthesis \end{array} \tag{16}$$

Client.
A client does not create nor destroy tokens, nor does it enter its critical section without holding a token. A client eventually returns all tokens it is sent.

$$\begin{array}{ll} \llparenthesis & invariant.(true\ ) \\ \rhd & invariant.(c.\mathsf{tok} = \sum_{e \in c.\mathcal{E}} \texttt{\#rcv}(\mathsf{tok}, e, c) - \sum_{e \in c.\mathcal{E}} \texttt{\#sent}(\mathsf{tok}, c, e)) \wedge \\ & invariant.(c.\mathsf{tok} = \underline{ord}(c\texttt{.crit})) \wedge \\ & \langle \forall\ e, k : e \in c.\mathcal{E} \wedge k \in \mathbb{N} :: \texttt{\#rcv}(\mathsf{tok}, e, c) \geq k \leadsto \texttt{\#sent}(\mathsf{tok}, c, e) \geq k \rangle\ \rrparenthesis \end{array} \tag{17}$$

If its server does not send more tokens than requested, all tokens that are delivered are eventually received.

$$\begin{array}{ll} \llparenthesis & invariant.(\langle \forall\ e : e \in c.\mathcal{E} :: \texttt{\#del}(\mathsf{tok}, e, c) \leq \texttt{\#sent}(\mathsf{req}, c, e) \rangle) \\ \rhd & \langle \forall\ e, k : e \in c.\mathcal{E} \wedge k \in \mathbb{N} :: \texttt{\#del}(\mathsf{tok}, e, c) \geq k \leadsto \texttt{\#rcv}(\mathsf{tok}, e, c) \geq k \rangle\ \rrparenthesis \end{array} \tag{18}$$

4.4.3 Proof of Client.

In this section we prove that the algorithm given for the client in Section 4.3 meets the specification given by the modified rely-guarantee pairs (17) and (18) in Section 4.4.2.

Informal Proof Description. Specification pair (17) relies only on the environment satisfying *true* and so represents the invariants of the client process, independent of the environment. It follows directly from the initial conditions (the channels are all empty and the client holds no tokens) and the annotation of the client algorithm (4.3).

Specification pair (18) for the client says that if the number of tokens delivered to the client from the environment is bounded by the number of requests sent by the client, then the client will eventually return all the tokens delivered to it. This follows directly from the algorithm of the client, which waits for a token to arrive after every request it sends. The tokens received are returned immediately (that is, the critical section of a client is finite).

Formal Proof. Again, the guarantee properties of (17) are invariant properties of the client process.

$$\begin{array}{ll} & \mathit{invariant}.(c.\mathsf{tok} = \sum_{e \in c.\mathcal{E}} \texttt{\#rcv}(\mathsf{tok}, e, c) - \sum_{e \in c.\mathcal{E}} \texttt{\#sent}(\mathsf{tok}, c, e)) \\ = & \quad \{\ 11\ \} \\ & \mathit{invariant}.(\mathit{true}\ ) \end{array}$$

Also:

$$\begin{array}{ll} & \mathit{invariant}.(c.\mathsf{tok} = \underline{\mathit{ord}}(c.\texttt{crit})) \\ = & \quad \{\ 12\ \} \\ & \mathit{invariant}.(\mathit{true}\ ) \end{array}$$

And finally:

$$\begin{array}{ll} & \langle \forall\ e, k : e \in c.\mathcal{E} \wedge k \in \mathbb{N} :: \texttt{\#rcv}(\mathsf{tok}, e, c) \geq k \leadsto \texttt{\#sent}(\mathsf{tok}, c, e) \geq k \rangle \\ = & \quad \{\ \text{client algorithm}\ \} \\ & \mathit{invariant}.(\mathit{true}\ ) \end{array}$$

We next prove the client satisfies rely-guarantee pair (18).

$$\begin{array}{ll} & \langle \forall\ e, k : e \in c.\mathcal{E} \wedge k \in \mathbb{N} :: \texttt{\#del}(\mathsf{tok}, e, c) \geq k \leadsto \texttt{\#rcv}(\mathsf{tok}, e, c) \geq k \rangle \\ \Leftarrow & \quad \{\ \text{calculus, 1, and 2}\ \} \\ & \langle \forall\ e : e \in c.\mathcal{E} :: (e, c).\overline{\mathsf{tok}} \leadsto \underline{\mathrm{inc}}\ \ \texttt{\#rcv}(\mathsf{tok}, e, c) \rangle \\ \Leftarrow & \quad \{\ \text{client algorithm}\ \} \\ & \langle \forall\ e : e \in c.\mathcal{E} :: (e, c).\overline{\mathsf{tok}} \leadsto (e, c).\overline{\mathsf{tok}} \wedge c.\texttt{sent\_req} \rangle \\ \Leftarrow & \quad \{\ \text{calculus}\ \} \\ & \mathit{invariant}.(\langle \forall\ e : e \in c.\mathcal{E} :: (e, c).\overline{\mathsf{tok}} \Rightarrow c.\texttt{sent\_req} \rangle) \\ = & \quad \{\ 13\ \} \\ & \mathit{invariant}.(\langle \forall\ e : e \in c.\mathcal{E} :: (e, c).\overline{\mathsf{tok}} \Rightarrow \texttt{\#rcv}(\mathsf{tok}, e, c) < \texttt{\#sent}(\mathsf{req}, c, e) \rangle) \\ \Leftarrow & \quad \{\ \text{calculus}\ \} \\ & \mathit{invariant}.(\langle \forall\ e : e \in c.\mathcal{E} :: \texttt{\#del}(\mathsf{tok}, e, c) \leq \texttt{\#sent}(\mathsf{req}, c, e) \rangle) \end{array}$$

□

4.4.4 Proof of Server.

In this section we prove that the algorithm given for the server in Section 4.3 meets the specification given by the modified rely-guarantee pairs (14), (15), and (16) in Section 4.4.2.

Informal Proof Description. Specification pair (14) relies only on the environment satisfying *invariant.*(*true*) and so represents the invariants of the server process, independent of the environment. It follows directly from the initial conditions (the channels are all empty and the server holds s.`tok_init` tokens) and the annotation of the server algorithm (4.3).

Specification pair (15) for the server says that if the client eventually returns all the tokens sent to it, then all the tokens sent to the server are eventually passed on to the next server. This follows from the server algorithm since it sends tokens returned by the client directly on to the next server.

Specification pair (16) for the server says that if the environment eventually returns all the tokens that are sent to it, then the server will eventually satisify (with tokens) all the requests sent to it. That the server receives all requests sent to it follows from the fact that the server does not suspend forever, which in turn follows from the fact that received requests are eventually satisfied. A server eventually satisfies an outstanding request because tokens are continuously circulating, and a hungry server passes the next token on to its client.

Formal Proof. The formal proof for the server follows the pattern demonstrated in the previous section with the client process. The proof is included in the appendix.

4.5 Proof of Program Specification

4.5.1 Composition

We establish the safety and progress requirements given by 4, 5, and 6 by showing that the guarantee clauses are invariant properties of the computation, given the ring architecture described in the introduction. That is, we show that each rely clause is an invariant property of the computation. This is done either directly, or by establishing that the rely clause is implied by a guarantee clause whose rely has already been shown to be program property. Notice that these proof obligations are dictated by the architecture in question: a ring of servers, each with a single client (ie the guarantee clauses of all neighbours with channels to a process must imply the rely clause of the process). By the main theorem of modified rely-guarantees, this establishes the conjuntion of the guarantee clauses as a property of the program.

Server Rely (14) is a Program Property. Trivial since this rely is *invariant.*(*true*).
□

Client Rely (17) is a Program Property. Trivial since this rely is *invariant.*(*true*).
□

Client Rely (18) is a Program Property.

$$\begin{array}{ll} & \mathit{true} \\ = & \quad \{ \text{ Server's rely-guarantee (14) } \} \\ & \langle \forall\ c : c \in \mathcal{C} :: \texttt{\#sent}(\texttt{tok}, c.s, c) \le \texttt{\#del}(\texttt{req}, c, c.s) \rangle \\ \Rightarrow & \quad \{ \text{ 1 } \} \end{array}$$

$$
\begin{array}{ll}
 & \langle \forall\, c : c \in \mathcal{C} :: \texttt{\#del}(\textsf{tok}, c.s, c) \le \texttt{\#del}(\textsf{req}, c, c.s) \rangle \\
\Rightarrow & \qquad \{\ 1\ \} \\
 & \langle \forall\, c : c \in \mathcal{C} :: \texttt{\#del}(\textsf{tok}, c.s, c) \le \texttt{\#sent}(\textsf{req}, c, c.s) \rangle
\end{array}
$$

□

Server Rely (15) is a Program Property.

$$
\begin{array}{ll}
 & \mathit{true} \\
= & \qquad \{\ \text{Client's rely-guarantees (18) and (17)}\ \} \\
 & \langle \forall\, s, k : s \in \mathcal{S} \wedge k \in \mathbb{N} :: \texttt{\#del}(\textsf{tok}, s, s.c) \ge k \leadsto \texttt{\#rcv}(\textsf{tok}, s, s.c) \ge k \rangle \wedge \\
 & \langle \forall\, s, k : s \in \mathcal{S} \wedge k \in \mathbb{N} :: \texttt{\#rcv}(\textsf{tok}, s, s.c) \leadsto \texttt{\#sent}(\textsf{tok}, s.c, s) \rangle \\
\Rightarrow & \qquad \{\ \text{calculus}\ \} \\
 & \langle \forall\, s, k : s \in \mathcal{S} \wedge k \in \mathbb{N} :: \texttt{\#del}(\textsf{tok}, s, s.c) \ge k \leadsto \texttt{\#sent}(\textsf{tok}, s.c, s) \ge k \rangle \\
\Rightarrow & \qquad \{\ 1\ \} \\
 & \langle \forall\, s, k : s \in \mathcal{S} \wedge k \in \mathbb{N} :: \texttt{\#sent}(\textsf{tok}, s, s.c) \ge k \leadsto \texttt{\#sent}(\textsf{tok}, s.c, s) \ge k \rangle \\
\Rightarrow & \qquad \{\ 3\ \} \\
 & \langle \forall\, s, k : s \in \mathcal{S} \wedge k \in \mathbb{N} :: \texttt{\#sent}(\textsf{tok}, s, s.c) \ge k \leadsto \texttt{\#del}(\textsf{tok}, s.c, s) \ge k \rangle
\end{array}
$$

□

Server Rely (16) is a Program Property. We introduce the notation $s.out.out$ to mean the out process of s's out process. By extension, $s.out^k$ is the out process that is k hops away from s. Because the architecture is a ring of N servers, we have $s.out^{N-1} = s.in$ and $s.out^N = s$.

$$
\begin{array}{ll}
 & \mathit{true} \\
= & \qquad \{\ \text{Server } s.out\text{'s rely-guarantee (15)}\ \} \\
 & \langle \forall\, s, k : s \in \mathcal{S} \wedge k \in \mathbb{N} :: \texttt{\#del}(\textsf{tok}, s, s.out) \ge k \leadsto \\
 & \qquad \texttt{\#sent}(\textsf{tok}, s.out, s.out^2) \ge k + s.out\,.\texttt{tok\_init} \rangle \\
\Rightarrow & \qquad \{\ 3\ \} \\
 & \langle \forall\, s, k : s \in \mathcal{S} \wedge k \in \mathbb{N} :: \texttt{\#del}(\textsf{tok}, s, s.out) \ge k \leadsto \\
 & \qquad \texttt{\#del}(\textsf{tok}, s.out, s.out^2) \ge k + s.out\,.\texttt{tok\_init} \rangle \\
\Rightarrow & \qquad \{\ \text{Server } s.out^2\text{'s rely-guarantee (15)}\ \} \\
 & \langle \forall\, s, k : s \in \mathcal{S} \wedge k \in \mathbb{N} :: \texttt{\#del}(\textsf{tok}, s, s.out) \ge k \leadsto \\
 & \qquad \texttt{\#sent}(\textsf{tok}, s.out^2, s.out^3) \ge k + s.out\,.\texttt{tok\_init} + s.out^2.\texttt{tok\_init} \rangle \\
\Rightarrow & \qquad \{\ 3\ \} \\
 & \ldots \\
\Rightarrow & \\
 & \langle \forall\, s, k : s \in \mathcal{S} \wedge k \in \mathbb{N} :: \texttt{\#del}(\textsf{tok}, s, s.out) \ge k \leadsto \\
 & \qquad \texttt{\#del}(\textsf{tok}, s.out^{N-1}, s.out^N) \ge k + s.out\,.\texttt{tok\_init} + \ldots + s.out^{N-1}.\texttt{tok\_init} \rangle \\
= & \qquad \{\ \text{ring architecture}\ \} \\
 & \langle \forall\, s, k : s \in \mathcal{S} \wedge k \in \mathbb{N} :: \texttt{\#del}(\textsf{tok}, s, s.out) \ge k \leadsto \\
 & \qquad \texttt{\#del}(\textsf{tok}, s.in, s) \ge k + \texttt{tok\_init} - s\,.\texttt{tok\_init} \rangle \\
\Rightarrow & \qquad \{\ 1\ \} \\
 & \langle \forall\, s, k : s \in \mathcal{S} \wedge k \in \mathbb{N} :: \texttt{\#sent}(\textsf{tok}, s, s.out) \ge k \leadsto \\
 & \qquad \texttt{\#del}(\textsf{tok}, s.out, s) \ge k + \texttt{tok\_init} - s\,.\texttt{tok\_init} \rangle
\end{array}
$$

□

Main Program Property. By the composition theorem for modified rely-guarantee, we therefore have the following as a property of the program:

$$
\begin{array}{ll}
& \forall\langle\, s \mid s \in \mathcal{S} \,\triangleright\, \mathit{invariant}.(\texttt{\#sent}(\mathsf{tok},s,s.c) \le \texttt{\#del}(\mathsf{req},s.c,s)) \\
& \quad \wedge\ \mathit{invariant}.(s.\mathsf{tok} = s.\texttt{tok\_init} - \texttt{\#sent}(\mathsf{tok},s,s.\mathit{out}) - \texttt{\#sent}(\mathsf{tok},s,s.c) \\
& \qquad\qquad + \texttt{\#rcv}(\mathsf{tok},s.\mathit{in},s) + \texttt{\#rcv}(\mathsf{tok},s.c,s)) \\
& \quad \wedge\ \langle \forall\, k : k \in \mathbb{N} :: \texttt{\#del}(\mathsf{tok},s.\mathit{in},s) \ge k \leadsto \texttt{\#sent}(\mathsf{tok},s,s.\mathit{out}) \ge k + s.\texttt{tok\_init}\rangle \\
& \quad \wedge\ \langle \forall\, k : k \in \mathbb{N} :: \texttt{\#del}(\mathsf{req},s.c,s) \ge k \leadsto \texttt{\#sent}(\mathsf{tok},s,s.c) \ge k\rangle\ \rangle \\
\wedge & \forall\langle\, c \mid c \in \mathcal{C} \,\triangleright\, \mathit{invariant}.(c.\mathsf{tok} = \texttt{\#rcv}(\mathsf{tok},c.s,c) - \texttt{\#sent}(\mathsf{tok},c,c.s)) \\
& \quad \wedge\ \mathit{invariant}.(c.\mathsf{tok} = \underline{\mathit{ord}}(c.\texttt{crit})) \\
& \quad \wedge\ \langle \forall\, k : k \in \mathbb{N} :: \texttt{\#rcv}(\mathsf{tok},c.s,c) \ge k \leadsto \texttt{\#sent}(\mathsf{tok},c,c.s) \ge k\rangle \\
& \quad \wedge\ \langle \forall\, k : k \in \mathbb{N} :: \texttt{\#del}(\mathsf{tok},c.s,c) \ge k \leadsto \texttt{\#rcv}(\mathsf{tok},c.s,c) \ge k\rangle\ \rangle
\end{array} \tag{19}
$$

4.5.2 Proofs of Safety and Progress

Proof of 4.

$$
\begin{array}{ll}
& \sum_{s\in\mathcal{S}}(s.\mathsf{tok} + (s,s.\mathit{out}).\mathsf{tok}) + \sum_{c\in\mathcal{C}}(c.\mathsf{tok} + (c.s,c).\mathsf{tok} + (c,c.s).\mathsf{tok}) \\
= & \qquad \{\ 19\ \} \\
& \sum_{s\in\mathcal{S}}(s.\texttt{tok\_init} - \texttt{\#sent}(\mathsf{tok},s,s.\mathit{out}) - \texttt{\#sent}(\mathsf{tok},s,s.c) + \texttt{\#rcv}(\mathsf{tok},s.\mathit{in},s) + \\
& \qquad \texttt{\#rcv}(\mathsf{tok},s.c,s) + \texttt{\#sent}(\mathsf{tok},s,s.\mathit{out}) - \texttt{\#rcv}(\mathsf{tok},s,s.\mathit{out})) + \\
& \sum_{c\in\mathcal{C}}(\texttt{\#rcv}(\mathsf{tok},s.c,c) - \texttt{\#sent}(\mathsf{tok},c,s.c) + \texttt{\#sent}(\mathsf{tok},c.s,c) - \\
& \qquad \texttt{\#rcv}(\mathsf{tok},c.s,c) + \texttt{\#sent}(\mathsf{tok},c,c.s) - \texttt{\#rcv}(\mathsf{tok},c,c.s)) \\
= & \qquad \{\ \text{ring architecture}\ \} \\
& \sum_{s\in\mathcal{S}}(s.\texttt{tok\_init} - \texttt{\#sent}(\mathsf{tok},s,s.c) + \texttt{\#rcv}(\mathsf{tok},s.c,s)) + \\
& \sum_{c\in\mathcal{C}}(\texttt{\#sent}(\mathsf{tok},c.s,c) - \texttt{\#rcv}(\mathsf{tok},c,c.s)) \\
= & \qquad \{\ \text{calculus}\ \} \\
& \sum_{s\in\mathcal{S}} s.\texttt{tok\_init} \\
= & \qquad \{\ \text{calculus}\ \} \\
& \texttt{tok\_init}
\end{array}
$$

□

Proof of 5.

$$
\begin{array}{ll}
& \langle \forall\, c : c \in \mathcal{C} :: c.\texttt{crit} \Rightarrow (c.\mathsf{tok} > 0)\rangle \\
\Leftarrow & \qquad \{\ \text{calculus}\ \} \\
& \langle \forall\, c : c \in \mathcal{C} :: c.\mathsf{tok} = \underline{\mathit{ord}}(c.\texttt{crit})\rangle \\
= & \qquad \{\ 19\ \} \\
& \mathit{true}
\end{array}
$$

□

Proof of 6.

$$\langle \forall\, c,k : c \in \mathcal{C} \wedge k \in \mathbb{N} :: \texttt{\#sent}(\mathsf{req},c,c.s) \ge k \leadsto \texttt{\#rcv}(\mathsf{tok},c.s,c) \ge k\rangle$$

$$
\begin{array}{ll}
= & \quad \{ \ 1 \text{ and } 3 \ \} \\
& \langle \forall\ c, k : c \in \mathcal{C} \wedge k \in \mathbb{N} :: \texttt{\#del}(\textsf{req}, c, c.s) \geq k \leadsto \texttt{\#rcv}(\textsf{tok}, c.s, c) \geq k \rangle \\
\Leftarrow & \quad \{ \ 19 \ \} \\
& \langle \forall\ c, k : c \in \mathcal{C} \wedge k \in \mathbb{N} :: \texttt{\#sent}(\textsf{tok}, c.s, c) \geq k \leadsto \texttt{\#rcv}(\textsf{tok}, c.s, c) \geq k \rangle \\
= & \quad \{ \ 1 \text{ and } 3 \ \} \\
& \langle \forall\ c, k : c \in \mathcal{C} \wedge k \in \mathbb{N} :: \texttt{\#del}(\textsf{tok}, c.s, c) \geq k \leadsto \texttt{\#rcv}(\textsf{tok}, c.s, c) \geq k \rangle \\
= & \quad \{ \ 19 \ \} \\
& \mathit{true}
\end{array}
$$

□

5 Second Example: Synchronization With Gossip

This section describes the gossip algorithm, gives a pseudocode implementation of a gossip process, and specifies the behaviour of this process using modified rely-guarantee. For the sake of brevity, the complete proof of correctness is not given.

The gossip algorithm can be used to synchronize a collection of processes. Two processes are considered to be neighbours if they are connected by a channel. There is a special process that initiates the gossip algorithm. The system is described by an undirected graph $G = (V, E)$, where V is the set of processes, and E is the set of channels. I, the initiator process is considered to be part of the graph. Since the graph is undirected and connected, there is a path between any pair of processes. Let $a \ \underline{nbr} \ b$ be the relation "a and b are neighbours". Let $R.(p, q)$ be defined as q, and $L.(p, q)$ be p.

5.1 Specifications

5.1.1 Safety

Synchronization. We must ensure that when the initiator receives a message from all its neighbors, all processes in V have completed the gossip algorithm.

$$\langle \forall\ u : u \ \underline{nbr} \ I :: \texttt{\#sent}(\textsf{msg}, u, I) > 0 \rangle \Rightarrow \langle \forall\ u :: u \text{ has completed the gossip algorithm} \rangle \tag{20}$$

5.1.2 Progress

Completion. The only progress requirement is that after the initiator sends a message to all its neighbors, it eventually receives a message from all its neighbors.

$$\langle \forall\ u : u \ \underline{nbr} \ I :: \texttt{\#sent}(\textsf{msg}, I, u) > 0 \rangle \hookrightarrow \langle \forall\ u : u \ \underline{nbr} \ I :: \texttt{\#rcv}(\textsf{msg}, u, I) > 0 \rangle \tag{21}$$

5.2 Algorithm

5.2.1 Pseudocode

```
Message Type:
  PROCESS_ID;
```

```
Initiator()  {
  myid = my process id;
  neighborhood = set of process_ids of neighbors;
  // val = 1
  foreach p in neighborhood
    send myid to p;
  foreach p in neighborhood
    receive p;
  // val = 2
}

Gossip()  {
  myid = my process id;
  neighborhood = set of process_ids of neighbors;
  // val = 0
  receive parent;
  // val = 1
  foreach p in neighborhood - {parent}
    send myid to p;
  foreach p in neighborhood - {parent}
    receive p;
  send myid to parent;
  // val = 2
}
```

5.3 Invariants, Process Properties, and Transient Properties

We define the following 2 graphs:

$$T = (\{p : p.val > 0 :: p\}, \{p : (p.val > 0) \wedge (p \neq I) :: (p, p.parent)\}) \tag{22}$$

$$T' = (\{p : p.val = 1 :: p\}, \{p : (p.val = 1) \wedge (p \neq I) :: (p, p.parent)\}) \tag{23}$$

The tree T' is clearly a sub-tree of T, since $R.T' \subseteq R.T$.

5.3.1 Global Invariants

Messages. There is at most one message sent on a channel.

$$\langle \forall\, u, v : u\ \underline{nbr}\ v :: \#\mathtt{sent}(\mathsf{msg}, u, v) \leq 1 \rangle \tag{24}$$

Tree of processes. The subgraph participating in the gossip algorithm is a tree, and a process sends a message only after it is part of the tree.

$$T \text{ is a tree} \wedge \langle \forall\, u, v :: \#\mathtt{sent}(\mathsf{msg}, u, v) > 0 \Rightarrow u \in L.T \rangle \tag{25}$$

Tree of ones. The subgraph of processes that have value "1" form a tree. Note that the subgraph of "1" processes first grows, and then shrinks. Also, if a process has value "2", all its children in the tree have value "2".

$$T' \text{ is a tree} \wedge \langle \forall\, p :: p.val = 2 \Rightarrow \langle \forall\, q : (q, p) \in R.T :: q.val = 2 \rangle \rangle \tag{26}$$

Possible Configurations. The following invariant states the possible configurations of the graph. Every process has value "2", or there is a process that has value "1" whose children in the tree T all have value "2".

$$\langle \forall\ p :: p.val = 2\rangle \vee \langle \exists\ u :: (u.val = 1) \wedge \langle \forall\ v : (v, u) \in R.T :: v.val = 2\rangle\rangle \tag{27}$$

5.3.2 Process Properties

The following local properties assert that a process will complete certain actions from a given state no matter what the process is composed with.

Broadcast. After the first message has been received, a process will send messages to all its neighbors except the process from which it received the first message.

$$(u.val = 1) \hookrightarrow \langle \forall\ u : u\ \underline{nbr}\ v \wedge (v \neq u.parent) :: \texttt{\#sent}(\textsf{msg}, u, v) > 0\rangle \tag{28}$$

Messages are received. Every delivered message received eventually.

$$(\texttt{\#del}(\textsf{msg}, w, u) > 0) \hookrightarrow (\texttt{\#rcv}(\textsf{msg}, w, u) > 0) \tag{29}$$

Termination. If all neighboring processes deliver a message, then eventually the algorithm will terminate.

$$\langle \forall\ u : u\ \underline{nbr}\ v :: \texttt{\#del}(\textsf{msg}, v, u) > 0\rangle \hookrightarrow (u.val = 2) \tag{30}$$

5.3.3 Transient Properties

Growing tree. A process that has value "0" that has a neighbor with value "1" will eventually change its value.

$$\langle \forall\ u, v : u\ \underline{nbr}\ v :: transient.(u.val = 0 \wedge v.val = 1)\rangle \tag{31}$$

Shrinking tree. A process that has value "1" and whose children in the tree have value "2" will eventually have value "2".

$$\langle \forall\ u :: transient.(u.val = 1 \wedge \langle \forall\ v : u\ \underline{nbr}\ v :: v.val \geq 1\rangle \wedge \langle \forall\ v : (v, u) \in R.T :: v.val = 2\rangle)\rangle \tag{32}$$

5.4 Proof

The correctness of the process properties given above can be confirmed from the text of the program. These properties are entirely local to the processes and so the validation can be carried out in isolation. The correctness of the computation as a whole, however, requires the validation of the global invariants and the global transient properties. These properties can not be established by an implication ladder of modified rely-guarantee properties because they cannot be expressed as the conjunction of local properties.

The complete proof for this algorithm can be found in [MSC94].

6 Discussion

Evaluation The first example illustrates the principal strength of this approach. Each process type is specified, implemented, and verified in isolation. The program is then demonstrated to meet the mutual exclusion specification by considering only the composition of modified rely-guarantee clauses and by creating an implication ladder. Nothing more than the specifications of the client and server processes, the architecture of their interconnection, and the channel properties given in 3.4 are needed in this phase of program verification.

The second example illustrates the principal shortcoming of this approach. The specification of this problem involves defining a tree of processes. This is a global property which cannot be directly expressed as a conjunction of local properties. For this reason, the proof cannot be directly built as an implication ladder of modified rely-guarantee properties. The principal problem with the second example is that the exact topology of the tree of processes in the computation is non-deterministic. If the algorithm were modified to use a fixed tree, we could prove its correctness without a global proof.

Nevertheless, this specification remains a useful construct, in particular as a precise definition of the obligation of the programmer implementing the gossip processes. The hard part of the larger proof, involving global properties in addition to modified rely-guarantee clauses, requires some sophistication, but the specification of the individual process can still be expressed as modified rely-guarantee clauses, providing a clear contract between the validation argument and the implementation.

Reusability Because each process is verified in isolation, the code and proof for a process can be easily reused in another computation, if the specification is appropriate. For example, for the mutual exclusion example, a new architecture consisting of a single central server with a collection of clients can be implemented by designing a new server processes, but keeping the client process presented in this paper.

Since the proof that a process meets its specification is often the hardest part of these proofs (compared to the proof that the rely clauses are all program properties, for example), this kind of proof reuse is important. Reusing the proof in this fashion results in a system that has been formally proven to be correct while keeping the additional proof obligation manageable.

Abstraction Modified rely-guarantee pairs also provide a clear and precise definition of the behavior of the environment of a process. They can be used as interface specifications for channels that lead to the environment. For instance, suppose we have a distributed computation that implements a distributed database. A user sends queries to the database by sending messages on channels to a local process that is connected to the distributed database. A user of the database only requires that the local process eventually responds to each query. The details of the computation that implements the database should be completely transparent to the user. Such a specification can be easily given using modified rely-guarantee pairs.

7 Conclusions

The emphasis of this approach is on the decomposition of a distributed program into individual processes and on the independent specification and verification of each of these processes. These specifications are then composed according to the architecture (ie according to how these processes are plugged together) yielding a global program property. The properties of the computation as a whole are implied by this global program property.

The principal strength of the modified rely-guarantee specification notation we propose is the ability to specify and verify processes in isolation. By combining the ability to rely on progress properties of the environment while restricting attention to local properties, we obtain a powerful specification construct and yet a simple corresponding proof methodology.

A Algorithm for Mutual Exclusion

In this section we give the algorithms for both the clients and the server. The program for initializing and connecting these components is not given here for the sake of brevity.

```
Message Type:
  enum {TOK, REQ} MSG;

Server (int s_tok_init, process_id client_id,
        process_id s_out, process_id s_in)  {
  for (int i=0; i<s_tok_init; i++)
    send TOK to s_out;
  int s_tok = 0;
  MSG m;
  repeat
    m = receive MSG where (MSG==REQ && source==client_id)
                       || (MSG==TOK && source==s_in);
    if (m==REQ)  {
      receive MSG where (MSG==TOK && source==s_in);
      send TOK to client_id;
      m = receive MSG where MSG==TOK;
      while (m.source!=client_id)  {  //m.source==s\_in
        send TOK to s_out;
        m = receive MSG where MSG==TOK;
      }
      send TOK to s_out;
    }
    else  {   //m==TOK
      send TOK s_out;
    }
  end repeat
}

Client (process_id server_id)  {
  int c_tok = 0;
  repeat
    non_critical();
    send REQ to server_id;
    receive MSG where MSG==TOK;
    c_tok++;
    begin_critical();
    end_critical();
    send TOK to server_id;
    c_tok--;
  end repeat
}
```

Next, we give the algorithm as expressed by a series of guarded commands.

Message Types:
  int TOK
  process_id REQ

```
Server (int s.tok_init, process_id s.c, process_id s.out, process_id s.in)
    initially:
        int s.tok = 0
        #sent(tok,s, s.out) = s.tok_init
        boolean s.hungry = false
        boolean s.clienttok = false
    actions:
        ¬s.hungry ∧ ¬s.clienttok ∧ (s.c, s).r̅e̅q̅ →
            rcv REQ from s.c
            s.hungry= true
        ¬s.hungry ∧ (s.in, s).t̅o̅k̅ →
            rcv TOK from s.in
            s.tok++
            send TOK to s.out
            s.tok- -
        s.hungry ∧ (s.in, s).t̅o̅k̅ →
            rcv TOK from s.in
            s.tok++
            send TOK to s.c
            s.tok- -
            s.hungry= false
            s.clienttok= true
        s.clienttok ∧ (s.c, s).t̅o̅k̅ →
            rcv TOK from s.c
            s.tok++
            send TOK to s.out
            s.tok- -
            s.clienttok= false

Client (process_id c.s)
    initially:
        int c.tok = 0
        boolean c.sent_req = false
        boolean c.crit = false
        boolean c.terminated = false
    actions:
        ¬c.sent_req ∧ ¬c.crit →
            c.terminated= true
        ¬c.sent_req ∧ ¬c.crit ∧ ¬c.terminated →
            send REQ(my_id) to c.s
            c.sent_req= true
        c.sent_req ∧ (c.s, c).t̅o̅k̅ →
            rcv TOK
            c.tok++
            c.sent_req= false
            c.crit= true
            begin_critical()
        c.crit →
            end_critical()
            c.crit= false
            send TOK(1) to c.s
            c.tok- -
```

B Proof of Mutual Exclusion Server

The guarantee properties of pair (14) are invariant properties of the server process and do not depend on the environment at all.

$$
\begin{array}{ll}
 & invariant.(\texttt{\#sent}(\mathsf{tok}, s, s.c) \leq \texttt{\#del}(\mathsf{req}, s.c, s)) \\
\Leftarrow & \qquad \{\ 10\ \} \\
 & invariant.(\texttt{\#rcv}(\mathsf{req}, s.c, s) \leq \texttt{\#del}(\mathsf{req}, s.c, s)) \\
= & \qquad \{\ 2\ \} \\
 & invariant.(true\ )
\end{array}
$$

Also:

$$
\begin{array}{ll}
 & invariant.(s.\mathsf{tok} = s\,\texttt{.tok\_init} + \sum_{e \in s.\hat{\mathcal{E}}} \texttt{\#rcv}(\mathsf{tok}, e, s) - \sum_{e \in s.\check{\mathcal{E}}} \texttt{\#sent}(\mathsf{tok}, s, e)) \\
= & \qquad \{\ 7\ \} \\
 & invariant.(true\ )
\end{array}
$$

We now present a lemma that will be used in the last two proofs.
Lemma S1.

$$
\begin{array}{ll}
 & s\texttt{.clienttok} \leadsto \neg s\texttt{.clienttok} \\
\Leftarrow & \\
 & \langle \forall\, k : k \in \mathbb{N} :: \texttt{\#sent}(\mathsf{tok}, s, s.c) \geq k \leadsto \texttt{\#del}(\mathsf{tok}, s.c, s) \geq k \rangle
\end{array}
$$

Proof of S1.

$$
\begin{array}{ll}
 & s\texttt{.clienttok} \leadsto \neg s\texttt{.clienttok} \\
\Leftarrow & \qquad \{\ 9 \text{ and server algorithm }\ \} \\
 & \texttt{\#sent}(\mathsf{tok}, s, s.c) > \texttt{\#rcv}(\mathsf{tok}, s.c, s) \leadsto s\texttt{.clienttok} \wedge (s.c, s).\mathsf{tok} \\
\Leftarrow & \qquad \{\ \langle \forall\, k : k \in \mathbb{N} :: \texttt{\#rcv}(\mathsf{tok}, s.c, s) = k \ \ \textbf{unless} \ \ (s.c, s).\mathsf{tok} \rangle\ \} \\
 & \langle \forall\, k : k \in \mathbb{N} :: \texttt{\#sent}(\mathsf{tok}, s, s.c) \geq k \leadsto \texttt{\#del}(\mathsf{tok}, s.c, s) \geq k \rangle
\end{array}
$$

We next show that the server satisfies modified rely-guarantee pair (15):

$$
\begin{array}{ll}
 & \langle \forall\, k : k \in \mathbb{N} :: \texttt{\#del}(\mathsf{tok}, s.in, s) \geq k \leadsto \texttt{\#sent}(\mathsf{tok}, s, s.out) \geq k + s\texttt{.tok\_init} \rangle \\
\Leftarrow & \qquad \{\ \text{calculus}\ \} \\
 & \langle \forall\, k : k \in \mathbb{N} :: \texttt{\#del}(\mathsf{tok}, s.in, s) \geq k \leadsto \texttt{\#rcv}(\mathsf{tok}, s.in, s) \geq k \rangle \wedge \\
 & \langle \forall\, k : k \in \mathbb{N} :: \texttt{\#rcv}(\mathsf{tok}, s.in, s) \geq k \leadsto \texttt{\#sent}(\mathsf{tok}, s, s.out) \geq k + s\texttt{.tok\_init} \rangle \\
= & \qquad \{\ \text{server algorithm}\ \} \\
 & \langle \forall\, k : k \in \mathbb{N} :: \texttt{\#rcv}(\mathsf{tok}, s.in, s) \geq k \leadsto \texttt{\#sent}(\mathsf{tok}, s, s.out) \geq k + s\texttt{.tok\_init} \rangle \\
\Leftarrow & \qquad \{\ 8\ \} \\
 & s\texttt{.clienttok} \leadsto \neg s\texttt{.clienttok} \\
\Leftarrow & \qquad \{\ \text{Lemma S1}\ \} \\
 & \langle \forall\, k : k \in \mathbb{N} :: \texttt{\#sent}(\mathsf{tok}, s, s.c) \geq k \leadsto \texttt{\#del}(\mathsf{tok}, s.c, s) \geq k \rangle
\end{array}
$$

Finally, we prove that the server satisfies modified rely-guarantee pair (16).

$$
\begin{array}{ll}
 & \langle \forall\ k : k \in \mathbb{N} :: \texttt{\#del}(\mathsf{req}, s.c, s) \geq k \leadsto \texttt{\#sent}(\mathsf{tok}, s, s.c) \geq k \rangle \\
\Leftarrow & \qquad \{ \text{ calculus } \} \\
 & \langle \forall\ k : k \in \mathbb{N} :: \texttt{\#del}(\mathsf{req}, s.c, s) \geq k \leadsto \texttt{\#rcv}(\mathsf{req}, s.c, s) \geq k \rangle \ \wedge \\
 & \langle \forall\ k : k \in \mathbb{N} :: \texttt{\#rcv}(\mathsf{req}, s.c, s) \geq k \leadsto \texttt{\#sent}(\mathsf{tok}, s, s.c) \geq k \rangle \\
\Leftarrow & \qquad \{ \text{ server algorithm } \} \\
 & (s\texttt{.hungry} \leadsto \neg s\texttt{.hungry}) \wedge (s\texttt{.clienttok} \leadsto \neg s\texttt{.clienttok}) \\
\Leftarrow & \qquad \{ \text{ Lemma S1 } \} \\
 & (s\texttt{.hungry} \leadsto \neg s\texttt{.hungry}) \ \wedge \\
 & \langle \forall\ k : k \in \mathbb{N} :: \texttt{\#sent}(\mathsf{tok}, s, s.c) \geq k \leadsto \texttt{\#del}(\mathsf{tok}, s.c, s) \geq k \rangle \\
\Leftarrow & \qquad \{ \text{ server algorithm } \} \\
 & (s\texttt{.hungry} \leadsto s\texttt{.hungry} \wedge (s.in, s).\mathsf{tok}) \ \wedge \\
 & \langle \forall\ k : k \in \mathbb{N} :: \texttt{\#sent}(\mathsf{tok}, s, s.c) \geq k \leadsto \texttt{\#del}(\mathsf{tok}, s.c, s) \geq k \rangle \\
\Leftarrow & \qquad \{ \ s\texttt{.hungry} \ \ \textbf{unless} \ \ s\texttt{.hungry} \wedge (s.in, s).\mathsf{tok} \ \} \\
 & (\mathit{true} \ \leadsto (s.in, s).\mathsf{tok}) \ \wedge \\
 & \langle \forall\ k : k \in \mathbb{N} :: \texttt{\#sent}(\mathsf{tok}, s, s.c) \geq k \leadsto \texttt{\#del}(\mathsf{tok}, s.c, s) \geq k \rangle \\
\Leftarrow & \qquad \{ \text{ calculus } \} \\
 & \langle \forall\ k : k \in \mathbb{N} :: \mathit{true} \ \leadsto \texttt{\#del}(\mathsf{tok}, s.in, s) \geq k \rangle \ \wedge \\
 & \langle \forall\ k : k \in \mathbb{N} :: \texttt{\#sent}(\mathsf{tok}, s, s.c) \geq k \leadsto \texttt{\#del}(\mathsf{tok}, s.c, s) \geq k \rangle \\
\Leftarrow & \qquad \{ \text{ calculus and } \texttt{tok\_init} > 0 \ \} \\
 & \langle \forall\ k : k \in \mathbb{N} :: \texttt{\#del}(\mathsf{tok}, s.in, s) \geq k \leadsto \texttt{\#sent}(\mathsf{tok}, s, s.out) \geq k + s\texttt{.tok\_init} \rangle \ \wedge \\
 & \langle \forall\ k : k \in \mathbb{N} :: \texttt{\#sent}(\mathsf{tok}, s, s.out) \geq k \leadsto \\
 & \qquad \texttt{\#del}(\mathsf{tok}, s.in, s) \geq k + \texttt{tok\_init} - s\texttt{.tok\_init} \rangle \ \wedge \\
 & \langle \forall\ k : k \in \mathbb{N} :: \texttt{\#sent}(\mathsf{tok}, s, s.c) \geq k \leadsto \texttt{\#del}(\mathsf{tok}, s.c, s) \geq k \rangle \\
\Leftarrow & \qquad \{ \text{ proof of (15) } \} \\
 & \langle \forall\ k : k \in \mathbb{N} :: \texttt{\#sent}(\mathsf{tok}, s, s.out) \geq k \leadsto \\
 & \qquad \texttt{\#del}(\mathsf{tok}, s.in, s) \geq k + \texttt{tok\_init} - s\texttt{.tok\_init} \rangle \ \wedge \\
 & \langle \forall\ k : k \in \mathbb{N} :: \texttt{\#sent}(\mathsf{tok}, s, s.c) \geq k \leadsto \texttt{\#del}(\mathsf{tok}, s.c, s) \geq k \rangle
\end{array}
$$

$\square$

References

[AL93] M. Abadi and L. Lamport. Composing Specifications. *ACM Transactions on Programming Languages and Systems*, 15(1):73–132, January 1993.

[BLCGP92] T. Berners-Lee, R. Cailliau, J. Groff, and B. Pollermann. World Wide Web: The Information Universe. *Electronic Networking: Research, Applications, and Policy*, 1(2), 1992.

[Cha94] K.M. Chandy. Properties of Concurrent Programs. *Formal Aspects of Computing*, 6(6):607–619, 1994.

[CM88] K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, Reading, MA, 1988.

[CR97] K.M. Chandy and A. Rifkin. Systematic Composition of Objects in Distributed Internet Applications. January 1997. Submitted to *Proceedings of the 30th Hawaii International Conference on System Sciences.*

[DS90] E.W. Dijkstra and C.S. Scholten. *Predicate Calculus and Program Semantics.* Springer-Verlag, New York, 1990.

[GJS96] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison-Wesley Developers Press, Sunsoft Java Series, 1996.

[Hoa69] C.A.R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–583, October 1969.

[Lam94] L. Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[MSC94] R. Manohar, P.A.G. Sivilotti, and K.M. Chandy. *Synchronization Using Gossip.* Internal Note RVM-14, August 1994.

[Obj95] Object Management Group. *The Common Object Request Broker: Architecture and Specification (CORBA).* OMG, 1995. Revision 2.0.