Local Rely-Guarantee Specifications

Murat Demirbas* Paolo A.G. Sivilotti* Rajit Manohar†

* Department of Computer & Information Science
The Ohio State University
Columbus, Ohio 43210 USA

† School of Electrical & Computer Engineering
Cornell University
Ithaca, NY 14853 USA

Abstract

We present a specification notation for components of concurrent systems and an accompanying proof methodology for reasoning about the composition of these components. Our specification construct enforces a disciplined use of rely-guarantee pairs and in turn enables us to prove (and later to reuse) component properties without having to examine its environment. Our proof methodology requires rely-guarantee properties to be composed explicitly by building an acyclic implication ladder and in turn enables us to do compositional proofs of both safety and liveness properties. Two examples (namely l-mutual exclusion on rings and distributed gossip synchronization) illustrate the simplicity and compositionality of our specification construct, and demonstrate how our methodology can be used to create structured and reusable proofs of safety and liveness properties of distributed systems.

Keywords: Specifications, Composition, Rely-guarantee, Distributed systems.

1 Introduction

Driven by demands for price-performance, enhanced communication, reactivity, and robustness, there has been a proliferation of loosely-coupled distributed systems, as reflected in the emergence of standards such as CORBA and technologies such as Java RMI and .NET. These technologies enable publishing, locating, binding, and using remote components by ensuring that the distributed-component interactions are all *syntactically* well-formed.

Syntactic compatibility alone, however, does not result in high-confidence distributed software. The formation of a correct distributed system requires, in addition, that the interactions be *semantically* well-formed and formally verified. Typical verification methods for distributed systems such as deduction [1–3] and model checking [4] are noncompositional and require a study of the entire system. Hence, they are effective only on small examples and do not scale up for large systems.

In the compositional verification literature, rely-guarantee is one of the best studied approaches. The intuition for using rely-guarantee specifications for a component M is that M needs to behave correctly only when its environment behaves correctly. A rely-guarantee specification for M is (R, G) where R denotes the conditions that M relies on its environment to provide, and G denotes the conditions that M guarantees in return. Rely-guarantee specifications enable application developers 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. The difficulty in applying rely-guarantee approaches to composition is that both R and G can refer to variables that are global to the computation, and as a result, one might need to consider all legal environments to be able to prove such a rely-guarantee property.

In this paper, we propose a rely-guarantee specification construct that departs from the traditional rely-guarantee approach in two significant ways:

1. all predicates must be on *local* component state only,
2. properties are composed by explicitly building an acyclic implication ladder.

The first item is a restriction on traditional rely-guarantee specifications. This restriction enables us to prove important properties about a component without having to examine its environment. This isolation simplifies the proof of component properties, and also enables us to reuse these properties when the component is composed with different environments.

The implication ladder in the second item gives a total ordering of the properties such that each rely is implied by the truth of all rely and guarantee properties previously listed. This enables us to compose *liveness* properties as well as *safety* properties.

Organization of the paper. We first present a brief summary of related work in Section 2. After presenting the preliminaries in Section 3, we define our local rely-guarantee specification construct and then derive the related composition theorems in Section 4. The methodology is explored by specifying and validating two synchronization problems which demonstrate the strength of the approach in independent component analysis and proof reuse. In Section 5, we present our first illustration, a solution to the l-mutual exclusion problem on a ring. In Section 6, we present our second illustration, a solution to the distributed gossip synchronization problem. Finally, in Section 7, we comment on the applicability and reusability of our methodology. We relegate some of the detailed proofs to the Appendices.

2 Related work

Here, we present a brief summary of some related work in the rely-guarantee literature.

Proofs of network of processes [5] (Misra and Chandy, 81): In this paper, a proof method based on specifying every process by a rely-guarantee pair is given for distributed programs. Network specification is proven by applying inference rules (based on Hoare's trace theory) to the specifications of component processes. This work takes a partial-correctness approach, and does not address liveness guarantees. This work can be considered as origin of rely-guarantee literature.

A proof technique for rely-guarantee properties [6] (Stark, 85): Stark's proof technique rules out circular dependencies in liveness conditions; one can derive global liveness properties of a system of concurrent processes from a set of local liveness properties. The intuition is to find a set of specifications that "cuts" the dependence between a component M_i and its environment. He likens this cut-set to finding a loop invariant, and does not offer a technique that can identify a cut-set. Note that our implication ladder rule in our proof technique results in such a cut-set.

Conjoining specifications [7] (Abadi and Lamport, 93): Temporal Logic of Actions (TLA) is used for describing systems; parallel composition is represented as conjunction. The main theorem states that if all the "rely" specifications (R, $\forall i :: R_i$) are *safety* properties, then circularity in "composition principle" is safe (i.e., does not lead to circular reasoning).

Distributed algorithms [8] (Lynch, 96): In this book a model of computation in asynchronous distributed systems, namely input/output automata, is introduced. This model is similar to our work in that properties about automota are proved in complete isolation and are independent of the environment that the automaton is to be composed with. A composition theorem is given for safety properties, whereas in our work we handle liveness properties as well. Using the "preserves" property, another composition theorem similar to rely-guarantee reasoning is presented, however, this theorem is also limited to composition of safety properties only.

Lazy compositional verification [9] (Shankar, 98): Shankar has also made the observation that unrestricted use of rely-guarantee specifications are bound to suffer from making references to the environment that the component will be composed with: "The difficulty in applying rely-guarantee approaches to composition is that they require component guarantee properties to be strong enough to imply any potential (future) environmental constraints. It is obviously not easy to anticipate all the potential constraints that might be placed on a component by the other components in a system."

Shankar introduces the term "lazy composition" where the component properties are proved by composing that component with an abstract environment. Let P_i be the component with the rely-guarantee spec (R_i, G_i). In lazy composition, G_i of P_i is proved of the system "$P_i||E_i$" where E_i captures the expected behavior of the abstract environment of P_i. Note that, G_i might not hold for "$P_i||P_j$" (since P_j might not satisfy the constraint E_i), however G_i holds for "$P_i||(P_j \wedge E_i)$". This way the proof obligation that P_j satisfies E_i can be lazily delayed till sufficient detail about the system is available.

Crudely speaking, this approach can be summarized as delaying the problem to surface rather than preventing it in the first place. In our work we prevent this problem to occur by restricting the rely-guarantee properties to only local properties of components.

3 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*. For simplicity of presentation, we assume that 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

Our definition of the model of computation gives the following properties on the ghost variables characterizing channels.

$$
\begin{array}{lrcl}
(1) & \#\texttt{sent}(c) & \geq & \#\texttt{del}(c) \\
(2) & \#\texttt{del}(c) & \geq & \#\texttt{rcv}(c) \\
(3) & \#\texttt{sent}(c) \geq k & \leadsto & \#\texttt{del}(c) \geq k
\end{array}
$$

Predicates. 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 [10]).

Properties. 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.

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.

Using **next** property enables us to deal with stuttering problem [11] trivially.

Definition 2 (stable). Safety property **stable**.p holds for a program if once p becomes true, it remains true from that point on.

$$\textbf{stable}.p \;\triangleq\; p \;\textbf{next}\; p$$

Definition 3 (invariant). Safety property **invariant**.p holds for a program if p holds initially, and if p remains true for the entire computation.

$$\textbf{invariant}.p \;\triangleq\; \textbf{initially}.p \wedge \textbf{stable}.p$$

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 4 (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.

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.

$$p \leadsto q \quad \triangleq \quad \textbf{invariant}.(p \;\Rightarrow\; \textbf{transient}.(\neg q))$$

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.

$$p \hookrightarrow q \quad \triangleq \quad p \leadsto q \wedge \textbf{stable}.q$$

Notation. Our proof format is from [12]. We denote quantification using the expression

$(Q\, x \;:\; r.x \;:\; t.x \;)$

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

$(Q\, x \;::\; t.x \;)$

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.

4 Proof Methodology

In this section, we first present the difficulty in proving the traditional rely-guarantee specifications. We then define our local rely-guarantee specification construct and derive the related composition theorems for safety and progress properties.

4.1 Compositionality of rely-gurantee specifications

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 [13], is an example of an any-component property (i.e., property holds for a system if it holds for any component of the system) 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:

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

(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.

Local 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:

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

Rely-guarantee properties which conform to this restriction, are *local 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. Next we present a theorem which allows us to restate the above definition as: $(R \triangleright G).P = R.P \ \Rightarrow \ G.P$

Theorem 1 (Compositionality). *Let $(R \triangleright G)$ be a property of program P. Then, $(R \triangleright G)$ is a property of $P \parallel Q$ for any program Q.*

Proof. The proof is obtained by structural induction on properties as defined in Section 3. □

We are simpler than TLA [2] since our model can be expressed using TLA, and there are properties of TLA we cannot express.

4.2 Composition theorem

Our composition theorem for local rely-guarantee specifications is valid not only for the safety properties but also for liveness properties as well.

Theorem 2 (Composition theorem). *Let $(R_1 \triangleright G_1), \ldots, (R_n \triangleright G_n)$ be a collection of local rely-guarantee properties of a program that relies on property R. Assume that, for all k,*

$$R \wedge (\forall j \ : \ j < k \ : \ R_j \wedge G_j \ \Rightarrow \ R_k \ )$$

holds. Then,

$$(R \triangleright (\forall i \ :: \ R_i \wedge G_i \ ))$$

is a property of the program.

Proof. The proof is obtained by induction on i. As the induction hypothesis at stage k of the construction, we assume that

$$R \wedge (\forall j : j < k : R_j \wedge G_j \Rightarrow R_k)$$

holds and conclude from the definition of rely-guarantee that G_k holds. $\square$

Restricting the composition theorem above to merely safety properties permits us to establish the following theorem:

Theorem 3 (Composing safety properties). *Let $(R_1 \triangleright G_1), \ldots, (R_n \triangleright G_n)$ be a collection of local 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,*

$$R \wedge (\forall j : j \neq k : G_j \Rightarrow R_k)$$

holds. Then,

$$(R \triangleright (\forall j :: R_j \wedge G_j))$$

is a property of the system.

This is the same theorem as in [7], and the proof is obtained by induction on the sequence of states in the computation.

5 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 5.

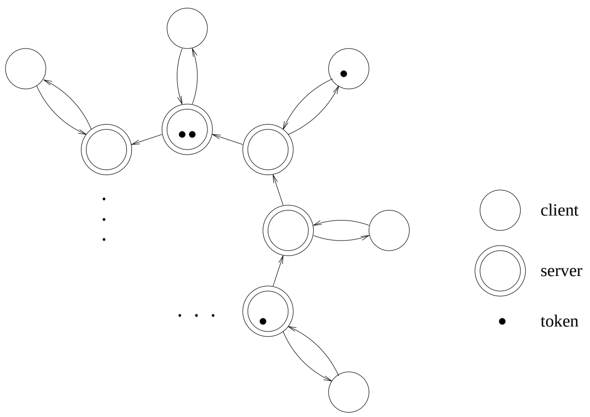


Figure 1: Clients Communicate with a Ring of Servers

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.

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 $(p,q).\mathsf{tok}$.

5.1 Problem statement

5.1.1 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.

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

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

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

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

5.1.2 Progress

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

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

5.2 Algorithm

We present the algorithm as an action system with pre and post conditions. 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 $s.$`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 $s.\mathtt{hungry} = \mathbf{false}$
boolean $s.\mathtt{clienttok} = \mathbf{false}$
actions:
$\neg s.\mathtt{hungry} \land \neg s.\mathtt{clienttok} \land (s.c, s).\overline{\mathsf{req}} \to$
$\underline{\text{inc}}\ \ \#\mathtt{rcv}(\mathsf{req}, s.c, s) \land s.\mathtt{hungry}$
$\neg s.\mathtt{hungry} \land (s.in, s).\overline{\mathsf{tok}} \to$
$\underline{\text{inc}}\ \ \#\mathtt{rcv}(\mathsf{tok}, s.in, s) \land \underline{\text{inc}}\ \ \#\mathtt{sent}(\mathsf{tok}, s, s.out)$
$s.\mathtt{hungry} \land (s.in, s).\overline{\mathsf{tok}} \to$
$\underline{\text{inc}}\ \ \#\mathtt{rcv}(\mathsf{tok}, s.in, s) \land \underline{\text{inc}}\ \ \#\mathtt{sent}(\mathsf{tok}, s, s.c) \land \neg s.\mathtt{hungry} \land s.\mathtt{clienttok}$
$s.\mathtt{clienttok} \land (s.c, s).\overline{\mathsf{tok}} \to$
$\underline{\text{inc}}\ \ \#\mathtt{rcv}(\mathsf{tok}, s.c, s) \land \underline{\text{inc}}\ \ \#\mathtt{sent}(\mathsf{tok}, s, s.out) \land \neg s.\mathtt{clienttok}$

Client (process_id $c.s$)
initially:
$c.\mathsf{tok} = 0$
$c.\mathtt{sent\_req} = \mathbf{false}$
$c.\mathtt{crit} = \mathbf{false}$
$c.\mathtt{terminated} = \mathbf{false}$
actions:
$\neg c.\mathtt{sent\_req} \land \neg c.\mathtt{crit} \to$
$c.\mathtt{terminated}$
$\neg c.\mathtt{sent\_req} \land \neg c.\mathtt{crit} \land \neg c.\mathtt{terminated} \to$
$\underline{\text{inc}}\ \ \#\mathtt{sent}(\mathsf{req}, c, c.s) \land c.\mathtt{sent\_req}$
$c.\mathtt{sent\_req} \land (c.s, c).\overline{\mathsf{tok}} \to$
$\underline{\text{inc}}\ \ \#\mathtt{rcv}(\mathsf{tok}, c.s, c) \land \underline{\text{inc}}\ \ c.\mathsf{tok} \land \neg c.\mathtt{sent\_req} \land c.\mathtt{crit}$
$c.\mathtt{crit} \to$
$\underline{\text{inc}}\ \ \#\mathtt{sent}(\mathsf{tok}, c, c.s) \land \underline{\text{dec}}\ \ c.\mathsf{tok} \land \neg c.\mathtt{crit}$

5.3 Process properties and invariants

5.3.1 Invariants

The following invariants can be easily established.

Server.

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

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

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

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

Client.

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

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

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

5.3.2 Specifications

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

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.

$$
\begin{array}{ll}
(& \textbf{invariant}.(\textbf{true}) \\
\triangleright & \textbf{invariant}.(c.\mathsf{tok} = \sum_{e \in c.\mathcal{E}} \texttt{\#rcv}(\mathsf{tok},e,c) - \sum_{e \in c.\mathcal{E}} \texttt{\#sent}(\mathsf{tok},c,e)) \wedge \\
& \textbf{invariant}.(c.\mathsf{tok} = \underline{ord}(c.\texttt{crit})) \wedge \\
& (\forall e,k \,:\, e \in c.\mathcal{E} \wedge k \in \mathbb{N} \,:\, \texttt{\#rcv}(\mathsf{tok},e,c) \geq k \leadsto \texttt{\#sent}(\mathsf{tok},c,e) \geq k\ )\ )
\end{array} \tag{14}
$$

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

$$
\begin{array}{ll}
(& \textbf{invariant}.(\,(\forall e \,:\, e \in c.\mathcal{E} \,:\, \texttt{\#del}(\mathsf{tok},e,c) \leq \texttt{\#sent}(\mathsf{req},c,e)\ )) \\
\triangleright & (\forall e,k \,:\, e \in c.\mathcal{E} \wedge k \in \mathbb{N} \,:\, \texttt{\#del}(\mathsf{tok},e,c) \geq k \leadsto \texttt{\#rcv}(\mathsf{tok},e,c) \geq k\ )\ )
\end{array} \tag{15}
$$

Proof of client. Specification pair (14) 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 (5.2).

Specification pair (15) 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).

The formal proof is included in Appendix A.

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.

$$
\begin{array}{ll}
(& \textbf{invariant}.(\textbf{true}) \\
\triangleright & \textbf{invariant}.(\texttt{\#sent}(\mathsf{tok},s,s.c) \leq \texttt{\#del}(\mathsf{req},s.c,s)) \wedge \\
& \textbf{invariant}.(s.\mathsf{tok} = s.\texttt{tok\_init} - \sum_{e \in s.\check{\mathcal{E}}} \texttt{\#sent}(\mathsf{tok},s,e) + \sum_{e \in s.\hat{\mathcal{E}}} \texttt{\#rcv}(\mathsf{tok},e,s))\ )
\end{array} \tag{16}
$$

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

$$
\begin{array}{ll}
(& (\forall k \,:\, k \in \mathbb{N} \,:\, \texttt{\#sent}(\mathsf{tok},s,s.c) \geq k \leadsto \texttt{\#del}(\mathsf{tok},s.c,s) \geq k\ ) \\
\triangleright & (\forall k \,:\, k \in \mathbb{N} \,:\, \texttt{\#del}(\mathsf{tok},s.in,s) \geq k \leadsto \texttt{\#sent}(\mathsf{tok},s,s.out) \geq k + s.\texttt{tok\_init}\ )\ )
\end{array} \tag{17}
$$

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.

$$
\begin{array}{ll}
(& (\forall k \,:\, k \in \mathbb{N} \,:\, \texttt{\#sent}(\mathsf{tok},s,s.c) \geq k \leadsto \texttt{\#del}(\mathsf{tok},s.c,s) \geq k\ ) \wedge \\
& (\forall k \,:\, k \in \mathbb{N} \,:\, \texttt{\#sent}(\mathsf{tok},s,s.out) \geq k \leadsto \texttt{\#del}(\mathsf{tok},s.in,s) \geq k + \texttt{tok\_init} - s.\texttt{tok\_init}\ ) \\
\triangleright & (\forall k \,:\, k \in \mathbb{N} \,:\, \texttt{\#del}(\mathsf{req},s.c,s) \geq k \leadsto \texttt{\#sent}(\mathsf{tok},s,s.c) \geq k\ )\ )
\end{array} \tag{18}
$$

Proof of server. Specification pair (16) 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 $s.\texttt{tok\_init}$ tokens) and the annotation of the server algorithm (5.2).

Specification pair (17) 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 (18) 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.

The formal proof is included in Appendix B.

5.4 Proof of program specification

5.4.1 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 local rely-guarantees, this establishes the conjuntion of the guarantee clauses as a property of the program.

Server Rely (16) is a program property. Trivial since this rely is **invariant**.(**true**).
□

Client Rely (14) is a program property. Trivial since this rely is **invariant**.(**true**).
□

Client Rely (15) is a program property.

$$
\begin{array}{ll}
 & \textbf{true} \\
= & \quad \{ \text{ Server's rely-guarantee (16) } \} \\
 & (\forall c : c \in \mathcal{C} : \texttt{\#sent}(\mathsf{tok}, c.s, c) \le \texttt{\#del}(\mathsf{req}, c, c.s)\ ) \\
\Rightarrow & \quad \{\ 1\ \} \\
 & (\forall c : c \in \mathcal{C} : \texttt{\#del}(\mathsf{tok}, c.s, c) \le \texttt{\#del}(\mathsf{req}, c, c.s)\ ) \\
\Rightarrow & \quad \{\ 1\ \} \\
 & (\forall c : c \in \mathcal{C} : \texttt{\#del}(\mathsf{tok}, c.s, c) \le \texttt{\#sent}(\mathsf{req}, c, c.s)\ ) \\
\Box &
\end{array}
$$

Server Rely (17) is a program property.

$$\textbf{true}$$

$$
\begin{array}{ll}
= & \quad \{ \text{ Client's rely-guarantees (15) and (14) } \} \\
 & (\forall s,k \,:\, s \in \mathcal{S} \wedge k \in \mathbb{N} \,:\, \texttt{\#del}(\mathsf{tok},s,s.c) \geq k \leadsto \texttt{\#rcv}(\mathsf{tok},s,s.c) \geq k\ ) \wedge \\
 & (\forall s,k \,:\, s \in \mathcal{S} \wedge k \in \mathbb{N} \,:\, \texttt{\#rcv}(\mathsf{tok},s,s.c) \leadsto \texttt{\#sent}(\mathsf{tok},s.c,s)\ ) \\
\Rightarrow & \quad \{ \text{ calculus } \} \\
 & (\forall s,k \,:\, s \in \mathcal{S} \wedge k \in \mathbb{N} \,:\, \texttt{\#del}(\mathsf{tok},s,s.c) \geq k \leadsto \texttt{\#sent}(\mathsf{tok},s.c,s) \geq k\ ) \\
\Rightarrow & \quad \{ \ 1 \ \} \\
 & (\forall s,k \,:\, s \in \mathcal{S} \wedge k \in \mathbb{N} \,:\, \texttt{\#sent}(\mathsf{tok},s,s.c) \geq k \leadsto \texttt{\#sent}(\mathsf{tok},s.c,s) \geq k\ ) \\
\Rightarrow & \quad \{ \ 3 \ \} \\
 & (\forall s,k \,:\, s \in \mathcal{S} \wedge k \in \mathbb{N} \,:\, \texttt{\#sent}(\mathsf{tok},s,s.c) \geq k \leadsto \texttt{\#del}(\mathsf{tok},s.c,s) \geq k\ ) \\
\Box &
\end{array}
$$

Server Rely (18) 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 $s.out^N = s$.

$$
\begin{array}{ll}
 & \textbf{true} \\
= & \quad \{ \text{ Server } s.out\text{'s rely-guarantee (17) } \} \\
 & (\forall s,k \,:\, s \in \mathcal{S} \wedge k \in \mathbb{N} \,:\, \texttt{\#del}(\mathsf{tok},s,s.out) \geq k \leadsto \\
 & \quad \texttt{\#sent}(\mathsf{tok},s.out,s.out^2) \geq k + s.out.\texttt{tok\_init}\ ) \\
\Rightarrow & \quad \{ \ 3 \ \} \\
 & (\forall s,k \,:\, s \in \mathcal{S} \wedge k \in \mathbb{N} \,:\, \texttt{\#del}(\mathsf{tok},s,s.out) \geq k \leadsto \\
 & \quad \texttt{\#del}(\mathsf{tok},s.out,s.out^2) \geq k + s.out.\texttt{tok\_init}\ ) \\
\Rightarrow & \quad \{ \text{ Server } s.out^2\text{'s rely-guarantee (17) } \} \\
 & (\forall s,k \,:\, s \in \mathcal{S} \wedge k \in \mathbb{N} \,:\, \texttt{\#del}(\mathsf{tok},s,s.out) \geq k \leadsto \\
 & \quad \texttt{\#sent}(\mathsf{tok},s.out^2,s.out^3) \geq k + s.out.\texttt{tok\_init} + s.out^2.\texttt{tok\_init}\ ) \\
\Rightarrow & \quad \{ \ 3 \ \} \\
 & \ldots \\
\Rightarrow & \\
 & (\forall s,k \,:\, s \in \mathcal{S} \wedge k \in \mathbb{N} \,:\, \texttt{\#del}(\mathsf{tok},s,s.out) \geq k \leadsto \\
 & \quad \texttt{\#del}(\mathsf{tok},s.out^{N-1},s.out^N) \geq k + s.out.\texttt{tok\_init} + \ldots + s.out^{N-1}.\texttt{tok\_init}\ ) \\
= & \quad \{ \text{ ring architecture } \} \\
 & (\forall s,k \,:\, s \in \mathcal{S} \wedge k \in \mathbb{N} \,:\, \texttt{\#del}(\mathsf{tok},s,s.out) \geq k \leadsto \\
 & \quad \texttt{\#del}(\mathsf{tok},s.in,s) \geq k + \texttt{tok\_init} - s.\texttt{tok\_init}\ ) \\
\Rightarrow & \quad \{ \ 1 \ \} \\
 & (\forall s,k \,:\, s \in \mathcal{S} \wedge k \in \mathbb{N} \,:\, \texttt{\#sent}(\mathsf{tok},s,s.out) \geq k \leadsto \\
 & \quad \texttt{\#del}(\mathsf{tok},s.out,s) \geq k + \texttt{tok\_init} - s.\texttt{tok\_init}\ ) \\
\Box &
\end{array}
$$

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

$$
\begin{array}{ll}
 & (\forall s \,:\, s \in \mathcal{S} \,:\, \textbf{invariant}.(\texttt{\#sent}(\mathsf{tok},s,s.c) \leq \texttt{\#del}(\mathsf{req},s.c,s)) \\
 & \quad \wedge\, \textbf{invariant}.(s.\mathsf{tok} = s.\texttt{tok\_init} - \texttt{\#sent}(\mathsf{tok},s,s.out) - \texttt{\#sent}(\mathsf{tok},s,s.c) \\
 & \qquad\qquad + \texttt{\#rcv}(\mathsf{tok},s.in,s) + \texttt{\#rcv}(\mathsf{tok},s.c,s)) \\
 & \quad \wedge\ (\forall k \,:\, k \in \mathbb{N} \,:\, \texttt{\#del}(\mathsf{tok},s.in,s) \geq k \leadsto \texttt{\#sent}(\mathsf{tok},s,s.out) \geq k + s.\texttt{tok\_init}\ ) \\
 & \quad \wedge\ (\forall k \,:\, k \in \mathbb{N} \,:\, \texttt{\#del}(\mathsf{req},s.c,s) \geq k \leadsto \texttt{\#sent}(\mathsf{tok},s,s.c) \geq k\ )\ ) \\
\wedge & (\forall c \,:\, c \in \mathcal{C} \,:\, \textbf{invariant}.(c.\mathsf{tok} = \texttt{\#rcv}(\mathsf{tok},c.s,c) - \texttt{\#sent}(\mathsf{tok},c,c.s)) \\
 & \quad \wedge\, \textbf{invariant}.(c.\mathsf{tok} = \underline{ord}(c.\texttt{crit}))
\end{array}
$$

$$\wedge\ (\forall k\ :\ k \in \mathbb{N}\ :\ \texttt{\#rcv}(\mathsf{tok}, c.s, c) \geq k \leadsto \texttt{\#sent}(\mathsf{tok}, c, c.s) \geq k\ )$$

(19) $$\wedge\ (\forall k\ :\ k \in \mathbb{N}\ :\ \texttt{\#del}(\mathsf{tok}, c.s, c) \geq k \leadsto \texttt{\#rcv}(\mathsf{tok}, c.s, c) \geq k\ )\ )$$

5.4.2 Proofs of safety and progress

Proof of 4.

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

□

Proof of 5.

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

□

Proof of 6.

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

□

6 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.

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.

6.1 Problem statement

6.1.1 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.

(20) $(\forall u : u \ \underline{nbr} \ I : \#\texttt{rcv(u)}I > 0) \Rightarrow (\forall u :: u$ has completed the gossip algorithm $)$

6.1.2 Progress

Completion. The only progress requirement is that eventually the initiator and the processes complete the gossip algorithm.

(21) $\textbf{true} \hookrightarrow (\forall u, I :: u, I$ has completed the gossip algorithm $)$

6.2 Algorithm

```
Message Type:
  PROCESS_ID;

Initiator()  {
  myid = my process id;
  // val = 0
  neighborhood = set of process_ids of neighbors;
  // val = 1
  foreach p in neighborhood
    send myid to p;
  // val = 2
  foreach p in neighborhood
    receive p;
  // val = 3
}

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;
  // val = 2
  foreach p in neighborhood - {parent}
    receive p;
  send myid to parent;
  // val = 3
}
```

$\textbf{initially}.((\forall u :: val.u = 0) \wedge val.I = 0$
$\wedge(\forall u, v :: \#\text{sent}(mesg, u, v) = \#\text{rcv}(mesg, v, u) = \#\text{del}(mesg, v, u) = 0))$

6.3 Process properties and invariants

As usual we write process properties in local rely-guarantee format. All the properties in this section has a rely clause of **true**. That is, they assert that a process will complete the guarantee clause regardless of the environment it is composed with. We call such properties *certificates*. We denote a certificate P as $(\ \triangleright\ P)$.

6.3.1 Channel properties

CS1. All the messages sent to the channel by the sender are eventually delivered. All the delivered messages are those sent by the sender.

$$\begin{array}{ll}(\ \triangleright & (\forall u, v : u\ \underline{nbr}\ v :: \#\text{sent}(msg, u, v) > 0 \hookrightarrow \#\text{del}(msg, u, v) > 0) \\ \quad\wedge & (\forall u, v : u\ \underline{nbr}\ v :: \#\text{del}(msg, u, v) > 0 \ \Rightarrow\ \#\text{sent}(msg, u, v) > 0)\ )\end{array}$$

6.3.2 Initiator properties

IS1. The initiator is phase 0 or 1 iff it has neither sent nor received any message.

$$\begin{array}{ll}(\ \triangleright & val.I = 0 \vee val.I = 1 \\ & \equiv\ (\forall u : u\ \underline{nbr}\ I :: \#\text{sent}(msg, I, u) = 0\ \wedge\ \#\text{rcv}(msg, u, I) = 0)\ )\end{array}$$

IS2. From phase 0 the initiator eventually proceeds to phase 1. (Actually, we do not distinguish phase 0 and 1 of the initiator.)

$$(\ \triangleright \quad val.I = 0 \leadsto val.I = 1\ )$$

IS3. The initiator is in phase 2 if it has sent messages to all its neighbors and has not yet received any message back.

$$\begin{array}{ll}(\ \triangleright & val.I = 2 \\ & \equiv\ (\forall u : u\ \underline{nbr}\ I :: \#\text{sent}(msg, I, u) > 0\ \wedge\ \#\text{rcv}(msg, u, I) = 0)\ )\end{array}$$

IS4. From phase 1 the initiator eventually sends messages to all its neighbors and proceeds to phase 2.

$$(\ \triangleright \quad val.I = 1 \leadsto val.I = 2\ )$$

IS5. The initiator is in phase 3 if it has received messages from all its neighbors.

$$\begin{array}{ll}(\ \triangleright & (\forall u : u\ \underline{nbr}\ I :: \#\text{rcv}(msg, u, I) > 0)\ ) \\ & \Rightarrow val.I = 3\ )\end{array}$$

IS6. All the messages delivered by the incoming channels of the initiator I are eventually received by I, and I receives only the messages delivered by its incoming channels.

$$\begin{array}{ll}(\ \triangleright & (\forall u : u\ \underline{nbr}\ I :: \#\text{del}(msg, u, I) > 0 \hookrightarrow \#\text{rcv}(msg, u, I) > 0) \\ \quad\wedge & (\forall v : u\ \underline{nbr}\ I :: \#\text{rcv}(msg, u, I) > 0 \ \Rightarrow\ \#\text{del}(msg, u, I) > 0)\ )\end{array}$$

6.3.3 Process properties

US1. A process u is in phase 0 if it has neither sent nor received any messages and its parent is undefined.

$$(\triangleright \quad val.u = 0 \\ \equiv (\forall v : u\ \underline{nbr}\ v :: \#\mathrm{sent}(msg, u, v) = 0\ \wedge\ \#\mathrm{rcv}(msg, v, u) = 0) \wedge parent.u = \perp)$$

US2. If u is in phase 0 and receives its first message from process x, then u sets its parent to be x. Once $parent.u$ is set, it cannot be reset.

$$(\triangleright \quad val.u = 0 \wedge (\exists x : x\ \underline{nbr}\ u :: \#\mathrm{rcv}(msg, x, u) > 0) \\ \hookrightarrow parent.u = x)$$

US3. u is in phase 1 if the parent of u is defined and u has not sent any messages.

$$(\triangleright \quad val.u = 1 \\ \equiv (\forall v : u\ \underline{nbr}\ v :: \#\mathrm{sent}(msg, u, v) = 0) \wedge parent.u \neq \perp)$$

US4. If u is in phase 0 and receives a message, then u eventually proceeds to phase 1.

$$(\triangleright \quad val.u = 0 \wedge (\exists x : x\ \underline{nbr}\ u :: \#\mathrm{rcv}(msg, x, u) > 0) \\ \leadsto val.u = 1)$$

US5. u is in phase 2 iff u has sent messages to all its neighbors (except its parent) and has not yet received messages back from all its neighbors.

$$(\triangleright \quad val.u = 2 \\ \equiv \quad (\forall v : u\ \underline{nbr}\ v \wedge v \neq parent.u :: \#\mathrm{sent}(msg, u, v) > 0) \\ \wedge \quad (\exists v : u\ \underline{nbr}\ v : \#\mathrm{rcv}(msg, v, u) = 0))$$

US6. From phase 1 u eventually sends messages to all its neighbors and proceeds to phase 2.

$$(\triangleright \quad val.u = 1 \leadsto val.u = 2)$$

US7. u is in phase 3 if it has sent a message back to its parent.

$$(\triangleright \quad \#\mathrm{sent}(msg, u, parent.u) > 0 \\ \Rightarrow val.u = 3)$$

US8. If u is in phase 2 and has received messages back from all its neighbors and has sent a message to its parent, then u eventually proceeds to phase 3.

$$(\triangleright \quad val.u = 2 \wedge (\forall v : u\ \underline{nbr}\ v :: \#\mathrm{rcv}(msg, x, u) > 0) \wedge\ \#\mathrm{sent}(msg, u, parent.u) > 0 \\ \leadsto val.u = 3)$$

US9. All the messages delivered by the incoming channels of u are eventually received by u, and u receives only the messages delivered by its incoming channels.

$$(\triangleright \quad (\forall v : u\ \underline{nbr}\ v :: \#\mathrm{del}(msg, v, u) > 0 \hookrightarrow \#\mathrm{rcv}(msg, v, u) > 0) \\ \wedge \quad (\forall v : u\ \underline{nbr}\ v :: \#\mathrm{rcv}(msg, v, u) > 0 \quad \Rightarrow \quad \#\mathrm{del}(msg, v, u) > 0))$$

6.3.4 Global invariants

Lemma 1 (GS1). $(\forall u, v : u\ \underline{nbr}\ v \wedge parent.v = u :: val.u = 3 \ \Rightarrow\ val.v = 3)$

Proof.
$$
\begin{array}{ll}
 & u\ \underline{nbr}\ v \wedge parent.v = u \wedge val.u = 3 \\
\Rightarrow & \quad \{\ \text{US7}\ \} \\
 & \#\text{rcv}(mesg, v, u) > 0 \\
\Rightarrow & \quad \{\ \text{US9, CS1}\ \} \\
 & \#\text{sent}(mesg, v, u) > 0 \\
\Rightarrow & \quad \{\ parent.v = u,\ \text{US7}\ \} \\
 & val.v = 3
\end{array}
$$

□

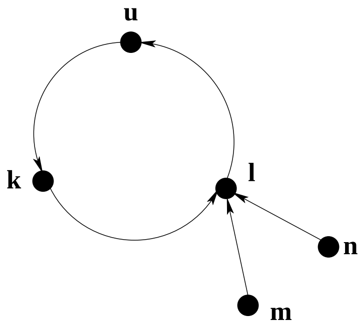
Lemma 2 (GS2). $(\forall u, v : u\ \underline{nbr}\ v \wedge parent.v = u :: val.v = 1 \ \Rightarrow\ val.u = 2)$

Proof.
$$
\begin{array}{ll}
 & u\ \underline{nbr}\ v \wedge parent.v = u \wedge val.v = 1 \\
\Rightarrow & \quad \{\ \text{US3}\ \} \\
 & \#\text{rcv}(mesg, v, u) > 0 \\
\Rightarrow & \quad \{\ \text{US9, CS1}\ \} \\
 & \#\text{sent}(mesg, v, u) > 0 \\
\Rightarrow & \quad \{\ \text{US1, US3}\ \} \\
 & val.u \geq 2 \\
\Rightarrow & \quad \{\ val.v = 1,\ \text{GS1}\ \} \\
 & val.u = 3
\end{array}
$$

□

Lemma 3 (GS3). *GS3::* $(\forall u : val.u \geq 1 :: (\exists m : m \in N :: parent^m.u = I))$

Proof. Assume for a contradiction that $(\exists u : val.u \geq 1 :: (\forall m : M \in N :: parent^m.j \neq I))$. Since there are a finite number of processes, u should be involved in a cycle.



In the figure, the arrows represent the *parent* relation. Without loss of generality select u to be a process in the cycle. Since the *parent* relation remains unchanged for $val \geq 1$ {from US2 }, when $val.u = 1$ the parent relation would be the same as in the figure. When $val.u = 1$, from GS2 it follows that $val.k \geq 2$, $val.l \geq 2$, and thus, $val.u \geq 2$. This is a contradiction, and hence, $(\forall u : val.u \geq 1 :: (\exists m : m \in N :: parent^m.u = I))$ follows. □

6.4 Proof of program specification

Theorem 4 (Safety). $(\forall u : u\ \underline{nbr}\ I :: \#rcv(msg, u, I) > 0) \ \Rightarrow\ ((\forall u :: val.u = 3) \wedge val.I = 3)$

Proof. $(\forall u : u \ \underline{nbr}\ I :: \#\mathrm{rcv}(msg, u, I) \geq 0)$
$\Rightarrow$ { IS5 }
$val.I = 3$
$\Rightarrow$ { GS1, GS3, induction on the height of u in *parent* tree }
$(\forall u :: val.u = 3)$

$\square$

Theorem 5 (Liveness). **true** $\hookrightarrow ((\forall u :: val.u = 3) \wedge val.I = 3)$

Proof. **true**
$\Rightarrow$ { GS5 }
$(\forall u :: val.u = 0) \wedge val.I = 0$
$\leadsto$ { IS2, IS4 }
$(\forall u :: val.u = 0) \wedge val.I = 2$
$\hookrightarrow$ { IS3, CS1, US9, US4, US6 }
$(\forall u : u \ \underline{nbr}\ I :: val.u \geq 2)$
$\hookrightarrow$ { GS3, previous step, induction on the height of u in *parent* tree }
$(\forall u :: val.u \geq 2)$
$\Rightarrow$ { GS3, tree structure }
$(\forall u :: val.u \geq 2) \wedge (\exists v :: (\forall u : u \ \underline{nbr}\ v :: v \neq parent.u))$
$\hookrightarrow$ { US8 }
$(\forall u :: val.u \geq 2) \wedge (\exists v :: (\forall u : u \ \underline{nbr}\ v :: v \neq parent.u \vee val.u = 3) \wedge val.v = 3)$
$\hookrightarrow$ { US7, US8, induction on the height of v in *parent* tree }
$(\forall u :: val.u = 3)$
$\hookrightarrow$ { GS3, US7, CS1, IS6 }
$(\forall u :: val.u = 3) \wedge val.I = 3$

$\square$

7 Concluding Remarks

The l-mutual exclusion and distributed gossip synchronization examples illustrate the principal strength of this approach. In both examples, each process type is specified, implemented, and verified in isolation. The program is then demonstrated to meet the problem specification by considering only the composition of local 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 are needed in this phase of program verification.

The principal strength of the local 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.

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 l-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 process, 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.

References

[1] K. M. Chandy and J. Misra. *Parallel Program Design*. Addison-Wesley Publishing Company, 1988.

[2] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[3] E. A. Emerson. Temporal and modal logic. *Handbook of Theoretical Computer Science*, B: Formal Models and Semantics:996–1072, 1990.

[4] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[5] J. Misra and K. M. Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, 7(4):417–426, 1981.

[6] E. W. Stark. A proof technique for rely/guarantee properties. *Foundations of software technology and theoretical computer science, Lecture Notes in computer science 106*, pages 369–391, 1985.

[7] M. Abadi and L. Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–534, May 1995.

[8] Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.

[9] N. Shankar. Lazy compositional verification. *Lecture Notes in Computer Science*, 1536:541–564, 1998.

[10] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, October 1969.

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

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

[13] K.M. Chandy. Properties of concurrent programs. *Formal Aspects of Computing*, 6(6):607–619, 1994.

A Proof of Mutual Exclusion Client

The guarantee properties of (14) are invariant properties of the client process.

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

Also:

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

And finally:

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

We next prove the client satisfies rely-guarantee pair (15).

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

□

B Proof of Mutual Exclusion Server

The guarantee properties of pair (16) are invariant properties of the server process and do not depend on the environment at all.

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

Also:

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

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

Lemma S1.

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

Proof of S1.

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

We next show that the server satisfies local rely-guarantee pair (17):

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

Finally, we prove that the server satisfies local rely-guarantee pair (18).

$$
\begin{array}{ll}
& (\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\ ) \\
\Leftarrow & \qquad \{ \text{ calculus } \} \\
& (\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\ ) \wedge \\
& (\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\ ) \\
\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 \\
& (\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\ ) \\
\Leftarrow & \qquad \{ \text{ server algorithm } \} \\
& (s.\texttt{hungry} \leadsto s.\texttt{hungry} \wedge (s.in,s).\mathsf{tok}) \wedge \\
& (\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\ ) \\
\Leftarrow & \qquad \{ \ s.\texttt{hungry} \ \textbf{unless} \ s.\texttt{hungry} \wedge (s.in,s).\mathsf{tok} \ \} \\
& (\textbf{true} \leadsto (s.in,s).\mathsf{tok}) \wedge \\
& (\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\ ) \\
\Leftarrow & \qquad \{ \text{ calculus } \} \\
& (\forall k \,:\, k \in \mathbb{N} \,:\, \textbf{true} \leadsto \texttt{\#del}(\mathsf{tok},s.in,s) \geq k\ ) \wedge \\
& (\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\ ) \\
\Leftarrow & \qquad \{ \text{ calculus and } \texttt{tok\_init} > 0 \ \} \\
& (\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}\ ) \wedge \\
& (\forall k \,:\, k \in \mathbb{N} \,:\, \texttt{\#sent}(\mathsf{tok},s,s.out) \geq k \leadsto \\
& \quad \texttt{\#del}(\mathsf{tok},s.in,s) \geq k + \texttt{tok\_init} - s.\texttt{tok\_init}\ ) \wedge \\
& (\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\ ) \\
\Leftarrow & \qquad \{ \text{ proof of (17) } \} \\
& (\forall k \,:\, k \in \mathbb{N} \,:\, \texttt{\#sent}(\mathsf{tok},s,s.out) \geq k \leadsto \\
& \quad \texttt{\#del}(\mathsf{tok},s.in,s) \geq k + \texttt{tok\_init} - s.\texttt{tok\_init}\ ) \wedge \\
& (\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\ )
\end{array}
$$

□