Causing Communication Closure: Safe Program Composition with Reliable Non-FIFO Channels[*]

Kai Engelhardt[†] Yoram Moses[‡]

May 6, 2009

A semantic framework for analyzing safe composition of distributed programs is presented. Its applicability is illustrated by a study of program composition when communication is reliable but not necessarily FIFO. In this model, special care must be taken to ensure that messages do not accidentally overtake one another in the composed program. We show that barriers do not exist in this model. Indeed, no program that sends or receives messages can automatically be composed with arbitrary programs without jeopardizing their intended behavior. Safety of composition becomes context-sensitive and new tools are needed for ensuring it. A notion of *sealing* is defined, where if a program P is immediately followed by a program Q that seals P then P will be communication-closed—it will execute as if it runs in isolation. The investigation of sealing in this model reveals a novel connection between Lamport causality and safe composition. A characterization of sealable programs is given, as well as efficient algorithms for testing if Q seals P and for constructing a seal for a significant class of programs. It is shown that every sealable program that is open to interference on $O(n^2)$ channels can be sealed using $O(n)$ messages.

1. Introduction

Much of the distributed algorithms literature is devoted to solutions for individual tasks. Implicitly it may appear that these solutions can be readily combined to create larger applications. Composing such solutions is not, however, automatically guaranteed to maintain their correctness and their intended behavior. For example, algorithms are typically designed under the assumption that they begin executing in a well-defined initial global state in which all channels are empty. In most cases, the algorithms are not guaranteed to terminate in such a state. Another inherent feature of distributed systems is that, even though they are often designed in clearly separated phases, these phases typically execute concurrently. For instance, Lynch writes in [Lyn96, p. 523]:

[*] A preliminary version appeared as [EM05a]. Work was partially supported by ARC Discovery Grant RM02036.

[†] `kaie@cse.unsw.edu.au`, School of Computer Science and Engineering, The University of New South Wales, and NICTA, Sydney, NSW 2052, Australia. National ICT Australia is funded through the Australian Government's *Backing Australia's Ability* initiative, in part through the Australian Research Council.

[‡] `moses@ee.technion.ac.il`, Department of Electrical Engineering, Technion, Haifa, 32000 Israel. Work on this paper happened during a sabbatical visit to the School of Computer Science and Engineering, The University of New South Wales, Sydney, NSW 2052, Australia.

> "An MST algorithm can be used to solve the leader-election problem [. . .]. Namely, after establishing an MST, the processes participate in the *STtoLeader* protocol to select the leader. Note that the processes do not need to know when the MST algorithm has completed its execution throughout the network; it is enough for each process i to wait until it is finished locally, [. . .]."

In general, when two phases, such as implementations of an MST algorithm and of the *STtoLeader* algorithm, are developed independently and then executed in sequence, one phase may confuse messages originating from the other with its own messages. Perhaps the first formal treatment of this issue was via the notion of *communication-closed layers* introduced by Elrad and Francez in [EF82]. Consider a program $P = P_1 \parallel \ldots \parallel P_n$ consisting of n concurrent processes $P_i = Q_i;L_i;Q'_i$, the execution of which is, intuitively, divided into three phases, Q_i, L_i, and Q'_i. Elrad and Francez define $L = L_1 \parallel \ldots \parallel L_n$ to be a *communication-closed layer (CCL) in P* if under no execution of P does a command in some L_i communicate with a command in any Q_j or Q'_j [EF82]. If a program P can be decomposed into a sequence of CCLs then every execution of P can be viewed as a concatenation of executions of P's layers in order. Hence, reasoning about P can be reduced to reasoning about its layers in isolation. This approach has been investigated further and applied to a variety of problems by Janssen, Poel, and Zwiers [JPZ91, Jan94, Jan95, PZ92]. Stomp and de Roever considered related notions in the context of synchronous communication [SdR94]. Gerth and Shrira considered the issue of using distributed programs as off-the-shelf components to serve as layers in larger distributed programs [GS86]. They observe that the above definition of CCL is made with respect to the whole program P as context, and hence is unsuitable for off-the-shelf components. They solve the problem by defining L to be a *General Tail Communication Closed (GTCC)* layer if, roughly speaking, for *all* layers $T_1 \parallel \ldots \parallel T_n$ we have that L is a CCL in $L_1;T_1 \parallel \ldots \parallel L_n;T_n$. Since this definition does not refer to the surrounding program context of a layer, it asserts a certain quality of composability. Sequentially composing GTCC layers guarantees that each one of them is a CCL.

We develop a framework for defining and reasoning about various notions central to the design of CCLs in different models of communication. The communication model used in most of the literature concerning CCLs is that of reliable FIFO channels. In practice, channels often fail to satisfy this assumption. Three main sources of imperfection are loss, reordering, and duplication of messages by a channel. This paper studies the impact of message reordering on the design of CCLs. Our communication model, which we call Rel, will therefore assume that channels neither lose nor duplicate messages but message delivery is not necessarily FIFO. As we shall see, in Rel, the CCL property depends in an essential way on Lamport causality [Lam78]. Indeed, to ensure CCL, causality is *all* that is needed in Rel, whereas either duplication or loss already mandate the need for headers in messages [FL90, EM05c].

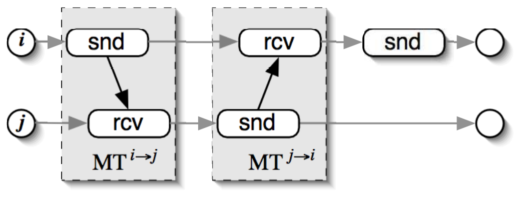


Figure 1: $\textsc{mt}^{j\to i}$ seals $\textsc{mt}^{i\to j}$.

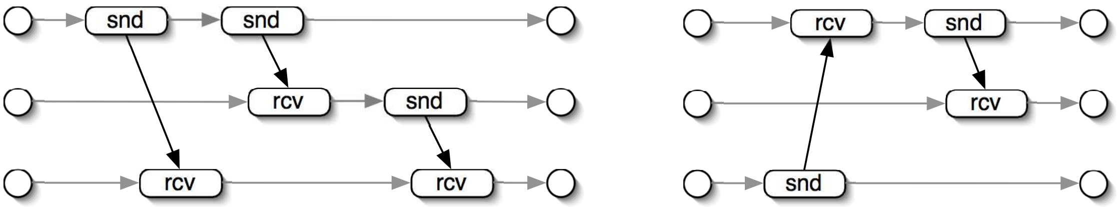
Consider for instance the task of transmitting a message m from process i to process j where it is stored in variable x. The task is accomplished by i performing $\textsc{snd}^{i\to j}_m$ to send the message and j performing $\textsc{rcv}^{j\leftarrow i}_x$ to receive it into variable x. This implementation denoted $\textsc{mt}^{i\to j}_{m\to x}$ (for *Message-Transmit*) works fine in isolation. Composing two copies[1] of $\textsc{mt}^{i\to j}$, however, does not guarantee the same behavior as executing the first to completion and then executing the second. Since communication is not FIFO, the second message sent by i could be the first one received by j. On the other hand, if $\textsc{mt}^{i\to j}$ is followed by $\textsc{mt}^{j\to i}$ no such interference occurs. Moreover, no later program can ever interfere with the first $\textsc{mt}^{i\to j}$ in this pair. Of course the second program, $\textsc{mt}^{j\to i}$, is still susceptible to interference, e.g., by another $\textsc{mt}^{j\to i}$. In fact, non-trivial programs are never safe from interference in Rel. As we shall show, for any terminating program P transmitting a message from i to j there is a program Q

[1] We omitted the subscript in $\textsc{mt}^{i\to j}$. Whenever a parameter is irrelevant to the point being made, we tend to omit it.

potentially interfering with communication in P. One consequence is that no terminating program that sends messages can be a GTCC layer.

The above discussion suggests that it is necessary to inspect the next layer in order to determine whether a given layer is a CCL. In fact, we shall define a notion of a program Q *sealing* its predecessor P, which will ensure that P is a CCL in P immediately followed by Q. For example, $\mathrm{MT}^{j\to i}$ seals $\mathrm{MT}^{i\to j}$ and vice versa. Intuitively, Q seals P if Q guarantees that no message sent after P can be received in P. Let us consider why $\mathrm{MT}^{j\to i}_{\mathrm{ACK}}$ seals $\mathrm{MT}^{i\to j}$. Suppose that a later message is sent on the channel from i to j as in Fig. 1. This send is performed only after the message sent in the opposite direction has been received by i, which in turn must have been sent after the first message has been received by j. Consequently, j's receive event must precede i's sending of the later message. Therefore, the later message cannot compete with the earlier one. A message transmitted in the opposite direction is often called an *acknowledgment*. More interesting examples of sealing are presented in Figures 2(a) and 3. For a decomposition of a program P into a sequence of ℓ layers $L^{(1)},\ldots,L^{(\ell)}$, it follows that if $L^{(k+1)}$ seals $L^{(k)}$ for all $1 \leq k < \ell$ then each layer $L^{(k)}$ is a CCL in P.

In [Lam78] Lamport defined causality among events of asynchronous message passing systems. Causality implies temporal precedence. As discussed above, transmitting an acknowledgment guarantees that the receive of the first message causally precedes any later sends on the same channel. Observe that the same effect could be obtained by other means ensuring the intended precedence. For instance, a causal chain consisting of a sequence of messages starting at j, going through a number of intermediate processes, and ending at i could be used just as well. While this *transitive* form of acknowledgment appears to be inefficient, a given message can play a role in a number of transitive acknowledgments. Fig. 2(a) illustrates a program consisting of the transmission of three messages over three different channels. It is sealed using transitive acknowledgments by the program displayed in Fig. 2(b), which sends only two messages.



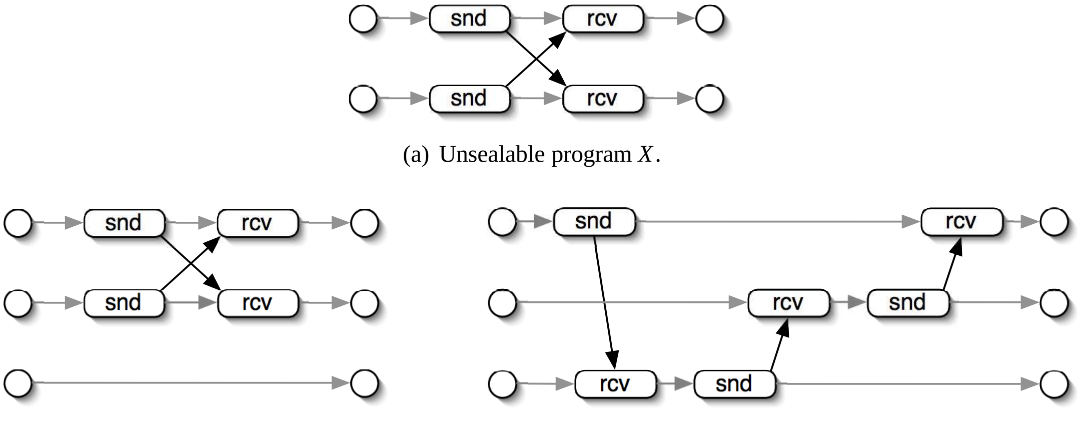
(a) A program P.

(b) A seal for P.

Figure 2: An example of sealing.

Indeed, we shall later show how $O(n)$ messages can usefully substitute for $\Omega(n^2)$ acknowledgments. Not all programs can be sealed. We shall later prove that program X shown in Fig. 3(a) is unsealable. The same program executed in the presence of a third process as in Fig. 3(b) is, however, sealable. Any seal of this program will necessarily use transitive acknowledgments as discussed above. See Fig. 3(c) for an illustration of one way this program can be sealed.

Contributions. The first main contribution of this paper is in the presentation of a framework studying safe composition of layers of distributed programs in different models of communication. Within the framework we define notions including CCL and barriers. Moreover, it is possible to define new notions such as sealing that play an important role in ensuring safe composition. In this paper the power of the framework is illustrated by a comprehensive study of safe composition in REL. In a companion paper [EM05b] the framework is used to define additional notions that are used to study safe composition in FIFO-models with duplicating and/or lossy channels.



(a) Unsealable program X.

(b) A sealable program P'.

(c) A seal for P'.

Figure 3: An example of a program for two processes that is unsealable unless a third process is added.

Our second main contribution is in identifying the notion of sealing and demonstrating its central role in the design of CCLs in REL. We study the theory of sealing in REL and present the following results.

- Sealable straight-line programs are completely characterized.
- A definition of the sealing *signature* of straight-line programs is given, which characterizes the sealing behavior of a program concisely, for both purposes, sealing and being sealed. The size of the signature is $O(n^2)$.
- An algorithm for deciding whether Q seals P based only on their signatures is presented.
- An algorithm for constructing seals for sealable straight-line programs is presented. It produces seals that perform less than $3n$ message transmissions even though $\Omega(n^2)$ channels may need to be sealed.

The restriction to straight-line programs is motivated by the undecidability of the corresponding problems for general programs. Specifically, the halting problem can be reduced to each of these problems for general programs. As far as communication closure is concerned, straight-line programs already display most of the interesting aspects relevant to the subject of sealing.

2. A Model of Distributed Programs with Layering

In this section we define a simple language for writing message-passing concurrent programs. Its composition operator "$*$" is called *layering*. Layering subsumes the two more traditional operators ";" and "$||$" (as discussed by Janssen in [Jan94]). The meaning of $P * Q$ is that each process i first executes its share of P and then proceeds directly to execute its share of Q. In particular, layering does not impose any barrier synchronization between P and Q. In other words, in $P * Q$ process i need not wait for any other processes to finish their shares of P before moving on to Q. Consequently, programs execute between *cuts* rather than global states. We shall define a notion $r[c,d] \Vdash P$ of a program P *occurring* over an *interval* $r[c,d]$ between the cuts c and d of a run r.

Our later analysis will be concerned with CCLs P. Thus we need to ensure that no message crosses any initial or final cut of an interval over which P occurs. A concise way of capturing this formally is via a new language construct, the *silent cut* , $\wr$. Writing $\wr$ specifies that all communication channels are empty at this cut. In other words, no statement to the left of the $\wr$ can communicate with a statement to the right. If P is a

CCL in a given larger program L then every execution of P in L is also an execution of $\wr P \wr$. In other words, P can be substituted for $\wr P \wr$ in L.[2] We adopt a standard notion of *refinement* to indicate substitutability of programs. Program P *refines* program Q if every execution of P over an interval $r[c,d]$ is also one of Q, regardless of what happens before c and after d. The notions of "$*$", "$\wr$", and refinement provide a unified language for defining notions of safe composition. The programming language and its semantics are formally defined as follows.

2.1. Syntax

Let $n \in \mathbb{N}$ and $\mathbb{P} = \{1,\ldots,n\}$ be a set of processes. Throughout the paper n will be reserved for denoting the number of processes. Let $(Var_i)_{i\in\mathbb{P}}$ be mutually disjoint sets of *program variables (of process i)* not containing the name h_i which is reserved for i's *communication history*. Let $Expr_i$ be the set of arithmetic expressions over Var_i. Let $\mathcal{L}$ be propositional logic over atoms formed from expressions with equality "$=$" and less-than "$<$". We define a syntactic category *Prg* of *programs*:

$$Prg \ni P \ ::= \ \varepsilon \mid x := \mathsf{e} \mid \textsc{snd}_{\mathsf{e}}^{i\to j} \mid \textsc{rcv}_{x}^{j\leftarrow i} \mid [\phi] \mid \wr \mid P * P \mid P + P \mid P^{\omega}$$

where $x \in Var_i$, $\mathsf{e} \in Expr_i$, $i, j \in \mathbb{P}$, and $\phi \in \mathcal{L}$.

The intuitive meaning of these constructs is as follows. The symbol ε denotes the *empty program*. It takes no time to execute. *Assignment statement* $x := \mathsf{e}$ evaluates expression e and assigns its value to variable x. The $\textsc{snd}_{\mathsf{e}}^{i\to j}$ statement sends a message containing the value of e on the channel from i to j. Communication is asynchronous, and sending is non-blocking. The $\textsc{rcv}_{x}^{j\leftarrow i}$ statement, however, blocks until a message arrives on the channel from i to j. It takes a message off the channel and assigns its content to x. The *guard* $[\phi]$ expresses a constraint on the execution of the program: in a run of the program, ϕ must hold at this location. Guards take no time to execute. The program $\wr$ is a guard-like constraint stating that all channels must be empty at this location. Formally, our propositional language $\mathcal{L}$ is not expressive enough to define $\wr$ as a guard because formulas are not capable of refering to channel contents. The operation "$*$" represents *layered composition* following Janssen et al. [Jan95]. Layering statements of distinct processes is essentially the same as parallel composition whereas layering of statements of the same process corresponds to sequential composition. We tend to omit "$*$" when no confusion will arise. The symbol "$+$" denotes nondeterministic choice. By P^{ω} we denote zero or more (possibly infinitely many) repetitions of program P.[3]

2.2. Semantics

A *send record (for i)* is a triple $(i \to j, v)$, which records sending a message with contents v from i to the receiver j. Similarly, $(j \leftarrow i, v)$ is a *receive record (for j)*. A *local state (for process i)* is a mapping from Var_i to values and from h_i to a sequence of send and receive records for i. A *local run (for process i)* is an infinite sequence of local states. We identify an *event (of i)* with the transition from one local state in a local run of i to the next. An event is either a *send*, a *receive*, or an *internal* event. A *(global) run* is a tuple $r = ((r_i)_{i\in\mathbb{P}}, \delta_r)$ of local runs — one for each process — plus an injective *matching function* δ_r associating a send event with each receive event in r. The mapping δ_r is restricted such that:[4]

[2] In place of the silent cut $\wr$ the preliminary version of this paper [EM05a] used a *phase quantifier* τ. Program τP roughly corresponds to our $\wr P \wr$.

[3] Using guards, choices, and repetition it is possible to define **if** ϕ **then** P **else** Q **fi** as an abbreviation for $[\phi]P + [\neg\phi]Q$ and **while** ϕ **do** P **od** for $([\phi]P)^{\omega}[\neg\phi]$. The results in this paper also hold for a language based on **if** and **while** instead of $[.]$, $+$, and $^{\omega}$.

[4] Our choice of execution model is closely related to the more standard one of infinite sequences of global states, representing an *interleaving* of moves by processes. Our conditions on δ_r guarantee the existence of such an interleaving. In general, each of our runs represents an equivalence class of interleavings.

1. If $\delta_r(e) = e'$ and e is a receive event of process j resulting in the appending of $(j \leftarrow i, v)$ to j's message history then e' is a send event of process i appending the corresponding send record $(i \rightarrow j, v)$ to i's message history.

2. Lamport's causality relation $\xrightarrow{\text{L}}$ induced by δ_r on the events of r, as defined below, is an irreflexive partial order, hence acyclic.

The first condition captures the property that messages are not corrupted in transit. The fact that the function δ_r is total precludes the reception of spurious messages, whereas injectivity ensures that messages are not duplicated in transit. Further restrictions on δ_r can be made to capture additional properties of the communication medium such as reliability, FIFO, fairness, etc.

We say that $r \in \textsc{Rel}$ if no unmatched send event is succeeded by infinitely many matched send events on the same channel.

In [Lam78] Lamport defined a "happened before" relation $\xrightarrow{\text{L}}$ on the set of events occurring in a run r of a distributed system. The relation $\xrightarrow{\text{L}}$ is defined as the smallest transitive relation subsuming (1) the total orders on the events of process i given by the r_i, and (2) the relation $\{ (e_1, e_2) \mid \delta_r(e_2) = e_1 \}$ between send and receive events induced by the matching function δ_r.

2.2.1. Cuts and Channels

Write $\mathbb{N}_+$ for $\mathbb{N} \cup \{\infty\}$. A *cut* is a pair (r, c) consisting of a run r and a $\mathbb{P}$-indexed family $c = (c_i)_{i \in \mathbb{P}}$ of $\mathbb{N}_+$-elements. We write "$\leq$" for the component-wise extension of the natural ordering on $\mathbb{N}_+$ to cuts within the same run. A cut is *finite* if all its components are.

Say that an event e performed by process i is *in* a cut (r, c) if e occurs in r_i at an index no larger than c_i, and e occurs *outside* of (r, c) if the index is larger than c_i. A cut (r, c) corresponds to the, possibly implausible, situation in which the events in the cut have occurred for each process $i \in \mathbb{P}$. We define the *channel* $\mathsf{chan}_{i \rightarrow j}$ at a cut (r, c) to be the set of i's send events to j and j's receive events from i in (r, c) that are not matched by δ_r to any event also in (r, c). Finally, a formula $\phi \in \mathcal{L}$ *holds at* (r, c), and we write $(r, c) \models \phi$, if ϕ holds in standard propositional logic when, for each $i \in \mathbb{P}$, program variables in Var_i are evaluated in the local states $r_i(c_i)$ if c_i is finite, and are considered unspecified otherwise.[5]

Observe that a cut can, in general, be fairly arbitrary. In particular, there is no requirement that all messages that are received before a cut is reached were sent before the cut. This is deliberate. There are, of course, many instances in which more structured cuts may be of interest. Indeed, we can define a cut (r, c) to be *consistent* if every $\xrightarrow{\text{L}}$ predecessor of an event in the cut (r, c) is also in the cut. Moreover, in this work we make use of a stronger property of cuts—that all channels are *empty* at the cut.

2.2.2. Semantics of Programs

We define the meaning of programs by stating when a program occurs over an interval. An *interval* consists of two cuts (r, c) and (r, d) over the same run with $c \leq d$, which we denote for simplicity by $r[c, d]$. An event is *in* $r[c, d]$ if it is in (r, d) but not in (r, c). We define the occurrence relation $\Vdash$ between intervals and programs by induction on the structure of programs. The interesting cases are those of $*$ and $\wr$. Formally, program $P \in Prg$ *occurs* over interval $r[c, d]$, denoted $r[c, d] \Vdash P$, iff:[6]

$r[c, d] \Vdash \varepsilon$ if $c = d$.

$r[c, d] \Vdash x := \mathsf{e}$ if $d = c[i \mapsto c_i + 1]$ and $r_i(d_i) = r_i(c_i)[x \mapsto v]$, where v is the value of e in $r_i(c_i)$.

[5] Recall that local states assign values to local variables.

[6] We shall denote by $f[a \mapsto b]$ the function that agrees with f on everything but a, and maps a to b.

$r[c,d] \Vdash \mathrm{SND}^{i\to j}_{\mathsf{e}}$ if $d = c[i \mapsto c_i + 1]$ and $r_i(d_i) = r_i(c_i)[h_i \mapsto r_i(c_i)(h_i) \cdot \langle (i \to j, v) \rangle]$, where v is the value of e in $r_i(c_i)$.

$r[c,d] \Vdash \mathrm{RCV}^{i\leftarrow j}_{x}$ if $d = c[i \mapsto c_i + 1]$ and $r_i(d_i) = r_i(c_i)[h_i \mapsto r_i(c_i)(h_i) \cdot \langle (i \leftarrow j, v) \rangle, x \mapsto v]$.

$r[c,d] \Vdash [\phi]$ if $c = d$ and $(r,c) \models \phi$.

$r[c,d] \Vdash \wr$ if $c = d$ and no communication event in (r,c) is matched by δ_r with an event outside (r,c).[7]

$r[c,d] \Vdash P * Q$ if there exists c' satisfying $c \le c' \le d$ such that $r[c,c'] \Vdash P$ and $r[c',d] \Vdash Q$.

$r[c,d] \Vdash P + Q$ if $r[c,d] \Vdash P$ or $r[c,d] \Vdash Q$.

$r[c,d] \Vdash P^{\omega}$ if, intuitively, an infinite or finite number (possibly zero) of iterations of P occur over $r[c,d]$. More formally, $r[c,d] \Vdash P^{\omega}$ if there exists a finite or infinite sequence $(c^{(k)})_{k\in I}$ such that I is a non-void prefix of $\mathbb{N}_+$, $c^{(0)} = c$, $c^{(k)} \le c^{(k')}$ for all $k < k' \in I$, $\bigsqcup_{k\in I} c^{(k)} = d$, and $r[c^{(k)}, c^{(k+1)}] \Vdash P$ for all $k, k+1 \in I$.

The program semantics is insensitive to deadlocks because deadlocking executions are not represented by runs. We deliberately chose to ignore deadlocks to simplify the presentation and focus on the main aspects of composition. Whether a program deadlocks can be analyzed using standard techniques [Lyn96, p. 635f].

General assumption. *From now onward, we shall only consider programs that are deadlock-free.*

2.2.3. Refinement

We shall capture various assumptions about properties of systems by specifying sets of runs. For instance, REL is the class of runs with reliable communication, and RELFI is its subclass in which channels are also FIFO.

Given a set Γ of runs, we say that P *refines* Q *in* Γ, denoted $P \le_{\Gamma} Q$, iff $r[c,d] \Vdash P$ implies $r[c,d] \Vdash Q$, for all $r \in \Gamma$ and $c,d \in (\mathbb{N}_+)^{\mathbb{P}}$. In other words, every execution of P (in a Γ run) is also one of Q, regardless of what happens before and after. Therefore, we may replace Q by P in any larger program context. This definition of refinement is thus appropriate for stepwise top-down development of programs from specifications. The refinement relation on programs is transitive (in fact a pre-order) and all programming constructs are monotone w.r.t. the refinement order.

3. Capturing Safe Composition

The silent cut program $\wr$ allows us to delineate the interactions that a layer can have with other parts of the program. When combined with refinement it is useful for defining various notions central to the study of safe composition, as we now illustrate.

CCL. We can express that the program L is a CCL in the program $P * L * Q$ w.r.t. Γ by:

$$\wr P * L * Q \wr \quad \le_{\Gamma} \quad P \wr L \wr Q \ .$$

In words, any isolated execution of $P * L * Q$ will have the property that all communication in L is internal and hence L executes as in isolation. This definition is context-sensitive.

[7] I.e., no receive in the cut (r,c) is mapped by δ_r to a send outside of the cut, and no receive from outside is mapped to a send in the cut.

Barriers. More modular would be a notion that guarantees safe composition regardless of the program context. One technique to ensure that two consecutive layers do not interfere with each other is to place a barrier B between them. Formally, program B is a *barrier* in Γ if

$$\wr P * B * Q \quad \leq_\Gamma \quad P \wr B \wr Q \text{ , for all } P, Q.$$

Traditionally, barriers have been used to synchronize the progression through phases by enforcing that no process could start its $n+1^{\mathrm{st}}$ task before all the other processes had completed their n^{th} tasks. This could be formilzed by requiring that, if $r[c,c'] \Vdash \wr P$, $r[c',d'] \Vdash B$, and $r[d',d] \Vdash Q$, then all events in (r,c') necessarily $\overset{L}{\rightarrow}$-precede all events not in (r,d'), for all runs $r \in \Gamma$, and programs P, Q.

TCC. Some programs can be safely composed without the need for communication-closedness [EF82, JZ92]. Depending on the model Γ, there may be programs P that safely compose with all following layers. We say that P *is tail communication closed (TCC)* in Γ if,

$$\wr P \quad \leq_\Gamma \quad P \wr \ .$$

Thus, if P is TCC then any execution of P starting in empty channels will also end with all channels empty. Therefore TCC programs can be readily composed.[8] It is straightforward to check that the programs ε, $[\phi]$, $x := \mathsf{e}$, and $P\wr$ are TCC in any Γ. Moreover, if P and Q are TCC in Γ then so are $P+Q$, $P*Q$, and P^ω.

Observe that every barrier B in Γ is in particular TCC in Γ.

Seals. In many models of interest, only trivial programs are TCC. This is the case, for example, in Rel, as shown in Section 4 below. In such models, an alternative methodology is required for determining when it is safe to compose given programs. Next we define a notion of *sealing* that formalizes the concept of program S serving as an impermeable layer between P and later phases such that no later communication will interact with P. We say that *S seals P in* Γ if,

$$\wr P * S \quad \leq_\Gamma \quad P \wr S \ .$$

Thus, if S seals P in Γ then neither S nor any later program can interfere with communication in P. If S seals P and Q seals S, then S will behave in $\wr P * S * Q$ as it does in isolation. Sealing allows incremental program development while maintaining CCL-style composition.

Lemma 1

1. If both P and P' are sealed by S in Γ then so is $P+P'$.
2. If both S and S' seal P in Γ then $S+S'$ (properly) seals P in Γ.
3. If S seals P in Γ then $S*Q$ seals P in Γ.
4. If both S seals P and S' seals S in Γ, then S' seals $P*S$ in Γ.
5. If P seals itself in Γ then P seals P^ω in Γ.
6. TCC subsumes sealing: P is TCC in Γ iff all programs seal P in Γ.

It follows from this lemma that, if program P can be decomposed into a sequence of ℓ layers $L^{(1)}, \ldots, L^{(\ell)}$, and in addition $L^{(k+1)}$ seals $L^{(k)}$ for all $1 \leq k < \ell$, then each layer $L^{(k)}$ is a CCL in P.

For example, as discussed in the introduction, any program of the form $\textsc{mt}^{j\to i}$ seals any program of the form $\textsc{mt}^{i\to j}$ in Rel. Consequently, a program of the form $\textsc{mt}^{i\to j} * \textsc{mt}^{j\to i}$ seals itself in Rel. On the other hand, the shorter program $\textsc{mt}^{i\to j}$ does not seal itself in Rel—in an execution of $\textsc{mt}^{i\to j} * \textsc{mt}^{i\to j}$ the two messages sent by i could be received in the reverse order of sending.

[8] TCC follows and is closely related to the notion of GTCC introduced by Gerth and Shrira [GS86]. The main difference is that their notion is defined w.r.t. a set of initial states.

Proper Seals. Suppose that $\mathbb{P} = \{1,2\}$ and $x_i \in \mathit{Var}_i$ for $i \in \mathbb{P}$. Then the program $Q = \textbf{while}\ \mathit{true}\ \textbf{do}\ (x_1 := 5 * x_2 := 17)\ \textbf{od}$ is TCC in RELFI, a CCL in REL, and seals any program in REL. For it necessarily *diverges*, that is, it occurs only over intervals $r[c,d]$ with non-finite d. This implies that no layer following Q has any impact on the semantics of the whole program. It follows trivially that no communication of a later layer can interfere with anything before. Programs such as Q are not particularly useful as seals, in contrast to ones that seal without diverging. This motivates the following definition. We say that S is a *proper seal* of P in Γ if S seals P and S never diverges after P. That is, for all $r \in \Gamma$ and c,d,d', whenever $r[c,d] \Vdash \wr P$, and $r[d,d'] \Vdash \wr S$ and d is finite then so is d'.

For instance, since $\textsc{mt}^{i\to j}$ is a terminating program that seals $\textsc{mt}^{j\to i}$ in REL, it is in particular a proper seal.

4. Case Study: Safe Composition in REL

We now consider safe composition in the model REL. Communication events can cause a program *not* to be TCC in REL. For example, reconsider the program $\textsc{mt}^{i\to j}_{\mathsf{e}\to x} = \textsc{snd}^{i\to j}_{\mathsf{e}} * \textsc{rcv}^{j\leftarrow i}_{x}$. It is TCC in RELFI but not TCC in REL. That $\textsc{mt}^{i\to j}$ is not TCC in REL is no coincidence. Next we show that no terminating program performing any communication whatsoever is TCC in REL.

Theorem 2 *If $r[c,d] \Vdash P$ for some $r \in$ REL and finite c,d such that all channels are empty in (r,c) and there is at least one send or receive event in $r[c,d]$, then P is not TCC in* REL.

Proof: Assume that $r[c,d] \Vdash P$ where $r \in$ REL, c,d are finite, all channels are empty at (r,c) and there is a send or receive event in $r[c,d]$. If there is a non-empty channel in (r,d) the claim is immediate since a matching communication event following P could interact with P. Otherwise, every message sent in $r[c,d]$ is received in $r[c,d]$. Since P is deadlock-free by the general assumption, there are processes whose last communication event in $r[c,d]$ is a receive. W.l.o.g. let i be such a process and assume that its last receive is of a message v sent by j into variable $x \in \mathit{Var}_i$.

A run $r' \in$ REL that equals r up to d can be constructed such that $r'[d,d'] \Vdash \wr \textsc{snd}^{j\to i}_{\mathsf{e}} * \textsc{rcv}^{i\leftarrow j}_{x} \wr$, where e evaluates to v in $r_j(d_j)$. So the same message is transmitted twice between j and i. Let $r'' \in$ REL be the same as r', except for δ_r, which swaps the matching send events between the two receive events. For $Q = \textsc{snd}^{j\to i}_{\mathsf{e}} * \textsc{rcv}^{i\leftarrow j}_{x}$ it follows that $r''[c,d'] \Vdash \wr P * Q \wr$ but $r''[c,d'] \nVdash \wr P \wr Q \wr$. The claim follows. ■

Since a barrier is necessarily TCC we immediately obtain

Corollary 3 No program can serve as a barrier in REL.

Having shown that TCC and thus barriers are not generally useful notions in REL, we turn our attention to (proper) sealing. It is instructive that not all terminating programs can be properly sealed in REL:

Lemma 4 If $\mathbb{P} = \{1,2\}$ then the program $X = \textsc{snd}^{1\to 2}_{x+1} * \textsc{snd}^{2\to 1}_{y+1} * \textsc{rcv}^{1\leftarrow 2}_{x} * \textsc{rcv}^{2\leftarrow 1}_{y}$ illustrated in Fig. 3(a) cannot be sealed properly in REL.

Proof: Assume, by way of contradiction, that S properly seals X in REL. Consider a run $r \in$ REL such that $r[(0,0),(2,2)] \Vdash X$ and $r[(2,2),d'] \Vdash S$ where d' is finite. If some process $i \in \mathbb{P}$ does not engage in any communication event in $r[(2,2),d']$ then S does not seal X since a send by process i performed at d'_i potentially interacts with X. Otherwise, let e_i be the first communication events of each process $i = 1,2$ in $r[(2,2),d']$. If one of the e_i is a send then, as before, this send can interact with X. Finally, if both e_i are receives then S causes a deadlock, contradicting the assumption that $r[(2,2),d'] \Vdash S$. ■

Our programming language *Prg* is Turing-complete. Since the halting problem for *Prg* can be reduced to sealability in $\textsc{Rel}$ we obtain

Theorem 5 *Sealability in* $\textsc{Rel}$ *is undecidable.*

Given this theorem we shall restrict our attention to more tractable subclasses of programs. Program P is *balanced (in* $\textsc{Rel}$*)* if, whenever $r[c,d] \Vdash P$ and all channels are empty at (r,c), then every channel contains an equal number of sends and receives at (r,d). Note that balanced programs are TCC in $\textsc{RelFi}$. The following theorem shows that in $\textsc{Rel}$ balance is a necessary prerequisite for being properly sealable.

Theorem 6 *In* $\textsc{Rel}$, *every non-divergent program that is properly sealable is also balanced.*

Proof: Let P and S be programs such that in $\textsc{Rel}$ P does not diverge and S properly seals P. Assume by way of contradiction that P is not balanced. Let $r \in \textsc{Rel}$ and c, c', d be such that $r[c,c'] \Vdash P$, $r[c',d] \Vdash S$, all channels are empty in (r,c), and, w.l.o.g., $\mathsf{chan}_{i\to j}$ contains k sends and m receives at (r,c') where $k \neq m$. Since S is a proper seal, there is neither a send nor a receive event in $r[c,c']$ matched with an event not in $r[c,c']$. Since every receive event must be matched to some event by δ_r it follows that $k > m$, that is, there are more sends than receives on $\mathsf{chan}_{i\to j}$ in $r[c,c']$. No receive in the seal can be matched to any of those sends. There exist $r' \in \textsc{Rel}$, $y \in Var_j$, and d' such that r' is the same as r up to d (hence $r'[c,d] \Vdash P * S$), $r'[d,d'] \Vdash \textsc{rcv}_j^{j\leftarrow i}$, and δ_r maps the receive event $r_j(d_j)$ to one of the send events of P that are unmatched in r. This match contradicts the assumption that S properly seals P. ■

Program P is said to *close* $\mathsf{chan}_{i\to j}$ *(in* $\textsc{Rel}$*)* if $\mathsf{chan}_{i\to j}$ is empty after P in any execution of P starting at a cut with empty channels. More formally this is expressed as follows. For all $r \in \textsc{Rel}$, if $r[c,d] \Vdash \wr P$ then $\mathsf{chan}_{i\to j}$ is empty in (r,d). A channel that is not closed is *open*. The state of a program's channels is the essential element in determining sealability.

Program P is *straight-line* if it contains neither nondeterministic choices nor loops nor guards. In other words, P is built from sends, receives, and assignments using layering only. Our focus in this section is on balanced straight-line programs, or *BSL* for short.

The *program graph* of a BSL P is a graph (V,E) that has a node for every send and receive event in P plus an initial dummy node $\textsc{fst}_i$ and a final dummy node $\textsc{lst}_i$ for each process i. The edge set E consists of the successor relation over events in the same process extended to the dummy nodes plus an edge between the k'th send and the k'th receive on channel $\mathsf{chan}_{i\to j}$, for all k, i, and j. All the graphs in Figures 2 and 3 are program graphs. The size of a BSL P's program graph is of the order of the size of the program.

Next we investigate the connection between program graphs and Lamport causality. We use E^+ to refer to the irreflexive transitive closure of E and call edges not containing dummy nodes *normal*. The subset of normal edges is denoted by N_E. In $\textsc{RelFi}$, the normal edges induce the full causality relation on the events of the program. As we shall show, in $\textsc{Rel}$ the normal edges of a program graph are also $\xrightarrow{\text{L}}$ edges.

Lemma 7 Let $r \in \textsc{Rel}$. Let P be a BSL with program graph (V,E). If $r[c,d] \Vdash \wr P$ then $N_E \subseteq \xrightarrow{\text{L}}$.

Proof: The only interesting normal edges are those between sends and receives of different processes. Consider the edge $(e_1, e_2) \in E$ between the k'th send and the k'th receive on $\mathsf{chan}_{i\to j}$. Let $r \in \textsc{Rel}$ such that $r[c,d] \Vdash \wr P$ and assume that $e_3 = \delta_r(e_2)$ is the ℓ'th send on $\mathsf{chan}_{i\to j}$ in P. We need to show that $e_1 \xrightarrow{\text{L}} e_2$. By definition of $\xrightarrow{\text{L}}$, we have that $e_3 \xrightarrow{\text{L}} e_2$. If $\ell = k$ then $e_3 = e_1$ and we are done. If $\ell > k$ then $e_1 \xrightarrow{\text{L}} e_3$ because e_1 is an earlier event of i than e_3 and the claim follows by transitivity of $\xrightarrow{\text{L}}$. Finally, suppose that $\ell < k$. This case is illustrated in Fig. 4. Consider the $k-1$ receives on $\mathsf{chan}_{i\to j}$ that precede e_2. They are all matched in r to sends by i. Since $\ell < k$ and e_3 is already matched to e_2, one of these receives, say e_4, must be matched to a send event e_5 that does not precede e_1. Since $e_5 \xrightarrow{\text{L}} e_4$ and $e_4 \xrightarrow{\text{L}} e_2$, it follows that $e_1 \xrightarrow{\text{L}} e_2$, as desired. ■

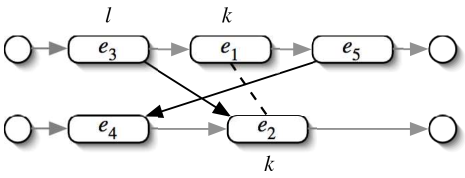


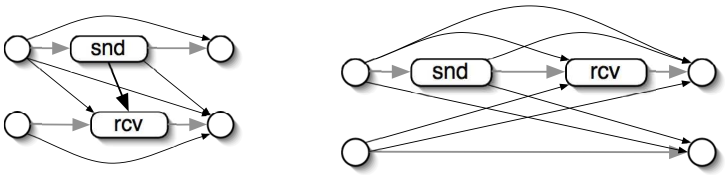
Figure 4: The case $\ell < k$ in the proof of Lemma 7.

Lemma 7 implies that all edges in $(N_E)^+$ will be $\xrightarrow{\mathsf{L}}$ edges in every run $r \in \textsc{Rel}$ of $\wr P * Q$. We note that $(N_E)^+$ is the largest set of edges with this property, because $(N_E)^+ = (\xrightarrow{\mathsf{L}} \cap V^2)$ if $r \in \textsc{RelFi}$.

A more concise representation than the program graph is called the *signature of P* and denoted by $\textsc{Sig}(P)$. It has size $O(n^2)$ while preserving the information necessary to decide what channels are left open, respectively closed, by P. Given the program graph (V,E) of a BSL P we can obtain $\textsc{Sig}(P)$ as follows. After calculating E^+, we remove all nodes except for the dummy nodes and the first send and last receive on each channel. The graph is further reduced by removing the node $\textsc{snd}^{i \to j}$ whenever $(\textsc{fst}_j, \textsc{snd}^{i \to j}) \in E^+$. Similarly, $\textsc{rcv}^{j \leftarrow i}$ is removed whenever $(\textsc{rcv}^{j \leftarrow i}, \textsc{lst}_i) \in E^+$. The sends and receives remaining in the signature are precisely the ones that could interfere with receives in a preceding layer or with sends in a succeeding layer.

The complexity of computing $\textsc{Sig}(P)$ is in $O(\|P\|^3)$ since it requires the causality relation obtained as the transitive closure of the edge relation of P's program graph. We remark that for BSLs P and Q, $\textsc{Sig}(P * Q)$ can be obtained from their respective signatures at a cost of $O(n^2)$.

Let P be a BSL and let $G = (V,E)$ be $\textsc{Sig}(P)$. Then P leaves channel $\mathsf{chan}_{i \to j}$ open iff $\textsc{rcv}^{j \leftarrow i} \in V$. For instance, the program $\textsc{mt}^{i \to j}$ leaves $\mathsf{chan}_{i \to j}$ open — there is a node $\textsc{rcv}^{j \leftarrow i}$ in $\textsc{Sig}(\textsc{mt}^{i \to j})$, which is depicted in Fig. 5(a). As we have shown earlier, $\textsc{mt}^{j \to i}$ seals $\textsc{mt}^{i \to j}$ in $\textsc{Rel}$, which implies that $\textsc{mt}^{j \to i}$ closes $\mathsf{chan}_{i \to j}$ once. Since $\textsc{mt}^{j \to i}$ does not re-open the channel, the $\textsc{rcv}^{j \leftarrow i}$ node found in $\textsc{Sig}(\textsc{mt}^{i \to j})$ is not present in the $\textsc{Sig}(\textsc{mt}^{i \to j} * \textsc{mt}^{j \to i})$ shown in Fig. 5(b).



(a) $\textsc{Sig}(\textsc{mt}^{i \to j})$

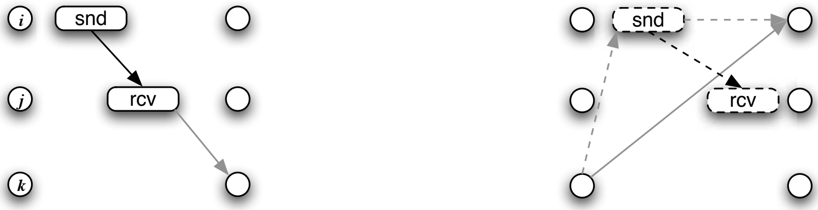
(b) $\textsc{Sig}(\textsc{mt}^{i \to j} * \textsc{mt}^{j \to i})$

Figure 5: Examples of signatures. Thin arrows denote transitive causality edges.

4.1. Deciding Sealing

Whether one BSL seals another can be decided on the basis of their signatures. Suppose BSL P leaves $\mathsf{chan}_{i \to j}$ open and Q seals P. Then, if Q sends on that channel, then P's last receive $\textsc{rcv}^{j \leftarrow i}$ on the channel must causally precede Q's first send $\textsc{snd}^{i \to j}$ on it. Otherwise, Q must ensure that any later send on $\mathsf{chan}_{i \to j}$ is causally preceded by P's last receive. This is guaranteed exactly if P's signature contains an edge $(\textsc{rcv}^{j \leftarrow i}, \textsc{lst}_k)$ and Q's signature contains an edge $(\textsc{fst}_k, \textsc{lst}_i)$, for some $k \in \mathbb{P}$. (See Fig. 6.)

Based on the above observation the following theorem characterizes sealing among BSLs.



(a) Channel $\mathsf{chan}_{i\to j}$ left open by P and a causality edge to $\textsc{lst}_k$.

(b) Sealing the channel by causality. The dashed part accounts for Q sending on $\mathsf{chan}_{i\to j}$.

Figure 6: Excerpts of the signatures of BSLs P and Q.

Theorem 8 *Let P and Q be BSLs and let $(V_P, E_P) = \textsc{Sig}(P)$ and $(V_Q, E_Q) = \textsc{Sig}(Q)$. Then Q properly seals P iff, for all $\textsc{rcv}^{j\leftarrow i} \in V_P$, there exists $k \in \mathbb{P}$ such that $(\textsc{rcv}^{j\leftarrow i}, \textsc{lst}_k) \in E_P$, $(\textsc{fst}_k, \textsc{lst}_i) \in E_Q$, and, if $\textsc{snd}^{i\to j} \in V_Q$ then $(\textsc{fst}_k, \textsc{snd}^{i\to j}) \in E_Q$.*

Proof: "$\Leftarrow$" Consider the channel $\mathsf{chan}_{i\to j}$. By construction, there is a node $\textsc{rcv}^{j\leftarrow i} \in V_P$ precisely if the channel is not closed by P. Suppose that $(\textsc{rcv}^{j\leftarrow i}, \textsc{lst}_k) \in E_P$ and $(\textsc{fst}_k, e) \in E_Q$ where $e = \textsc{snd}^{i\to j}$ if $\textsc{snd}^{i\to j} \in V_Q$ and $e = \textsc{lst}_i$ otherwise. Let $r \in \textsc{Rel}$ and c, d be such that $r[c,d] \Vdash \wr P * Q$. Let c' be such that $r[c,c'] \Vdash P$ and $r[c',d] \Vdash Q$. Let e_{R} in $r[c,c']$ be a $\textsc{rcv}^{j\leftarrow i}$ event. We shall prove that $e_{\mathrm{S}} = \delta_r(e_{\mathrm{R}})$ is also in $r[c,c']$. By definition of $\xrightarrow{\text{L}}$ we have that $e_{\mathrm{S}} \xrightarrow{\text{L}} e_{\mathrm{R}}$. First observe that e_{S} cannot be in (r,c) since $r[c,d] \Vdash \wr P * Q$ implies that δ_r cannot map e_{R} to an event in (r,c). Second, e_{S} cannot come after (r,c') because, as we shall show, that would imply $e_{\mathrm{R}} \xrightarrow{\text{L}} e_{\mathrm{S}}$. By transitivity, we would obtain $e_{\mathrm{R}} \xrightarrow{\text{L}} e_{\mathrm{R}}$, contradicting the irreflexivity of $\xrightarrow{\text{L}}$. Assume by way of contradiction that e_{S} is not in (r,c'). If e_{S} is in $r[c',d]$, that is, generated by Q, then $\textsc{snd}^{i\to j} \in V_Q$ represents a send event e'_{S}. This event is causally preceded by e_{R} because $(e_{\mathrm{R}}, \textsc{lst}_k) \in E_P$, $(\textsc{fst}_k, \textsc{snd}^{i\to j}) \in E_Q$, and $e_{\mathrm{S}} = e'_{\mathrm{S}}$ or $e'_{\mathrm{S}} \xrightarrow{\text{L}} e_{\mathrm{S}}$. Otherwise, that is, if e_{S} is not in (r,d), it is causally preceded by e_{R} because $(e_{\mathrm{R}}, \textsc{lst}_k) \in E_P$, $(\textsc{fst}_k, \textsc{lst}_i) \in E_Q$, and e_{S} is causally preceded by the last event of process i in $r[c,d]$. In either case, $e_{\mathrm{R}} \xrightarrow{\text{L}} e_{\mathrm{S}}$ follows by transitivity.

By now we have shown that δ_r does not match any receive in $r[c,c']$ to a send event not in $r[c,c']$. Because P is balanced this implies that all send events in $r[c,c']$ (i.e., the ones generated by P) must be matched with receive events in that interval. Thus, also $r[c,d] \Vdash \wr P \wr Q$.

"$\Rightarrow$" Suppose that $\textsc{rcv}^{j\leftarrow i} \in V_P$ and that there is no k such that $(\textsc{rcv}^{j\leftarrow i}, \textsc{fst}_k) \in E_P$ and $(\textsc{fst}_k, e) \in E_Q$ where $e = \textsc{snd}^{i\to j}$ if $\textsc{snd}^{i\to j} \in V_Q$ and $e = \textsc{lst}_i$ otherwise. We show that Q does not properly seal P. First consider the case $e = \textsc{snd}^{i\to j}$. For lack of a causal relationship between $\textsc{rcv}^{j\leftarrow i} \in V_P$ and $\textsc{snd}^{i\to j} \in V_Q$ they can be matched in an interval $r[c,d]$ over which $\wr P * Q \wr$ occurs, violating the sealing property. Finally consider the remaining case, $e = \textsc{lst}_i$. Again, for lack of a causal relationship between $\textsc{rcv}^{j\leftarrow i} \in V_P$ and $\textsc{lst}_i \in V_Q$, a subsequent send event can be matched with $\textsc{rcv}^{j\leftarrow i} \in V_P$, that is, there exist $r \in \textsc{Rel}$ and c, d such that $r[c,d] \Vdash \wr P * Q * \textsc{snd}_{53}^{i\to j} \wr$ and δ_r matches the last receive on channel $\mathsf{chan}_{i\to j}$ in P with the $\textsc{snd}_{53}^{i\to j}$. ■

Given the theorem above, the complexity of deciding whether Q seals P, given their signatures, is obviously determined by the size of P's signature, which we recall is $O(n^2)$.

4.2. A Characterization of Sealability

Observe that the set of channels closed by a BSL P when executed from a cut with empty channels is uniquely determined by P and can be derived from its signature. We can thus associate a *closed-channel* *graph* with each BSL . Formally, the closed-channel graph $C_P = (\mathbb{P}, E_P)$ of a BSL P is given by $(i, j) \in E_P$ iff $i \neq j$ and $\mathsf{chan}_{i \to j}$ is closed by P in REL. In the following we denote the undirected version of a graph G by G^{u}.

Theorem 9 (Sealability) *Let P be a BSL. Then P can be sealed properly in* REL *iff C_P^{u} is connected. Moreover, if P is properly sealable in* REL *then it can be sealed by a BSL that transmits less than $3n$ messages.*

Proof: "$\Rightarrow$" Suppose that C_P^{u} is not connected. Then $\mathbb{P}$ can be partitioned into two non-void sets, A and $\overline{A}$, such that there is no channel closed by P between (elements of) the two sets. Assume, by way of contradiction, that the program S properly seals P. Since S is a seal, every message sent in S along a channel not closed by P must be causally preceded by all receives on that channel in P. This holds in particular for all channels between A and $\overline{A}$. There must be such receives in P for each of the channels not closed by P. To establish the causal precedences, S must transmit messages. Unless S transmits messages between A and $\overline{A}$, it cannot seal P. Consider one of the causally minimal sends of such a transmission in S. It can interfere with the last receive on that channel in P. Consequently, S does not seal P.

"$\Leftarrow$" The algorithm sketched as SEAL(P) takes a BSL P as input and outputs a proper seal for P if P is properly sealable.

```
SEAL$(P)$
1  $(V, E) \leftarrow$ CLOSED-CHANNELS$(P)$    $\triangleright$ This algorithm is presented in Appendix A.
2  $S \leftarrow \varepsilon$
3  pick $T \subseteq E$ s.t. $(\mathbb{P}, T)^{\mathrm{u}}$ forms a spanning tree of $V$
4  $v \leftarrow$ a node at the center of $T$
5  for $(w, w') \in T$ pointing away from $v$ s.t. $(w', w) \notin E$
6      do $S \leftarrow S * \mathrm{MT}^{w \to w'}$
7  add a converge-cast in $T$ to $S$
8  add a broadcast in $T$ to $S$
```

Let S be the result of SEAL(P). It consists of less than $3n$ instances of MT because every spanning tree contains $n-1$ edges and each of the three sub-phases, (a) lines 5–6, (b) the converge-cast, and (c) the broadcast transmits less than n messages. Each one of these MT instances transmits a message along a channel that is closed at the time of transmission. For phase (a) this follows from the selection criterion for these transmission in line 5. Phase (a) establishes that all channels between a node and its parent in the spanning tree are closed, thus phase (b) transmits on closed channels only. Similarly, phase (b) closes all channels between nodes and their children in the spanning tree, hence also phase (c) transmits on closed channels only. Finally, we need to show that every channel left open by P is closed at least once by S. Let (i, j) be such that P leaves $\mathsf{chan}_{i \to j}$ open. If $(i, j) \in T^{-1}$ then phase (a) closes the channel by sending on $\mathsf{chan}_{j \to i}$. Otherwise it is closed transitively by the subsequence of the converge-cast from j to the root v followed by the subsequence of the broadcast from v to i. ■

Observe that SEAL(P) constructs a tailor-made tree barrier S between P and any later program.

Example 10 Consider a phase $L = *_{i \in \mathbb{P}} L_i$. In L each process $i \neq 1$ sends a message to every other process $k \notin \{1, i\}$ before receiving the $n-2$ messages sent to it in this phase. Finally, process i transmits a message to process 1. We can define process i's program L_i more formally by

$$L_i \quad = \quad \left(*_{k \notin \{1,i\}} \mathrm{SND}^{i \to k} \right) * \left(*_{k \notin \{1,i\}} \mathrm{RCV}^{i \leftarrow k} \right) * \mathrm{SND}^{i \to 1} \quad .$$

Process 1 in turn receives those messages sent last in the L_i, that is:

$$L_1 \quad = \quad *_{i\neq 1}\,\mathrm{RCV}^{1\leftarrow i}$$

Executing L beginning with empty channels leaves n^2-3n+3 channels open. Nevertheless, L can be sealed efficiently by the program

$$S \quad = \quad *_{i\neq 1}(\mathrm{SND}^{1\rightarrow i} * \mathrm{RCV}^{i\leftarrow 1}) \ ,$$

which transmits $n-1$ messages. (See Fig. 7 for the program graph of S.)

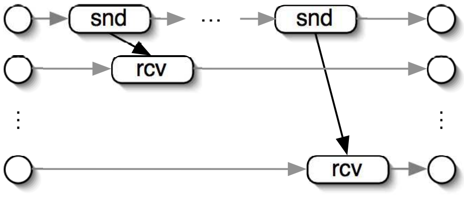


Figure 7: $O(n)$ transmissions close $\Omega(n^2)$ open channels.

5. Conclusion and Future Work

A subtle yet crucial issue in developing distributed applications is the safe composition of smaller programs into larger ones. The notion of CCL captures when a program works as if it were executed in isolation in the context of a given larger program. The literature on CCLs focused mostly on reliable FIFO communication. In that setting programs can be designed that are inherently CCLs in any program context.

Observe that neither termination detection nor barrier-style techniques can be applied in REL without careful inspection of the surrounding program context. Any such mechanism will form a layer in the resulting program which in turn must be shown to safely compose with the other layers. A popular approach to running distributed applications on non-RELFI systems is to construct an intermediate data-link layer providing RELFI communication to the application. This typically involves sealing every single message transmission from interference by previous and later layers. Popular algorithms for data-link achieve this by adding message headers and/or acknowledging every single message, thereby incurring a significant overhead [AAF$^+$94, WZ89]. As we show for REL, it is often possible to do better than that. Our analysis of sealing can be used to add the minimal amount of glue between consecutive layers to ensure that they compose safely, without changing the layers at all.

We have introduced a framework for studying safe program composition. It facilitates the formal definition of standard notions such as CCL, barriers, and TCC. Gerth and Shrira showed that—as a context-sensitive notion—CCL is unsuitable for compositional development of larger systems from off-the-shelf components. As we have shown, neither barriers nor TCC layers are useful for such development in REL, that is, when communication is reliable but not FIFO. In another paper [EM05b], we use essentially the same framework to investigate safe composition in models with message duplication or loss. Barriers and TCC layers are also absent in those models. The framework introduced here is used to define two more notions, namely fitting after and separating, that are more readily applicable in those models.[9] We illustrate our approach by applying it to the case of REL. Notably, the approach allows for seamless composition of programs without need for translation or headers.

[9] We say that P *fits after* Q if $\wr QP \leq_\Gamma \wr Q \wr P$. Program S *separates* P *from* Q if $\wr P * S * Q \wr \leq_\Gamma P \wr S \wr Q$.

The central notion introduced and explored in this paper is that of one program *sealing* another. Larger programs can be composed from smaller ones provided each smaller program seals its predecessor. For instance, recall that $\textsc{mt}^{i\to j} * \textsc{mt}^{j\to i}$ seals itself in $\textsc{Rel}$. Lemma 1.5 can be used to show that a program of the form **while** *true* **do** $\textsc{mt}^{i\to j} * \textsc{mt}^{j\to i}$ **od** can serve to transmit a sequence of values from i to j in $\textsc{Rel}$. Indeed, if the return messages from j to i are not merely acknowledgments, it can perform sequence exchange. The notion of sealing in $\textsc{Rel}$ is shown to be intimately related to Lamport causality. Based on this connection, we devise efficient algorithms for deciding and constructing seals for the class of straight-line programs.

Acknowledgment

We would like to thank Manuel Chakravarty, Yael Moses, and Ron van der Meyden for helpful comments on preliminary versions of this paper. Special thanks to Elena Blank for an observation that led to a simplification of the notation.

References

[AAF+94] Yehuda Afek, Hagit Attiya, Alan Fekete, Michael Fischer, Nancy A. Lynch, Yishay Mansour, Dai-Wei Wang, and Lenore Zuck. Reliable communication over unreliable channels. *Journal of the ACM*, 41(6):1267–1297, 1994.

[EF82] Tzilla Elrad and Nissim Francez. Decomposition of distributed programs into communication-closed layers. *Science of Computer Programming*, 2(3):155–173, December 1982.

[EM05a] Kai Engelhardt and Yoram Moses. Causing communication closure: Safe program composition with non-FIFO channels. In Pierre Fraigniaud, editor, *DISC 2005 19th International Symposium on Distributed Computing*, volume 3724 of *LNCS*, pages 229–243. Springer-Verlag, September 26–29 2005.

[EM05b] Kai Engelhardt and Yoram Moses. Safe composition of distributed programs communicating over order-preserving imperfect channels. In Ajit Pal, Ajay Kshemkalyani, Rajeev Kumar, and Arobinda Gupta, editors, *7th International Workshop on Distributed Computing IWDC 2005*, volume 3741 of *LNCS*, pages 32–44. Springer-Verlag, December 27–30 2005.

[EM05c] Kai Engelhardt and Yoram Moses. Single-bit messages are insufficient in the presence of duplication. In Ajit Pal, Ajay Kshemkalyani, Rajeev Kumar, and Arobinda Gupta, editors, *7th International Workshop on Distributed Computing IWDC 2005*, volume 3741 of *LNCS*, pages 25–31. Springer-Verlag, December 27–30 2005.

[FL90] Alan Fekete and Nancy A. Lynch. The need for headers: An impossibility result for communication over unreliable channels. In Jos C. M. Baeten and Jan Willem Klop, editors, *CONCUR '90: Theories of Concurrency: Unification and Extension*, volume 458 of *LNCS*, pages 199–215. Springer-Verlag, 1990.

[GS86] Rob Gerth and Liuba Shrira. On proving communication closedness of distributed layers. In Kesav V. Nori, editor, *Foundations of Software Technology and Theoretical Computer Science, Sixth Conference*, volume 241 of *LNCS*, pages 330–343, New Delhi, India, 18–20 December 1986. Springer-Verlag.

[Jan94] Wil Janssen. *Layered Design of Parallel Systems*. PhD thesis, University of Twente, 1994.

[Jan95] Wil Janssen. Layers as knowledge transitions in the design of distributed systems. In Uffe H. Engberg, Kim G. Larsen, and Arne Skou, editors, *Proceedings of the Workshop on Tools and Algorithms for the Construction and Analysis of Systems, TACAS* (Aarhus, Denmark, 19–20 May, 1995), number NS-95-2 in Notes Series, pages 304–318, Department of Computer Science, University of Aarhus, May 1995. BRICS.

[JPZ91] Wil Janssen, Mannes Poel, and Job Zwiers. Action systems and action refinement in the development of parallel systems. In Jos C. M. Baeten and Jan Frisco Groote, editors, *Proceedings of CONCUR '91, 2nd International Conference on Concurrency Theory, Amsterdam, The Netherlands*, volume 527 of *LNCS*, pages 298–316, 1991.

[JZ92] Wil Janssen and Job Zwiers. From sequential layers to distributed processes, deriving a minimum weight spanning tree algorithm, (extended abstract). In *Proceedings 11th ACM Symposium on Principles of Distributed Computing*, pages 215–227. ACM, 1992.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 7:558–565, 1978.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[PZ92] Mannes Poel and Job Zwiers. Layering techniques for development of parallel systems. In Gregor von Bochmann and David K. Probst, editors, *Computer Aided Verification, Fourth International Workshop, CAV '92*, volume 663 of *LNCS*, pages 16–29, Montreal, Canada, June 29 – July 1 1992. Springer-Verlag.

[SdR94] Frank A. Stomp and Willem-Paul de Roever. A principle for sequential reasoning about distributed algorithms. *Formal Aspects of Computing*, 6(6):716–737, 1994.

[WZ89] Da-Wei Wang and Lenore D. Zuck. Tight bounds for the sequence transmission problem. In *PODC '89: Proceedings of the eighth annual ACM Symposium on Principles of Distributed Computing*, pages 73–83. ACM Press, 1989.

A. Algorithms

```
PROGRAM-GRAPH($P$)
$V, E \leftarrow \bigcup_{i \in \mathbb{P}} \{\textsc{fst}_i, \textsc{lst}_i\}, \emptyset$      ▷ First and last dummy nodes for each process
$f \leftarrow \lambda i : \mathbb{P}.\textsc{fst}_i$      ▷ Book keeping for local precedence
▷ Add sends and receives with local precedence
for $e$ in $P$ from left to right where $e$ is of the form $\textsc{snd}^{i \to j}$ or $\textsc{rcv}^{i \leftarrow j}$
    do $V, E, f(i) \leftarrow V \cup \{e\}, E \cup \{(f(i), e)\}, e$
▷ Add precedence between last $i$-event and $i$'s last dummy node
for $i \in \mathbb{P}$
    do $E \leftarrow E \cup \{(f(i), \textsc{lst}_i)\}$
▷ Add precedence between FIFO matching sends and receives
for $e \in V$ the $k$'th event in $P$ of the form $\textsc{snd}^{i \to j}$ for some $i, j, k$
    do $E \leftarrow E \cup \{(e, e')\}$ where $e'$ is the $k$'th $\textsc{rcv}^{i \leftarrow j}$ event in $P$
return $(V, E)$
```

```
$\textsc{Deadlock-Free}(P)$
$V, E \leftarrow \textsc{Program-Graph}(P)$
return $\exists$ cycle in $E$
```

```
$\textsc{Sig}(P)$
$V, E \leftarrow \textsc{Program-Graph}(P)$
$E \leftarrow E^{+}$                                  $\triangleright$ Add irreflexive transitive closure
$\triangleright$ Remove all but minimal sends and maximal receives on open channels
$V \leftarrow V \setminus \{\, e \mid e$ is a $\textsc{snd}^{i \to j}$ event preceded by another such send or $\textsc{fst}_j \,\}$
$V \leftarrow V \setminus \{\, e \mid e$ is a $\textsc{rcv}^{j \leftarrow i}$ event that precedes another such receive or $\textsc{lst}_i \,\}$
return $(V, E \cap V^2)$
```

```
$\textsc{Is-Seal}(P, Q)$
$V_P, E_Q \leftarrow \textsc{Sig}(P)$
$V_Q, E_Q \leftarrow \textsc{Sig}(Q)$
for $(i, j) \in \mathbb{P}^2 \setminus \mathrm{id}_{\mathbb{P}}$ s.t. $\textsc{rcv}^{j \leftarrow i} \in V_P$
    do $e \leftarrow \begin{cases} \textsc{snd}^{i \to j} & \text{if } \textsc{snd}^{i \to j} \in V_Q \\ \textsc{lst}_i & \text{otherwise} \end{cases}$
       $\mathit{safe} \leftarrow \mathit{false}$
       for $k \in \mathbb{P} \setminus \{i\}$
           do $\mathit{safe} \leftarrow \mathit{safe} \vee ((\textsc{rcv}^{j \leftarrow i}, \textsc{lst}_k) \in E_P \wedge (\textsc{fst}_k, \textsc{snd}^{i \to j}) \in E_Q)$
       if $\neg\, \mathit{safe}$
         then return $\mathit{false}$
return $\mathit{true}$
```

```
$\textsc{Closed-Channels}(P)$
$V, E \leftarrow \mathbb{P}, \mathbb{P}^2 \setminus \mathrm{id}_{\mathbb{P}}$
$V', E' \leftarrow \textsc{Sig}(P)$
for $(i, j) \in E$
    do if $\textsc{rcv}^{j \leftarrow i} \in V'$ and $(\textsc{rcv}^{j \leftarrow i}, \textsc{lst}_i) \notin E'$
          then $E \leftarrow E \setminus \{(i, j)\}$
return $(V, E)$
```

```
$\textsc{Signature-Compose}(V_P, E_P, V_Q, E_Q)$
$\triangleright$ Sequentially compose the two signatures
$V \leftarrow \{\, e^{(X)} \mid e \in V_X \wedge X \in \{P, Q\} \,\}$
$E \leftarrow \{\, (e^{(X)}, f^{(Y)}) \in V^2 \mid X = Y \wedge (e, f) \in E_X \,\} \cup \left\{\, (\textsc{lst}_i^{(P)}, \textsc{fst}_i^{(Q)}) \;\middle|\; i \in \mathbb{P} \,\right\}$
$E \leftarrow E^{+}$
$\triangleright$ Remove dummy nodes between the two signatures
$V \leftarrow V \setminus \left\{\, \textsc{lst}_i^{(P)} \;\middle|\; i \in \mathbb{P} \,\right\} \setminus \left\{\, \textsc{fst}_i^{(Q)} \;\middle|\; i \in \mathbb{P} \,\right\}$
$\triangleright$ Remove all but the first sends and last receives
$V \leftarrow V \setminus \{\, e^{(Q)} \in V \mid e^{(P)} \in V \wedge e = \textsc{snd}^{i \to j} \,\} \setminus \{\, e^{(P)} \in V \mid e^{(Q)} \in V \wedge e = \textsc{rcv}^{j \leftarrow i} \,\}$
$\triangleright$ Remove sends and receives on closed channels
$V \leftarrow V \setminus \{\, e^{(Q)} \in V \mid (\textsc{fst}_j, e^{(Q)}) \in E \wedge e = \textsc{snd}^{i \to j} \,\} \setminus \{\, e^{(P)} \in V \mid (e^{(P)}, \textsc{lst}_j) \in E \wedge e = \textsc{rcv}^{j \leftarrow i} \,\}$
rename by dropping superscripts $(X)$
return $(V, E \cap V^2)$
```