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

Kai Engelhardt† Yoram Moses‡

October 19, 2008

A rigorous framework for analyzing safe composition of distributed programs is presented. It facilitates specifying notions of safe sequential execution of distributed programs in various models of communication. 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. None of its send or receive actions will match or interact with actions outside P.

The applicability of sealing 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. In this model 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. 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 class of straight-line programs. It is shown that every sealable program can be sealed using $O(n)$ messages. In fact, $3n-4$ messages are necessary and sufficient in the worst case, despite the fact that a sealable program may be open to interference on $\Omega(n^2)$ channels.

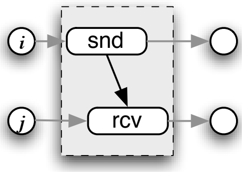
1 Introduction

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. Distributed systems applications are often designed in clearly separated phases, although these phases typically execute concurrently. For instance, using MST to refer to a minimum-weight spanning tree of the network and *STtoLeader protocol* to refer to a protocol using a spanning tree to elect a leader node, Lynch writes in [Lyn96, p. 523]:

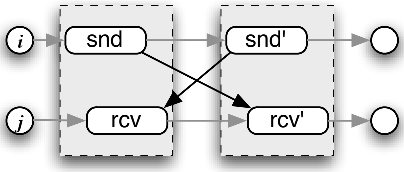
*A preliminary version appeared as [EM05a]. Work was partially supported by ARC Discovery Grant RM02036.

†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.



(a) Transmit once.



(b) Transmit twice.

Figure 1: Transmitting twice with message reordering.

> "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, that is, has output its set of incident edges in the MST."

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.

A customary way of avoiding such confusion is to introduce message headers or, more generally, ensure that different phases use disjoint message alphabets. Effectively, this amounts to allowing different phases to communicate on disjoint sets of virtual channels. A fundamental question considered in this paper is: when is it safe to compose phases that communicate using the same message alphabet? This is obviously a model-dependent issue, as safe composition depends in a crucial way on properties such as synchrony, reliability of communication, and whether channels are FIFO. Being able to establish safe composition can allow us to eliminate the overhead of erecting virtual channels.

An interesting application in which care is taken to ensure that communication obeys phase order is in the design of *synchronizers* by Awerbuch [Awe85]. Here, a program P designed for a round-synchronous message-passing system needs to be executed in an asynchronous setting. Running P without any modification will in general give rise to executions that do not appear in the synchronous setting, and may violate some of the intended properties of P. The synchronizers defined in [Awe85] introduce a control layer between every two rounds of P, which ensures that the phases of P are executed in the same manner in the asynchronous setting as in the synchronous one.

When communication is guaranteed to deliver messages in FIFO order, safe composition requires that a process be able to determine when a received message belongs to the following phase, thereby allowing a trasition from one phase to the next. Life is more interesting in models that allow messages to be reordered. Consider for instance the task of transmitting a message from process i to process j. The task is accomplished by i performing a send action and j performing a corresponding receive action. (See Fig. 1(a).) This implementation works fine in isolation. Composing two copies, 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. (See Figure 1(b).) Thus, executing programs sequentially in this model may be unsafe. As we now show, however, there are cases in which particular programs do compose in a safe manner.

Now consider a program in which a message transmission from i to j is immediately followed by one in the other direction, from j to i. (See Figure 2.) In this case each message sent is received in the phase it was sent. In fact, ssuming that the programs start out with empty channels, a stronger property is satsified in this example. Despite the fact that messages may be reordered by the communication channel, no matter which program is executed later on, the message sent in the first phase from i to j will never be received out of phase. Of course the second transmission is still susceptible to interference, e.g., by another transmission in the same direction. We think of the transmission from j to i as *sealing* the first transmission. More precisely (although formal definitions will come later on) we think of a

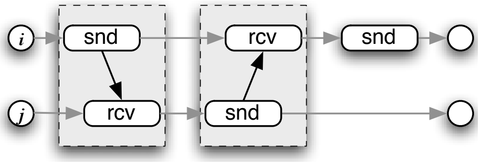


Figure 2: Sealing one transmission with another.

program Q as sealing another program P if when Q is placed immediately after P we are ensured that no communication operations in P will interact with ones in Q or in any later phase. In particular, the second transmission in Figure 2 is said to seal the first transmission.

Contributions. The first contribution of this paper is in presenting a framework for studying safe composition of layers of distributed programs in different models of communication. It introduces a novel specification construct called *silent cut* in the context of a semantics for refinement and layered composition of programs. Within the framework we can represent notions related to safe composition including the communication-closed layers of Elrad and Francez [EF82] and other related notions from the literature. Moreover, we define the notion of sealing among programs, and argue that it allows for a structured compositional approach to safe program construction. 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.

Our second contribution is a case study with a comprehensive analysis of sealing among straight-line programs in a model, which we denote by REL, with reliable communication over reordering channels. We study the theory of sealing in REL and present the following results.

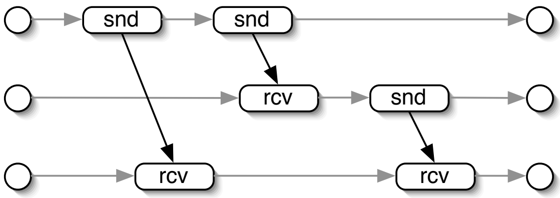
- Sealable straight-line programs are completely characterized.
- A definition of the sealing *signature* of straight-line programs is given, which characterizes the sealing behavior of a program concisely, for both purposes, sealing and being sealed. The size of the signature is $O(n^2)$.
- An algorithm for deciding whether Q seals P based only on their signatures is presented.
- The notion of sealing is shown to be closely related to Lamport causality [Lam78]. Thus, ensuring that Q seals P is tantamount to Q *causing* P to be a communication-closed layer.
- 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.
- Finally, a lower bound is proved showing that our sealing algorithm uses an optimal number of messages in the worst case.

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 many of the interesting aspects relevant to the subject of sealing.

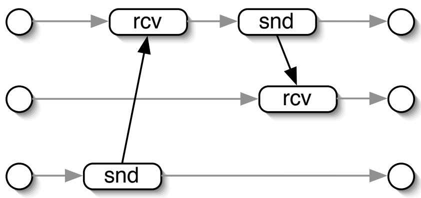
1.1 Further Intuition Regarding Sealing in REL

As illustrated by Figure 2, a message transmission from j to i seals one from i to j; let us consider why. Suppose that a later message is sent on the channel from i to j as in Fig. 2. 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*.

In [Lam78] Lamport defined (potential) causality among events of asynchronous message passing systems. Causality implies temporal precedence. As discussed above, receiving an acknowledgment before sending a new message to process j guarantees that the receive of the acknowledged message by j causally precedes any later sends on the same channel. Observe that the same effect could be obtained by other means. 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. Clearly, sending a chain of messages to obtain this *transitive* form of acknowledgment is an inefficient way of sending a single acknowledgement. Since a given message can play a role in a number of transitive acknowledgments, this can be used efficiently when many acknowledgements are required. Fig. 3(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. 3(b), which sends only two messages.



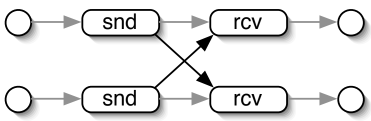
(a) A program P.



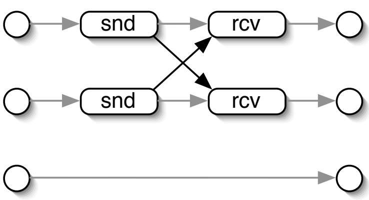
(b) A program Q sealing P.

Figure 3: An example of sealing.

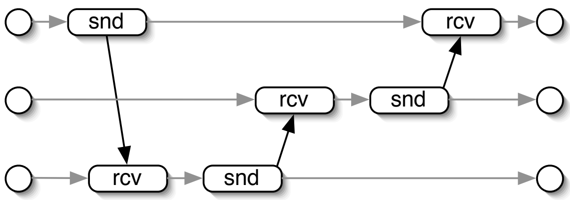
Not all programs can be sealed. For example, program X shown in Fig. 4(a) is unsealable. There are only two channels in the system in question. Intuitively, the first message that is sent on either channel after X might cause interference via message reordering. Hence, a potential seal Q for X cannot safely send messages after X. But if Q does not send messages, then a send performed in either direction after Q will be unsafe. In this case, again, Q would not have sealed X. The same program executed in the presence of a third process as in Fig. 4(b) is, however, sealable. Any seal of this program will necessarily use transitive acknowledgments as discussed above. See Fig. 4(c) for an illustration of one way this program can be sealed.



(a) Unsealable program X.



(b) A sealable program P'.



(c) A seal for P'.

Figure 4: An example of a program for two processes that is unsealable unless a third process is added.

2 Background and Related Work

Perhaps the first formal treatment of the phase composition problem was via the notion of *communication-closed layers* introduced by Elrad and Francez in [EF82]. Consider a program $P = P_1 \parallel \dots \parallel P_n$ consisting of n concurrent processes $P_i = Q_i; L_i; Q'_i$. Intuitively, P is divided into three phases, $Q = Q_1 \parallel \dots \parallel Q_n$, $L = L_1 \parallel \dots \parallel L_n$, and $Q' = Q'_1 \parallel \dots \parallel Q'_n$, executing concurrently with possible overlap. Elrad and Francez define L to be a *communication-closed layer (CCL)* in this program P if under no execution of P does a command in some L communicate with a command in any Q or Q' [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.

The notion of CCLs was further studied and couched in a refinement setting by Zwiers and his colleagues in a sequence of papers [JPZ91, Zwi91, FPZ93, PZ92, JZ93]. They define a notion of "conflict" between actions in a shared-variable model and consider *conflict composition* of programs, in which actions that do not conflict can be concurrent, while conflicting actions need to be executed in sequence. Using this notion, they present a refinement framework for developing concurrent and distributed applications top-down from specifications to implementations. The power and applicability of this framework were demonstrated in a number of case studies, treating well-known problems including Atomic (two-phase) Commitment [JZ92b] and the design of a minimum-weight spanning tree algorithm for asynchronous systems [JZ92a]. Janssen further extended this line of work by considering epistemic logics to specify CCLs [Jan95, Jan94]. 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. They seek to facilitate a methodology in which programs can be designed so that they can be composed in an arbitrary order while still ensuring safe composition. To this end, they define L to be a *General Tail Communication Closed (GTCC)* layer if, roughly speaking, for *all* layers $T = T_1 \parallel \dots \parallel T_n$ we have that L is a CCL in $L_1; T_1 \parallel \dots \parallel L_n; T_n$. Since this definition does not refer to the surrounding program context of a layer, it asserts a certain quality of composability. Composing GTCC layers sequentially guarantees that each one of them is a CCL in the resulting program. GTCC layers can then be used as off-the-shelf components in the design of more complex distributed programs.

3 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 "$\parallel$" (as discussed by Janssen et al. in [JPZ91]). 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, when programs are composed via such layering, a program's execution need not start and end at well-defined global states. Instead, a program executes between two *cuts*, where a cut specifies a local time point for each one of the processes. To account for this, 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. When $r[c,d] \Vdash P$, we think of the interval $r[c,d]$ as being an execution of P.

Our later analysis will be concerned with designing programs P that will be CCLs. 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* , denoted by '$\wr$'. 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.[1] We adopt a standard notion of *refinement* to indicate substitutability of programs. Program P *refines* program Q if, whenever P occurs over an interval $r[c,d]$ then also Q occurs over $r[c,d]$, 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.

3.1 Syntax

Let $n \in \mathbb{N}$ and $\Pi = \{1,\ldots,n\}$ be a set of processes. Throughout the paper n will be reserved for denoting the number of processes. Let $(Var_i)_{i\in\Pi}$ be nonempty and 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*:

$$Prg \ni P \ ::= \ \varepsilon \mid x := \mathsf{e} \mid \mathrm{SND}_{\mathsf{e}}^{i\to j} \mid \mathrm{RCV}_x^{j\leftarrow i} \mid [\phi] \mid \wr \mid P*P \mid P+P \mid P^{\omega}$$

where $x \in Var_i$, $\mathsf{e} \in Expr_i$, $i,j \in \Pi$, 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. If $x \in Var_i$, then we call the statement $x := x$ a $\mathbf{skip}_i$ statement. It is an action by i that has no effect. The $\mathrm{SND}_{\mathsf{e}}^{i\to j}$ 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 $\mathrm{RCV}_x^{j\leftarrow i}$ 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. [JPZ91]. 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 connected by $*$.

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 $^{\omega}$.

The fact that $\mathrm{RCV}^{j\leftarrow i}$ is a blocking action may seem very restrictive, since a process blocks on a particular channel until a message arrives. We are not assuming input enabledness (see [Lyn96]), in which a message delivery is not controlled by the receiving process. However, we can capture the case in which a process waits to receive a message that may arrive on an a channel whose identity in not known a priori. In fact, by using the choice operator $+$, we can capture an action such as $\mathrm{RCV}^{j\leftarrow i_1} + \cdots + \mathrm{RCV}^{j\leftarrow i_k}$ in which the process may receive on one of k channels.

3.2 Semantics

A *send record (for i)* is a triple $(i \to j, \mathrm{v})$, which records sending a message with contents v from i to the receiver j. Similarly, $(j \leftarrow i, \mathrm{v})$ 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 (representing i's communication history). 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 $r = ((r_i)_{i\in\Pi}, \delta_r)$ of local runs — one for

[1] 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 $\wr P \wr$.

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:[2]

1. If $\delta_r(e) = e'$ and e is a receive event of process j resulting in the appending of $(j \leftarrow i, \mathrm{v})$ to j's message history then e' is a send event of process i appending the corresponding send record $(i \rightarrow j, \mathrm{v})$ 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. By modifying these assumptions appropriately we can of course capture message corruption and/or duplication and the like. Moreover, further restrictions on δ_r can be made to capture additional properties of the communication medium such as reliability, FIFO, fairness, etc.

Intuitively, the REL model on which our case study in Section 5 will focus consists of the set of all runs in which communication channels are reliable although messages may be reordered. Since we are considering a setting in which receives are deliberate actions we cannot capture REL by saying simply that every message sent is guaranteed to be received. We need to rule out the case in which the intended receiver stopped performing receive actions on the channel. We shall capture this by saying that, on a channel in which infinitely many messages are delivered in a given run, every message sent is received. Formally, we say that $r \in$ REL if no unmatched send event on a given channel 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 local run 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.[3]

3.2.1 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 Π-indexed family $c = (c_i)_{i \in \Pi}$ 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 none of 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 that 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 $i \in \Pi$. Since a cut specifies an arbitrary set of local times for the processes, it is possible for a message to be received in a given cut without having been sent in the cut. Hence, to describe the state of a channel at a cut we choose to account both for messages that have been sent and not yet received *and* for ones that were received although not yet sent. Thus, we formally define the *channel* $\mathsf{chan}_{i \to j}$ 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 $i \in \Pi$, 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.[4]

While cuts can in general be arbitrary, there are, of course, many instances in which more restricted and well-behaved cuts may be of interest. Indeed, we can define a cut (r, c) to be *consistent* if every

---

[2] 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.

[3] In this paper we consider the $\xrightarrow{\text{L}}$ relation restricted to send and receive events. Ignoring internal events does not affect the relation among the communication actions.

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

$\xrightarrow{\mathsf{L}}$ predecessor of an event in the cut (r,c) is also in (r,c). Moreover, in this work we make use of $\wr$ to capture a stronger property of cuts—that all channels are *empty* at the cut.

3.2.2 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:[5]

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

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

$r[c,d] \Vdash \mathrm{SND}_{\mathsf{e}}^{i \to j}$ 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 $r_i(c_i)$.

$r[c,d] \Vdash \mathrm{RCV}_x^{i \leftarrow j}$ 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).[6]

$r[c,d] \Vdash P * Q$ if there exists c' satisfying $c \leq c' \leq d$ 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 $\mathbb{N}_+$, $c^{(0)} = c$, $c^{(k)} \leq c^{(k')}$ for all $k < k' \in I$, $\bigsqcup_{k \in I} c^{(k)} = d$, and $r[c^{(k)}, c^{(k+1)}] \Vdash P$ for all $k, k+1 \in I$.

3.2.3 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 $P \sqsubseteq_{\Gamma} Q$, iff $r[c,d] \Vdash P$ implies $r[c,d] \Vdash Q$, for all $r \in \Gamma$ and $c,d \in (\mathbb{N}_+)^{\Pi}$. In other words, every execution of P (in a Γ run) is also one of Q, regardless of what happens before and after. Therefore, in systems guaranteed to generate only runs in Γ, 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.

Deadlocks It is worth noting that our semantics explicitly models successful executions of programs, but does not directly represent deadlocks. The blocking actions that may appear in our programs are receives $\mathrm{RCV}_x^{i \leftarrow j}$ and guards $[\phi]$. An example of a deadlocked program is $D = \wr \mathrm{RCV}_x^{1 \leftarrow 2} * \mathrm{RCV}_y^{2 \leftarrow 1}$ in which each of the two processes waits to receive a message from the other, and there are no messages in transit between them. The receive actions will thus never take place, and neither of the processes will ever be able to perform another action. Observe that $r[c,d] \Vdash D$ will not hold for any interval $r[c,d]$.[7] A direct consequence of our definition of refinement is that a program that necessarily deadlocks, such

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

[6] 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.

[7] Formally, we have defined a run to contain an infinite local run for each process. Since a deadlocked process would give rise to a finite local run, no such execution is representable. But even if we were to allow finite local runs in a global run, the receive actions denied by the deadlock would never appear.

as the program D just described, is guaranteed to refine all programs. Thus, our notion of refinement is meaningful mainly for programs that do not deadlock. There is a broad literature on deadlocks and deadlock detection in distributed programs [CES71, Lyn96]. Our treatment below will apply mainly to non-deadlocking programs, with the task of detecting deadlocks being requiring treatment by additional machinery. For example, in Section 5.2 we characterize deadlocks for a particular class of programs under discussion, and provide an efficient algorithm based on Lemma 9 for detecting whether a program is prone to deadlock. In order to avoid the need of repeatedly mentioning deadlocks in the sequel, we shall make the following

General assumption: *Unless stated otherwise, programs are assumed to be deadlock-free.*

4 Capturing Safe Composition and Sealing

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:

$$\wr P * L * Q \wr \quad \sqsubseteq_\Gamma \quad P \wr L \wr Q \ .$$

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 if in isolation. This definition is context-sensitive, where the context consists of both P and Q.

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

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

Traditionally, barriers have been used to synchronize the progression through phases by enforcing that no process could start its $n+1^{\text{st}}$ task before all the other processes had completed their n^{th} tasks. This could be formalized 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 $\xrightarrow{\mathrm{L}}$-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, JZ92a]. 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

$$\wr P \quad \sqsubseteq_\Gamma \quad P \wr \ .$$

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.

[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.

Seals. In many models of interest, only trivial programs are TCC. This is the case, for example, in REL, as shown in Theorem 2. 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 actions will interact with P. We say that S *seals* P *in* Γ if

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

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 and synchronizers. It is of interest to consider the connection between Awerbuch's synchronizers in [Awe85] and sealing. Given a synchronous program $P = R_1 * \ldots * R_k$ consisting of k rounds, Awerbuch constructs a program $A = R_1 * C_1 * \ldots R_{k-1} * C_{k-1} * R_k$, which has a control layer C_i between the pair of rounds R_i and R_{i+1}. While this is perhaps not explicitly stated by Awerbuch, in order to be correct the control layers in the synchronizers must use messages from an alphabet that is disjoint from that of the original synchronous program. If we view the control messages as being sent on separate virtual channels, it is possible to show that each control layer C_i seals the preceding round R_i, and, moreover, is sealed by the following round R_{i+1} in RELFI where communication is reliable and channels are FIFO.

Sealing and program development. Sealing allows incremental program development while maintaining CCL-style composition. The following lemma captures a number of useful cases that facilitate the construction of seals and their 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 $S + S'$ 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 $1 \le k < \ell$, then each layer $L^{(k)}$ is a CCL in P.

Proper Seals. Suppose that $\Pi = \{1, 2\}$ and $x_i \in Var_i$ for $i \in \Pi$. Then the program $Q =$ **while** *true* **do** $(x_1 := 5 * x_2 := 17)$ **od** is TCC in RELFI, a CCL in REL, and seals any program in REL. This is so because Q 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 transmission of a single message from process i to process j is a program that cannot diverge and that seals a transmission in the opposite direction, the former is in particular a proper seal for the latter.

5 Case Study: Safe Composition in REL

We now consider safe composition in the model REL. A central property of REL programs is that communication events can cause a program *not* to be TCC in REL. Let us reconsider the message transmission program from Figure 1(a). In our programming language *Prg* it is expressed by $\mathrm{SND}_{\mathsf{e}}^{i\to j} * \mathrm{RCV}_x^{j\leftarrow i}$, which we shall denote by $\mathrm{MT}_{\mathsf{e}\to x}^{i\to j}$. It is TCC in RELFI but not TCC in REL. That $\mathrm{MT}^{i\to j}$ 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 $x \in Var_i$.

A run $r' \in$ REL that equals r up to d can be constructed such that $r'[d,d'] \Vdash \wr \mathrm{SND}_{\mathsf{e}}^{j\to i} * \mathrm{RCV}_x^{i\leftarrow j} \wr$, 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 $Q = \mathrm{SND}_{\mathsf{e}}^{j\to i} * \mathrm{RCV}_x^{i\leftarrow j}$ it follows that $r''[c,d'] \Vdash \wr P * Q \wr$ but $r''[c,d'] \not\Vdash \wr P \wr Q \wr$. 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 $\Pi = \{1,2\}$ then the program $X = \mathrm{SND}^{1\to 2} * \mathrm{SND}^{2\to 1} * \mathrm{RCV}^{1\leftarrow 2} * \mathrm{RCV}^{2\leftarrow 1}$ illustrated in Fig. 4(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 $i \in \Pi$ 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 REL we obtain

Theorem 5 *Sealability in* REL *is undecidable.*

(The detailed proof can be found in Appendix A.)

5.1 Sealability for Balanced, Straight-line Programs

Given Theorem 5 we shall restrict our attention to more tractable subclasses of programs. Program P is *balanced (in* 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 RELFI. A 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 *BSLs* for short.

The following theorem shows that in REL balance is a necessary prerequisite for being properly sealable.

Theorem 6 *In* REL*, every non-divergent program that is properly sealable is also balanced.*

Proof: Let P and S be programs such that in REL P does not diverge and S properly seals P. Assume by way of contradiction that P is not balanced. Let $r \in$ 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 $r' \in$ REL, $y \in Var_j$, and d' such that r' is the same as r up to d (hence $r'[c,d] \Vdash P * S$), $r'[d,d'] \Vdash \mathrm{RCV}_j^{j\leftarrow i}$, 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 graphs. 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 FST_i and a final dummy node 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 1–4 are program graphs except for Figure 1(b), which illustrates reordering of messages by having edges from the first send to the second receive and from the second send to the first receive. The size of a BSL's program graph is of the order of the size of the BSL .

Next we investigate the connection between program graphs and Lamport causality. We use E^+ to refer to the irreflexive transitive closure of a set of edges E. We call edges that do not contain dummy nodes *normal*, and denote the subset of normal edges by N_E. In RELFI, the normal edges induce the full causality relation on the events of the program. For ease of exposition, we shall often abuse notation slightly and identify events in an execution $r[c,d]$ of a BSL P with nodes of v of its program graph. This is unambiguous, since there is a one-to-one correspondence between the events and the nodes they are identified with. Moreover, we shall say that an event *e causally precedes e' in a program P*, with the intended meaning that $e \xrightarrow{\text{L}} e'$ holds in every REL execution $r[c,d]$ of $\wr P$.

We now show that in the REL model, normal edges of a program graph are guaranteed to be $\xrightarrow{\text{L}}$ edges.

Lemma 7 Let $r \in$ REL and 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 $r \in$ 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. 5. Consider the $k-1$ receives on $\mathsf{chan}_{i\to j}$ that precede e_2. They

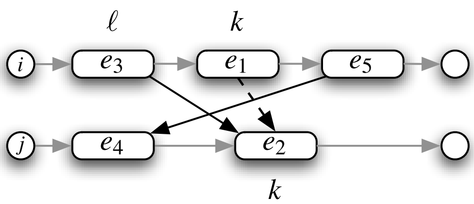


Figure 5: The case $\ell < k$ in the proof of Lemma 7.

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. ■

Lemma 7 implies that all edges in $(N_E)^+$ will be $\xrightarrow{\text{L}}$ edges in every run $r \in$ REL of $\wr P$. We note that $(N_E)^+$ is the largest set of edges with this property because $(N_E)^+$ captures precisely the *necessary* Lamport causality in REL. This is formally captured by the following.

Lemma 8 Let P be a BSL with program graph (V,E) and let v,w be non-dummy nodes of V. Then $(v,w) \in (N_E)^+$ iff $v \xrightarrow{\text{L}} w$ holds for all finite intervals $r[c,d]$ such that $r \in$ REL and $r[c,d] \Vdash \wr P$.

Proof: Let (V,E) be P's program graph, and let $v,w \in V$.

"$\Rightarrow$:" Suppose that $(v,w) \in (N_E)^+$. It follows that there is a finite sequence $v = v_0, v_1, \ldots, v_k = w$ of nodes of V such that $(v_i, v_{i+1}) \in N_E$. The claim follows by induction on k, by transitivity of $\xrightarrow{\text{L}}$, given Lemma 7.

"$\Leftarrow$:" The fact that P is a BSL implies by Lemma 11 that there is an interval satisfying $\hat{r}[c,d] \Vdash \wr P$ where $\hat{r}$ is a RELFI run. If $v \xrightarrow{\text{L}} w$ holds for all finite intervals $r[c,d]$ such that $r \in$ REL and $r[c,d] \Vdash \wr P$, then $v \xrightarrow{\text{L}} w$ holds, in particular, for the run $\hat{r}$. It follows that there is a finite sequence $v = e_0, \ldots, e_k = w$ of send and receive events in $\hat{r}[c,d]$ such that $e_i \xrightarrow{\text{L}} e_{i+1}$ because

1. either e_i is the j^{th} send event and e_{i+1} is the j^{th} receive event on the same channel for some j and hence $\delta_{\hat{r}}(e_{i+1}) = e_i$, or

2. e_{i+1} is the next send or receive event following e_i in some local run r_j.

In both cases $(e_i, e_{i+1}) \in N_E$ is immediate from the definition of program graphs. We conclude that $(v,w) \in (N_E)^+$, as desired. ■

Observe that the definition of the N_E edges matches send and receive events in FIFO order. The proof of Lemma 8 thus shows that, among communication events, RELFI executions of a program P have precisely the Lamport causality common to the intersection of all REL executions of P.

5.2 Deadlock in BSLs.

Since we shall be studying refinement for programs that are assumed to be deadlock-free, it is desirable to be able to identify these programs explicitly. At this point we are able to characterize and decide based on the structure of a BSL whether it has a deadlocking execution in REL. Recall that our program semantics does not capture deadlocks directly. We can, however, still define when a program is deadlocked at a cut outside of the semantics. We now give a definition for straight-line programs because that is all we need for the following result. Recall that the only blocking action in a straight-line program is a receive that waits on a particular channel. In the sequel we will use $\oplus_k 1$ to represent the successor function in $\{1, \ldots, k\}$, that is $j \oplus_k 1 = (j \bmod k) + 1$.

1. We say that a straight-line program Q is *deadlocked at the cut* (r,c) if there is a sequence $\langle i_1, \ldots, i_k \rangle$ of length $k > 1$ such that for every process i_j in the sequence, we have that (i) its first action in Q is of the form $\text{RCV}^{i_j \leftarrow i_{j \oplus_k 1}}$, and (ii) there is no send event in the channel $\mathsf{chan}_{i_{j\oplus_k 1} \to i_j}$ in the cut (r,c). Thus, every process i_j with $j < k$ waits for an unavailable message from i_{j+1}, and i_k waits for one from i_1.

2. We say that the program P *has a deadlocking execution* if there is a way to split P into two layers $P = P' * P''$ such that there exists an interval $r[c,d]$ satisfying both that $r[c,d] \Vdash \wr P'$ and P'' is deadlocked at (r,d).

We can now prove the following lemma, which provides the formal justification for the procedure DEADLOCK-FREE in Appendix B, deciding whether a program deadlocks.

Lemma 9 Let P be a BSL. Then P has a deadlocking execution in REL iff its program graph contains a cycle.

Proof: Let P be a BSL and let (V,E) be its program graph.

"⇒:" Suppose that P has a deadlocking execution in REL. Let $P = P' * P''$ and $r \in$ REL satisfy that $r[c,d] \Vdash \wr P'$ and P'' is deadlocked at (r,d). For ease of exposition, assume without loss of generality that the sequence of deadlocked processes is $\langle 1,\ldots,k\rangle$. For $j = 1,\ldots,k$, let e_j be j's first action in P'' (which by definition is of the form $\textsc{rcv}^{j \leftarrow j\oplus_k 1}$). Our goal is to show that there is an $(N_E)^+$ path from $e_{j\oplus_k 1}$ to e_j in P's program graph. From this it will follow that P's program graph has a cycle, as desired. Fix j. Suppose that process j's first action in the second subprogram P'' is the m_j^{th} receive on the channel $\mathsf{chan}_{j\to j\oplus_k 1}$ in P. Thus, there are $m_j - 1 \geq 0$ receives on the channel in the first subprogram P'. The assumption that j participates in the deadlock of P'' at (r,d) implies that the number of send events on $\mathsf{chan}_{j\to j\oplus_k 1}$ in P' is smaller than m_j, since otherwise there would necessarily be a send in the channel in (r,d). Since P is ballanced, there is an m_j^{th} send on the channel (performed by process $j \oplus_k 1$) in P. But that send must appear after the first action for $j \oplus_k 1$ (a receive) in P''. Thus, if we denote by e' the m_j^{th} send on the channel then we have that $(e_{j\oplus_k 1}, e') \in E$ because both are events on $j \oplus_k 1$ and e' is the later action. Moreover, by definition of the program graph we have that $(e', e_j) \in E$ since e' is the m_j^{th} send on the channel, and e_j is the m_j^{th} receive. We thus obtain an $(N_E)^+$ path from $e_{j\oplus_k 1}$ to e_j in P's program graph as desired, and we are done with the "⇒" direction.

"⇐:" Let $v_1,\ldots,v_k$ be a cycle in (V,E). Observe that this cycle does not contain dummy nodes because first dummy nodes have in-degree 0 and last dummy nodes have out-degree 0. Assume $r[c,d] \Vdash \wr P$ for some cuts c and d and a run $r \in$ REL. By Lemma 7, the cycle is also a cycle in $\xrightarrow{\text{L}}$. But any $\xrightarrow{\text{L}}$ relation induced by δ_r must be acyclic according to our definition of a run. It follows that $r[c,d] \not\Vdash \wr P$, contrary to our assumption. The lack of executions of $\wr P$ over any interval of any run in REL indicates that P necessarily deadlocks in REL. ■

Interestingly, composing non-deadlocking BSLs is guaranteed to yield a non-deadlocking BSL:

Lemma 10 If P and Q are non-deadlocking BSLs, then so is $P * Q$.

Proof: Assume that P and Q are BSLs whose program graphs are acyclic. Then $P * Q$ is clearly balanced, as it has the sum of $\textsc{snd}^{i\to j}$ and $\textsc{rcv}^{j\leftarrow i}$ actions in P and in Q, for all channels $\mathsf{chan}_{i\to j}$, and it is also immediately a straight-line program. Its program graph is acyclic, because we are concatenating two DAGs, connecting the first DAG's sinks with the second one's source nodes. The claim follows. ■

Program graphs mimic RELFI executions of a BSL. Indeed, we can show that BSLs have RELFI executions:

Lemma 11 If P is a non-deadlocking BSL, then there is an interval satisfying $r[c,d] \Vdash \wr P$ where r is a RELFI run.

Proof: Let P be a non-deadlocking BSL. Construct the run $r = ((r_i)_{i\in\Pi}, \delta_r)$ as follows. For every $i \in \Pi$ the local run r_i consists of executing the m_i actions in i's portion of P, followed by infinitely many $\mathbf{skip}_i$ actions. For every channel $\mathsf{chan}_{i\to j}$, the matching function δ_r matches the receives $\textsc{rcv}^{j\leftarrow i}$ on the channel with $\textsc{snd}^{i\to j}$ actions in FIFO order: The first receive to the first send, ..., the k^{th} receive to the k^{th} send, etc. The $\xrightarrow{\text{L}}$ relation in this case coincides with $(N_E)^+$, and since P does not deadlock, $\xrightarrow{\text{L}}$ is acyclic. Thus $r \in$ REL and since δ_r conforms with FIFO ordering, r is a run of RELFI. Finally, let $c = (0)_{i\in\Pi}$ and let (r,d) be the cut reached after executing P from (r,c) onward. It is straightforward to check that $r[c,d] \Vdash \wr P$, as desired. ■

5.2.1 Deciding Sealing

Roughly speaking, for Q to seal P it must guarantee that before any message is sent on a channel $\mathsf{chan}_{i\to j}$ after P is executed, the channel is emptied. This ensures that the later message will not participate in a race against a message sent by P over the same channel. Since P is balanced, it also guarantees that all receives on $\mathsf{chan}_{i\to j}$ in P are matched with sends in P, and none will be available to match later sends. Lamport causality plays an important role in determining sealing among BSLs in REL. Intuitively, if the last $\text{RCV}^{j\leftarrow i}$ in P causally precedes the first $\text{SND}^{i\to j}$ that takes place after P, then $\mathsf{chan}_{i\to j}$ is empty before the later send. Formally, the connection between sealing and causality is captured by the following:

Theorem 12 *Let P and Q be BSLs. Then Q properly seals P in* REL *iff for every channel* $\mathsf{chan}_{i\to j}$

1. *every* $\text{SND}^{i\to j}$ *action in Q must be causally preceded in $\wr P * Q$ by all* $\text{RCV}^{j\leftarrow i}$ *actions in P, and*

2. *the final event on i in $P * Q$ must be causally preceded in $\wr P * Q$ by all* $\text{RCV}^{j\leftarrow i}$ *actions in P.*

Proof: Let P and Q be BSLs, and let $i, j \in \Pi$. If P contains no $\text{SND}^{i\to j}$ action then, being a BSL, it contains no $\text{RCV}^{j\leftarrow i}$ and the claim holds for $\mathsf{chan}_{i\to j}$. Otherwise there are at least one $\text{SND}^{i\to j}$ and one $\text{RCV}^{j\leftarrow i}$ in P.

"⇒:" We start by proving that point 1 holds. Suppose that, for some $r \in \text{REL}$ and cuts c,d,e we have that $r[c,d] \Vdash \wr P$, $r[d,e] \Vdash Q$, and there are a send event $e_3 = \text{SND}^{i\to j}$ in $r[d,e]$ (hence an action of Q) and a receive event $e_2 = \text{RCV}^{j\leftarrow i}$ in $r[c,d]$ such that the send event e_2 does not causally precede the receive e_3. If one of P's communication events in $r[c,d]$ is matched by δ_r to an event after (r,d) then Q does not seal P, and the claim holds. So assume that this is not the case. Let e_1 be a $\text{SND}^{i\to j}$ event and e_4 a $\text{RCV}^{j\leftarrow i}$ event such that $\delta_r(e_4) = e_3$ and $\delta_r(e_2) = e_1$.[9] Our assumption about δ_r implies that e_1 is in (i.e., occurs before) the cut (r,d) and e_4 is outside (r,d). It follows that $e_1 \xrightarrow{\text{L}} e_2 \xrightarrow{\text{L}} e_4$ in r. Let δ' be the mapping obtained from δ_r by switching δ_r's values on e_2 and e_4, and leaving all other values intact. Thus, $\delta'(e_2) = e_3$ and $\delta'(e_4) = e_1$. Define $r' = ((r'_i)_{i\in\Pi}, \delta')$ where the the local runs r'_i contain the same same sequence of actions, respectively, as the local runs r_i do in r. Since $\delta' \neq \delta_r$, the values of variables may differ between r and r', but the sequences of operations are the same. A straightforward induction on the number of operations in P and Q shows that $r'[c,d] \Vdash \wr P$ and $r'[d,e] \Vdash Q$. We claim that the Lamport causality relation $\xrightarrow{\text{L}}$ in r', induced by δ', is acyclic. Assume, by way of contradiction, that it contains a cycle. This cycle must contain the edge (e_3, e_2) because any $\xrightarrow{\text{L}}$ path not including it is also a path in the $\xrightarrow{\text{L}}$ relation for r. But a cycle containing (e_3, e_2) implies the existence of a path establishing $e_2 \xrightarrow{\text{L}} e_3$ in r' and in r, contradicting the assumption that e_2 does not causally precede e_3 in r. Finally, since $r \in \text{REL}$ and δ' only swaps to message deliveries in δ_r, the fact that no unmatched send event in r on a given channel is succeeded by infinitely many matched send events on the same channel implies that the same holds for r'. It follows that $r' \in \text{REL}$ and r' is a witness to the fact that Q does not seal P, since the receive event e_2 of P is matched to a send event not in P.

To prove that point 2 holds, suppose there's $\text{RCV}^{j\leftarrow i}$ in P that does not causally precede the final event on i in $P * Q$. Define $Q' = Q * \text{MT}^{i\to j}$. The fact that P, Q, and $\text{MT}^{i\to j}$ are BSLs implies that $P * Q'$ is a BSL. Observe that the send-receive edge added to the program graph of $P * Q$ to obtain $P * Q'$ cannot participate in a cycle, since the new receive action has out-degree 1, and its only successor, LST_j, has out-degree 0. It follows that the graph of $P * Q'$ is acyclic iff that of $P * Q$ is. Hence, since $P * Q$ is assumed not to deadlock, neither does $P * Q'$. Lemma 11 implies that there is a RELFI run r and cuts c, d, e, and f such that $r[c,d] \Vdash \wr P$, $r[d,e] \Vdash Q$ and $r[e,f] \Vdash \text{MT}^{i\to j}$.

Lemma 8 and the fact that $(N_E)^+$ coincides with $\xrightarrow{\text{L}}$ for $r[c,f]$ yield that the last $\text{RCV}^{j\leftarrow i}$ in P does not causally precede the $\text{SND}^{i\to j}$ in $\text{MT}^{i\to j}$. From here on, a proof identical to that of point 1 shows that Q' does not seal P, and hence Q does not seal P.

[9] Strictly speaking, an event e_4 such that $\delta_r(e_4) = e_3$ need not exist. In this case, the same argument goes through if we omit all reference to e_4.

"$\Leftarrow$:" Assume that points 1 and 2 hold. Let $r \in$ REL and let c, d and e be cuts such that $r[c,d] \Vdash \wr P$ and $r[d,e] \Vdash Q$. We need to show that $r[c,d] \Vdash P\wr$, by demonstrating that all channels are empty at (r,d). Fix a channel $\mathsf{chan}_{i\to j}$. Recall that $\mathsf{chan}_{i\to j}$ is empty at c because $r[c,c] \Vdash \wr$. If there are no $\text{SND}^{i\to j}$ actions in P then there are no $\text{RCV}^{j\leftarrow i}$ actions since P is a BSL. In this case, the fact that $\mathsf{chan}_{i\to j}$ is empty at (r,c) implies that it is empty at (r,d). We can thus assume that there are $\text{SND}^{i\to j}$ and $\text{RCV}^{j\leftarrow i}$ events in P. The fact that $r[c,d] \Vdash \wr P$ implies that all $\text{RCV}^{j\leftarrow i}$ actions in P are matched to $\text{SND}^{i\to j}$ actions by δ_r. We claim that they cannot be matched to ones outside of (r,d). Observe that points 1 and 2 implies that every $\text{RCV}^{j\leftarrow i}$ in P (hence in (r,d)) causally precedes every $\text{SND}^{i\to j}$ outside of (r,d). A match of a $\text{RCV}^{j\leftarrow i}$ of P to a $\text{SND}^{i\to j}$ from the outside creates a cycle in the $\xrightarrow{\text{L}}$ relation in r, which is a contradiction. Finally, since (i) every $\text{RCV}^{j\leftarrow i}$ in P is matched by δ_r to a $\text{SND}^{i\to j}$ in P, (ii) δ_r is injective, and (iii) there numbers of $\text{SND}^{i\to j}$ actions in P is finite and equal to the number of $\text{RCV}^{j\leftarrow i}$ actions in P, we conclude that δ_r matches all $\text{SND}^{i\to j}$ actions in P to $\text{RCV}^{j\leftarrow i}$ actions in P, and so $\mathsf{chan}_{i\to j}$ is empty at (r,d), as claimed. ■

Intuitively, Theorem 12 says that for Q to seal P, it must ensure during its execution that all receives by P are matched. Moreover, if Q sends messages along a channel, then P's receives on that channel must be matched before Q sends on the channel. The channels are thus sure to be emptied after P before any later use.

Theorem 12 can be used to decide sealing among BSLs, given a way of determining causal precedence among actions in a program. Recall that Lemma 8 relates the program graph to Lamport causality. Combining the two results it is possible to derive a decision procedure for sealing based on program graphs. We will design a somewhat more efficient procedure of this, based on a concise summary of the program graph that we call the signature of a program. Signatures keep track of enough information for deciding sealing.

Signatures. Given the program graph (V_P, E_P) of a BSL P we obtain the *signature of P*, denoted by $\text{SIG}(P) = (V^{\text{s}}, E^{\text{s}})$, as follows (see algorithm SIG(P) in Appendix B).

- V^{s} contains of the $2n$ dummy nodes FST_i and LST_i, as well as nodes corresponding to sends and receives as follows. A $\text{SND}^{i\to j}$ node $v \in V_P$ is in V^{s} exactly if both v corresponds to the last $\text{SND}^{i\to j}$ in P and $(\text{FST}_j, v) \notin E_P^+$; similarly, a $\text{RCV}^{j\leftarrow i}$ node $w \in V_P$ is in V^{s} exactly if both w corresponds to the last $\text{RCV}^{j\leftarrow i}$ in P and $(w, \text{LST}_i) \notin E_P^+$.
- The edges of $\text{SIG}(P)$ are the result of restricting E_P^+ to the vertices in V^{s}.

Observe that the signature has size $O(n^3)$, and its size does not grow with that of the program graph. In many natural instances the signature is significantly smaller than that. The signature edges that connect dummy nodes and communication nodes are there to facilitate performing the tests in Theorem 12, while the remaining edges make it possible to construct $\text{SIG}(P * Q)$ from $\text{SIG}(P)$ and $\text{SIG}(Q)$ (see SIGNATURE-COMPOSE in Appendix B). As our focus is on deciding sealing, we can now combine Theorem 12 with Lemma 8 to obtain:

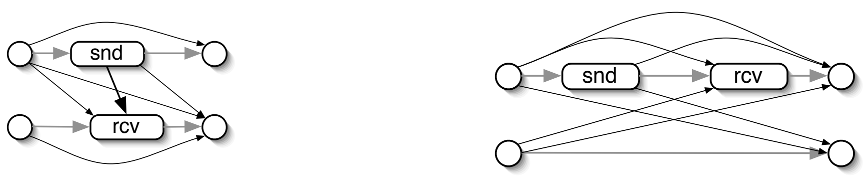
Theorem 13 *Let P and Q be BSLs and let* $\text{SIG}(P) = (V_P^{\text{s}}, E_P^{\text{s}})$ *and* $\text{SIG}(Q) = (V_Q^{\text{s}}, E_Q^{\text{s}})$. *Then Q properly seals P in* REL *iff, for all* $\text{RCV}^{j\leftarrow i} \in V_P^{\text{s}}$, *there exists* $k \in \Pi$ *such that (i)* $(\text{RCV}^{j\leftarrow i}, \text{LST}_k) \in (E_P^{\text{s}})^+$, $(\text{FST}_k, \text{LST}_i) \in (E_Q^{\text{s}})^+$, *and (ii) if* $\text{SND}^{i\to j} \in V_Q^{\text{s}}$ *then* $(\text{FST}_k, \text{SND}^{i\to j}) \in (E_Q^{\text{s}})^+$.

Proof: "$\Leftarrow$:" We will show that the conditions of Theorem 12 are satisfied for every channel. Notice that by transitivity of E^+ relations and of $\xrightarrow{\text{L}}$, we need only show the conditions for the last $\text{RCV}^{j\leftarrow i}$ event in P. Denote the program graphs of P and Q by (V_P, E_P) and (V_Q, E_Q), respectively, and fix a channel $\mathsf{chan}_{i\to j}$. If P has no communication actions over the channel, then both conditions of Theorem 12 vacuously hold. Suppose that P does communicate over the channel. If there is no node $\text{RCV}^{j\leftarrow i} \in V_P^{\text{s}}$ then, by construction of $\text{SIG}(P)$, $(w, \text{LST}_i) \in E_P^+$ where w is the last $\text{RCV}^{j\leftarrow i}$ in P. Lemma 8 implies that $w \xrightarrow{\text{L}} \text{LST}_i$ in all REL executions of P. Moreover, since $\text{RCV}^{j\leftarrow i} \xrightarrow{\text{L}} w$ for all other $\text{RCV}^{j\leftarrow i} \in V_P$ we have that $\text{RCV}^{j\leftarrow i} \xrightarrow{\text{L}} \text{LST}_i$ for all $\text{RCV}^{j\leftarrow i}$ in E_P. Both conditions 1 and 2 of Theorem 12 immediately follow. Finally, if there is a $w = \text{RCV}^{j\leftarrow i}$ in V_P^{s} then a similar argument applies: By the theorem's assumption, there exists $k \in \Pi$ such that (i) $(w, \text{LST}_k) \in (E_P^{\text{s}})^+$, $(\text{FST}_k, \text{LST}_i) \in (E_Q^{\text{s}})^+$, and (ii) if there is an event $v = \text{SND}^{i\to j}$ in V_Q^{s} then $(\text{FST}_k, v) \in (E_Q^{\text{s}})^+$. If $\hat{E}$ is the edge set of the program graph for $P * Q$, then we have that $(w, v) \in \hat{E}^+$. Using Lemma 8 we obtain, for all executions of $\wr P * Q$, that $w \xrightarrow{\text{L}} e$ for i's final action e in Q, and if Q performs an event $v = \text{SND}^{i\to j}$ then $w \xrightarrow{\text{L}} v$ as well.

"$\Rightarrow$:" Suppose that Q properly seals P, that $w \in V_P^{\text{s}}$ is a $\text{RCV}^{j\leftarrow i}$, and that there is no k such that $(w, \text{FST}_k) \in E_P^{\text{s}}$ and $(\text{FST}_k, e) \in E_Q^{\text{s}}$ where $e = \text{SND}^{i\to j}$ if $\text{SND}^{i\to j} \in V_Q^{\text{s}}$ and $e = \text{LST}_i$ otherwise. We claim that $w \xrightarrow{\text{L}} e$ does *not* hold in RELFI executions of $\wr P * Q$. This will complete the argument, by Theorem 12 and Lemma 11. Assume by way of contradiction that $w \xrightarrow{\text{L}} e$ in all REL executions of $\wr P * Q$. Then Lemma 8 implies that $(w, e) \in \hat{E}^+$, where as before $\hat{E}$ is the edge set of the program graph for $P * Q$. Since $w \in V_P$ and $e \in V_Q$, any path of $\hat{E}$ edges connecting w to e must contain an edge whose source is in V_P and target is in V_Q. This edge cannot connect a send to a receive, because such edges are internal to P or to Q for BSLs. The edge must connect two actions of the same process k, and it follows that $(w, \text{LST}_k) \in (E_P)^+$ and that $(\text{FST}_k, e) \in (E_Q)^+$. By the definition of signatures and the choice of the nodes involved, this implies that $(w, \text{LST}_k) \in E_P^{\text{s}}$ and that $(\text{FST}_k, e) \in E_Q^{\text{s}}$, contradicting the theorem's assumption for this part of the proof. ■

The complexity of computing $\text{SIG}(P)$ 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, $\text{SIG}(P * Q)$ can be obtained from their respective signatures at a cost of $O(n^3)$. Given Theorem 13, the complexity of deciding whether Q seals P, given their signatures, is obviously determined by the size of P's signature, and is thus $O(n^3)$. Thus, if signatures are pre-computed, then the complexity of deciding sealing becomes independent of the lengths of the programs. (See algorithm IS-SEAL(P,Q) in Appendix B.)

Let P be a BSL and let $G = (V, E)$ be $\text{SIG}(P)$. Then P leaves channel $\mathsf{chan}_{i\to j}$ open iff $\text{RCV}^{j\leftarrow i} \in V$. For instance, the program $\text{MT}^{i\to j}$ leaves $\mathsf{chan}_{i\to j}$ open — there is a node $\text{RCV}^{j\leftarrow i}$ in $\text{SIG}(\text{MT}^{i\to j})$, which is depicted in Fig. 6(a). As we have shown earlier, $\text{MT}^{j\to i}$ seals $\text{MT}^{i\to j}$ in REL, which implies that $\text{MT}^{j\to i}$ closes $\mathsf{chan}_{i\to j}$ once. Since $\text{MT}^{j\to i}$ does not re-open the channel, the $\text{RCV}^{j\leftarrow i}$ node found in $\text{SIG}(\text{MT}^{i\to j})$ is not present in the $\text{SIG}(\text{MT}^{i\to j} * \text{MT}^{j\to i})$ shown in Fig. 6(b).



(a) $\text{SIG}(\text{MT}^{i\to j})$

(b) $\text{SIG}(\text{MT}^{i\to j} * \text{MT}^{j\to i})$

Figure 6: Examples of signatures. Thin arrows denote transitive causality edges.

We now illustrate how signatures can be used to decide whether one BSL seals another. Suppose that the BSL P leaves $\mathsf{chan}_{i\to j}$ open and Q seals P (see both parts of Figure 7). Then, if Q sends on that channel, then P's last receive $\text{RCV}^{j\leftarrow i}$ on the channel must causally precede Q's first send $\text{SND}^{i\to j}$ on it. If Q does not send on the channel, then 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 $(\text{RCV}^{j\leftarrow i}, \text{LST}_k)$ and Q's signature contains an edge $(\text{FST}_k, \text{LST}_i)$, for some $k \in \Pi$.

5.2.2 A Characterization of Sealability

Program P is said to *close* $\mathsf{chan}_{i\to j}$ *(in* REL*)* if $\mathsf{chan}_{i\to j}$ is empty at the end of P in any execution of P starting from empty channels. channels. This is more formally expressed as follows. For all $r \in \text{REL}$, if

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

(b) Q sealing P by causality. The dashed edge accounts for Q sending on $\mathsf{chan}_{i\to j}$.

Figure 7: Excerpts of the signatures of BSLs P and Q.

$r[c,d] \Vdash \wr P$ then $\mathsf{chan}_{i\to j}$ is empty in (r,d). A channel that is not closed by P is left *open* by it. The state of a program's channels is the essential element in determining sealability. It is straightforward to check

Lemma 14 Let P be a BSL with program graph $G_P = (V_P, E_P)$ and signature $\textsc{Sig}(P) = (V_P^{\mathrm{s}}, E_P^{\mathrm{s}})$. Then $\mathsf{chan}_{i\to j}$ is closed by P iff either

- there is a $\textsc{snd}^{i\to j}$ in V_P^{s}, and the last $v = \textsc{rcv}^{j\leftarrow i}$ in G_P satisfies $(v, \textsc{lst}_i) \in E_P^{+}$, or
- there is no $\textsc{snd}^{i\to j}$ node in V_P^{s}.

Lemma 14 shows that the set of channels closed by a BSL P is uniquely determined by P and can be easily obtained from $\textsc{Sig}(P)$. We can thus associate a *closed-channel graph* with each BSL . Formally, the closed-channel graph $C_P = (\Pi, \textsc{cc}_P)$ of a BSL P is given by $(i,j) \in \textsc{cc}_P$ iff $i \neq j$ and $\mathsf{chan}_{i\to j}$ is closed by P in $\textsc{Rel}$. In the following we denote the undirected version of a graph G by G^{u}.

Theorem 15 (Sealability) *Let P be a BSL . Then P can be sealed properly in* $\textsc{Rel}$ *iff* C_P^u *is connected. Moreover, if P is properly sealable in* $\textsc{Rel}$ *then it can be sealed by a BSL that transmits at most* $3n-4$ *messages.*

Proof: "⇒:" Suppose that C_P^{u} is not connected. Then Π 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, Theorem 12, Lemma 8 and Lemma 14 imply that 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.

"⇐:" The algorithm sketched as $\textsc{Seal}(P)$ takes a BSL P as input and outputs a proper seal for P if P is properly sealable.

$\textsc{Seal}(P)$

$(V, \textsc{cc}_P) \leftarrow \textsc{Closed-Channels}(P)$ ▷ This algorithm is presented in Appendix B.
$S \leftarrow \varepsilon$
pick $T \subseteq \textsc{cc}_P$ s.t. $(\Pi, T)^{\mathrm{u}}$ forms a spanning tree of V
$v \leftarrow$ a node of T with in-degree at least 1 in T
for $(w, w') \in T$ pointing away from v s.t. $(w', w) \notin \textsc{cc}_P$
    do $S \leftarrow S * \textsc{mt}^{w\to w'}$
add a converge-cast in T to S
add a broadcast in T to S
return S

Let S be the result of $\text{SEAL}(P)$. Each one of these MT instances in S transmits a message along a channel that is closed at the time of transmission. For phase (a) in lines 5–6 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 the converge-cast phase (b) in line 7 transmits on closed channels only. Similarly, phase (b) closes all channels between nodes and their children in the spanning tree, hence also the broadcast phase (c) in line 8 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.

Phase (a) sends a message on every edge of the spanning tree pointing away from v. Since v was chosen such that at least one edge need not be redirected this requires at most $n-2$ messages. Each of the converge-cast phase (b) and the broadcast phase (c) transmits $n-1$ messages, one on every edge of T. Altogether, the seal S uses at most $3n-4$ messages. ■

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

Example 16 Consider a phase $P_n = *_{i\in\Pi} L_i$. In P_n each process $i \neq 1$ sends a message to every other process $k \neq i$ before receiving the $n-2$ messages sent to it in this phase. We can define process i's program L_i more formally by

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

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

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

Executing P_n beginning with empty channels leaves $(n-1)^2$ channels open. The $n-1$ channels from process 1 are closed. Nevertheless, P_n can be sealed efficiently by the program $\text{SEAL}(P_n)$ from the proof of Theorem 15 above. Since the channels from 1 to any of the other processes is closed by P_n, any process other than 1 can be chosen as the root v on line 4. (See Fig. 8 for the closed channel graph of P_n.)

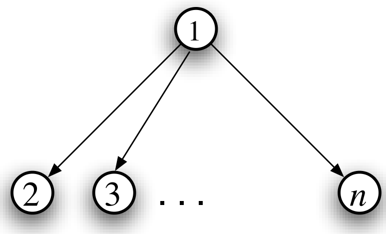


Figure 8: The closed channel graph of P_n.

In fact, we can use this example to show that the upper bound stated in Theorem 15 is tight.

Theorem 17 (lower bound) *For $n > 2$ there exists a sealable BSL P_n such that every proper seal S of P_n sends at least $3n-4$ messages.*

Proof: Let $n > 2$ and assume that S is a proper seal of the program P_n from Example 16. We shall show that S must send at least $3n-4$ messages. The seal S must close each of the open channels, and so must create a causal chain from every process $k \neq 1$ to all other processes. Consider a process $k \neq 1$. We first claim that k cannot send any messages before it has received a message from process 1. Since all outgoing channels from k are open at the end of P_n, we have that such a process k cannot send a message on a channel $k \to i$ before this channel is closed by a causal chain $i \rightsquigarrow k$. If $i \neq 1$, then such a causal chain must, in turn, end with a message arriving on an incoming channel $j \to k$. If $j = 1$ then we have that k is sending to i only after receiving from process 1 as claimed. Otherwise, the message sent on $j \to k$ requires that $j \to k$ has been closed in S. That would require a causal chain from k to j, which must start with a message sent by k. It follows that k can send no message and can receive no messages from processes other than 1 before it receives at least one message from process 1.

In order to create causal chains from every $k \neq 1$ to all other processes, S must have every such process send one or more messages. By the above, then, every process $k \neq 1$ must receive a message from process 1.

We denote by $caused(m)$ for a message m sent in S to be the set of processes for which there are causal chains to m's sender when m is sent. (Notice that, in particular, m's sender is in $caused(m)$.) Consider an ordering σ of all events in S that is a topological sort with respect to the normal edges N_E defined by its signature $\textsc{Sig}(S)$. With respect to the ordering defined by σ, we say that a message m sent to process k is a *clincher* if

$$\{2,\ldots,n\} \setminus \{k\} \quad \subseteq \quad \bigcup \left\{ \, caused(m') \;\middle|\; m' \text{ has been sent to } k \text{ up until and including } m \, \right\} \quad .$$

Thus, once every message sent up to and including m according to the order defined by σ is delivered to m's destination process k, the latter will have causal chains from all necessary processes. Observe that the seal S must generate a clincher for each of the processes $1,\ldots,n$. Let $i_2,\ldots,i_n$ be a permutation of the processes $2,\ldots,n$ ordered according to when they send their first message according to σ. Thus, i_ℓ is the $\ell-1^{\text{st}}$ sender among $\{2,\ldots,n\}$. We denote by m_ℓ the first message sent by i_ℓ. By definition, when m_ℓ is sent, none of the processes in $later(\ell) = \{i_{\ell+1},\ldots,i_n\}$ has sent any messages in S that causally precede the sending of m_ℓ. Therefore, there are no causal chains to i_ℓ from processes in $later(\ell)$ when m_ℓ is sent. It follows that $m_2,\ldots,m_{n-1}$ cannot be clinchers. Nevertheless, the need for causal chains from $2,\ldots,n$ to all processes implies that every process must receive a clincher. We can thus "charge" each process $i_\ell \in \{i_2,\ldots,i_{n-1}\}$ for three messages: its first message from 1, the message m_ℓ , and for the first clincher that i_ℓ receives. We charge i_n for its first message from 1, and we charge process 1 for the first clincher that it receives. (Since we charge each of 1 and i_n for only one message, we are not concerned what other roles this message may have, e.g., whether i_n's message from i is also a clincher.) Since all charged messages are distinct—we are charging for a message at most once—and since they are all necessarily sent by S, we conclude that S must send at least $3(n-2)+2 = 3n-4$ messages, as claimed. ■

6 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 a CCL captures when a program operates as if it were executed in isolation when invoked in the context of a given larger program. We have introduced a framework for studying safe program composition. Using a silent cut operator $\wr$ and a notion of refinement, it enables concise formal definitions of standard notions such as CCL, barriers, and TCC. Moreover, it facilitates the introduction and study of new building blocks for safe composition. 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. Moreover, in this case each of the subporgrams is guaranteed to be a CCL, and so their combined behavior can be deduced from the individual guarantees provided from each of the components. Within our framework, it is natural to consider issues of safe composition in different models of distributed computation. Notably, the approach allows for seamless composition of programs without need for translation or headers.

In the second part of this paper, we studied sealing and safe composition for BSLs (balanced, straight-line programs) in the REL model, in which message delivery is reliable but channels can reorder messages. While this is a restricted class of programs, the subtleties introduced by message reordering already make for a nontrivial underlying structure of seals for BSLs. Future work will consider extending this class to allow for branching (**if-then-else-fi**) and looping. These introduce additional subtleties, but will the underlying structure discovered in the current analysis appears to remain valid in the broader setting. In another paper [EM05b], we use essentially the same framework to investigate safe composition in models with FIFO channels that may be prone to message duplication or loss. As in the case of REL, 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.[10]

We provided tools for deciding whether a BSL P in REL is sealable, for deciding if Q seals P, as well as for constructing a seal for P if P is sealable. These are based on the *program graph* and *signature* of a BSL . We proved that $O(n)$ messages are necessary and sufficient for sealing a sealable BSL in REL. In particular, this means that a linear number of acknowledgements can be used to seal a quadratic number of open channels. The notion of sealing in 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. To see how sealing can be used to design useful programs in REL from simpler ones, recall that $\text{MT}^{i\to j} * \text{MT}^{j\to i}$ seals itself in REL. Lemma 1.5 can be used to show that a program of the form **while** *true* **do** $\text{MT}^{i\to j} * \text{MT}^{j\to i}$ **od** will correctly transmit a sequence of values from i to j in REL. Indeed, if the return messages from j to i are not merely acknowledgments, this can perform sequence exchange.

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.

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 a very careful reading leading to many comments and an observation that led to a simplification of the notation. We thank the anonymous referees for their detailed and most useful comments.

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.

[Awe85] Baruch Awerbuch. Complexity of network synchronization. *Journal of the ACM*, 32(4):804–823, 1985.

[CES71] Edward G. Coffman, Jr., M. J. Elphick, and Arie Shoshani. System deadlocks. *ACM Computing Surveys*, 3(2):67–78, 1971.

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

[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.

[FPZ93] Maarten Fokkinga, Mannes Poel, and Job Zwiers. Modular completeness for communication closed layers. In Eike Best, editor, *CONCUR '93: 4th International Conference on Concurrency Theory*, volume 715 of *LNCS*, pages 50–65, Hildesheim, Germany, 23–26 August 1993. Springer-Verlag.

[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.

[JZ92a] 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.

[JZ92b] Wil Janssen and Job Zwiers. Protocol design by layered decomposition: A compositional approach. In Jan Vytopil, editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 571 of *LNCS*, pages 307–326. Springer-Verlag, 1992.

[JZ93] Wil Janssen and Job Zwiers. Specifiying and proving communication closedness in protocols. In André A. S. Danthine, Guy Leduc, and Pierre Wolper, editors, *PSTV*, volume C-16 of *IFIP Transactions*, pages 323–339. North-Holland, 1993.

[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.

[Zwi91] Job Zwiers. Layering and action refinement for timed systems. In J. W. de Bakker, Cornelis Huizing, Willem P. de Roever, and Grzegorz Rozenberg, editors, *Real-Time: Theory in Practice, REX Workshop*, volume 600 of *LNCS*, pages 687–723. Springer-Verlag, 1991.

A Proofs

Proof of Lemma 1: First observe that non-deterministic choice distributes over layering, that is, $(P+P')*Q = P*Q+P'*Q$ and $P*(Q+Q') = P*Q+P*Q'$.

1. Assume that both P and P' are sealed by S in Γ, that is, $\wr P*S \sqsubseteq_\Gamma P\wr S$ and $\wr P'*S \sqsubseteq_\Gamma P'\wr S$ by the definition of sealing.

$$\begin{aligned} \wr(P+P')*S &\sqsubseteq_\Gamma \wr(P*S+P'*S) && \text{distributivity}\\ &\sqsubseteq_\Gamma \wr P*S+\wr P'*S && \text{distributivity}\\ &\sqsubseteq_\Gamma P\wr S+P'\wr S && \text{assumption}\\ &\sqsubseteq_\Gamma (P+P')\wr S && \text{distributivity}\end{aligned}$$

2. Assume that both S and S' seal P in Γ, that is, $\wr P*S \sqsubseteq_\Gamma P\wr S$ and $\wr P*S' \sqsubseteq_\Gamma P\wr S'$.

$$\begin{aligned} \wr P*(S+S') &\sqsubseteq_\Gamma \wr P*S+\wr P*S' && \text{distributivity}\\ &\sqsubseteq_\Gamma P\wr S+P\wr S' && \text{assumption}\\ &\sqsubseteq_\Gamma P\wr(S+S') && \text{distributivity}\end{aligned}$$

3. Assume that S seals P in Γ, that is, $\wr P*S \sqsubseteq_\Gamma P\wr S$. By monotony of $*$ with respect to refinement, also $S*Q$ seals P in Γ.

4. Assume that S seals P and S' seals S in Γ, that is, (1) $\wr P*S \sqsubseteq_\Gamma P\wr S$ and (2) $\wr S*S' \sqsubseteq_\Gamma S\wr S'$.

$$\begin{aligned} \wr P*S*S' &\sqsubseteq_\Gamma P\wr S*S' && \text{assumption (1) and monotony}\\ &\sqsubseteq_\Gamma P*S\wr S' && \text{assumption (2) and monotony}\end{aligned}$$

5. Assume that P seals itself (in Γ). By induction we show that P also seals P^i, for any number $i \in \mathbb{N}$ of layered compositions of P. The base case is $P^0 = \varepsilon$, which is sealed by every program. The inductive case is an application of point 4 of this lemma. Recall that non-deterministic choice is modelled as union in our semantics. Next observe that the proof of point 1 works for arbitrary unions, too. Finally, the only difference between P^ω and the infinite choice between the P^i where $i \in \mathbb{N}$ is that the latter lacks the infinite repetition, P^∞ as a non-deterministic choice. It remains to be shown that P also seals P^∞.

6. "$\Rightarrow$:" Assume that P is TCC in Γ, i.e., $\wr P \sqsubseteq_\Gamma P \wr$. Let Q be a program. Then $\wr P * Q \sqsubseteq_\Gamma P \wr Q$ by monotony of layering with respect to the refinement order.

   "$\Leftarrow$:" Assume that every program Q seals P in Γ. Then this is in particular true for the empty program ε, i.e., $\wr P * \varepsilon \sqsubseteq_\Gamma P \wr \varepsilon$. Since ε is a right unit of layering, it follows that $\wr P \sqsubseteq_\Gamma P \wr$. ■

Proof of Theorem 5: We reduce the halting problem to sealability. Suppose D is a decider for sealability of programs in Prg. Let Prg_i be the class of programs of our language Prg for the single process $\Pi = \{i\}$ containing neither sends nor receives. Observe that halting problem for Prg_i is undecidable already because we have full arithmetic over the integers in our programming language. We construct a decider H for Prg_1 as follows.

$H =$ "on input $\langle P \rangle$ where $P \in Prg_1$:

1. Construct $P' \in Prg_2$ as a variant of P for process 2 by replacing variables of process 1 by variables of process 2.

2. Run D on $\langle X * P * P' \rangle$, where $X = \textsc{snd}^{1\to 2}_{x+1} * \textsc{snd}^{2\to 1}_{y+1} * \textsc{rcv}^{1\leftarrow 2}_{x} * \textsc{rcv}^{2\leftarrow 1}_{y}$ from Lemma 4.

3. *Accept* if D rejects and *reject* if D accepts."

Thus, if D deciding sealability exists then H decides the halting problem for Prg_1. It follows that sealability is undecidable. ■

B Algorithms

PROGRAM-GRAPH(P)

1 $V, E \leftarrow \bigcup_{i \in \Pi}\{\textsc{fst}_i, \textsc{lst}_i\}, \emptyset$ $\quad\triangleright$ First and last dummy nodes for each process
2 $f \leftarrow \lambda i : \Pi.\textsc{fst}_i$ $\quad\triangleright$ Book keeping for local precedence
3 $\triangleright$ 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 $V, E, f(i) \leftarrow V \cup \{e\}, E \cup \{(f(i), e)\}, e$
4 $\triangleright$ Add precedence between last i-event and i's last dummy node
  for $i \in \Pi$
    do $E \leftarrow E \cup \{(f(i), \textsc{lst}_i)\}$
5 $\triangleright$ 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
6 **return** (V, E)

DEADLOCK-FREE(P)

1 $V, E \leftarrow$ PROGRAM-GRAPH(P)
2 **return** $\neg\exists$ cycle in E

SIG(P)

1 $V, E \leftarrow$ PROGRAM-GRAPH(P)
2 $E \leftarrow E^+$ $\quad\triangleright$ Add irreflexive transitive closure
3 $\triangleright$ Remove all but minimal sends and maximal receives on open channels
  $V \leftarrow V \setminus \{\, e \mid e \text{ is a } \textsc{snd}^{i\to j} \text{ event preceded by another such send or } \textsc{fst}_j \,\}$
  $V \leftarrow V \setminus \{\, e \mid e \text{ is a } \textsc{rcv}^{j\leftarrow i} \text{ event that precedes another such receive or } \textsc{lst}_i \,\}$
4 **return** $(V, E \cap V^2)$

IS-SEAL(P,Q)

$V_P, E_P \leftarrow$ SIG(P)
$V_Q, E_Q \leftarrow$ SIG(Q)
for $(i,j) \in \Pi^2 \setminus \mathrm{id}_\Pi$ s.t. $\mathrm{RCV}^{j \leftarrow i} \in V_P$
 do $e \leftarrow \begin{cases} \mathrm{SND}^{i \to j} & \text{if } \mathrm{SND}^{i \to j} \in V_Q \\ \mathrm{LST}_i & \text{otherwise} \end{cases}$
  safe $\leftarrow$ *false*
  for $k \in \Pi \setminus \{i\}$
   do $safe \leftarrow safe \vee ((\mathrm{RCV}^{j \leftarrow i}, \mathrm{LST}_k) \in E_P \wedge (\mathrm{FST}_k, \mathrm{SND}^{i \to j}) \in E_Q)$
  if $\neg$ *safe*
   then return *false*
return *true*

CLOSED-CHANNELS(P)

$V, E \leftarrow \Pi, \Pi^2 \setminus \mathrm{id}_\Pi$
$V', E' \leftarrow$ SIG(P)
for $(i,j) \in E$
 do if $\mathrm{RCV}^{j \leftarrow i} \in V'$ and $(\mathrm{RCV}^{j \leftarrow i}, \mathrm{LST}_i) \notin E'$
  then $E \leftarrow E \setminus \{(i,j)\}$
return (V, E)

SIGNATURE-COMPOSE(V_P, E_P, V_Q, E_Q)

$\triangleright$ Sequentially compose the two signatures
$V \leftarrow \left\{ e^{(P)} \;\middle|\; e \in V_P \right\} \cup \left\{ e^{(Q)} \;\middle|\; e \in V_Q \right\}$
$E \leftarrow \left\{ (e^{(P)}, f^{(P)}) \;\middle|\; (e,f) \in V_P \right\} \cup \left\{ (e^{(Q)}, f^{(Q)}) \;\middle|\; (e,f) \in V_Q \right\} \cup \left\{ (\mathrm{LST}_i^{(P)}, \mathrm{FST}_i^{(Q)}) \;\middle|\; i \in \Pi \right\}$
$E \leftarrow E^+$
$\triangleright$ Remove dummy nodes between the two signatures
$V \leftarrow V \setminus \left\{ \mathrm{LST}_i^{(P)} \;\middle|\; i \in \Pi \right\} \setminus \left\{ \mathrm{FST}_i^{(Q)} \;\middle|\; i \in \Pi \right\}$
$\triangleright$ Remove all but the first sends and last receives
$V \leftarrow V \setminus \left\{ e^{(Q)} \in V \;\middle|\; e^{(P)} \in V \wedge e = \mathrm{SND}^{i \to j} \right\} \setminus \left\{ e^{(P)} \in V \;\middle|\; e^{(Q)} \in V \wedge e = \mathrm{RCV}^{j \leftarrow i} \right\}$
$\triangleright$ Remove sends and receives on closed channels
$V \leftarrow V \setminus \left\{ e^{(Q)} \in V \;\middle|\; (\mathrm{FST}_j, e^{(Q)}) \in E \wedge e = \mathrm{SND}^{i \to j} \right\} \setminus \left\{ e^{(P)} \in V \;\middle|\; (e^{(P)}, \mathrm{LST}_j) \in E \wedge e = \mathrm{RCV}^{j \leftarrow i} \right\}$
rename by dropping superscripts (X)
return $(V, E \cap V^2)$