Robustness against rendezvous

ANONYMOUS AUTHOR(S)

CCS Concepts: • **Software and its engineering → General programming languages**;

Additional Key Words and Phrases: Concurrency, Serializability

ACM Reference format:
Anonymous Author(s). 2017. Robustness against rendezvous. *Proc. ACM Program. Lang.* 1, 1, Article 1 (January 2017), 8 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

2 OVERVIEW

3 MESSAGE PASSING SYSTEMS

We fix arbitrary sets $\mathbb{P}$ and $\mathbb{V}$ of process ids and message payloads. For given sets $\mathbb{P}$ and $\mathbb{V}$, we fix the sets

$$S = \{\text{send}(p, q, v) : p, q \in \mathbb{P}, v \in \mathbb{V}\}, \text{ and } R = \{\text{rec}(q, v) : q \in \mathbb{P}, v \in \mathbb{V}\}$$

of *send actions* and *receive actions*. Each send action $\text{send}(p, q, v)$ combines two process ids $p, q \in \mathbb{P}$ denoting the sender and the receiver of the message, respectively, and a message payload $v \in \mathbb{V}$. Receive actions specify the process $q \in \mathbb{P}$ receiving the message, and the message payload $v \in \mathbb{V}$. We write $\text{send}(p, q, \_)$ and $\text{rec}(q, \_)$ when the message payload is unconstrained. We denote the process executing a send/receive action $a \in S \cup R$ by $\text{proc}(a)$, i.e., $\text{proc}(a) = p$ for every $a = \text{send}(p, q, v)$, and $\text{proc}(a) = q$ for every $a = \text{rec}(q, v)$, and the destination of a send action $s \in S$ by $\text{dest}(s)$, i.e., $\text{dest}(\text{send}(p, q, v)) = q$.

A message passing system consists of a finite set of processes that communicate via messages. Each process is described as a state machine that evolves by executing send or receive actions, or local actions (specified as ϵ transitions) and it is equipped with an instance of some fixed collection data type, e.g., FIFO queue, that acts as a message buffer (storing incoming messages before being received).

Formally, a *message passing system* is a tuple $A = (L, \delta, l_0, \mathbb{T})$ where L is a set of local process states, $\delta \subseteq L \times (S \cup R \cup \{\epsilon\}) \times L$ is a transition relation describing the evolution of all processes, l_0 is the initial state of every process, and $\mathbb{T}$ is a collection data type with interface $\text{add}(m)$ for adding an element m to the collection and $\text{rem}()$ that removes and returns an element from the collection (this method returns only when the collection is non-empty, otherwise it blocks).

A configuration $c = \langle \vec{l}, \vec{b} \rangle$ is a vector of local states $\vec{l}$ together with a vector of message buffers $\vec{b}$ that are instances of the collection data type $\mathbb{T}$. These two vectors are indexed by elements of $\mathbb{P}$. For a vector $\vec{x}$, let $\vec{x}_p$ denote the element of $\vec{x}$ of index p. The state of the process p is defined by the local state $\vec{l}_p$ and the message buffer $\vec{b}_p$.

We fix an arbitrary set $\mathbb{M}$ of message identifiers, and the sets

$$S_{id} = \{s_i : s \in S, i \in \mathbb{M}\} \text{ and } R_{id} = \{r_i : r \in R, i \in \mathbb{M}\}$$

2017. 2475-1421/2017/1-ART1 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

$$\textsc{send}\ \frac{m = (i, v) \qquad i \in \mathbb{M}\ \text{fresh} \qquad l \in \delta(\vec{l}_p, \text{send}(p, q, v)) \neq \emptyset}{\vec{l}, \vec{b} \xrightarrow{\text{send}_i(p,q,v)} \vec{l}[\vec{l}_p \leftarrow l], \vec{b}[\vec{b}_q.\ \text{add}(m)]}$$

$$\textsc{receive}\ \frac{(i, v) = \vec{b}_q.\ \text{rem}() \qquad l \in \delta(\vec{l}_q, \text{rec}(q, v)) \neq \emptyset}{\vec{l}, \vec{b} \xrightarrow{\text{rec}_i(q,v)} \vec{l}[\vec{l}_q \leftarrow l], \vec{b}[\vec{b}_q.\ \text{rem}()]}$$

$$\textsc{local}\ \frac{l \in \delta(\vec{l}_p, \epsilon) \neq \emptyset}{\vec{l}, \vec{b} \xrightarrow{\epsilon} \vec{l}[\vec{l}_p \leftarrow l], \vec{b}}$$

Fig. 1. The asynchronous semantics of a message passing system A. Above, $\vec{l}[\vec{l}_p \leftarrow l]$ denotes an update of $\vec{l}$ where the element of index p is replaced by l. Also, $\vec{b}[\vec{b}_q.\ \text{add}(m)]$ denotes the vector of message buffers obtained from $\vec{b}$ by calling the method $\text{add}(m)$ of the element of index q.

of indexed actions; an indexed action combines a send/receive action with a message identifier $i \in \mathbb{M}$. Message identifiers are used to pair send and receive actions. We denote the message id of an indexed send/receive action a by $\text{msg}(a)$. Indexed send and receive actions $s \in S_{id}$ and $r \in R_{id}$ are *matching*, written $s \mapsto r$, when $\text{msg}(s) = \text{msg}(r)$. For notational convenience, we often associate $\mathbb{M}$ with $\mathbb{N}$, e.g., writing $\text{send}_1(p, q, v)$ and $\text{rec}_2(q', v')$ in place of $\text{send}_{i_1}(p, q, v)$ and $\text{rec}_{i_2}(q', v')$.

The transition relation $\rightarrow$ in Figure 1 is determined by a message passing system A, and maps a configuration c_1 to another configuration c_2 and indexed action $a \in S_{id} \cup R_{id}$ or ϵ. SEND transitions ... RECEIVE transitions ... LOCAL transitions TODO FILL IN

An *execution* of a system A under the asynchronous semantics to configuration c_n is a configuration sequence $c_0 c_1 \ldots c_n$ such that $c_m \xrightarrow{a_{m+1}} c_{m+1}$ for $0 \leq m < n$. We call the sequence $a_1 \ldots a_n$ the *trace* of $c_0 c_1 \ldots c_n$ and we say that c_n is reachable in A under the asynchronous semantics, with trace $a_1 \ldots a_n$. The *reachable local state vectors* of A, denoted Asynch-States(A), is the set of local state vectors in reachable configurations. The set of traces of A under the asynchronous semantics is denoted by AsynchTr(A).

Given a trace t, a send action s in t is called an *unmatched send* when t contains no receive action r with $s \mapsto r$. A trace t is called *matched* when it contains no unmatched send.

4 SYNCHRONIZABILITY

We define a notion of robustness, called *synchronizability*, which ensures that even if the system uses buffers to store messages it provides the illusion that messages are received instantaneously, as in a synchronous (rendez-vous) semantics. This is analogous to the atomicity criterion for concurrent shared-memory systems, where even though transactions can interleave, every execution is "equivalent" to an execution where transactions happen atomically without interference.

We define a conflict relation $\prec$ on actions in $S_{id} \cup R_{id}$ that relates every two actions of the same process and every send with the matching receive. Formally,

$$a \prec a' \text{ iff } \text{proc}(a) = \text{proc}(a') \text{ or } a \mapsto a'$$

A permutation t' of a trace t is *conflict-preserving* when every pair a and a' of actions of t appear in the same order in t' whenever $a \prec a'$. From now on, whenever we use the term permutation we mean conflict-preserving permutation. Intuitively, a message passing system can not distinguish between two traces, one being a conflict-preserving permutation of the other.

$$\textsc{send}\ \frac{l \in \delta(\vec{l}_p, \mathrm{send}(p, q, v)) \neq \emptyset \qquad \mathrm{empty}(\vec{b}_j)\ \text{for all}\ j}{\vec{l}, \vec{b} \xRightarrow{\mathrm{send}(p,q,v)} \vec{l}[\vec{l}_p \leftarrow l], \vec{b}[\vec{b}_q . \mathrm{add}(v)]} \qquad \textsc{local}\ \frac{l \in \delta(\vec{l}_p, \epsilon) \neq \emptyset}{\vec{l}, \vec{b} \xrightarrow{\epsilon} \vec{l}[\vec{l}_p \leftarrow l], \vec{b}}$$

$$\textsc{receive}\ \frac{v = \vec{b}_q . \mathrm{rem}() \qquad l \in \delta(\vec{l}_q, \mathrm{rec}(q, v)) \neq \emptyset \qquad \mathrm{empty}(\vec{b}_j)\ \text{for all}\ j \neq q}{\vec{l}, \vec{b} \xRightarrow{\mathrm{rec}(q,v)} \vec{l}[\vec{l}_q \leftarrow l], \vec{b}[\vec{b}_q . \mathrm{rem}()]}$$

Fig. 2. The synchronous semantics of a message passing system A. The predicate $\mathrm{empty}(\vec{b}_j)$ denotes the fact that the buffer $\vec{b}_j$ is empty, i.e, $\vec{b}_j . \mathrm{rem}()$ is not enabled.

Example 4.1. The following traces

$$\mathrm{send}_1(p_1, q, \_)\ \mathrm{send}_2(p_2, q, \_)\ \mathrm{rec}_1(q, \_)\ \mathrm{rec}_2(q, \_)$$
$$\mathrm{send}_1(p_1, q_1, \_)\ \mathrm{send}_2(p_2, q_2, \_)\ \mathrm{rec}_2(q_2, \_)\ \mathrm{rec}_1(q_1, \_)$$

have the following conflict-preserving permutations, respectively:

$$\mathrm{send}_1(p_1, q_1, \_)\ \mathrm{rec}_1(q, \_)\ \mathrm{send}_2(p_2, q, \_)\ \mathrm{rec}_2(q, \_)$$
$$\mathrm{send}_1(p_1, q_1, \_)\ \mathrm{rec}_1(q_1, \_)\ \mathrm{send}_2(p_2, q_2, \_)\ \mathrm{rec}_2(q_2, \_)$$

Lemma 4.2. *For a given trace t,* $\mathrm{AsynchTr}(A)$ *contains every conflict-preserving permutation t' of t when $t \in \mathrm{AsynchTr}(A)$.*

A trace t is called *synchronous* when every send $s \in S_{id}$ is immediately followed by the matching receive. Formally, for every $0 \leq k < |t| - 1$, if $t_k \in S_{id}$ then $t_{k+1} \in R_{id}$ and $t_k \mapsto t_{k+1}$.

Informally, a trace t is called *synchronizable* when there exists a conflict-preserving permutation of t which is synchronous. The presence of unmatched send actions introduces a subtlety: a trace is considered synchronizable by supposing that every message is eventually received. To account for this, we define a *completion* of a trace t to be any sequence $t \cdot t'$ where t' contains only receive actions that match one of the unmatched send actions in t.

Definition 4.3. A trace t is called *synchronizable* when there exists a synchronous conflict-preserving permutation of a completion of t.

Example 4.4. The traces in Example 4.1 are synchronizable while the following are not:

$$\mathrm{send}_1(p, q, \_)\ \mathrm{send}_2(p, q, \_)\ \mathrm{rec}_2(q, \_)\ \mathrm{rec}_1(q, \_)$$
$$\mathrm{send}_1(p, q, \_)\ \mathrm{send}_2(q, p, \_)\ \mathrm{rec}_2(p, \_)\ \mathrm{rec}_1(q, \_)$$
$$\mathrm{send}_1(p, q, \_)\ \mathrm{send}_2(q, p, \_)$$

The latter trace demonstrates the use of completions in defining synchronizability: none of its completions, i.e., one of the first two traces, cannot be permuted to a synchronous trace.

A message passing system A is synchronizable when every trace $t \in \mathrm{AsynchTr}(A)$ is synchronizable.

The transition relation $\Rightarrow$ in Figure 1 defines a new semantics for a message passing system A where a receive action of a process q is enabled only if the buffers of all the other processes are empty and the buffer of process q contains exactly one message. Executions and reachable local state vectors under the new semantics, denoted by Synch-States(A), are defined as in Section 3. The set of traces of A under the new semantics is denoted by Synch-Tr(A).

TODO DESCRIBE TRANSITIONS

Lemma 4.5. *The synchronous semantics of a message passing system A admits every synchronous trace of A under the asynchronous semantics.*

TODO DEFINE THE CONDITION XXX THAT EVERY MESSAGE IS EVENTUALLY RECEIVED

Lemma 4.6. *For a given synchronizable XXX message passing system A,* Asynch-States(A) = Synch-States(A).

Proof. TODO □

5 MONITORING SYNCHRONIZABILITY

Checking that a given trace is synchronizable can be done by tracking a "conflict-graph" like in conflict serializability [] to show that every pair of matching send and receive actions can be executed atomically. Roughly, the nodes of this graph correspond to such pairs of matching actions, and the edges to conflicts between actions from one pair and another. Then, a trace is synchronizable iff the conflict-graph is acyclic. Dealing with unmatched sends and completions poses a technical difficulty since the conflict-graph should contain enough additional edges to anticipate for the possible completions.

Before defining conflict-graphs, we define a notion of action-graph that represents the order between all conflicting actions in some given trace.

Definition 5.1 (Action-Graph). The *action-graph* of a trace t is the directed graph $G_t = \langle V, E\rangle$ where there is a node in V for each action in t, and E contains an edge from u to v iff $\mathrm{act}(u) \prec \mathrm{act}(v)$ and $\mathrm{act}(u)$ occurs before $\mathrm{act}(v)$ in t (where $\mathrm{act}(v)$ is the action of trace t corresponding to the graph node v).

For two actions a and a' in a trace t, $a \rightsquigarrow a'$ denotes the fact that G_t contains a path from the node representing a to that representing a'.

Lemma 5.2. *The* action-graph *of a trace is acyclic.*

Definition 5.3 (Conflict-Graph). The *conflict-graph* of a trace t is the directed graph $CG_t = \langle V', E'\rangle$ where V' includes one node for each pair of matching send and receive actions, and each unmatched send action in t, and we have $(v, v') \in E'$ iff one of the following holds

(1) there exist actions $a \in \mathrm{act}(v)$ and $a' \in \mathrm{act}(v')$ such that $(a, a') \in E$ where $G_t = (V, E)$ is the action-graph of t (and $\mathrm{act}(v)$ is the set of actions of trace t corresponding to the graph node v), or
(2) $\mathrm{act}(v)$ contains a send action s_1 and $\mathrm{act}(v')$ contains an unmatched send action s_2 such that $\mathrm{dest}(s_1) = \mathrm{dest}(s_2)$,
(3) $\mathrm{act}(v)$ contains an action a and $\mathrm{act}(v')$ contains an unmatched send action s_1 with $\mathrm{proc}(a) = \mathrm{dest}(s_1)$.

The second and the third item above apply to unmatched traces and they represent conflicts occurring in specific completions. TODO GIVE A HINT. Before discussing these completions, a direct consequence of previous results on conflict serializability [] is that a matched trace is synchronizable whenever its conflict-graph is acyclic.

Lemma 5.4. *A matched trace is synchronizable iff its conflict-graph is acyclic.*

We consider a notion of *canonical* completion where intuitively, the order between the receive actions added by the completion is consistent with the order of the send actions in t. Formally, a *completion* $t \cdot t'$ of a trace t is called canonical when for every two unmatched send actions $\mathrm{send}_1(p_1, q_1, \_)$ and $\mathrm{send}_2(p_2, q_2, \_)$ occurring in this order in t, the suffix t' contains $\mathrm{rec}_1(q_1, \_)$ before $\mathrm{rec}_2(q_2, \_)$.

Lemma 5.5. *A trace t is synchronizable iff the canonical completion of t is synchronizable.*

Proof. The "if" direction is trivial. For the reverse, let t be a synchronizable trace. Therefore, there exists a completion $t \cdot t'$ which is synchronizable. If $t \cdot t'$ is canonical, then the proof is finished. Otherwise, let $\text{send}_1(p_1, q_1, \_)$ and $\text{send}_2(p_2, q_2, \_)$ be two unmatched send actions occurring in this order in t such that $\text{rec}_2(q_2, \_)$ occurs before $\text{rec}_1(q_1, \_)$ in t'. W.l.o.g., we assume that there is no other action of q_1 or q_2 in between $\text{rec}_2(q_2, \_)$ and $\text{rec}_1(q_1, \_)$. Otherwise, for instance, if there is an action $\text{rec}_3(q_2, \_)$ in between $\text{rec}_2(q_2, \_)$ and $\text{rec}_1(q_1, \_)$, then either $\text{send}_2(p_2, q_2, \_)$ occurs before $\text{send}_3(p_3, q_2, \_)$ (for some p_3) and we can choose $\text{send}_1(p_1, q_1, \_)$ and $\text{send}_3(p_3, q_2, \_)$ satisfying this property, or $\text{send}_3(p_3, q_2, \_)$ occurs before $\text{send}_2(p_2, q_2, \_)$ and we can choose $\text{send}_2(p_2, q_2, \_)$ and $\text{send}_3(p_3, q_2, \_)$ satisfying this property.

Note that, since $\text{send}_1(p_1, q_1, \_)$ occurs before $\text{send}_2(p_2, q_2, \_)$, it is impossible that $\text{send}_2(p_2, q_2, \_) \rightsquigarrow \text{send}_1(p_1, q_1, \_)$. By Lemma 5.4, the conflict graph of $t \cdot t'$ is acyclic. Let $t \cdot t''$ be a permutation of $t \cdot t'$ where $\text{rec}_2(q_2, \_)$ and $\text{rec}_1(q_1, \_)$ are swapped (i.e., $\text{rec}_1(q_1, \_)$ occurs before $\text{rec}_2(q_2, \_)$). If $q_1 \neq q_2$, then this permutation is conflict-preserving and the conflict graph of $t \cdot t''$ is the same as the one of $t \cdot t'$. Now, assume that $q_1 = q_2$. Then, the conflict-graph of $t \cdot t''$ is the conflict-graph of $t \cdot t'$ where the edge from the node u representing $\{\text{send}_2(p_2, q_2, \_), \text{rec}_2(q_2, \_)\}$ to the node v representing $\{\text{send}_1(p_1, q_1, \_), \text{rec}_1(q_1, \_)\}$ is replaced with an edge from v to u. Since the conflict-graph of $t \cdot t'$ is acyclic, the action graph of $t \cdot t'$ contains no path from $\text{send}_1(p_1, q_1, \_)$ to $\text{send}_2(p_2, q_2, \_)$ (otherwise, this path together with the edge from $\text{rec}_2(q_2, \_)$ to $\text{rec}_1(q_1, \_)$ will induce a conflict-graph cycle). Therefore, the only path in $CG_{t \cdot t'}$ between u and v is the single-edge path from u to v, and changing the direction of this edge will result in a graph that remains acyclic. Consequently, the conflict-graph of $t \cdot t''$ is acyclic. Applying this reasoning further for every pair of unmatched send operations in t shows that the canonical completion of t is synchronizable. □

The second and the third item in Definition 5.3 represent conflicts occurring in canonical completions of the trace t.

Lemma 5.6. *For every trace t, CG_t is isomorphic to $CG_{t'}$ where t' is the canonical completion of t.*

The following is a direct consequence of Lemmas 5.5 and 5.6.

Theorem 5.7. *A trace t is synchronizable iff CG_t is acyclic.*

Theorem 5.7 can be used to define a runtime monitor checking for synchronizability. The monitor maintains the conflict-graph of the trace produced by the message passing system and checks whether it contains some cycle. While this approach requires dealing with the complexity of the asynchronous semantics (the unbounded message buffers), the next section shows that this is not necessary. Synchronizability violations, if any, can be exposed by executing the system under the *synchronous* semantics.

6 CHECKING SYNCHRONIZABILITY

Verifying synchronizability for a message passing system using a monitor that maintains the conflict-graph of a trace is undecidable since it requires dealing with known undecidable features, e.g., FIFO queues. We show in this section that checking synchronizability can be reduced to a reachability problem in a system that executes under the synchronous semantics, and thus doesn't use unbounded message buffers. This result is based on the following ideas

- consider a class of borderline violations to synchronizability, i.e., traces which are not synchronizable and every strict prefix is synchronizable.

Causal delivery violation:

$$\text{send}_1(p_1, q, \_) \rightsquigarrow \text{send}_2(p_2, q, \_) \text{ and } \text{rec}_2(q, \_) \rightsquigarrow_{lossy} \text{rec}_1(q, \_)$$

Exchange pattern:

$$\text{send}_1(p_1, q_1, \_) \rightsquigarrow_{lossy} \text{rec}_2(q_2, \_) \text{ and } \text{send}_2(p_2, q_2, \_) \rightsquigarrow_{lossy} \text{rec}_1(q_1, \_)$$

Fig. 3. Synchronizability violation patterns.

- starting from the original system A, define a new system A_0 executing under the synchronous semantics, which simulates borderline synchronizability violations of A, if any, and A_0 goes to an error state whenever such a violation exists.

6.1 Synchronizability Violation Patterns

We show that any trace violating synchronizability contains one of the two violation patterns in Figure 3. Intuitively, the first pattern describes a violation to *causal delivery* where two causally related messages sent to the same process are received in an order different from that in which they were sent. Here, the causal order between messages is defined by the paths in the action graph (ignoring unmatched sends, the predicate $\rightsquigarrow_{lossy}$ is exactly $\rightsquigarrow$). The second violation pattern describes a situation in which two messages are sent concurrently and each message is received after the other one is sent. This situation arises for instance in distributed protocols which alternate between phases in which all processes send messages to their peers and phases in which they receive messages from their peers. In these cases, the sends and receives in the exchange pattern are causally related since they are executed by the same process.

We say that a trace contains a causal delivery violation or an exchange pattern if it contains the actions and the action-graph paths specified in Figure 3. In the context of matched traces, the predicate $\rightsquigarrow_{lossy}$ is exactly $\rightsquigarrow$ (describing action-graph paths). Otherwise, the interpretation of $\rightsquigarrow_{lossy}$ takes into consideration the fact that the receive action in the right-hand side can occur in the completion of an unmatched trace. Thus, $a \rightsquigarrow_{lossy} \text{rec}_1(q, \_)$ holds in a trace t when a is an action of t and either $a \rightsquigarrow \text{rec}_1(q, \_)$ (which implies that $\text{rec}_1(q, \_)$ is also an action of t), or t contains an action b such that $\text{proc}(b) = q$ and $a \rightsquigarrow b$, or $\text{proc}(a) = q$.

Theorem 6.1. *A trace is not synchronizable iff it contains a causal delivery violation or an exchange pattern.*

Proof. The "only if" direction is trivial. For both violation patterns, the two pairs of matching send/receive actions (or one pair and an unmatched send, in the case of the causal delivery violations) will define a conflict graph cycle.

For the "if" direction, let t be a non-synchronizable trace. Let us first assume that t is matched. Then, the conflict graph of t contains two nodes v_1 and v_2 such that there is a path from v_1 to v_2 and vice-versa. Let $\text{act}(v_i) = \{\text{send}_i(p_i, q_i, \_), \text{rec}_i(q_i, \_)\}$, for $i \in [1, 2]$. Then, one of the following holds:

- $\text{send}_1(p_1, q_1, \_) \rightsquigarrow \text{send}_2(p_2, q_2, \_)$ and $\text{rec}_2(q_2, \_) \rightsquigarrow \text{rec}_1(q_1, \_)$, which corresponds to a violation to causal delivery when $q_1 = q_2$. Otherwise, when $q_1 \neq q_2$, we show that this cycle induces another (smaller cycle) that contains a causal delivery violation. Thus, consider the path between $\text{rec}_2(q_2, \_)$ and $\text{rec}_1(q_1, \_)$. By the definition of the conflict relation, the last conflict in this path is necessarily between another action a of q_1 (i.e., $\text{proc}(a) = q_1$) and $\text{rec}_1(q_1, \_)$. Otherwise, this path will contain $\text{send}_1(p_1, q_1, \_)$ and we would have $\text{send}_1(p_1, q_1, \_) \rightsquigarrow \text{send}_2(p_2, q_2, \_) \mapsto \text{rec}_2(q_2, \_)$ and $\text{rec}_2(q_2, \_) \rightsquigarrow \text{send}_1(p_1, q_1, \_)$ which is

impossible by Lemma 5.2 (the action graph would be cyclic). Now, the path between $\mathrm{rec}_2(q_2, \_)$ and a will necessarily contain a conflict due to matching (i.e., a pair of matching send/receive actions); otherwise, $q_1 = q_2$. Let $\mathrm{send}_3(p_3, q_3, \_) \mapsto \mathrm{rec}_3(q_3, \_)$ be the last such conflict (according to the order in this path). Since it is the last such conflict, we have that $q_3 = q_1$. Therefore, we have $\mathrm{rec}_3(q_1, \_) \rightsquigarrow \mathrm{rec}_1(q_1, \_)$, and $\mathrm{send}_1(p_1, q_1, \_) \rightsquigarrow \mathrm{send}_2(p_2, q_2, \_) \mapsto \mathrm{rec}_2(q_2, \_) \rightsquigarrow \mathrm{send}_3(p_3, q_1, \_)$ which is equivalent to $\mathrm{send}_1(p_1, q_1, \_) \rightsquigarrow \mathrm{send}_3(p_3, q_1, \_)$. The latter is a violation to causal delivery.

- $\mathrm{send}_1(p_1, q_1, \_) \rightsquigarrow \mathrm{rec}_2(q_2, \_)$ and $\mathrm{send}_2(p_2, q_2, \_) \rightsquigarrow \mathrm{rec}_1(q_1, \_)$ which is an exchange pattern.
- $\mathrm{send}_1(p_1, q_1, \_) \rightsquigarrow \mathrm{rec}_2(q_2, \_)$ and $\mathrm{rec}_2(q_2, \_) \rightsquigarrow \mathrm{send}_1(p_1, q_1, \_)$ which is impossible by Lemma 5.2 because it would imply a cyclic action graph.
- $\mathrm{rec}_1(q_1, \_) \rightsquigarrow \mathrm{send}_2(p_2, q_2, \_)$ and $\mathrm{rec}_2(q_2, \_) \rightsquigarrow \mathrm{send}_1(p_1, q_1, \_)$ which is impossible, by Lemma 5.2.

TODO EXTEND TO UNMATCHED TRACES □

6.2 Simulating Borderline Violations

A violation to synchronizability t is called *borderline* when every strict prefix of t is synchronizable. We first show that any borderline synchronizability violation of a message passing system A can be simulated by the synchronous semantics of another message passing system B computable in polynomial time.

Let t be a borderline synchronizability violation. By Theorem 6.1, t contains either a causal delivery violation or an exchange pattern. Therefore, we have

$$t = t_1 \cdot \mathrm{send}_1(p_1, q_1, \_) \cdot t_2 \cdot \mathrm{rec}_1(q_1, \_)$$

where the prefix $t' = t_1 \cdot \mathrm{send}_1(p_1, q_1, \_) \cdot t_2$ is a synchronizable trace. We first define a system lossy(A) whose synchronous semantics simulates every synchronizable trace of A, and then the system B which simulates the complete trace t. The meaning of the term "simulation" is defined later on.

A trace t is called *synchronous but lossy* when every receive $r \in R_{id}$ is immediately preceded by the matching send. Formally, for every $0 < k \leq |t| - 1$, if $t_k \in R_{id}$ then $t_{k-1} \in S_{id}$ and $t_{k-1} \mapsto t_k$.

Lemma 6.2. *A trace t is synchronizable whenever there exists a synchronous but lossy permutation of t.*

A synchronous but lossy trace t can be rewritten to a synchronous trace by redirecting all unreceived messages to a fixed additional process π_0. Let remove-lossiness(t) be the trace obtained from t by replacing every unmatched $\mathrm{send}_i(p, q, v)$ with $\mathrm{send}_i(p, \pi_0, v) \cdot \mathrm{rec}_i(\pi_0, v)$.

We define the system lossy(A) whose

admits remove-lossiness(t')

By Lemma 4.5, the synchronous semantics of A admits all synchronous traces. However, it may not admit any conflict-preserving permutation

synchronizable traces are not admitted by the synchronous semantics because some of the sent messages are never received, i.e., such traces contain send actions without matching receives. However, they are admitted by the synchronous semantics of another system lossy(A) where messages can be non-deterministically redirected to a fixed additional process π_0. More precisely, given a synchronizable trace

Synch-Tr(lossy(A)) contains a

Thus, if $A = (L, \delta, l_0, \mathbb{T})$ then lossy(A) $= (L, \delta', l_0, \mathbb{T})$ where

$$\delta' = \delta \cup \{(l, \mathrm{send}(p, \pi_0, v), l'), (l_0, \mathrm{rec}(\pi_0, v), l_0) : \exists\, \mathrm{send}(p, q, v) \in S.\ (l, \mathrm{send}(p, q, v), l') \in \delta\}.$$

In words, every send transition to some process q is cloned to a send transition to process π_0 together with a corresponding receive transition of process π_0.

Lemma 6.3. *fsdfsf*

To deal with the last receive action, the trace t can be transformed into a synchronous trace t' by redirecting the message $\langle i_1, j_1, v_1\rangle$ to a distinguished process p that relays it to j_1 later, i.e.,

$$t' = t_1 \cdot \mathrm{send}_{i_1}(i_1, p, v_1) \cdot \mathrm{rec}_p(i_1, p, v_1) \cdot t_2 \cdot \mathrm{send}_p(p, j_1, v_1) \cdot \mathrm{rec}_{j_1}(p, j_1, v_1)$$

Lemma 6.4. *The trace t' is a synchronous trace when t is a borderline violation.*

In the following, we define a message passing system B whose synchronous semantics admits t' whenever A admits t (under the asynchronous semantics).

By Lemma 4.5, the prefix t' is admitted by the synchronous semantics of A, i.e., $t' \in \text{Synch-Tr}(A)$. Thus, if $A = (L, \delta, l_0, \mathbb{T})$ then $B = (L \uplus \{l_j : j \in \mathbb{V}\}, \delta_2, l_0, \mathbb{T})$ where

- every send transition to some process j is cloned to a send transition to process p and a corresponding receive transition of process p (the local state of p stores the identifier of process j), i.e.,

$$\delta_1 = \delta \cup \{(l, \mathrm{send}_i(i, p, v), l), (l_0, \mathrm{rec}_p(i, p, v), l_j) : \exists\, \mathrm{send}_i(i, j, v) \in S.\ (l, \mathrm{send}_i(i, j, v), l') \in \delta\}.$$

- every receive transition of some process j is cloned to a send transition of process p and a corresponding receive transition from process p, i.e.,

$$\delta_2 = \delta_1 \cup \{(l_j, \mathrm{send}_p(p, j, v), l_0), (l, \mathrm{rec}_j(p, j, v), l') : \exists\, \mathrm{rec}_j(i, j, v) \in R.\ (l, \mathrm{rec}_j(i, j, v), l') \in \delta\}.$$

TODO CAN WE HAVE PROBLEMS IF THE is IN THE SEND AND RECEIVE TRANSITIONS ARE NOT THE SAME. MAYBE WE SHOULD REMOVE i FROM MESSAGES AND LEAVE ONLY PAYLOAD.

6.3 Checking for Synchronizability Violation Patterns

We define a monitor in the form of a register automaton (a finite state machine equipped with a set of registers) that

7 EXPERIMENTAL EVALUATION

8 RELATED WORK

9 CONCLUSIONS