Robustness against rendezvous

Abstract.

1 Introduction

2 Overview

3 Message passing systems

TODO ASSUME PAYLOADS INCLUDE A MESSAGE IDENTIFIER

We fix arbitrary sets $\mathbb{I}$ and $\mathbb{P}$ of process ids and message payloads. A *message* is a triple $m = \langle i, j, p\rangle$ where $i \in \mathbb{I}$ denotes the source, $j \in \mathbb{I}$ the destination, and $p \in \mathbb{P}$ the payload. The set of messages is denoted by M. The source, destination, and payload of a message m are denoted by $\mathsf{src}(m)$, $\mathsf{dst}(m)$, $\mathsf{pay}(m)$, respectively. For given sets $\mathbb{I}$ and $\mathbb{P}$, we fix the sets

$$S = \{\mathrm{send}_i(m) : i \in \mathbb{I}, m \in M, i = \mathsf{src}(m)\}, \text{ and } R = \{\mathrm{rec}_j(m) : j \in \mathbb{I}, m \in M, j = \mathsf{dst}(m)\}$$

of *send actions* and *receive actions*; each send action $\mathrm{send}_i(m)$ combines a process id $i \in \mathbb{I}$ with a message $m \in M$ whose source is the same process i. We denote the message of a send/receive action a by $\mathsf{msg}(a)$, and the process id indexing the action by $\mathsf{proc}(a)$. Send and receive actions $s \in S$ and $r \in R$ are *matching*, written $s \mapsto r$, when $\mathsf{msg}(s) = \mathsf{msg}(r)$.

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 processed).

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 add(m) for adding an element m to the collection and rem() that removes and returns an element from the collection (this method returns only when the collection is non-empty). To simplify the exposition, we assume that each receive action is enabled in every local state, i.e., for every $l \in L$ and every $r \in R$, there exists $l' \in L$ such that $(l, r, l') \in \delta$.
TODO GIVE MORE EXPLANATIONS

A configuration $c = \langle \boldsymbol{l}, \boldsymbol{b}\rangle$ is a vector of local states $\boldsymbol{l}$ together with a vector of message buffers $\boldsymbol{b}$ that are instances of the collection data type $\mathbb{T}$. For a vector

$$\text{SEND}\quad \frac{m = \langle i, j, p\rangle \qquad \delta(\boldsymbol{l}_i, \text{send}_i(m)) \neq \emptyset}{\boldsymbol{l}, \boldsymbol{b} \xrightarrow{\text{send}_i(m)} \boldsymbol{l}[\boldsymbol{l_i} \leftarrow \delta(\boldsymbol{l}_i, \text{send}_i(m))], \boldsymbol{b}[\boldsymbol{b}_j.\ \text{add}(m)]}$$

$$\text{RECEIVE}\quad \frac{m = \boldsymbol{b}_j.\ \text{rem}() \qquad \delta(\boldsymbol{l}_j, \text{rec}_j(m)) \neq \emptyset}{\boldsymbol{l}, \boldsymbol{b} \xrightarrow{\text{rec}_j(m)} \boldsymbol{l}[\boldsymbol{l_j} \leftarrow \delta(\boldsymbol{l}_j, \text{rec}_j(m))], \boldsymbol{b}[\boldsymbol{b}_j.\ \text{rem}()]}$$

$$\text{LOCAL}\quad \frac{\delta(\boldsymbol{l}_i, \epsilon) \neq \emptyset}{\boldsymbol{l}, \boldsymbol{b} \xrightarrow{\epsilon} \boldsymbol{l}[\boldsymbol{l_i} \leftarrow \delta(\boldsymbol{l}_i, \epsilon)], \boldsymbol{b}}$$

Fig. 1: The asynchronous semantics of a message passing system A. Above, $\boldsymbol{l}[\boldsymbol{l_i} \leftarrow \delta(\boldsymbol{l}_i, \text{send}_i(m))]$ denotes an update of $\boldsymbol{l}$ where the ith element is replaced by one of the local states in $\delta(\boldsymbol{l}_i, \text{send}_i(m))$. Also, $\boldsymbol{b}[\boldsymbol{b}_j.\ \text{add}(m)]$ denotes the vector of message buffers obtained from $\boldsymbol{b}$ by calling the method $\text{add}(m)$ of the jth element.

$\boldsymbol{x}$, let $\boldsymbol{x}_i$ denote the ith element of $\boldsymbol{x}$. The state of a process is defined by a local state $\boldsymbol{l}_i$ and a message buffer $\boldsymbol{b_i}$, for some i.

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 action $a \in S \cup R$. 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).

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 $S \cup R$ that relates every two actions of the same process and every send with the corresponding receive. Formally,

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

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 $a \prec a'$. Intuitively, a

$$\text{SEND}\quad \frac{m = \langle i, j, p\rangle \qquad \delta(\boldsymbol{l}_i, \text{send}_i(m)) \neq \emptyset}{\boldsymbol{l}, \boldsymbol{b} \xRightarrow{\text{send}_i(m)} \boldsymbol{l}[\boldsymbol{l_i} \leftarrow \delta(\boldsymbol{l}_i, \text{send}_i(m))], \boldsymbol{b}[\boldsymbol{b}_j.\ \text{add}(m)]} \qquad \text{LOCAL}\quad \frac{\delta(\boldsymbol{l}_i, \epsilon) \neq \emptyset}{\boldsymbol{l}, \boldsymbol{b} \xrightarrow{\epsilon} \boldsymbol{l}[\boldsymbol{l_i} \leftarrow \delta(\boldsymbol{l}_i, \epsilon)], \boldsymbol{b}}$$

$$\text{RECEIVE}\quad \frac{m = \boldsymbol{b}_j.\ \text{rem}() \qquad \delta(\boldsymbol{l}_j, \text{rec}_j(m)) \neq \emptyset \qquad \boldsymbol{b_i}.\text{rem}() \text{ is not enabled, for all } i \neq j}{\boldsymbol{l}, \boldsymbol{b} \xRightarrow{\text{rec}_j(m)} \boldsymbol{l}[\boldsymbol{l_j} \leftarrow \delta(\boldsymbol{l}_j, \text{rec}_j(m))], \boldsymbol{b}[\boldsymbol{b}_j.\ \text{rem}()]}$$

Fig. 2: The synchronous semantics of a message passing system A.

message passing system can not distinguish between two traces, one being the conflict-preserving permutation of the other.

Example 1. The following traces

$$\text{send}_{i_1}(i_1, j_1, \_)\ \text{send}_{i_2}(i_2, j_1, \_)\ \text{rec}_{j_1}(i_1, j_1, \_)\ \text{rec}_{j_1}(i_2, j_1, \_)$$
$$\text{send}_{i_1}(i_1, j_1, \_)\ \text{send}_{i_2}(i_2, j_2, \_)\ \text{rec}_{j_2}(i_2, j_2, \_)\ \text{rec}_{j_1}(i_1, j_1, \_)$$

have the following conflict-preserving permutations, respectively:

$$\text{send}_{i_1}(i_1, j_1, \_)\ \text{rec}_{j_1}(i_1, j_1, \_)\ \text{send}_{i_2}(i_2, j_1, \_)\ \text{rec}_{j_1}(i_2, j_1, \_)$$
$$\text{send}_{i_1}(i_1, j_1, \_)\ \text{rec}_{j_1}(i_1, j_1, \_)\text{send}_{i_2}(i_2, j_2, \_)\ \text{rec}_{j_2}(i_2, j_2, \_)$$

Lemma 1. *For a given trace t,* AsynchTr(A) *contains every conflict-preserving permutation t' of t when $t \in$* AsynchTr(A).

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

Definition 1. *A trace t is called* synchronizable *when there exists a synchronous conflict-preserving permutation of t.*

Example 2. The traces in Example 1 are synchronizable while the following are not:

$$\text{send}_{i_1}(i_1, j_1, v_1)\ \text{send}_{i_1}(i_1, j_1, v_2)\ \text{rec}_{j_1}(i_1, j_1, v_2)\ \text{rec}_{j_1}(i_2, j_1, v_1)$$
$$\text{send}_{i_1}(i_1, j_1, \_)\ \text{send}_{j_1}(j_1, i_1, \_)\ \text{rec}_{i_1}(j_1, i_1, \_)\ \text{rec}_{j_1}(i_1, j_1, \_)$$

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

The transition relation $\Rightarrow$ in Figure 1 defines a new semantics for a message passing system A where a receive for process i is enabled only if the buffers of all the other processes are empty. 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).

Lemma 2. *For a given synchronizable message passing system A,* Asynch-States(A) = Synch-States(A).

Proof. TODO HERE WE USE THE ASSUMPTION ABOUT RECEIVE ENABLEDNESS

Checking that a given trace is synchronizable can be done by tracking a "conflict-graph" like in serializability where transactions are pairs of sends and corresponding receives. A trace is synchronizable iff the conflict-graph is acyclic. Formally,

Definition 2 (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).*

Intuitively, one can think of the action-graph of a trace t as a structure that represents the order between all conflicting actions in t. 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'.

Definition 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 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).*

Theorem 1. *(from [**?**]) A trace t is synchronizable iff CG_t is acyclic.*

5 Checking Synchronizability

Verifying a (bounded state) 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 boarderline violations to synchronizability, i.e., traces which are not synchronizable and every strict prefix is synchronizable.
- starting from the original system A, define a new system A_0 executing under the synchronous semantics, which simulates minimal synchronizability violations of S, if any (S_0 goes to an error state whenever such a violation exists).

Causal delivery violation:

$$\mathrm{send}_{i_1}(i_1, j, v_1) \rightsquigarrow \mathrm{send}_{i_2}(i_2, j, v_2) \text{ and } \mathrm{rec}_j(i_2, j, v_2) \rightsquigarrow \mathrm{rec}_j(i_1, j, v_1)$$

Exchange pattern:

$$\mathrm{send}_{i_1}(i_1, j_1, v_1) \rightsquigarrow \mathrm{rec}_{j_2}(i_2, j_2, v_2) \text{ and } \mathrm{send}_{i_2}(i_2, j_2, v_2) \rightsquigarrow \mathrm{rec}_{j_1}(i_1, j_1, v_1)$$

Fig. 3: Synchronizability violation patterns.

5.1 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. 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 protocols solving consensus 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.

Theorem 2. *A trace is not synchronizable iff it contains a causal delivery violation or an exchange pattern.*

Proof. TODO

5.2 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 $\mathit{something}(A)$ computable in polynomial time.

Let t be a borderline synchronizability violation. By Theorem 2, t contains either a causal delivery violation or an exchange pattern. Therefore, we have

$$t = t_1 \cdot \mathrm{send}_{i_1}(i_1, j_1, v_1) \cdot t_2 \cdot \mathrm{rec}_{j_1}(i_1, j_1, v_1)$$

where the prefix $t' = t_1 \cdot \mathrm{send}_{i_1}(i_1, j_1, v_1) \cdot t_2$ is a synchronous trace. The prefix t' may contain send actions without matching receives and it is not admitted as it is by the synchronous semantics of A, i.e., $t' \notin \text{Synch-Tr}(A)$. We show however that the configurations reached in A by an execution with trace t' can be reached in another message passing system A' even under the synchronous semantics.

Thus, A' is obtained from A by non-deterministically redirecting messages to some distinguished process p that accepts any message. Formally, if $A = (L, \delta, l_0, \mathbb{T})$ then $A' = (L \uplus \{l_p\}, \delta_2, l_0, \mathbb{T})$ where

- every send transition to some process j is cloned to a send transition to process p, i.e.,

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

- the process p can receive any message

$$\delta_2 = \delta_1 \cup \{(l_p, \mathrm{rec}_p(i,p,v), l_p) : \exists\, \mathrm{send}_i(i,p,v) \in S.\ (l, \mathrm{send}_i(i,p,v), l') \in \delta_1\}.$$

For a given synchronous trace t, let redirect-orphans(t) be the trace obtained from t by replacing every unmatched send action $\mathrm{send}_i(i,j,v)$ with $\mathrm{send}_i(i,p,v)\mathrm{rec}_p(i,p,v)$.

TODO REDIRECT ORPHANS PRESERVES ACYCLICITY PRECISELY

Lemma 3. *Let $(\boldsymbol{l}, \boldsymbol{b})$ be a configuration reachable in A with a synchronous trace t' (under the asynchronous semantics). Then, $(\boldsymbol{l}, \boldsymbol{\emptyset})$ is reachable in A' with trace* redirect-orphans(t').

To simulate the last receive in t, we transform A' such that any message sent to some process j_1 can be redirected to some distinguished process i_0 who relays it to j_1 non-deterministically at a later time. We constrain the system such that it can redirect only one message.

TODO NEEDS A MONITOR FOR IMPOSING THIS CONSTRAINT. BUT MAYBE WE DON'T NEED IT IF WE ARE JUST INTERESTED IN DEFINING A SYSTEM THAT ADMITS BORDERLINE VIOLATIONS (AND ANYTHING IN ADDITION)

5.3 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

6 Experimental Evaluation

7 Related Work

8 Conclusions