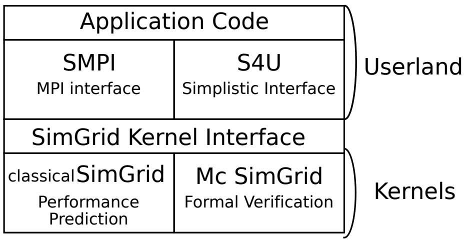
Formal Semantics of the SimGrid Simulator

June 25, 2018

This document tries to formally express the semantic of applications that can be executed in SimGrid, such as MPI applications. The long term goals is to find better reduction algorithms for MPI applications in Mc SimGrid, the model-checker embedded within the SimGrid framework.

SimGrid is a simulator of distributed applications. Several user interfaces are proposed, ranging from the classical and realistic MPI formalism, to less realistic simgrid-specific APIs that ease the expression of theoretical distributed algorithms. These user interfaces are built upon a common interface, that is implemented either on top of a performance simulator, or on top of a model-checker exploring exhaustively all possible outcomes from a given initial situation.



The distributed application is represented in SimGrid as a set of **actors**, representing processes or threads of real applications, or MPI ranks. These actors interact with each other either through message passing, or with classical synchronization objects (such as mutexes or semaphores), or through executions on CPUs and read/write operations on disks.

Even if it simulates distributed applications, SimGrid proposes a shared memory model: all actors share the same memory space. To simulate distributed settings, most of the simulated applications simply ensure that they only use variables that are local to each actor, without any program global variables. Enforcing the memory separation at application level allows the kernel to deal with shared-memory and distributed-memory primitives in the same way. It also permits to partially abstract the studied simulated infrastructure: the distributed services that are not relevant to the study can easily be abstracted as centralized components.

From the formal point of view, a major advantage of the SimGrid framework is that all user interfaces are implemented on top of a very small amount of kernel primitives. In this document, we are interested in formalizing these operations and their inter-dependencies, that will be useful for partial-order reduction methods in model-checking.

This document is organized as follows. Section 1 formally defines the programming model offered by the SimGrid kernel using TLA+. It specifies the semantic of every offered operation types through their effects on the system. Section 3 defines an event system with these operations, exploring the causality, conflict and independence relations between the defined events. Section 5 presents how the MPI semantic is implemented on top of the SimGrid kernel.

1 System State Definition

A distributed system is a tuple $P = \langle \mathsf{Actors}, \mathsf{Network}, \mathsf{Synchronization} \rangle$ in which $\mathsf{Actors} = \{A_1, A_2, ...A_n\}$ is a set of n actors. Actors do not have a global shared memory nor a global clock. The execution of an actor A_i consists of an alternate sequence of local states and actions $s_{i,0} \xrightarrow{a_0} s_{i,1} \xrightarrow{a_1} s_{i,2}... \xrightarrow{a_{n-1}} s_{i,n}$ (firing a_i from local state $s_{i,j}$, the state of actor A_i changes from $s_{i,j}$ to $s_{i,j+1}$). All the actions in one actor are totally ordered by the causal relation. The subsystem Network provides facilities for the Actors able exchange messages with each other while subsystem Synchronization composed of several mutexes to synchronize actors when they access shared resources.

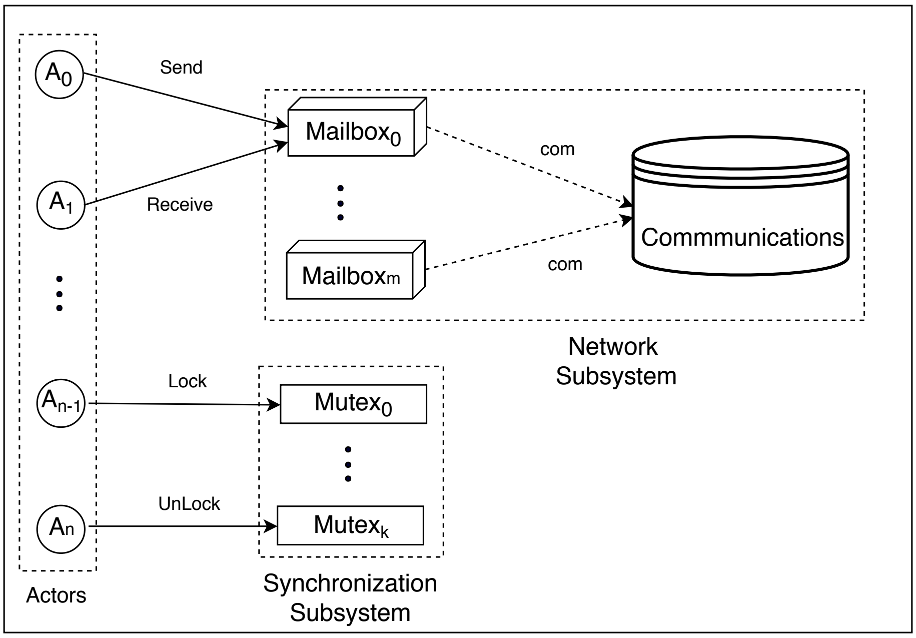


Figure 1: Three main elements in the system: Actors, Network and Synchronization

Specification 1 (TLA+ specification of functions and variables of the system)

We formally describe the system P in a formal specification language called TLA+ [LMTY02] by making the specification for P. Specification 1 is a part of TLA+ specification presenting variables, data structures and functions used for modeling the system. The specification focus on the modeling of the actions and states of the system. To distinguish between actors we use a constant *Actor*, it is treated as a array storing sequence of ids (aId). Each actor has its own memory, and the memory can be accessed through the variable *memory* which indexed by actor ids. Each instruction in the set *Instr* correspond an action defined in the next part of specification. The variable *pc* is an instruction array presenting the current instruction of the actors. Based on the value of the instruction of $pc[aId]$, the correspond action is invoked to execute by the actor with id is Aid. Hence, an actor can execute a consequence of actions by changing their instruction in variable *pc*. Variables Comunications, waitingQueue, Request will be used to expressed the state of Network and Synchronization subsystems while the functions will be called when defining the actions of the actors.

1.1 Network Subsystem

The state of the network subsystem is defined as a pair $\langle \mathsf{Mailboxes}, \mathsf{Communications} \rangle$, where

- $\mathsf{Mailboxes} = \{\mathsf{mailbox}_1, \mathsf{mailbox}_2, ...\mathsf{mailbox}_m\}$ is a set of m mailboxes, each $\mathsf{mailbox}_i$ is an infinite queue storing *send* and *receive* requests of agents, it is considered as a rendez-vous where *send* and *receive* requests meet. For a given mailbox, corresponding requests are stored with a FIFO policy. It means that when a *send* request coming to the mailbox, the oldest *receive* is selected to combine with the coming *send*, producing a ready communication in Communications (the same process for receive requests). Hence, at the same time there are only one kind of pending requests: send or receive requests.

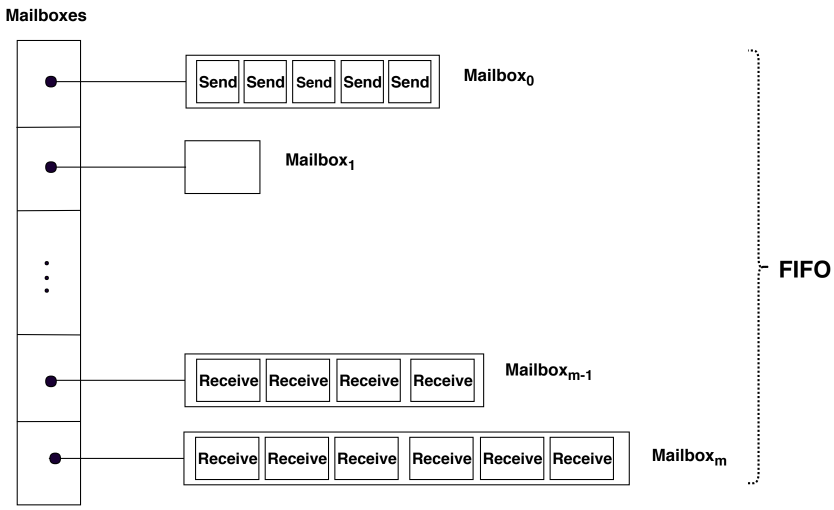


Figure 2: Mailboxes includes m FIFO mailboxes containing send or receive requests

Specification 2 (TLA+ specification of the Mailboxes)

CONSTANTS *NbMailbox*

VARIABLES *Mailboxes*

$Init \triangleq \ \wedge Mailboxes = [i \in NbMailbox \mapsto \{\}]$

- Communications is a set of individual communications, each of them describing a data exchange between two actors. A communication with the "ready" status is formed when a *send* request matches with a *receive* request in a particular mailbox. While a "ready" communicationis ready for exchanging data between two actors, a communication whose status is "done" presents that the data was transmitted and the communication has finished.

Specification 3 (TLA+ specification the Communications)

VARIABLES *Communications*

$Comm \triangleq [id : Nat,$
$mb : NbMailbox,$
$status : \{\text{“ready”}, \text{“done”}\},$
$src : Actors,$

$dst : Actors,$
$data\_src : Addr,$
$data\_dst : Addr]$

$TypeInv \triangleq \land Communications \subseteq Comm$

$Init \triangleq \land Communications = \{\}$

Four action types are defined in Network subsystem to support actors communicate with each other. They are *AsyncSend*, *AsyncReceive*, *WaitAny* and *TestAny*. An actors start a communication by firing a *AsyncSend* or *AsyncReceive*; however, data are really exchanged between two actors after firing *WaitAny* or *TestAny* actions. The specification of the actions in $TLA+$ are as follows:

Four action types are defined in Network subsystem to support actors communicate with each other. They are *AsyncSend*, *AsyncReceive*, *WaitAny* and *TestAny*. An actors start a communication by firing a *AsyncSend* or *AsyncReceive*; however, data are really exchanged between two actors after firing *WaitAny* or *TestAny* actions. The specification of the actions in $TLA+$ are as follows:

- *AsyncSend*

Specification 4 (TLA+ specification *AsyncSend*)

$AsyncSend(Aid, mb, data\_r, comm\_r) \triangleq$
$\land Aid \in Actors$
$\land mb \in NbMailbox$
$\land data\_r \in Addr$
$\land comm\_r \in Addr$
$\land pc[Aid] \in SendIns$

If a matching "receive" request exists in the mailbox(mb), choose the oldest one and complete the Sender fields and set the communication to the "ready" state

$\land \lor \exists\, request \in Mailboxes[mb] :$
$\quad \land request.status = \text{“receive”}$
$\quad \land \forall\, d \in Mailboxes[mb] : d.status = \text{“receive”} \implies request.id \leq d.id$
$\quad \land Communications' =$
$\qquad Communications \cup \{[request \text{ EXCEPT}$
$\qquad\quad !.status = \text{“ready”},$
$\qquad\quad !.src = Aid,$
$\qquad\quad !.data\_src = data\_r]\}$
$\quad \land Mailboxes' = [Mailboxes \text{ EXCEPT } ![mb] = Mailboxes[mb] \setminus \{request\}]$
$\quad \land memory' = [memory \text{ EXCEPT } ![Aid][comm\_r] = request.id]$
$\quad \land \text{UNCHANGED } \langle comId \rangle$

Otherwise (i.e. no matching AsyncReceive communication request exists, create a AsyncSend request and push it in the set Communications.

$\lor \land \neg\exists\, req \in Mailboxes[mb] : req.status = \text{“receive”}$
$\quad \land \text{LET } request \triangleq$
$\qquad [id \mapsto comId,$
$\qquad mb \mapsto mb,$
$\qquad status \mapsto \text{“send”},$
$\qquad src \mapsto Aid,$

$$
\begin{array}{l}
\qquad\qquad\qquad dst \mapsto NoActor, \\
\qquad\qquad\qquad data\_src \mapsto data\_r, \\
\qquad\qquad\qquad data\_dst \mapsto NoAddr] \\
\quad \text{IN} \\
\qquad \land\ Mailboxes' = [Mailboxes \text{ EXCEPT } ![mb] = Mailboxes[mb] \cup \{request\}] \\
\qquad \land\ memory' = [memory \text{ EXCEPT } ![Aid][comm\_r] = request.id] \\
\qquad \land\ \text{UNCHANGED } \langle Communications \rangle \\
\qquad \land\ comId' = comId + 1 \\
\land\ \exists\, ins \in Instr : pc' = [pc \text{ EXCEPT } ![Aid] = ins] \\
\land\ \text{UNCHANGED } \langle waitingQueue, Requests \rangle
\end{array}
$$

An actor drops an asynchronous *send* request to a particular mailbox by firing an *AsyncSend* action. If there are pending *receive* requests in the mailbox, the *send* request will be matched with the oldest receive request to form a communication with "ready" status in the Communications. In the other hand, there is no pending receive in the mailbox, a communication (request) whose status is "send" is created in the Communications and the request is stored in the mailbox.

- *AsyncReceive*

Specification 5 (TLA+ specification of *AsyncReceive*)

$$
\begin{array}{l}
AsyncReceive(Aid, mb, data\_r, comm\_r) \triangleq \\
\land\ Aid \in Actors \\
\land\ mb \in NbMailbox \\
\land\ data\_r \in Addr \\
\land\ comm\_r \in Addr \\
\land\ pc[Aid] \in ReceiveIns
\end{array}
$$

If a matching "send" request exists in the mailbox mb, choose the oldest one and, complete the receiver's fields and set the communication to the "ready" state

$$
\begin{array}{l}
\land\ \lor\ \exists\, request \in Mailboxes[mb] : \\
\qquad \land\ request.status = \text{"send"} \\
\qquad \land\ \forall\, d \in Mailboxes[mb] : d.status = \text{"send"} \implies request.id \leq d.id \\
\qquad \land\ Communications' = \\
\qquad\qquad Communications \cup \{[request \text{ EXCEPT} \\
\qquad\qquad\qquad\qquad\qquad !.status = \text{"ready"}, \\
\qquad\qquad\qquad\qquad\qquad !.dst = Aid, \\
\qquad\qquad\qquad\qquad\qquad !.data\_dst = data\_r]\} \\
\qquad \land\ Mailboxes' = [Mailboxes \text{ EXCEPT } ![mb] = Mailboxes[mb] \setminus \{request\}] \\
\qquad \land\ memory' = [memory \text{ EXCEPT } ![Aid][comm\_r] = request.id] \\
\qquad \land\ \text{UNCHANGED } \langle comId \rangle
\end{array}
$$

Otherwise (i.e. no matching AsyncSend communication request exists), create a "receive" request and push it in the Communications.

$$
\begin{array}{l}
\quad \lor\ \land\ \neg\exists\, req \in Mailboxes[mb] : req.status = \text{"send"} \\
\qquad\qquad \land\ \text{LET } request \triangleq \\
\qquad\qquad\qquad\qquad [id \mapsto comId, \\
\qquad\qquad\qquad\qquad status \mapsto \text{"receive"}, \\
\qquad\qquad\qquad\qquad dst \mapsto Aid, \\
\qquad\qquad\qquad\qquad data\_dst \mapsto data\_r]
\end{array}
$$

$$
\begin{array}{l}
\quad\quad\quad\quad \text{IN} \\
\quad\quad\quad\quad\quad \wedge\ Mailboxes' = [Mailboxes\ \text{EXCEPT}\ ![mb] = Mailboxes[mb] \cup \{request\}] \\
\quad\quad\quad\quad\quad \wedge\ memory' = [memory\ \text{EXCEPT}\ ![Aid][comm\_r] = request.id] \\
\quad\quad\quad\quad\quad \wedge\ \text{UNCHANGED}\ \langle Communications \rangle \\
\quad\quad\quad\quad\quad \wedge\ comId' = comId + 1 \\
\wedge\ \exists\, ins \in Instr : pc' = [pc\ \text{EXCEPT}\ ![Aid] = ins] \\
\wedge\ \text{UNCHANGED}\ \langle waitingQueue, Requests \rangle
\end{array}
$$

Actors use *AsyncReceive* to post an asynchronous receive request on a mailbox; the way a receive request processed is the same as a send request treated. The receive request is combined with a matching send to form a ready communication in Communications. Otherwise, if there is no pending send, a communication that lacks the sender's information is created.

- *TestAny*

Specification 6 (TLA+ specification of *TestAny*)

$$
\begin{array}{l}
TestAny(Aid, comms, ret\_r) \triangleq \\
\wedge\ Aid \in Actors \\
\wedge\ ret\_r \in Addr \\
\wedge\ pc[Aid] \in TestIns \\
\wedge\ \vee\ \exists\, comm\_r \in comms, c \in Communications : c.id = memory[Aid][comm\_r] \wedge \\
\quad\quad \textit{If the communication is "ready" the data is transfered, return ValTrue} \\
\quad\quad \vee\ \wedge\ c.status = \text{"ready"} \\
\quad\quad\quad\quad \wedge\ memory' = [memory\ \text{EXCEPT}\ ![c.dst][c.data\_dst] = \\
\quad\quad\quad\quad\quad\quad\quad\quad memory[c.src][c.data\_src], \\
\quad\quad\quad\quad\quad\quad\quad\quad ![Aid][ret\_r] = ValTrue] \\
\quad\quad\quad\quad \wedge\ Communications' = \\
\quad\quad\quad\quad\quad\quad (Communications \setminus \{c\}) \cup \{[c\ \text{EXCEPT}\ !.status = \text{"done"}]\} \\
\quad\quad \textit{Else if the cummunication is already done, keep Communications unchanged, return ValTrue} \\
\quad\quad \vee\ \wedge\ c.status = \text{"done"} \\
\quad\quad\quad\quad \wedge\ memory' = [memory\ \text{EXCEPT}\ ![Aid][ret\_r] = ValTrue] \\
\quad\quad\quad\quad \wedge\ \text{UNCHANGED}\ \langle Communications \rangle \\
\quad \vee\ \neg\exists\, comm\_r \in comms, c \in Communications : c.id = memory[Aid][comm\_r] \\
\quad\quad \textit{If no communication is "ready" or "done", return ValFalse} \\
\quad\quad\quad\quad \wedge\ c.status \in \{\text{"ready"}, \text{"done"}\} \\
\quad\quad\quad\quad \wedge\ memory' = [memory\ \text{EXCEPT}\ ![Aid][ret\_r] = ValFalse] \\
\quad\quad\quad\quad \wedge\ \text{UNCHANGED}\ \langle Communications \rangle \\
\textit{Test is non-blocking since in all cases pc[AId] is incremented} \\
\wedge\ \exists\, ins \in Instr : pc' = [pc\ \text{EXCEPT}\ ![Aid] = ins] \\
\wedge\ \text{UNCHANGED}\ \langle waitingQueue, Requests, Mailboxes, comId \rangle
\end{array}
$$

Although communications are established in the Communications by the combination of *send* and *receive* requests, data are really transferred between actors by firing *TestAny* or *WaitAny* actions. *TestAny* test a set of communications, returning either true or false depending on the status of the communications. If at least one communication in the set has a "ready" or "done" status, *WaitAny* action returns true, otherwise false is given.

As stated, data exchange is performed in *TestAny* command. If a communication has a "ready" status and it is selected to test then data is copied from the source actor to the destination actor (both actors concern the communication) , and the status of the communication is assigned to "done".

- *WaitAny*

Specification 7 (TLA+ specification of *WaitAny*)

$WaitAny(Aid, comms) \triangleq$
$\quad \land Aid \in Actors$
$\quad \land pc[Aid] \in WaitIns$
$\quad \land \exists comm\_r \in comms, c \in Communications : c.id = memory[Aid][comm\_r] \land$
$\qquad \lor \land c.status = \text{"ready"}$
Data is transfered to destination, then update status of the communication to "done"
$\qquad\quad \land memory' = [memory \text{ EXCEPT } ![c.dst][c.data\_dst] = memory[c.src][c.data\_src]]$
$\qquad\quad \land \quad Communications' = (Communications \setminus \{c\}) \cup \{[c \text{ EXCEPT } !.status = \text{"done"}]\}$
$\qquad \lor \land c.status = \text{"done"}$
$\quad \land \text{UNCHANGED } \langle memory, Communications \rangle$
In both cases, pc[Aid] is incremented
$\quad \land \exists ins \in Instr : pc' = [pc \text{ EXCEPT } ![Aid] = ins]$
$\quad \land \text{UNCHANGED } \langle waitingQueue, Requests, Mailboxes, comId \rangle$

A *WaitAny* action have the same function as *TestAny* actions in transferring data, but it does not return any values. Moreover, it is only enabled if at least one communication in the set has a "ready" status. So, it can blocks it's actor in the case there is no ready communication in the set.

1.2 Synchronization subsystem

The state of the Synchronization subsystem is defined by Mutexes. Mutexes $= \{m_1, m_2 ... m_k\}$ is a set of k asynchronous mutexes. The Mutexes are used to synchronize the actors. An actor A_i declares it's interest on a mutex m_j by executing the action *MutexAsyncLock*(A_i, m_j) while the mutex remembers that interest by adding the id of the actor to it's waiting queue. This queue also follows a FIFO policy. We say that a mutex m is *busy* if there is at least one actor in it's waiting queue, otherwise it is *free*. The state of the Mutexes can be indicated by state of all the mutexes, more formally the state of Mutexes $= \{state_1, state_2 ... state_k\}$ where $state_i \in$ {"free", "busy" }.

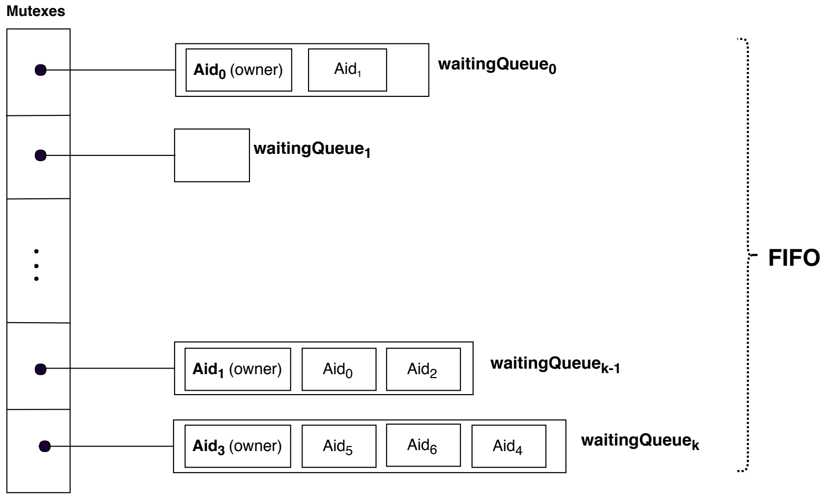


Figure 3: Each mutex uses a waiting queue consisting of actor identifiers

In this model mutexes are asynchronous, similar to communication, in the sense that requesting a mutex is not blocking. In the synchronization subsystem, there are four actions allowing actors to interact with the Mutexes, namely *MutexAsyncLock*, *MutexUnlock*, *MutexWait* and *MutexTest*. They are described formally by *TLA+* as follows:

- *MutexAsyncLock*

Specification 8 (TLA+ specification of *MutexAsyncLock*)

$MutexAsyncLock(Aid, mid, req\_a) \triangleq$
$\land\ Aid \in Actors$
$\land\ pc[Aid] \in LockIns$
$\land\ mid \in Mutexes$
$\land\ req\_a \in Addr$

If Actor ¡Aid¿ has no pending request on mutex ¡mid¿, create a new one

$\land\ \lor\ \land \neg isMember(Aid, waitingQueue[mid])$
$\qquad \land\ Requests' = [Requests \text{ EXCEPT } ![Aid] = Requests[Aid] \cup \{mid\}]$
$\qquad \land\ memory' = [memory \text{ EXCEPT } ![Aid][req\_a] = mid]$
$\qquad \land\ waitingQueue' = [waitingQueue \text{ EXCEPT } ![mid] =$
$\qquad\qquad Append(waitingQueue[mid], Aid)]$

Otherwise i.e. actor ¡Aid¿ already has a pending request on mutex ¡mid¿, keep the variables unchanged

$\quad \lor\ \land\ isMember(Aid, waitingQueue[mid])$
$\qquad \land \text{ UNCHANGED } \langle waitingQueue, memory, Requests\rangle$

MutexAsyncLock is never blocking, in any case, pc[Aid] is incremented

$\land\ \exists\, ins \in Instr : pc' = [pc \text{ EXCEPT } ![Aid] = ins]$
$\land \text{ UNCHANGED } \langle Communications, Mailboxes, comId\rangle$

MutexAsyncLock(A_i, m_j) is executed by an actor A_i when the actor wants to acquire a mutex m_j. After firing *MutexAsyncLock*, the id i of the actor will be added to the tail of the mutex's queue. Hence, if the mutex is free, the actor is the first in the mutex's waiting queue, and it becomes the *owne* of the mutex, otherwise it is a *waiting* actor. However, unlike classical mutexes, when an actor is waiting for a mutex, it is not blocked, and to identify which mutexes it has asked for, the mutex's id j is added to it's *Requests* set .

TJ: add this in the actors state?

- *MutexUnlock*

Specification 9 (TLA+ specification of *MutexAsyncLock*)

$MutexUnlock(Aid, mid) \triangleq$
$\land\ Aid \in Actors$
$\land\ mid \in Mutexes$
$\land\ pc[Aid] \in UnlockIns$

If ¡Aid¿ makes a "valid" unlock on ¡mid¿ (either owner or not) remove any linking between them

$\land\ isMember(Aid, waitingQueue[mid])$
$\land\ waitingQueue' = [waitingQueue \text{ EXCEPT } ![mid] = Remove(Aid, waitingQueue[mid])]$
$\land\ Requests' = [Requests \text{ EXCEPT } ![Aid] = Requests[Aid] \setminus \{mid\}]$
$\land\ \exists\, ins \in Instr : pc' = [pc \text{ EXCEPT } ![Aid] = ins]$
$\land\ \text{UNCHANGED } \langle memory, Communications, Mailboxes, comId \rangle$

MutexUnlock is used to remove an interest on a mutex by an actor. Either the actor is the owner or not, this command can be fired by the mutex, deleting the id of the actor from the mutex's queue and removing the mutex's id from actor's request set.

- *MutexTest*

Specification 10 (TLA+ specification of *MutexTest*)

$MutexTest(Aid, req\_a, test\_a) \triangleq$
$\land\ Aid \in Actors$
$\land\ pc[Aid] \in MtestIns$
$\land\ test\_a \in Addr$
$\land\ \exists\, req \in Requests[Aid] : req = memory[Aid][req\_a] \land$

If the actor is the owner then return true

$\quad \lor \land\ isHead(Aid, waitingQueue[req])$
$\quad\quad \land\ memory' = [memory \text{ EXCEPT } ![Aid][test\_a] = ValTrue]$

Else if it is not the onwer then return false

$\quad \lor \land\ \neg isHead(Aid, waitingQueue[req])$
$\quad\quad \land\ memory' = [memory \text{ EXCEPT } ![Aid][test\_a] = ValFalse]$

$\land\ \exists\, ins \in Instr : pc' = [pc \text{ EXCEPT } ![Aid] = ins]$
$\land\ \text{UNCHANGED } \langle waitingQueue, Requests, Communications, Mailboxes, comId \rangle$

An actor can check if it is the owner of a mutex that he has previously asked for access to (id of the mutex is included in the actor's request set). To realize this, it can use the *MutexTest* action, returning true if the id of the actor is the first element of the mutex's waiting queue, otherwise the returned value is false.

- *MutexWait*

Specification 11 (TLA+ specification of *MutexWait*)

$MutexWait(Aid, req\_a) \triangleq$

$\wedge\ Aid \in Actors$
$\wedge\ req\_a \in Addr$
$\wedge\ pc[Aid] \in MwaitIns$
$\wedge\ \exists\, req \in Requests[Aid] : req = memory[Aid][req\_a] \wedge isHead(Aid, waitingQueue[req])$
$\wedge\ \exists\, ins \in Instr : pc' = [pc \text{ EXCEPT } ![Aid] = ins]$
$\wedge\ \text{UNCHANGED } \langle memory, waitingQueue, Requests, Communications, Mailboxes, comId\rangle$

While a *MutexTest* can be fired by an actor without a condition ensuring the actor is the owner, a *MutexWait* will not return until it's actor owns the mutex passed to it. Hence, a waiting actor can be blocked when trying to execute a *MutexWait* action.

1.3 Summary

Actor = local state (containing variables and PC) + program (sequence of actions)

Subsystem	Network	Synchronization
Resource	Mailbox	Mutex
Activity	Communication	Request
Actions	Send=*AsyncSend* +*WaitAny* Recv=*AsyncReceive* +*WaitAny* *TestAny*	Lock=*MutexAsyncLock* +*MutexWait* *MutexUnlock* *MutexTest*

TAP: Please review the next paragraph, I'm not confident with this definition of LocalCompute.

Beside of the mentioned actions, a program in SimGrid can have local computations named *LocalComputation* actions. Such actions do not intervene with shared objects (Mailboxes, Mutexes and Communications), and they can be responsible for I/O tasks. it's specification is as follows:

Specification 12 (TLA+ specification of *LocalComputation*)

$Local(Aid) \triangleq$
$\wedge\ Aid \in Actors$
$\wedge\ pc[Aid] \in LocalIns$
Change value of memory[Aid][a]
$\wedge\ memory' \in [Actors \rightarrow [Addr \rightarrow Nat]]$
$\wedge\ \forall\, p \in Actors, a \in Addr : memory'[p][a] \neq memory[p][a]$
Ensure that memory[Aid][a] is not where actor Aid strores communication
$\implies p = Aid \wedge a \notin CommBuffers(Aid)$
$\wedge\ \exists\, ins \in Instr : pc' = [pc \text{ EXCEPT } ![Aid] = ins]$
$\wedge\ \text{UNCHANGED } \langle Communications, waitingQueue, Requests, Mailboxes, comId\rangle$

$Local(Aid) \triangleq$
$\wedge\ Aid \in Actors$
$\wedge\ pc[Aid] \in LocalIns$
Change value of memory[Aid][a]
$\wedge\ memory' \in [Actors \rightarrow [Addr \rightarrow Nat]]$
$\wedge\ \forall\, p \in Actors, a \in Addr : memory'[p][a] \neq memory[p][a]$
Ensure that memory[Aid][a] is not where actor Aid strores communication
$\implies p = Aid \wedge a \notin CommBuffers(Aid)$

$\land \exists\, ins \in Instr : pc' = [pc \text{ EXCEPT } ![Aid] = ins]$
$\land$ UNCHANGED $\langle Communications, waitingQueue, Requests, Mailboxes, comId\rangle$

---

2 Independence theorems

Let enabled(s) presents the set of actions enabling at state s. We have the definition of independence as follows:

Definition 1
$\mathbf{I(a_1, a_2)}$ *denotes that actions* a_1 *and* a_2 *are independent. This is true if they satisfy following conditions ([God96]):*

(Def 1.1) The execution order of independent actions does not change their overall result.

$\forall s$ where $a_1, a_2 \in enabled(s), \exists$ uniq s' such that $(s \xrightarrow{a_1 a_2} s' \land s \xrightarrow{a_2 a_1} s')$.

(Def 1.2) Executing a given action does not enable nor disable any action that is independent with it.

$\forall s$ where $a_1 \in enabled(s)$ and $s \xrightarrow{a_1} s'$, $(a_2 \in enabled(s) \Leftrightarrow a_2 \in enabled(s'))$

Definition 2
$\mathbf{D(a_1, a_2)}$ *denotes that actions* a_1 *and* a_2 *are not independent. They are said to be dependent.*

In the following, we prove several independence theorems, specifying classes of actions that are always independent in our M_P system. Here, we only give a sketch of the proof of the theorems while the full proof (based on the TLA$^+$ of actions) is deferred to Appendix X.

Theorem 2.1
An AsyncSend action and an AsyncReceive action are independent.

Proof. Let a_s and a_r be respectively be an *AsyncSend* and an *AsyncReceive* actions. If a_s and a_r occur on differing mailboxes, they are trivially independent because there is not shared state between differing mailboxes.

Let's assume that they occur on the same mailbox. Let's prove that Def 1.1 is true in all cases. We use the fact that the mailbox contains a FIFO queue which can either be empty, or contain only send actions, or only receive actions. (i) If the mailbox's queue initially is empty, a_s and a_r will be matched together. (ii) If the mailbox contains send actions before a_s and a_r are triggered, a_r will be matched with the first of the pre-existing send actions and a_s will be added to the tail of the queue. (iii) Conversely, if the mailbox initially contains receive actions, a_s will be matched with the first of these recv actions, and a_r will be queued. In all cases, the outcome does not depend on the relative order of a_s and a_r, and Def 1.1 is true.

In addition, since send and receive actions are never disabled once they exist, Def 1.2 is trivially true. We thus conclude that $I(a_s, a_r)$. □

Theorem 2.2
Two AsyncSend actions, or two AsyncReceive actions sending requests to different mailboxes are independent.

Proof. Since two *AsyncSend* concern different mailboxes, they change the state of different mailboxes. So, changing their execution orders get the same overall state. Besides, an *AsyncSend* can neither disable nor enable other *AsyncSend*. For those reasons two *AsyncSend* are independent. Similarly, two *AsyncReceive* actions are trivially independent. □

Theorem 2.3
Two WaitAny actions, two TestAny actions, a WaitAny action and a TestAny action are independent.

Proof. Let's start with two *WaitAny* actions. Let w_1 and w_2 be the first *WaitAny* and the second *WaitAny* respectively. (i) Assuming that w_1 waits communication com_1 while w_2 waits communication com_2. With any execution order, the status of com_1 changes to "done" after executing w_1 while the status of com_2 changes to "done" because of firing w_2. Data is copied from senders to receivers. (ii) If the actions are related to the same communication, the execution of w_1 forces the status of the communication to "done". Data of the sender is copied to the receiver. After that w_2 is executed. Due to the status of the communication is "done", w_2 returns and nothing happens afterward. Conversely, if executing w_2 before w_1, the status also changes to "done" and data is still sent, nothing different from the previous order. From the above claims, changing the execution order get the same final state, Def 1.1 is true. Besides, because two *WaitAny* actions can not enable or disable each other, Def 1.2 is true. Hence, we have $\mathrm{I}(w_1, w_2)$. Similarly, we have the proof for two *TestAny* actions, and for a *WaitAny* and a *TestAny*. □

Theorem 2.4
A WaitAny action or a TestAny action and a AsyncSend action or a AsyncReceive action are independent if they concern different communication.

Proof. Let's start with a *WaitAny* and an *AsyncSend* . (i) If they concern different mailboxes, they are trivially independent since there is not shared states between mailboxes and one can not enable or disable other. (ii) Assuming they concern the same mailbox. In the first order we execute the *WaitAny* before *AsyncSend*. The status of communication that waited by *WaitAny* is changed to "done", the data is sent. After that a send request posted to the mailbox. Depending on the state of the mailbox (empty or not), the send request can be matched with a pending receive request, or it will be queued into the mailbox. In the reverse execution order, we get the same outcome, the send request is treated similarly, the status change to "done", and the data is copied from the sender to the receiver. It means that the final outcome is not effected by the execution orders. Concerning the Def 1.2, it is true since both actions can not enable or disable other. Hence, they are independent.
The proof for a *WaitAny* and an *AsyncReceive*, a *TestAny* and an *AsyncSend*, or a *TestAny* and an *AsyncReceive* are similar.

□

Theorem 2.5
Two MutexAsyncLock actions are independent if they concern different mutexes.

Proof. Since two actions modify the state of different mutexes, the final outcome is stable. Changing the execution order obtain the same final state. Besides, the Def 1.2 is always true for the pair actions. Hence, we conclude the theorem. □

Theorem 2.6
A MutexAsyncLock action and MutexUnlock action are independent.

Proof. If they touch different mutexes, they are trivially independent since there is no shared state, and they are enabled and disabled independently. In contrary, they touch the same mutex. We firstly examine the execution order where *MutexAsyncLock* execute before *MutexUnlock*. In this case the id of the actor that fire the *MutexAsyncLock* will be added to the mutex's waiting queue while the id of the mutex is removed from the *Requests* of the actor that fires the

MutexUnlock. Similarly, with the reverse order we get the same outcome. In addition, the Def 1.2 is true, sine they can not enable and disable other. ☐

Theorem 2.7
A MutexAsyncLock action or MutexUnlock action is independent with a AsyncSend action, a AsyncReceive action, a TestAny action or a WaitAny action.

Proof. We start by proving the independence relation between a *MutexAsyncLock* an *AsyncSend*. Similar to Theorem 2.5, two actions modify state of different objects. While the *MutexAsyncLock* modifies the state of a mutex, the *AsyncSend* modifies the state of a mailbox and Communications. Hence, we get the same final state. Besides, one action can not be disabled or enabled by firing other one. So, they are independent. Similarly, A *MutexAsyncLock* action is independent with a *AsyncReceive* action, a *TestAny* action or a *WaitAny* action. ☐

Theorem 2.8
Two MutexWait actions, two MutexTest actions, a MutexWait action and MutexTest action are independent.

Proof. Regarding to two *MutexWait* actions, they just modify local state of their actors. So, commuting their execution does not change the modification. Besides, one are not enabled or disabled by the execution of other one. Hence, they are independent. The proof for two *MutexTest* actions, or a *MutexWait* action and *MutexTest* action is similar. ☐

Theorem 2.9
A MutexAsyncLock action and a MutexWait or MutexTest action are independent.

Proof. Let's prove for a *MutexAsyncLock* and a *MutexWait*. Not depend on the order of execution, after executing the *MutexAsyncLock*, the id of the mutex is added to *Requests* of the actor that fires the *MutexAsyncLock* while the id of the actor is queued to mutex's waiting queue. The state of *MutexWait*'s actor is changed because the actor's PC chages to next action. From the above mentioned, we obtain the final outcome although we change the execution order. In addition, with these two actions, the Def 1.2 is true since the execution of *MutexAsyncLock* is not effected by *MutexWait* and reverse. Hence, we the theorem is proved. For the pair *MutexAsyncLock* and *MutexTest* is proved similarly. ☐

Theorem 2.10
A LocalComputation action is independent with all other actions.

Proof. *LocalComputation* is trivially independent with other actions sine they modify different part of the system, and one do not disable or enable other one. ☐

3 SimGrid simulations as an Event System

This section we present how we build the unfolding [RSSK15], a Labelled Prime Event Structure (LES for short) of a concurrent program under an independence relation, for our distributed system P by adapting the method in [NRS$^+$18]. We start by defining happen before relations considered as causality relation in LESs. After that, we recap the definition of LES, and finally all steps for constructing an unfolding are described with a simple example illustrating the steps.

3.1 Happened-before relation

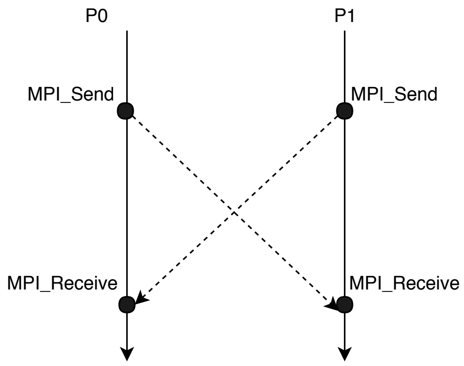
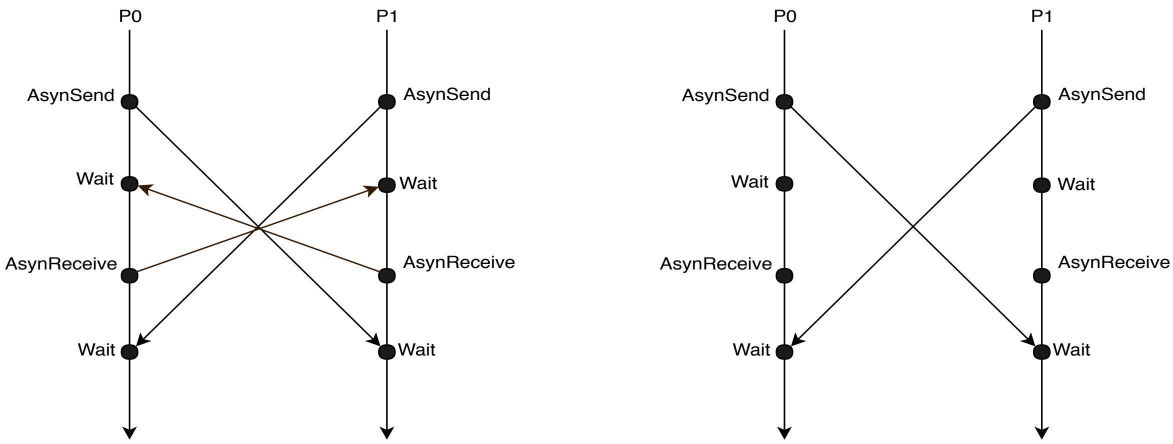


Figure 4: A MPI program with a potential deadlock

In SimGrid, actions are atomic and actions in the same actor are totally ordered, but actions in the whole system are partially ordered (partial order relation). Hence, there are different instances (runs) of the system, and the relation between actions are considered in a given instance. To adapt to the reality, happened-before relation in SimGrid is flexible. Let's look at an example in Figure 3.1. A MPI program including two processes, and process P0 sends a massage to process P1 (doted line). Before receiving the massage from P0, P1 also sends a massage to P0. At first glance, we may think that there is a deadlock in the program since there is a dependency cycle. However, in practice, depending on the size of the massages exchanged by the processes, the deadlock may appears or not. MPI_Send and MPI_Receive are blocking functions in MPI, but they may or may not block. MPI_Send is ambiguously define, it will not return until the buffer passed to it can be reused. For sending small enough massages, those massages eagerly sent before the calling MPI_Receive from receiving processes (eager protocol). Hence, MPI_Sends are not blocked until matching MPI_Receive have been posted. Actually, if the massages are small, MPI can easily find spaces in internal storage to save them before they are really sent. On the other hand, working with large massages, blocking communications are used, MPI_Sends must wait for matching MPI_Receives. Therefor, the scenario in Figure 3.1 may or may not has a deadlock. SimGrid covers both cases, users can chose optionally two modes detecting or not deadlocks in the same situation with the above scenario. This conversion can be done easily by switching between two happened_before definitions.

Definition 3
The happened-before relation denoted by $\rightarrow$ can be defined based on two relations immediate precede and remotely precede:

- *Immediate precede (denoted by $\prec$) : Two actions e and f in the same actor A_i, $e \prec f$ if the occurrence of e precedes the occurrence of f in the actor A_i*
- *Remotely precede (denoted by $\rightsquigarrow$): Action e is the* AsyncSend *or* AsyncReceive *action of actor A_i, action f is the Wait action of the actor A_j, $e \rightsquigarrow f$ if e and f concerns the same communication request.*
- *The 'happened before' is a transitive relation including immediate precede and remotely precede.*



(a) deadlock (b) dealock free

Figure 5: Happened- before relations between actions

The diagram in Figure 3.1(a) illustrates happened_before relations (denoted by arrow lines) between actions in the program based on Definition 3. In SimGrid a MPI_Send is simulated by a *AsyncSend* and a *WaitAny* while a MPI_Receive comprises a *AsyncReceive* and a *WaitAny*. Obviously, there is a happened_before relation cycle in the diagram; the cycle includes the first *WaitAny* and *AsyncReceive* of P0, the first *WaitAny* and the *AsyncReceive* of P1. The existing of cycle leads to a deadlock in the program. In the case we do not want to capture the deadlock, Definiton 4 is used.

Definition 4
The happened-before relation denoted by $\rightarrow$ can be defined based on two relations immediate precede and remotely precede:

- *Immediate precede (denoted by $\prec$) : Two actions e and f in the same actor A_i, $e \prec f$ if the occurrence of e precedes the occurrence of f in the actor A_i*
- *Remotely precede (denoted by $\rightsquigarrow$): Action e is the AsyncSend action of actor A_i, action f is the Wait action of the actor A_j, $e \rightsquigarrow f$ if e and f concerns the same communication request.*
- *The 'happened before' is a transitive relation including immediate precede and remotely precede.*

Using Definition 4 and presenting the happened_before relation between actions of the program in Figure 3.1(b), we can see that there are no any cycle, then the program is deadlock-free.

The happened-before relation is not a total order on the actions, two actions may not related by a happened-before relation. In that case, we say that they are concurrent denoted by the symbol $\|$. For example, in the above figures, since $\neg$(*AsyncSend* of P0 $\rightsquigarrow$ *AsyncSend* of P1) and $\neg$(*AsyncSend* of P0 $\prec$ *AsyncSend* of P1) then *AsyncSend* of P0 $\|$ *AsyncSend* of P1.

3.2 Event structures

Labelled Prime Event Structure (PES for short). This section recaps the definition of labelled event structures [RSSK15]

Definition 5
A labelled event structure on a set label L is a tuple $\mathcal{E} = \langle E, <, \#, h \rangle$ where

- *E is a set of events.*
- *$<$ is a partial order relation on E, called causality relation.*
- *$h : E \to L$ is a labeling function assigning each event in E to a label in L.*
- *# is an irreflexive symmetric relation called conflict relation such that for every event e $\in$ E, the set causes(e) = { e' $\in$ E: e'< e} is finite (the set of predecessors of e is finite), and for every events e, f, g $\in$ E, if e # f and g < f then e # g.*

Intuitively, the causality relation expresses the happened-before while two events are *conflict* if they are not in one executions. Each label in L is an action, and if two events neither are *conflict* nor *causality* related, they are *concurrent*. An important notion in PESs is *configuration*. A subset of events C of E is a *configuration* if for every event e $\in$ C, *causes*(e) $\subseteq$ C (all events in causes of e belong to C) and $\forall e, e' \in C, \neg(e \# e')$ (there is no conflict relation in C). We use $Conf(\mathcal{E})$ to present the set of all configuration in $\mathcal{E}$.

4 Unfolding Semantics

Unfolding Semantics (Unfolding for short). Unfolding Semantics is proposed in [NRS+18] and can describe behavior of a distributed system under independence relations. Indeed, a unfolding is a LES where each maximal configuration corresponds to a Mazurkiewicz trace. In the *unfolding*, each event e is presented by a pair $e = \langle a, H\rangle$ where a is an action in P, and H is a history of the event e, denoting that action t occurs after the history H. For a given configuration C, an event e is called maximal event of C if event e is not in the history of other events, more formally e is a maximal event if $\nexists\ e' \in C$ such that $e < e'$. Let $MaxEvt(C)$ is a set comprised of all maximal events of C. Formally, $MaxEvt(C) = \{e \in C$: $\nexists\ e' \in C$ and $e < e'$ }. We use $state(C)$ to express the state of application P obtained by executing all actions related to events in C and keep the causality relations in C, and $enabled(state(C))$ denotes all actions that enable at $state(C)$

For a given distributed system P and independence relation between actions in P, the unfolding of P denoted by $U_P = \langle E, <, \#, h\rangle$, the Algorithm 1 illustrates how we build the unfolding of P.

Algorithm 1: Building unfolding[NRS+18]

> Set all elements E, $<$, #, and h are empty
> **repeat**
>   **foreach** *subset* E' $\in$ E *and* E' *is a configuration* **do**
>   **foreach** *action* a $\in$ $enabled(state(E'))$ **do**
>   **if** $D(a, h(e'))$ *holds for all* $e' \in$ $MaxEvt(E')$ **then**
>   1. Add a new event e = $\langle a, E'\rangle$ to E.
>   2. Update $<$, # and h as following:
>   – set $e' < e$ if $e' \in E'$.
>   – set $e' \# e$ if $D(a, h(e')$ and e' $\in$ $E \setminus E$'.
>   – set $h(e) = a$.
>   **end**
>   **end**
>   **end**
> **until** *No new event added to* E;

Figure 4 displays a distributed program P composed of three actors. Actor0 and Actor2 send a *send* request to the $mailBox1$ while Actor1 posts a *receice* request on the $mailBox1$. After sending the request, both Actor0 and Actor1 wait the request by firing a *WaitAny* command. Actor1 also declares his interest on the mutex $mutex1$ by executing *MutexAsyncLock*. The unfolding of the program is described in the right of Figure 4. At initial step (denoted by #) $U_P = \langle \emptyset, \emptyset, \emptyset, \emptyset \rangle$. There is only one configuration C $= \emptyset$ in $Conf(U_p)$. After that events $e_1 = \langle \mathit{AsyncSend}, \emptyset \rangle$, $e_2 = \langle \mathit{AsyncReceive}, \emptyset \rangle$ and $e_3 = \langle \mathit{AsyncSend}, \emptyset \rangle$ are created. We create those events because they have not precede events, and there is no maximal event in C to check the dependence condition. We now have four configurations $\{e_1\}$, $\{e_2\}$, $\{e_3\}$, $\{e_1, e_2\}$, $\{e_2, e_3\}$ and $\emptyset$. For the configuration $\{e_1, e_2\}$ we can create event e_6 since $pre(e_6) = e_1$ ($e_1 \in \{e_1, e_2\}$), the communication is ready, and action $\langle 0, \mathit{WaitAny} \rangle$ is dependent with $h(e_1)$ and $h(e_2)$. Similarly, we can create events e_4, e_5, e_7, e_8, e_9 and e_{10}. Note that, we have a configuration $\{e_2, e_3, e_8\}$, the maximal events of this configuration are e_2 and e_8, and $h(e_2)$ $h(e_8)$ are dependently related to action $\langle 0, \mathit{WaitAny} \rangle$; however, we can not create a new event by combining that configuration with action $\langle 0, \mathit{WaitAny} \rangle$ because the communication *com* is not ready for processing in this configuration. The communication *com* is not ready since there is no pending receive request to march with sending request of Actor0, the receive request already marched with the sending request of Actor2. We also have the causality relations depicted by arrows, for example $e_1 < e_4$, $e_1 < e_5$, $e_2 < e_6$. Events belong to different configurations are related by conflict relations, for example $e_1 \# e_3$, $e_5 \# e_7$ or $e_3 \# e_9$. In this unfolding, there are two maximal configurations, they are $\{e_1, e_2, e_4, e_5, e_6, e_9\}$, $\{e_2, e_3, e_7, e_8, e_{10}\}$, and they correspond two Mazurkiewicz traces of the system.

```
Actor0:     com = AsynSend(mailBox1)
                      Wait(com)

Actor1:   com = AsynReceive(mailBox1)
                      Wait(com)
                 MutexAsyncLock(mutex1)

Actor2:        AsynSend(mailBox1)
```

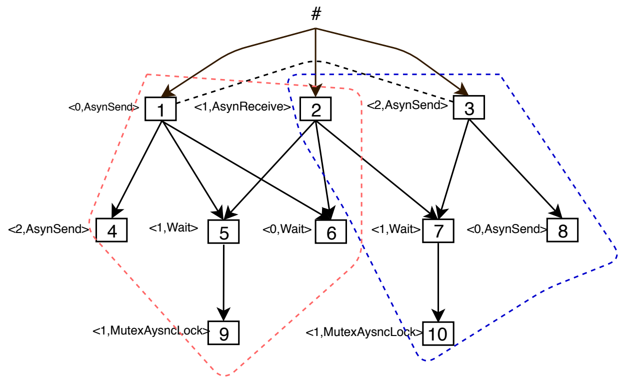


Figure 6: A (toy) program (left), and it's unfolding semantics (right)

4.1 Determining enabled actions

In Algorithm 1, building unfolding needs compute enabled transitions at state of configurations. Given a configuration C, determining enabled action at $state(C)$ can be done trivially by executing all actions related to events in C to obtain the $state(C)$ (executing from initial state of P and preserving causality in C), and then checking enabled actions at $state(C)$. However, this solution is very expensive since normally the size of C is large. This section we introduce an efficient method to determining enabled actions at a state of a configuration.

For an action a in actor A, let $pre(a)$ denotes the action is right before a in the actor A, and $next(a)$ refers to the next action after a. For event $e = \langle a, C \rangle$, we use a_e to imply the action a that related to event e. Given a configuration C and actor A, we use $maximalEvent(A, C)$ imply an event $e = \langle a, C \rangle$ in C such that $\nexists$ an event $e' = \langle a', C \rangle$ in C such that $pre(a') =$

a, and in this case we say that event e is the maximal event of actor A in C. Intuitively, $maximalEvent(A, C)$ is the last occurrence of actor A in the configuration C

In the most cases, an action a of actor A is enabled at state of C if it is the next action of the action related to the maximal event of A in C. For example, in the Figure 1, action *MutexWait* is enabled at $state(\{e_2, e_3, e_7\})$ since $maximalEvent(Actor1, C) = e_7$ and $next($*WaitAny*$) =$ *MutexWait* (here *WaitAny* is the action of event e_7). However, in some other cases, the above condition is not enough to ensure an action becomes enabled. For example, action *WaitAny* of Actor0 is not enabled at $state(\{e_2, e_3, e_7, e_8, e_{10}\})$ although $maximalEvent(Actor0, C) = e_8$ and $next($*AsyncSend*$) =$ *WaitAny*. The action *WaitAny* is not enabled sine the communication *com* is not ready. Given a *WaitAny* action and assume it wait a *send* request, to ensure it is enabled at a configuration, there is a available *receive* can marches with the *send*. We can compare the number of receive request (they must concern the same mailbox with the send request) in the configuration with the number of send request (concern the same mailbox with the send) in the history of event whose action is the send. If the later number is smaller than the former one then the *WaitAny* is enabled at the configuration.

4.2 UDPOR

TAP: Present Unfolding DPOR here, the most important here is how to create new events from current configuration C. From the theory in the paper, for each subset of C, check which actions are enabled at state($subset_i$(C)). however, how to have the state state($subset_i$(C)). The worst case is try to run all actions in $subset_i$(C), but it will be very expensive. A smarter solution is based on happended-before relation

5 MPI Implementation

TODO: explain here how MPI is implemented on top of the previously described API

References

[AAJS14] Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos F. Sagonas. Optimal dynamic partial order reduction. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 373–384, 2014.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008.

[FG05] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 110–121, 2005.

[God96] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer, 1996.

[GW93] Patrice Godefroid and Pierre Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. *Formal Methods in System Design*, 2(2):149–164, 1993.

[LMTY02] Leslie Lamport, John Matthews, Mark R. Tuttle, and Yuan Yu. Specifying and verifying systems with TLA+. In *Proceedings of the 10th ACM SIGOPS European Workshop, Saint-Emilion, France, July 1, 2002*, pages 45–48, 2002.

[NRS+18] Huyen T. T. Nguyen, César Rodríguez, Marcelo Sousa, Camille Coti, and Laure Petrucci. Quasi-optimal partial order reduction. *CoRR*, abs/1802.03950, 2018.

[Pel98] Doron A. Peled. Ten years of partial order reduction. In *Computer Aided Verification, 10th International Conference, CAV '98, Vancouver, BC, Canada, June 28 - July 2, 1998, Proceedings*, pages 17–28, 1998.

[PJQ17] Anh Pham, Thierry Jéron, and Martin Quinson. Verifying MPI applications with simgridmc. In *Proceedings of the First International Workshop on Software Correctness for HPC Applications, CORRECTNESS@SC 2017, Denver, CO, USA, November 12, 2017*, pages 28–33, 2017.

[RSSK15] César Rodríguez, Marcelo Sousa, Subodh Sharma, and Daniel Kroening. Unfolding-based partial order reduction. In *26th International Conference on Concurrency Theory, CONCUR 2015, Madrid, Spain, September 1.4, 2015*, pages 456–469, 2015.

[Val89] Antti Valmari. Stubborn sets for reduced state space generation. In *Advances in Petri Nets 1990 [10th International Conference on Applications and Theory of Petri Nets, Bonn, Germany, June 1989, Proceedings]*, pages 491–515, 1989.

Appendices

Theorem .1
An $\mathsf{AsyncSend}$ action and an $\mathsf{AsyncReceive}$ action are independent.

Proof. Suppose both *AsyncSend* and *AsyncReceive* operations are enabled at a state s = ⟨ Communications, Mailboxes, memory, pc, waitingQueue, *Requests* ⟩. Let a_s and a_r be respectively the *AsyncSend* and the *AsyncReceive* actions, and let $actor_s$ and $actor_r$ be the actors execute a_s and a_r restrictively. Let's firstly prove that Def 1.1 is true.

(i)If they occur on different mailboxes, and suppose that a_s occur on $\mathsf{mailbox}_i$ and a_r occur on $\mathsf{mailbox}_j$. Let's check a situation where a_s is followed by a_r. We have $s \xrightarrow{a_s} s_1 \xrightarrow{a_r} s_2$, where the state s_1 = ⟨$\mathsf{Communications}_1$, $\mathsf{Mailboxes}_1$, memory, pc_1, *waitingQueue*, *Requests* ⟩ and and the state s_2 = ⟨$\mathsf{Communications}_2$, $\mathsf{Mailboxes}_2$, memory, pc_2, *waitingQueue*, *Requests* ⟩. When firing a_r, depending on the state of $\mathsf{mailbox}_i$, the send request can be either added to the LIFO queue of the $\mathsf{mailbox}_i$ if there is no pending request receive, or marched with the first receive request in the queue to form a ready communication *comm* in Communications. Hence, in the state s_1, Communications and Mailboxes are replaced by $\mathsf{Communications}_1$ ($\mathsf{Communications}_1$ = Communications $\cup$ $\{comm\}$) and $\mathsf{Mailboxes}_1$ respectively. Similarly, the request receive a_r is treated in the same way, it can be queued or marched with a pending send in the $\mathsf{mailbox}_j$. In reverse, executing a_s before a_r obtain the same outcome (state s_2).

(ii) If both the request are posted on $\mathsf{mailbox}_i$, we have following cases:

- If the mailbox is empty, $s \xrightarrow{a_s} s_1$ where the state s_1 = ⟨$\mathsf{Communications}_1$, $\mathsf{Mailboxes}_1$, memory, pc_1, *waitingQueue*, *Requests* ⟩ in which $\mathsf{mailbox}_i = \{a_r\}$, the program counter array transform from pc (at sate s) to pc_1 (at state s_1) sine the program counter of a_s changes to the next instruction. After that firing a_r from s_1 we have $s_1 \xrightarrow{a_r} s_2$ where s_2 = ⟨$\mathsf{Communications}_2$, $\mathsf{Mailboxes}_2$, memory, pc_2, *waitingQueue*, *Requests* ⟩. Since there is a pending send request on the mailbox (a_s), the request a_r is marched with the send request to create a ready communication *comm* in Communications($\mathsf{Communications}_1$ = Communications $\cup$ $\{comm\}$). If commuting the execution between a_s and a_r, we obtain the same final state (state s_2).

- If the are some pending sends in the mailbox $mailbox_i$, $s \xrightarrow{a_s} s_1$ where the state s_1 = ⟨$\mathsf{Communications}_1$, $\mathsf{Mailboxes}_1$, memory, pc_1, *waitingQueue*, *Requests* ⟩. The send request a_s is added to the tail of the mailbox ($\mathsf{mailbox}_i$ = Append ($mailbox_i$, $\{a_s\}$). After that firing a_r at s_1, $s_1 \xrightarrow{a_r} s_2$ where the state s_2 = ⟨$\mathsf{Communications}_2$, $\mathsf{Mailboxes}_2$, memory, pc_2, *waitingQueue*, *Requests* ⟩. Because there are some pending send requests in the mailbox, a_r is combined with the first pending send to construct a ready communication in the Communications. We also reach the final state if apply the reverse order.

- If the are some pending receive requests in the mailbox $mailbox_i$, $s \xrightarrow{a_s} s_1$ where the state s_1 = ⟨$\mathsf{Communications}_1$, $\mathsf{Mailboxes}_1$, memory, pc_1, *waitingQueue*, *Requests* ⟩. The send request a_s is marched with the first pending receive, creating a ready communication in Communications. When a receive request arriving to the mailbox because of executing a_r, it will be appended to the tail of the queue (($\mathsf{mailbox}_i$ = Append ($mailbox_i$, $\{a_r\}$)). If executing a_r before a_s we also obtain the overall state. It means that the final outcome state is not effected by the execution orders.

For the second condition (Def 1.2), based on the condition making a_s enabled

```
(/\ rdv \in RdV /\ data_r \in Addr/\ comm_r \in Addr /\ pc[aId] \in SendIns)
```

the send action a_r can not be enabled or disabled by the receive action a_r, and vice versa a_r is not controlled by a_s.

$\square$

Theorem .2
Two $\mathsf{MutexAsyncLock}$ *actions are independent if they concern different mutexes.*

Proof. We will prove that both conditions of the Definition 1 are satisfied. Let's prove that Def 1.1 is true. Suppose both $\mathsf{MutexAsyncLock}$ (a_1, m_1) and $\mathsf{MutexAsyncLock}$ (a_2, m_2) operations are enabled at a state s = ⟨ Communications, Mailboxes, memory, pc, waitingQueue, Requests ⟩. We firstly examine a execution order where $\mathsf{MutexAsyncLock}$ (p_1, m_1) is executed before $\mathsf{MutexAsyncLock}$ (p_2, m_2), and after that, reverse order is checked. We have four cases as follows.

- If the id of $actor_1$ and $actor_2$ are in included in waitingQueue[m1] and waitingQueue[m2] respectively. This is the simplest situation, and in any order, there is nothing change in the system except the program counter of actor a1 and a2 move to the next instruction.
- If the id of $actor_1$ is not in waitingQueue[m1] and the id $actor_2$ is included waitingQueue[m2]. We have $s \xrightarrow{\mathsf{MutexAsyncLock}(p_1, m_1)} s_1$, where s_1 = ⟨ Communications, Mailboxes, memory, pc_1, $waitingQueue_1$, Requests ⟩, in which the queue $waitingQueue_1[m_1]$ contains the id of $actor_1$, and the program counter of $actor_1$ changes to next instruction. After that, executing $\mathsf{MutexAsyncLock}$ (a_2, m_2) only replaces the program counter of $actor_2$ by the next instruction: $s_1 \xrightarrow{\mathsf{MutexAsyncLock}(a_2, m_2)} s_2$, where s_2 = ⟨ Communications, Mailboxes, memory, pc_2, $waitingQueue_1$, Requests ⟩. If we commute the order we will get the final outcome state s_2.
- If the id of $actor_1$ is in waitingQueue[m1], and the id $actor_2$ is not included waitingQueue[m2]. We have $s \xrightarrow{\mathsf{MutexAsyncLock}(p_1, m_1)} s_1$, where s_1 = ⟨ Communications, Mailboxes, memory, pc_1, $waitingQueue$, Requests ⟩, everything are unchanged except the program counter of $actor_1$ changes to the next instruction. After that, firing $\mathsf{MutexAsyncLock}$ (a_2, m_2) $s_1 \xrightarrow{\mathsf{MutexAsyncLock}(a_2, m_2)} s_2$, where s_2 = ⟨ Communications, Mailboxes, memory, pc_2, $waitingQueue_1$, Requests ⟩. This execution adds id of $actor_2$ to $waitingQueue_1[m_2]$ and $actor_2$'s program counter is moved to the next instruction. The reversed order leads to the same state s_2.
- If the id of $actor_1$ and $actor_2$ are not contained in waitingQueue[m1] and waitingQueue[m2] respectively. We have $s \xrightarrow{\mathsf{MutexAsyncLock}(a_1, m_2)} s_1 \xrightarrow{\mathsf{MutexAsyncLock}(a_2, m_2)} s_2$, where s_1 = ⟨ Communications, Mailboxes, memory, pc_1, $waitingQueue_1$, Requests ⟩ and s_2 = ⟨ Communications, Mailboxes, memory, pc_2, $waitingQueue_2$⟩. While $waitingQueue_1[m_1]$ contains $actor_1$, $waitingQueue_2[m_2]$ includes $actor_2$. The array of program counter pc turns to pc_1 because the change of program counter of $actor_1$, and because of the moving that value of $actor_2$ to the next instruction makes pc_1 transforms to pc_2. For the opposite order where $\mathsf{MutexAsyncLock}$ (a_1, m_1) goes before $\mathsf{MutexAsyncLock}$ (a_2, m_2) brings the same overall outcome.

Concerning second condition Def 1.2, based on the specification (conditions for enable), it is trivially conclude one can not enable or disable other one. $\square$