A Map of Asynchronous Communication Models

Florent Chevrou[1], Aurélie Hurault[1], Shin Nakajima[2], and Philippe Quéinnec[1]

[1] Université de Toulouse – IRIT, Toulouse, France first.last@irit.fr
[2] National Institute of Informatics, Tokyo, Japan nkjm@nii.ac.jp

Abstract. Asynchronous communication encompasses a variety of features besides the decoupling of send and receive events. Those include message-ordering policies which are often crucial to the correctness of a distributed algorithm. This paper establishes a map of communication models that exhibits the relations between them along two axes of comparison: the strength of the ordering property and the level of abstraction of the specification. This brings knowledge about which model can be substituted by another without breaking any safety property. Furthermore, it brings flexibility and ready-to-use modules when developing correct-by-construction distributed systems where model decomposition exposes the communication component. Both criteria of comparison are covered by refinement. We consider seven ordering policies and we model in Event-B these communication models at three levels of abstraction. The proofs of refinement between all of these are mechanized in Rodin.

Keywords: asynchronous communication · formal verification · refinements of communication models · Event-B

1 Introduction

A classic way to develop distributed algorithms is to start with a global goal, such as mutual exclusion or global agreement. A distributed version of the algorithm is then derived, either directly or by progressive transformation of the specification, e.g. by refinement. This approach dates back to early work by Dijkstra [11], Chandy-Misra with UNITY [7], Back and Kurki-Suonio with action systems [5], or Lamport with TLA$^+$ [19]. It is still bustling in the correct-by-construction community and Event-B [1] is a framework which embodies this methodology. At one point in the development process, communication is explicitly introduced, to express the flow of information from one site to another, and it eventually takes the form of message exchanges. When the development is conducted with formal verification, the properties of the communication are shown to be sufficient for the correctness of the algorithm. However, it is often unclear what are the specific properties of this communication that are necessary to ensure the correctness of the algorithm. Especially, it may be difficult to replace one communication model with another without doing again the complete proof.

The present work aims at alleviating these difficulties for asynchronous point-to-point communication with message ordering policies. These policies control

message deliveries based on past events or involved peers, and their relative strength forms a hierarchy of communication models. To this end, we use simulation: if a model M_1 simulates another model M_2, M_2 has less non-determinism, hence fewer behaviors. Thus a safety property proved under M_1 in a given system will hold if M_1 is substituted by M_2 (there is no guarantee of preservation for liveness properties). A distributed application is refined up to the point where communication is introduced. Then, model decomposition isolates the communication part and the hierarchy is used to choose an adequate ordering policy.

There exist several approaches to decomposition in Event-B [14]: shared-variable [3], shared-event [6] and modularization [16]. Our map is well suited for shared-event decomposition, where variables are partitioned and a set of events are synchronized and shared by sub-models. During the refinement of a system, asynchronous communication appears via two events: send and receive. These events are isolated in a sub-model to be refined using the results of this paper.

Nevertheless, the proposed one-dimensional scale is not sufficient as several communication models may realize the same ordering policy. They often have little in common: some directly map the ordering property on high level data structures such as distributed executions while others will make use of ad-hoc concrete approaches (e.g. counters on messages) from which the property arises. Mapping the communication models depending on their level of abstraction completes the approach. Therefore, we draw a bidimensional map of communication models and use refinement as a common ground for the two orthogonal comparison criteria: refinement for simulation, and data refinement for concretization. Our results, summed up in the map in Figure 1, are proved and mechanized in Event-B. Regarding decomposition, this means once an ordering policy has been chosen across the simulation direction, the model can be refined across the concretization direction as part of a correct-by-construction development.

The outline of this paper is the following. Section 2 recalls basic definitions of the theory of asynchronous distributed systems and their modeling in Event-B. Section 3 presents seven communication models and proves the hierarchy of their ordering policies, based on simulation, using refinement. Section 4 presents variants of the models based on message histories, and proves that the hierarchy still holds. Section 5 refines them one step further towards practical concrete models. Section 6 discusses proof effort and localization. Section 7 provides related work.

The page http://hurault.perso.enseeiht.fr/MenagerieOfRefinements contains all the models discussed in this paper and gives indications to replay the proofs.

2 Distributed Systems

2.1 Distributed Executions

An asynchronous message-passing distributed system is composed of a set of peers that exchange messages. This paper considers point-to-point communication where a message has exactly one sender and at most one receiver. A distributed execution is a partially ordered set of events, where events are communication events: message send events and message receive events; internal

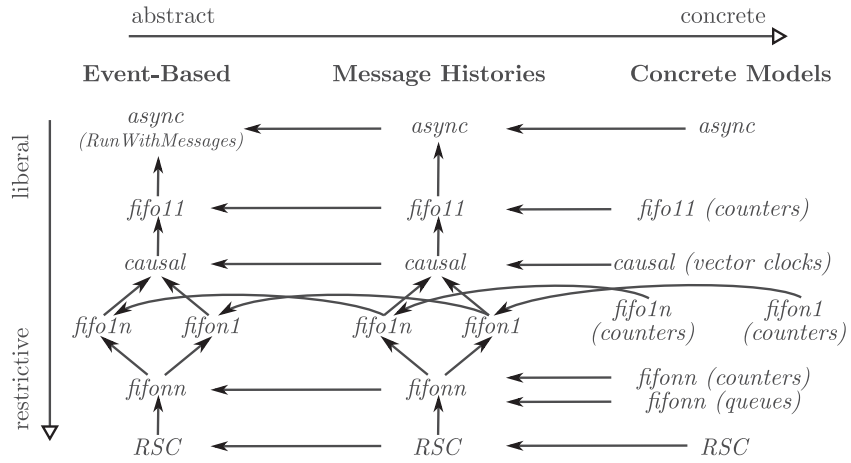

Fig. 1. Map of the Asynchronous Communication Models. A black arrow means "refines". The two axis of refinement are the level of abstraction (data refinement) and the strength of the ordering policy (reduction of non-determinism).

events are ignored. The partial order is named the causal order [18,23], and it abstracts independent events. Events occur on peers: a labeling function states where the event e has occurred. Assuming interleaving of independent events and no true concurrency, a run is a linear extension of a distributed execution.

Let `PEER` be the set of peers, `MESSAGE` an enumerable set of messages identifiers, and $\texttt{COM} \triangleq \{Send, Receive\}$ the communication labels.

Definition 1. *A distributed execution $(E, \prec_c, com, mes, peer)$ is a partially ordered set with labeling functions, where E is an enumerable set, $\prec_c$ is a partial order on E, and com, mes and $peer$ are labeling functions from E to `COM`, `MESSAGE` and `PEER`. An event e that occurs on $peer(e)$ is either the sending or the reception $(com(e))$ of message $mes(e)$. $(E, \prec_c, com, mes, peer)$ satisfies:*

- *no message is sent or received more than once:*
  $\forall e, e' \in E : \big(com(e) = com(e') \wedge mes(e) = mes(e')\big) \Rightarrow e = e'$
- *a receive event is preceded by a send event:*
  $\forall e \in E : com(e) = Receive$
  $\quad \Rightarrow \exists e' \in E : \big(com(e') = Send \wedge mes(e') = mes(e) \wedge e' \prec_c e\big)$
- *events occurring on the same peer are totally ordered:*
  $\forall e, e' \in E : peer(e) = peer(e') \Rightarrow e \prec_c e' \vee e' \prec_c e$

Definition 2. *A run $\sigma = (E, \prec_c, <_\sigma, com, mes, peer)$ extends a distributed execution $(E, \prec_c, com, mes, peer)$: $(E, <_\sigma)$ is a linear extension of $(E, \prec_c)$.*

2.2 Event-B

A model in Event-B [1] is an abstract state machine containing state variables, invariants, and events (the word "event" refers either to an element of a dis-

tributed execution or a part of an Event-B machine – a transition predicate –; the context hopefully makes it clear which is which). An event E parameterized by x has the form `EVENT` E `ANY` x `WHERE` $G(v,x)$ `THEN` $A(v,x)$ `END`, where $G(v,x)$ is the guard of the event and $A(v,x)$ an action changing the values of v. In this paper, actions are deterministic assignments of the form $v := expr$ where v is a state variable. `INITIALISATION` specifies the initial state of a machine. A machine can be related to an Event-B context (`SEES`) that specifies sets, constants, axioms and theorems.

The main concept of the Event-B method is the refinement (`REFINES`) of machines. It consists of a refinement of the events: the guards may be weakened and the behavior must conform to the abstract event. New events refine the special event called "skip". The Rodin tool [2] generates proof obligations for the refinements and the preservation of the invariants by the events.

In Event-B, $x_1 \mapsto x_2$ denotes a pair (x_1, x_2). Relations are sets of pairs. $dom(r)$ and $ran(r)$ denote the domain and range of a relation r. $E \leftrightarrow F$ denotes the set of relations between E and F, $E \twoheadrightarrow F$ the set of total surjective relations, and $E \rightarrow F$ total functions from E to F. The relation $r_1; r_2$ denotes the forward composition of relations r_1 and r_2. "$\lhd$" is the domain restriction operator such that given a relation r and a set E, $E \lhd r \triangleq \{x \mapsto y \mid x \mapsto y \in r \wedge x \in E\}$. "$\hookleftarrow$" is the domain subtraction operator such that given a relation r and a set E', $E' \hookleftarrow r \triangleq \{x \mapsto y \mid x \mapsto y \in r \wedge x \notin E'\}$. "$\leftarrow$" is the overriding operator such that given relations r_1 and r_2, $r_1 \leftarrow r_2 \triangleq r_2 \cup (dom(r_2) \hookleftarrow r_1)$. $\mathbb{P}(E)$ denotes the powerset of E.

2.3 Event-B Distributed Executions

First, we introduce each feature of asynchronous communication through a series of initial refinements: concept of "events", "happening", and "past", then the pairing of two events (communication), localization of the events (distribution) and causality (distributed executions), linearization of the executions (totally ordered runs), and eventually messages which label the exchanges. This paper skips these preliminary refinements. The resulting machine called `RunWithMessages`, presented in Figure 2, is a boilerplate for any asynchronous point-to-point communication model. It conforms to the distributed executions and runs of Definitions 1 and 2. By playing with the guards, other communication paradigms can be specified (e.g. synchronous communication, multicast, join).

The machine relies on sets defined in the contexts: `EVENT` the set of event identifiers labeled by elements of `PEER`, `MESSAGES`, and `COM` (`Send` or `Receive`). Events are labeled by variables `peerOf`, `mesOf`, and `comOf` once they are introduced in the machine (e.g. $e \mapsto Send \in comOf$). New communication events are introduced by the actions of the two Event-B events `send` and `receive`. They are stored in the `past` variable and the labeling functions evolve according to the parameters of `send` and `receive`: the peer it has occurred on and the exchanged message. Additionally, variables `prec` (partial causal order $\prec_c$) and `run` (total order $<_\sigma$) log the dependencies between the events which serves to specify the communication properties (including ordering policies in future machines).

```
MACHINE E_RunWithMessages
SEES E_Messages
VARIABLES past peerOf prec run mesOf comOf
INVARIANTS // (excerpt)
  TmesOf: mesOf ∈ past →MESSAGE                                          //
  TpeerOf: peerOf ∈ past →PEER          TcomOf: comOf ∈ past →COM        // Type invariants
  Tprec:   prec ∈ past ⇸⇸past             Trun:  run ∈ past ⇸⇸past       //

  // prec is reflexive , transitive , and anti−symmetric. So is run (omitted).
  inv1: (past ◁ id) ⊆ prec        inv2: prec ; prec ⊆ prec        inv3: prec ∩ prec^−1 ⊆ id
  // Events occurring on the same peer are totally ordered
  inv4: ∀ e1, e2 · e1∈ past ∧e2∈ past ∧peerOf(e1) = peerOf(e2) ⇒e1 ↦ e2 ∈ prec ∨e2 ↦ e1 ∈ prec
  inv5: prec ⊆ run  // run extends prec.
  inv6: ∀ e1, e2 · e1 ∈ past ∧ e2 ∈ past ⇒ e1 ↦ e2 ∈ run ∨e2 ↦ e1 ∈ run // run is total.
  // No message is sent or received more than once.
  inv7: ∀ e1,e2 · e1∈ past ∧e2∈ past ∧comOf(e1)=comOf(e2) ∧mesOf(e1)=mesOf(e2) ⇒e1 = e2
  inv8: ∀ e · e ∈ past ∧ comOf(e) = Receive // A receive event is preceded by a send event.
              ⇒ (∃ es · es ∈ past ∧ comOf(es) = Send ∧mesOf(e) = mesOf(es) ∧es ↦ e ∈ prec)

EVENT send ANY e p m // New event, Peer where the event occurs, Sent message
WHERE
  grd1: e ∈ EVENT \ past ∧ p ∈ PEER ∧ m ∈ MESSAGE
  grd3: m ∈ MESSAGE \ ran(mesOf) // m has not already been sent
THEN
  act1:  past := past ∪ {e}
  act2:  peerOf := peerOf ∪{e ↦ p}
  act3: prec := prec ∪ {e ↦ e} ∪ {ep · ep ∈ past ∧ (∃ ep2 · ep2 ∈ past ∧ peerOf(ep2) = p ∧ep ↦
ep2 ∈ prec) | ep ↦ e}
  act4: run := run ∪ {e ↦ e} ∪ {ep · ep ∈ past | ep ↦ e}
  act5: mesOf := mesOf ∪{e ↦ m}          act6: comOf := comOf ∪{e ↦ Send}

EVENT receive ANY e p m // new event; receiver; received message
WHERE
  grd1: e ∈ EVENT \ past ∧ p ∈ PEER ∧ m ∈ MESSAGE
  grd6: ∀ ep · ep ∈ past ∧ comOf(ep) = Receive ⇒mesOf(ep) ≠ m // m has not already been received
  grd7: ∃ es · es ∈ past ∧ comOf(es) = Send ∧mesOf(es) = m   // m has been sent
THEN // Same as send except
  act4:                  // The new event is causally after all the events that causally precede
    prec := prec         // an event from the same peer (penultimate line) and after all the events
      ∪ {e ↦ e}          // that causally precede the send event (last line).
      ∪ {ep · ep ∈ past ∧ (∃ ep2 · ep2 ∈ past ∧ peerOf(ep2) = p ∧ep ↦ ep2 ∈ prec) | ep ↦ e}
      ∪ {ep · ep ∈ past
          ∧ (∃ es · es ∈ past ∧ comOf(es) = Send ∧mesOf(es) = m ∧ep ↦ es ∈ prec) | ep ↦ e}
  act6: comOf := comOf ∪{e ↦ Receive}
```

Fig. 2. Event-B Machine for Asynchronous Point-to-Point Communication

3 Abstract Communication Models

The communication model specifies when a communication action (send or receive) is possible in order to ensure specific properties on the communication. We focus on message ordering properties (e.g. global ordering: all messages are received in their emission order). In this section, each abstract communication model is a machine based on `RunWithMessages` that is characterized by an ordering invariant on distributed executions or runs. We use this invariant to filter out the distributed executions and runs that do not abide to the ordering policy and keep all those that do.

The communication models constitute steps between fully asynchronous distributed communication (*async*) where sending and receiving a message is always

possible, partially ordered communication (*fifo11*, *causal*, *fifo1n*, *fifon1*), totally ordered communication (*fifonn*), and almost synchronous communication (*RSC*) where a message must be received immediately after it has been sent. We use simulation to define a hierarchy based on the strength of the delivery order. Stronger models have less non-determinism on the receptions. Machine `RunWithMessages` models asynchronous communication and corresponds to the *async* model. The other models impose more and more determinism on reception (and, for RSC, on send). The first column of Figure 1 accounts for the hierarchy of these models. Refinement is used to prove the simulation relations between the models. Note that these are *not* concretization refinements: no model can be called more (or less) concrete or realizable. Concretization of the communication models follow a specific path for each model and is described later.

3.1 Informal Specifications

In this paper, we study seven asynchronous point-to-point communication models. A detailed description with figures of each model is given in [10].

RSC Realizable with Synchronous Communication [8,17]. The send event of a message is immediately followed by the receive event of this message (viewed atomically, it corresponds to synchronous communication).

fifo n-n Messages are globally ordered and are delivered in their send order.

fifo 1-n Messages from the same peer are delivered in their send order.

fifo n-1 On a given peer, messages are received in their send order.

fifo 1-1 Messages between a couple of peers are delivered in their send order. Messages from/to different peers are independently delivered.

causal Messages are delivered according to the causality of their emission [18]. If a message m_1 is causally sent before a message m_2 (i.e. there exists a causal path from the first emission to the second one), then a peer cannot get m_2 before m_1.

async Fully Asynchronous. No order on message delivery is imposed. The machine `RunWithMessages` is this model.

3.2 Event-B Specifications

We consider the specifications of the communication models with events. Each communication model is characterized by an invariant that describes the ordering properties it ensures on the communication. The invariants of the models all introduce es_1 and es_2, the send events of two distinct messages, as well as er_1 and er_2, the corresponding receive events. The model-specific part imposes an order on the receive events (er_1 and er_2) based on the causal or run order of the send events (es_1 and es_2) and whether or not the sending or receiving peers are the same (same sending peer and same receiving peer for *fifo 1-1*, same sending peer for *fifo 1-n*, same receiving peer for *fifo n-1* and *causal*). For instance, the ordering invariant in the machine `CausalEvent` is:

```
// Given two transmissions of messages and the four corresponding events: es1 er1 and es2 er2
∀ es1, er1, es2, er2 · es1 ∈ past ∧ er1 ∈ past ∧ es2 ∈ past ∧ er2 ∈ past
    ∧ comOf(es1) = Send ∧comOf(es2) = Send ∧comOf(er1) = Receive ∧comOf(er2) = Receive
    ∧ mesOf(es1) = mesOf(er1) ∧ mesOf(es2) = mesOf(er2)
    // Model−specific part:
    ∧ es1 ↦ es2 ∈ prec            // If es1 CAUSALLY precedes es2
    ∧ peerOf(er1) = peerOf(er2) // and the corresponding RECEPTIONS occur on the SAME PEER
    ⇒ er1 ↦ er2 ∈ run             // then they must occur in the emission order.
```

Our next goal is to compare the communication models, by proving that some have less transitions than others (i.e. are more deterministic). Later in Sections 4 and 5, we derive more concrete specifications of these models. However, at this point, having machines that are as liberal as the ordering allows is important. Thus, the weakest preconditions of the ordering invariants are stipulated for the guards of the send and receive events. As the actions are assignments of the form $var := var \cup \{\cdots\}$, the computation of the weakest preconditions is trivial [12]. As an example, Figure 3 presents the resulting structure of the `CausalEvent` machine with a close up on the ordering guard of the receive event.

3.3 Proofs and Invariants

The difference between the models is an invariant directly related to the order of delivery and the associated weakest precondition used as a guard on the communication events. A proof of refinement consists in proving the logical implications between these invariants. Most of the time these proofs require little manual intervention thanks to auto-provers, post-tactics, and SMT solvers.

The refinements of *causal* in *fifo-n1* and *fifo-1n* need manual intervention with a specific invariant that states that two causally related events on different peers are necessary linked by (at least) one message. Informally, it means that causality between events on distinct peers only exists due to message exchanges.

```
∀ e1, e2 · e1 ↦ e2 ∈ prec ∧peerOf(e1) ≠ peerOf(e2) ⇒
(∃ es, er · e1 ↦ es ∈ prec ∧ es ↦ er ∈ prec ∧ er ↦ e2 ∈ prec ∧peerOf(e1) = peerOf(es)
        ∧ comOf(es) = Send ∧comOf(er) = Receive ∧mesOf(es) = mesOf(er))
```

4 History-based Models

In this section, we take one step forwards in the direction of concretization. These new specifications share a common framework in which the ordering properties rely upon keeping track of dependent messages in histories. This makes it easier to compare them much like in the previous section. Yet, the specifications are now operational and realistic enough to be implemented and used as such.

There are two directions involved in the mapping of these communication models. First, each history-based model relates to its execution-based counterpart: it is a concretization of the latter, which means the underlying ordering properties still hold, and we use refinement to prove it in Section 4.2. For example, `Fifo11History` is a concretization of `Fifo11Event`. Second, it is expected that the history-based communication models, which model the same ordering

```
MACHINE E_CausalEvent
REFINES E_fifo11 // (which refines E_RunWithMessages)
SEES E_Messages
VARIABLES past peerOf prec run mesOf comOf

INVARIANTS
    ... // Invariants from E_RunWithMessages
    ordering : // causal ordering invariant : see 3.2

EVENT send REFINES send ... // from E_RunWithMessage with additional invariant

EVENT receive REFINES receive // receive event with a
ANY e p m
WHERE
 grd1: e ∈ EVENT \ past
 grd2: p ∈ PEER
 grd3: m ∈ MESSAGE \ ran(mesOf)
  ... // guards from E_RunWithMessage
 // weakest precondition of the causal ordering invariant
 ordering : ∀ es1, er1, es2, er2 ·
          es1 ∈ past ∪ {e} ∧ er1 ∈ past ∪ {e}
        ∧ es2 ∈ past ∪ {e} ∧ er2 ∈ past ∪ {e}
        ∧ (comOf ∪{e ↦ Send})(es1) = Send
        ∧ (comOf ∪{e ↦ Send})(es2) = Send
        ∧ (comOf ∪{e ↦ Send})(er1) = Receive
        ∧ (comOf ∪{e ↦ Send})(er2) = Receive
        ∧ (mesOf ∪ {e ↦ m})(es1) = (mesOf ∪{e ↦ m})(er1)
        ∧ (mesOf ∪ {e ↦ m})(es2) = (mesOf ∪{e ↦ m})(er2)
        ∧ (peerOf ∪ {e ↦ p})(er1) = (peerOf ∪{e ↦ p})(er2)
        ∧ es1 ↦ es2 ∈ run ∪ {e ↦ e} ∪{ep · ep ∈ past | ep ↦ e}
        ⇒ er1 ↦ er2 ∈ run ∪ {e ↦ e} ∪{ep · ep ∈ past | ep ↦ e}
 THEN // same actions as E_RunWithMessage
    act1: past := past ∪{e}
    act2: peerOf := peerOf ∪{e ↦ p}
    act3: prec := prec ∪ ...
    act4: run := run ∪ ...
    act5: mesOf := mesOf ∪{e ↦ m}
    act6: comOf := comOf ∪{e ↦ Receive}
```

Fig. 3. Structure of the Causal Communication Model Described With Events. The machine corresponds to `E_RunWithMessages` with an additional ordering invariant and the associated guards.

policies, preserve the hierarchy of these ordering policies. Once again, the simulation relations (i.e. the reduction of the non-determinism of the communication events send and receive) are made explicit and proved by refinement. For instance `CausalEvent` is stronger than (refines) `Fifo11Event` and `CausalHistory` is stronger than (refines) `Fifo11History`.

4.1 Specifications with Histories

We consider specifications of the asynchronous point-to-point interaction models where communication occurs according to two parameterized events: $send(p, m, d)$ (peer p sends message m to an explicit peer d) and $receive(p, m)$ (peer p receives message m).

The models rely on a state variable `net` that contains messages in transit. Sent messages are labeled to carry information about the communication: the

origin peer, the destination peer, and the history of the message. The history of a message is the set of messages on which it depends, i.e. the set of messages which precede it. As two notions of precedence exist (causal/execution), two kinds of message histories are defined: namely causal and global.

Definition 3. *(Message Histories) For a run* $\sigma = (E, \prec_c, <_\sigma, com, mes, peer)$*, and a message* m*:*

$$hcOf(m) \triangleq \left\{ \begin{array}{l} m' \in MESSAGE : \exists e, e' \in E : \\ \quad com(e) = Send \wedge com(e') = Send \\ \wedge\ mes(e) = m \wedge mes(e') = m' \\ \wedge\ e' \prec_c e \end{array} \right\}$$

$$hgOf(m) \triangleq \left\{ \begin{array}{l} m' \in MESSAGE : \exists e, e' \in E : \\ \quad com(e) = Send \wedge com(e') = Send \\ \wedge\ mes(e) = m \wedge mes(e') = m' \\ \wedge\ e' <_\sigma e \end{array} \right\}$$

In the Event-B models, the message histories are built upon state variables $\texttt{hg} \subseteq \texttt{MESSAGE}$, the global history, and $\texttt{hc} \in \texttt{PEER} \to \mathbb{P}(\texttt{MESSAGE})$, the causal histories of each peer. When peer p sends a message m, the global history (`hgOf`) and the causal history (`hcOf`) of m are the current values of `hg` and of $\texttt{hc}(p)$. The new message is also added to the history state variables (`hg` and $\texttt{hc}(p)$). The causal history $\texttt{hc}(p)$ of peer p is updated when a message m is received to account for the causal relation induced by the transmission of the message from one peer to another: m and its causal history $\texttt{hcOf}(m)$ are added to $\texttt{hc}(p)$. The validity of these constructions with regard to the above definitions is stated as two invariants. The ordering properties of a model are determined by guards on the `send` and `receive` events that depend on the message histories, origin, and destination of a message.

4.2 Concretization

For each communication model, the refinement of the event-based model by the history-based model is split in two steps to facilitate the proofs. First, add new variables to hold histories and message destination (`net`, `hg`, `hc`, `hgOf`, `hcOf`, `destOf`), and replace the guards about events by guards about histories. Then, remove the now useless variables related to events (`past`, `prec`, `run`, ...).

As an example, Figure 4 is the resulting machine for the *causal* model. Its ordering invariant states that if m_1 and m_2 have the same destination, and m_1 was sent causally before m_2 (thus m_1 is in the causal history of m_2), then m_1 cannot be in transit when m_2 is not. This means that m_1 must be received before m_2. Accordingly, the ordering guard for receive allows to deliver a message m if there does not exist another message m_2 in transit, with same destination, and which is in the history of m.

Data refinement consist in proving that the model-specific guards on the communication events guarantee the ordering properties on the distributed executions. The proofs rely on the ordering invariant, and wisely formulated gluing

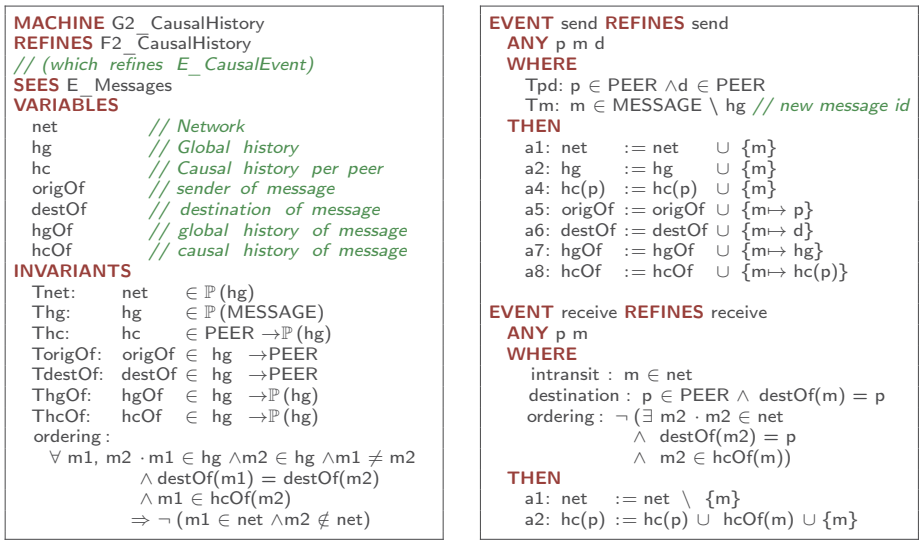

Fig. 4. History-Based Event-B Model for Causal Communication

and consistency invariants. The gluing invariants relate the state variables of the abstract (events, executions) and concrete machines (network, histories). For instance, a message m_1 is in the causal history of m_2 if the send event of m_1 is causally anterior to the send event of message m_2. The consistency invariants clarify links between the state variable of the concrete machine (e.g. if a message m_1 is in the causal history of m_2, it is also in its global history). Besides, significant manual interventions have to be carried out to supervise the proof process as the number of state variables and invariants misdirect the automatic provers. Finding the optimal formulation (e.g. proposition vs. contraposition), through trial and error, is a large portion of the proof effort.

5 Concrete Models

In the first approach, the models based on events directly translate the ordering policies of the communication models. The second approach using message histories is more concrete: the locality and transmission of data is taken into account with messages that carry their history. However, keeping trace of all the previously sent messages is still unrealistic in practice. Therefore we refine the models that use histories with concrete structures such as counters of messages or queues of messages.

5.1 Logical Clocks

Regarding the causal communication model as described in Figure 5, the causality relation can be explicit, using pruned causal histories [17] (in the worst case,

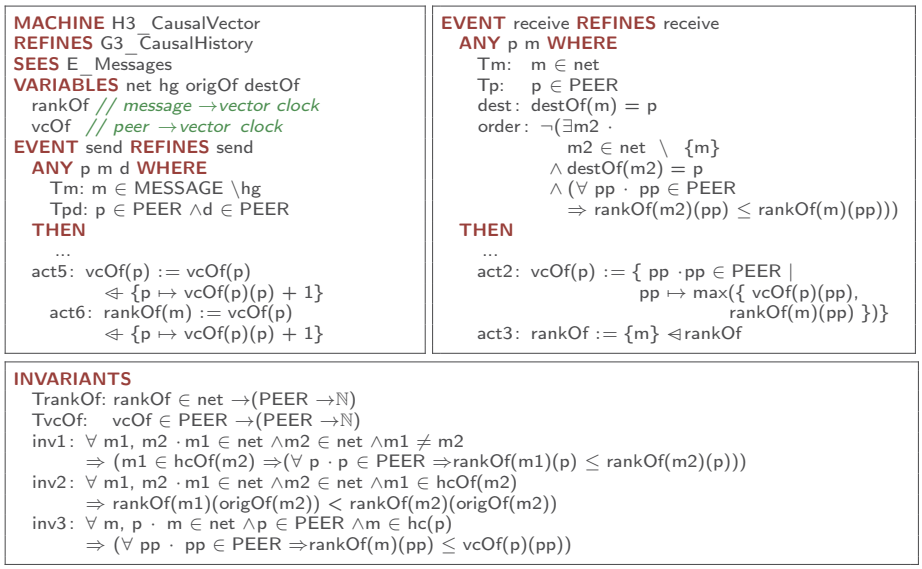

Fig. 5. Concrete Model for Causal Communication Using Vector Clocks

this is as costly as our version with histories), or derived from logical vector clocks of size n or matrix clocks of size $n \times n$ [23].

Every peer p has a vector clock $vcOf(p)$. For peers p and pp, $vcOf(p)(pp)$ holds the number of send events on pp that are in the current past of peer p. When a peer sends a message, it increments its own count ($vcOf(p)(p)$) and piggybacks its vector with the message. At reception, a peer updates every component of its vector with the max of its current value and of the component of the received vector. Thus, $vcOf(p)(pp)$ holds the number of messages sent by pp and known by p. A message m is in the causal history of m' iff every vector component of m is lower or equal than the one of m' (and at least one is strictly lower: distinct messages have different vectors). To ensure causal reception, a message can be delivered to a peer iff no other message exists for this peer with a lower vector.

The refinement of `CausalHistory` replaces the history variables with vector clocks. The events are refined to update these variables and, in the `receive` event, the guard built on histories is replaced with a property on the vectors. The refinement proof requires gluing invariants on causal histories and vector clocks.

5.2 Other Concretizations

As shown in Figure 1, other concrete models have been defined. The various *fifo** models are easily described with counters. If n denotes the number of peers, 2 counters (*fifonn*), $2 \times n$ counters (*fifon1* and *fifo1n*), or $2 \times n^2$ counters (*fifo11*)

are used to account for the ordinal rank of the last sent and last received messages in the system (*fifonn*), a peer (*fifon1* and *fifo1n*), or a couple of peers (*fifo11*). The ranks of the last received messages determine the rank of the messages that can be received. Alternatively, message queues can be used: if n denotes the number of peers, we need a global queue (*fifonn*), n inbox queues (*fifon1*), n outbox queues (*fifo1n*), or n^2 queues (*fifo11*).

6 Additional Remarks

6.1 Proof Effort

The full menagerie holds 42 machines, 41 refinements, 329 invariants, and more than 1400 proof obligations. Once the necessary invariants are stated, the large majority of these proof obligations are automatically proved by Rodin with SMT solvers (49 manual proofs, 3.5% of the proof obligations). The main difficulties are described below.

To make the proofs automatic, the trick is to find additional invariants. For instance, to prove that `RscHistory` refines `RscEvent`, the invariant

$$\forall\, e1,\ e2 \cdot e1 \mapsto e2 \in run \land comOf(e1) = Send \land e1 \neq e2 \Rightarrow mesOf(e1) \notin net$$

has to be made explicit (it says that if there exists at least one event after a send event e_1, then the message sent at e_1 is no longer in transit). As expected, the discovery of the necessary invariants is the hardest part in the proofs, and the largest part of our proof effort was devoted to this point. Our methodology consists in running the automatic provers and analyzing the failure (if any). After some case analysis of the disjunctions, a contradiction often appears in the hypotheses. This contradiction leads us to a relevant new invariant. Once stated and proved, this new invariant may, with good luck, suppress the unsuccessful branch and advance towards the fully automatic proof.

The refinements involving *causal* are never easy. One essential invariant is:

$$\forall\, e1,\ e2 \cdot e1 \mapsto e2 \in prec \land peerOf(e1) \neq peerOf(e2) \Rightarrow (\exists\, es, er \cdot e1 \mapsto es \in prec \land es \mapsto er \in prec \land er \mapsto e2 \in prec \land peerOf(e1) = peerOf(es) \land comOf(es) = Send \land comOf(er) = Receive \land mesOf(es) = mesOf(er))$$

It states that two causally related events on different peers are necessarily linked by (at least) one message, or conversely, that causality between peers only arises from message exchanges. This invariant had to be manually instantiated.

Lastly, concrete models need ad-hoc reasoning. For instance, Section 5.1 presents the specific invariants that are required to prove that `CausalVector` refines `CausalHistory`. These invariants are expected as they state that vector clocks encode causality. Nevertheless, the refinement proofs require to manually recall and instantiate these invariants.

6.2 Localization

The last point concerns the distributed nature of the communication models. The first abstract models, based on properties of the executions, are purely

logical and offer a global point of view of the communication models. The second models, based on histories, are actually directly implementable even if costly. The third concrete models offer realistic implementations. By looking at their definitions, one can distinguish two classes of communication models. The models *async*, *fifo11*, *causal* and *fifo1n* only need meta information piggybacked with the message and local knowledge available on the peer. On the other hand, *fifon1*, *fifonn* and *rsc* require global shared variables, and their implementation in a distributed system requires a central coordinator or totally ordered multicast.

7 Related Work

Asynchronous communication models in distributed systems are studied and compared in [17] (notion of ordering paradigm), [8] (notion of distributed computation classes), and [13] (for message sequence charts). Implementations of the basic communication models (*causal*, *fifo11*) using histories or clocks are explained in classic textbooks [17,23]. In our previous work, we have unified and extended these results in [10]. The goal was to develop a framework to mechanically verify algorithms [9], and to give a unified description of the models. However only the communication models with message histories were specified in TLA$^+$. All the Event-B models presented here are new, as well as the refinement relations leading to the distributed executions (Section 2), between the abstract communication models (Section 3), and between the concretizations (abstract model to history-based model to ad-hoc model, sections 4.2 and 5).

Formal verification of distributed algorithms have been conducted with success. However the hypotheses on the communication are often fuzzy or unclear and one has to dive deep into the proofs to identify them. For instance, [22] studies the topology maintenance in structured peer-to-peer networks. Different algorithms are studied, some assume FIFO channels and some do not. It is unclear why it is required, and if it is required for all channels.

Refinement has been used to verify distributed algorithms. [20] describes the addition of Byzantine resilience to standard Paxos. The proof is conducted by refinement of the distributed non-Byzantine algorithm and has been mechanically checked with the TLA$^+$ Proof System. Another approach is presented in [21]. Three versions of Paxos (the classic one, disk Paxos and Byzantine Paxos) are derived from an abstract, non-distributed algorithm.

The Event-B book [1] presents several examples of refinements of distributed algorithms. The *simple file transfer protocol* decomposes the atomic sending of a file in a sequence of send events, and uses counters to coordinate the progression. This protocol is later extended to handle loss and re-transmission with an alternating bit protocol. In this example, asynchronous communication appears implicitly during refinement, and properties of the communication are directly embedded in the resulting machine. A logical clock is used in the *routing algorithm for a mobile agent* to order the messages sent by a mobile agent while it moves. This example can be seen as the development of an ordered communication model down to a concrete localizable model. Lastly, the *leader election on a*

connected graph network deals with the difficulties of splitting an atomic action (in a shared-memory model) into several actions (in a message-passing model). This creates deadlocked states (a situation called *contention* in the algorithm) where two nodes are each waiting for the other to progress. This development is more concerned with providing a algorithmic solution in presence of non-atomic actions, than with the development of non-atomicity (i.e. messages).

[4] presents the development by refinement of snapshot algorithms. It starts with the specification of the snapshot problem, which is by essence a global property. A generic architecture with asynchronous communication is presented, which allows the derivation of several algorithms. At one point, the set of messages (which models fully asynchronous communication) is refined by FIFO queues (which models ordered communication). This leads to a simpler snapshot algorithm, which ends being the well-known Chandy-Lamport algorithm.

[15] describes the formal derivation of an algorithm for leader election in Event-B. The abstract model is centralized, and refinement introduces distribution. The behavioral part of the communication model first comprises two events, *send* and *receive* which directly access the state variable of the other peers. Then, a new refinement introduces new variables to decouple the peers and to get a "one-to-one asynchronous communication channel".

8 Conclusion

This paper provides a guide for the design of the communication component in the development of distributed systems and algorithms. It considers a wide range of asynchronous communication models that enforce message-ordering properties on the system and positions each one of them on a map of refinement relations. The map, shown in Figure 1, has two dimensions: it compares the models according to the strength of the underlying ordering properties and their level of abstraction. All these models are specified and the refinements proved in Event-B which paves the way for reusing part of the mechanization in a correct-by-construction development of a distributed system thanks to shared-event model decomposition. Our machines are indeed pluggable to any system where communication occurs according to two events send and receive with usual parameters (message, destination). A classic development process consists in introducing asynchronous communication which corresponds to our `RunWithMessages` machine, the root of our map, and make use of the rest of the map to strengthen the ordering policy depending on the needs, pursue the development towards the concrete practical specifications models (with counters or queues), or even substitute models afterwards knowing the safety properties are preserved. Besides, each one of the three sets of communication models we provide has its assets: the concrete models are close to practical implementations, the event-based models clearly translate the ordering policies which ease theoretical reasoning on the properties themselves, and the history-based models offer a compromise with operational descriptions that are implementable and yet remain uniform to ease formal reasoning.

References

1. Abrial, J.R.: Modeling in Event-B - System and Software Engineering. Cambridge University Press (2010)
2. Abrial, J., Butler, M.J., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: an open toolset for modelling and reasoning in Event-B. STTT **12**(6), 447–466 (2010)
3. Abrial, J., Hallerstede, S.: Refinement, decomposition, and instantiation of discrete models: Application to Event-B. Fundamenta Informaticae **77**(1-2), 1–28 (2007)
4. Andriamiarina, M.B., Méry, D., Singh, N.K.: Revisiting snapshot algorithms by refinement-based techniques. Computer Science and Information Systems **11**(1), 251–270 (2014)
5. Back, R., Kurki-Suonio, R.: Distributed cooperation with action systems. ACM Transactions on Programming Languages and Systems **10**(4), 513–554 (1988)
6. Butler, M.J.: Decomposition structures for Event-B. In: Integrated Formal Methods, 7th International Conference, IFM 2009. pp. 20–38 (2009)
7. Chandy, K.M., Misra, J.: Parallel Program Design: A Foundation. Addison-Wesley (1988)
8. Charron-Bost, B., Mattern, F., Tel, G.: Synchronous, asynchronous, and causally ordered communication. Distributed Computing **9**(4), 173–191 (Feb 1996)
9. Chevrou, F., Hurault, A., Quéinnec, P.: Automated verification of asynchronous communicating systems with TLA$^+$. Electronic Communications of the EASST (PostProceedings of AVoCS 2015) **72** (2015)
10. Chevrou, F., Hurault, A., Quéinnec, P.: On the diversity of asynchronous communication. Formal Aspects of Computing **28**(5), 847–879 (Sep 2016)
11. Dijkstra, E.W.: EWD851b – reducing control traffic in a distributed implementation of mutual exclusion (1983)
12. Dijkstra, E.W., Scholten, C.S.: Predicate Calculus and Program Semantics. Springer-Verlag New York, Inc. (1990)
13. Engels, A., Mauw, S., Reniers, M.A.: A hierarchy of communication models for message sequence charts. Science of Computer Programming **44**(3), 253–292 (2002)
14. Hoang, T.S., Iliasov, A., Silva, R., Wei, W.: A survey on Event-B decomposition. ECEASST **46** (2011)
15. Iliasov, A., Laibinis, L., Troubitsyna, E., Romanovsky, A.: Formal derivation of a distributed program in Event-B. In: ICFEM. Lecture Notes in Computer Science, vol. 6991, pp. 420–436. Springer (2011)
16. Iliasov, A., Troubitsyna, E., Laibinis, L., Romanovsky, A., Varpaaniemi, K., Ilic, D., Latvala, T.: Supporting reuse in Event B development: Modularisation approach. In: Abstract State Machines, Alloy, B and Z. pp. 174–188 (2010)
17. Kshemkalyani, A.D., Singhal, M.: Distributed Computing: Principles, Algorithms, and Systems. Cambridge University Press (Mar 2011)
18. Lamport, L.: Time, clocks and the ordering of events in a distributed system. Communications of the ACM **21**(7), 558–565 (Jul 1978)
19. Lamport, L.: Specifying Systems. Addison Wesley (2002)
20. Lamport, L.: Byzantizing Paxos by refinement. In: 25th International Symposium on Distributed Computing. LNCS, vol. 6950, pp. 211–224 (2011)
21. Lampson, B.W.: The ABCD's of Paxos. In: Symposium on Principles of Distributed Computing, PODC 2001. pp. 13–. ACM (2001)
22. Li, X., Misra, J., Plaxton, C.G.: Active and concurrent topology maintenance. In: 18th Int'l Symp. on Distributed Computing. LNCS, vol. 3274, pp. 320–334 (2004)
23. Raynal, M.: Distributed Algorithms for Message-Passing Systems. Springer (2013)