A Modular Framework for Verifying Versatile Distributed Systems

Florent Chevrou, Aurélie Hurault, Philippe Quéinnec

▶ **To cite this version:**

Florent Chevrou, Aurélie Hurault, Philippe Quéinnec. A Modular Framework for Verifying Versatile Distributed Systems. Journal of Logic and Algebraic Methods in Programming, 2019, 108, pp.24-46. 10.1016/j.jlamp.2019.05.008 . hal-02451058

HAL Id: hal-02451058
https://hal.science/hal-02451058v1

Submitted on 23 Jan 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Open Archive Toulouse Archive Ouverte

OATAO is an open access repository that collects the work of Toulouse researchers and makes it freely available over the web where possible

This is an author's version published in:
http://oatao.univ-toulouse.fr/24923

Official URL

DOI : https://doi.org/10.1016/j.jlamp.2019.05.008

To cite this version: Chevrou, Florent and Hurault, Aurélie and Quéinnec, Philippe *A Modular Framework for Verifying Versatile Distributed Systems.* (2019) Journal of Logical and Algebraic Methods in Programming, 108. 24-46. ISSN 2352-2208

Any correspondence concerning this service should be sent to the repository administrator: tech-oatao@listes-diff.inp-toulouse.fr

A modular framework for verifying versatile distributed systems ☆

Florent Chevrou, Aurélie Hurault, Philippe Quéinnec *

Université de Toulouse, IRIT, 2, rue Charles Camichel, 31000 Toulouse, France

Keywords:
Distributed systems
Asynchronous communication
Multicast
Compatibility checking
TLA$^+$

A B S T R A C T

Putting independent components together is a common design practice of distributed systems. Besides, there exists a wide range of interaction protocols that dictate how these components interact, which impacts their compatibility. However, the communication model itself always consists in a monolithic description of the rules and properties of the communication. In this paper, we propose a mechanized framework for the compatibility checking of compositions of peers where the interaction protocol can be fine tuned through assembly of basic properties on the communication. These include whether the communication is point-to-point, multicast or convergecast, which ordering-policies are to be applied, applicative priorities, bounds on the number of messages in transit, and so on. Among these properties, we focus on a generic description of multicast communication that encompasses point-to-point and one-to-all communication as special cases. The components that form the communication model are specified in TLA$^+$, and a system, composed of a communication model and a specification of the behavior of the peers (also in TLA$^+$), is checked with the TLA$^+$ model checker. Eventually we provide theoretical views on the relations between ordering-policies through the lenses of multicast and convergecast communication.

1. Context

1.1. Introduction

Distributed systems are a composition of individual components, the peers, that exchange messages and work towards a common goal. Their interactions are governed by a protocol, or communication model, that specifies whether the emission or the reception of a message is possible. For example, synchronous communication dictates that a message shall be sent and received at the same time (rendez-vous). In asynchronous communication, though, which this paper focuses on, the emission and the reception of a message do not happen simultaneously: the two events occur with a delay. This results in many possible interleavings of the communication events, some of which might jeopardize the compatibility or the correction of a composition of peers unless specific properties on the communication are met. Such properties include whether the communication is point-to-point, multicast or convergecast, numerous message-ordering policies that state that some messages have to be delivered in their emission order, bounds on the number of messages in transit, and applicative

☆ This work was supported by project PARDI ANR-16-CE25-0006.
* Corresponding author.
E-mail addresses: florent.chevrou@enseeiht.fr (F. Chevrou), aurelie.hurault@enseeiht.fr (A. Hurault), philippe.queinnec@enseeiht.fr (P. Quéinnec)

https://doi.org/10.1016/j.jlamp.2019.05.008

priorities ensuring that some messages or recipients have precedence over others. Any conjunction of these properties is a unique communication model. Yet, existing verification frameworks consider the interaction protocol to be an indivisible entity that may be, at best, parameterized (e.g. capacity of queues) or entirely substituted by another.

In this paper, we describe an extensible framework where the communication model is any desired conjunction of communication properties we call "micromodels". We allow for different combinations to apply on different parts of the distributed system: for instance multicast causally ordered communication on some of the peers and point-to-point capped FIFO ordered communication on another subsystem. Each micromodel is a transition system specified in TLA$^+$ whose transitions account for an emission or a delivery of a message and whose states may fit any convenient data structure, no matter how the rest of the communication is described. For instance, a simple specification of the micromodel corresponding to the property "there are at most n messages in transit" is a set in which a message is added after an emission, removed after a reception, and that prevents any further emissions when it contains n messages. As an example, it may coexist with a micromodel that enforces a message delivery order using queues. A system to verify consists of the product of the micromodels and the behavior of the peers, specified in TLA$^+$. The correctness of the system is checked with TLC, the TLA$^+$ model checker. This correctness is any linear temporal properties (safety and liveness) that TLA$^+$ can express.

The contributions of the paper are the following. We provide a library of TLA$^+$ modules that specify the behavior of various micromodels. First, physical micromodels deal with the multiplicities of delivery: point-to-point communication (a message is delivered to one peer), multicast communication (a message has several receivers) and convergecast communication (a peer receives a set of messages). One notable contribution is a generic specification (in one single micromodel) of multicast communication that encompasses point-to-point and one-to-all communication as special cases. Combined with these physical micromodels, the framework includes nine micromodels that control emission and reception. The complete files of the framework are available online [12]. Lastly, this paper includes a theoretical study that compare the expressive power of the message-ordering micromodels.

The outline of this paper follows: This introduction presents a running example. Section 2 provides an introduction to the TLA$^+$ specification language, Section 3 presents the overall design of our verification framework and the modular design of communication models, Section 4 details several micromodels: a universal micromodel of communication for both the point-to-point and multicast paradigms, a micromodel for convergecast communication, and several message-ordering models that are used in combination with the previous communication paradigms. Section 5 studies the relations between these message-ordering models with multicast and convergecast communication. Section 6 explores related work, and eventually Section 7 sums this work up and paves the way for further developments.

1.2. Running example: a conference reviewing system

As a running example, we present a conference reviewing system. This system is composed of peers that are the authors, the chairs of the program committee and the reviewers. Authors send their papers to all the PC chairs. An author can submit only one paper. Each chairperson attributes a paper number and takes responsibility for a part of the papers, based on this number. After the deadline has passed, the chairs reject new submissions and inform the authors. After the deadline, each chair independently sends his papers to some reviewers, waits for the reviews, and sends the acceptance result to the author. The system must ensure that it does not deadlock and that every author eventually receives a unique answer (either rejection for a late paper, or acceptance result if reviewed).

Fig. 1 shows a possible execution: three authors send their paper on channel `submission` to all the chairs (multicast communication – black plain arrows). Chair 1 will handle the odd messages (i.e. the one from authors 1 and 3) and chair 2 will handle the even messages (i.e. the one from author 2). The chairs forward the paper that they handle on channel `paper` to a set of two or three reviewers (multicast communication – blue dashed arrows). Paper 1 (from author 1) is sent by chair 1 to all three reviewers, and paper 2 (from author 2) is sent by chair 2 to reviewers 1 and 3. Author 3 submits her paper after the deadline and is rejected by chair 1. The chairs wait on channel `review` for at least two reviews by paper (convergecast communication – green dotted-dashed arrows), and send on channel `acceptation` the acceptance result to the authors (point-to-point communication – red dotted arrows).

2. TLA$^+$ specification language

TLA$^+$ [23] is a formal specification language based on untyped Zermelo-Fraenkel set theory for specifying data structures, and on the temporal logic of actions (TLA) for specifying dynamic behaviors. TLA$^+$ allows to specify symbolic transition systems with variables and *actions*. The TLA$^+$ toolbox contains the TLC model checker (an enumerative explicit-state model checker), the TLAPS proof assistant, and various tools such as a translator for the PlusCal Algorithm Language [24] into a TLA$^+$ specification.

Expressions rely on standard first-order logic, set operators, and several arithmetic modules. Hilbert's choice operator, written as $\textsc{choose}\ x \in S : p$, deterministically picks an arbitrary value in S which satisfies p, provided such a value exists (its value is undefined otherwise).

Functions are primitive objects in TLA$^+$, and tuples are a particular kind of function. The application of function f to an expression e is written as $f[e]$. The set of functions whose domain is X and whose co-domain is a subset of Y is written as $[X \to Y]$. The expression $\textsc{domain}\ f$ is the domain of the function f. The expression $[x \in X \mapsto e]$ denotes the function

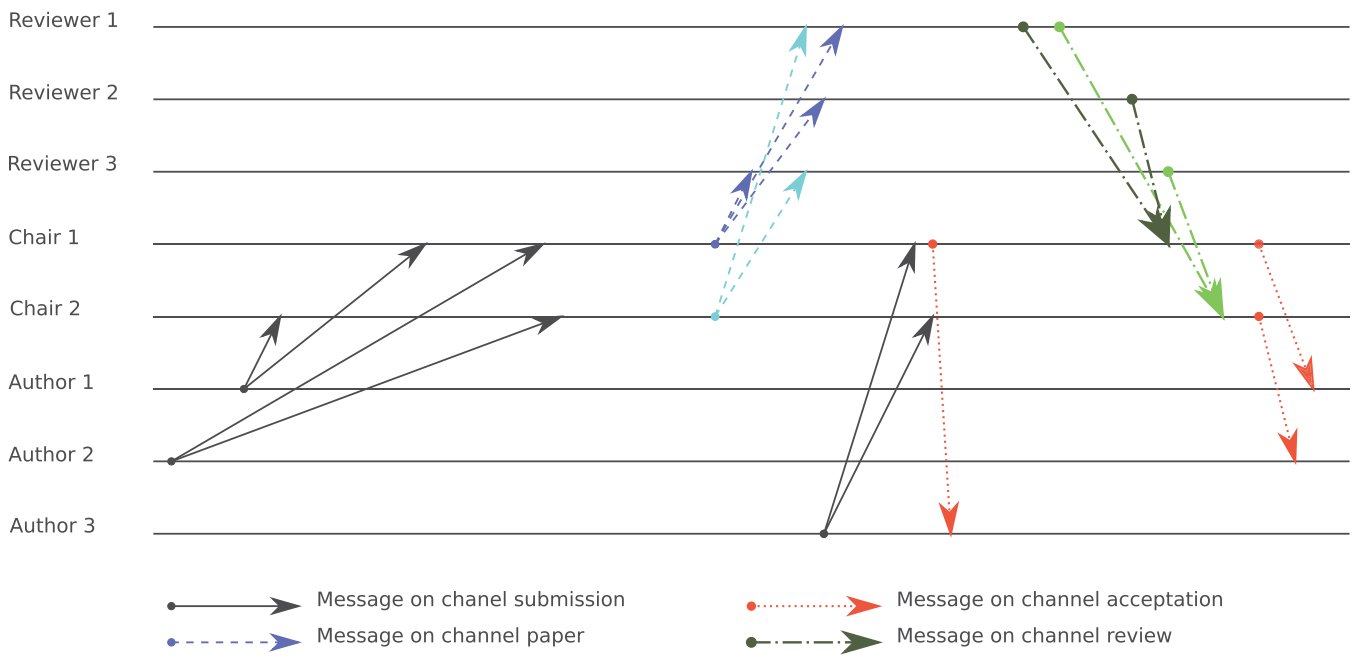

Fig. 1. An execution for the reviewing example. (For interpretation of the colors in the figure(s), the reader is referred to the web version of this article.)

with domain X that maps any $x \in X$ to e. The notation $[f \text{ EXCEPT } ![e_1] = e_2]$ is a function that is equal to the function f except at point e_1, where its value is e_2. Records are functions with domain the names of the fields. As a shorthand, $r[\text{“}field\text{”}]$ is written $r.field$, and $[r \text{ EXCEPT } !.field = val]$ is a new record that is equal to r except for the field $field$ which gets val. In the val expression, @ can be used to refer to the initial value of the field: $[r \text{ EXCEPT } !.count = @ + 1]$ means $[r \text{ EXCEPT } !.count = r.count + 1]$.

Modules are used to structure complex specifications. A module contains constant declarations, variable declarations, and definitions. A module can *extend* other modules, importing all their declarations and definitions. A module can also be an *instantiation* of another module. The module $MI \triangleq \text{INSTANCE } M \text{ WITH } q_1 \leftarrow e_1, q_2 \leftarrow e_2 \ldots$ is an instantiation of module M, where each symbol q_i is replaced by e_i (q_i are identifiers specifying constants or variables of module M, and e_i are expressions). Then $MI!x$ references the symbol x of the instantiated module.

Other than constant and variable declarations, a module contains definitions in the form $Op(arg_1, \ldots, arg_n) \triangleq exp$. This defines the symbol Op such that $Op(e_1, \ldots, e_n)$ equals exp, where each arg_i is replaced by e_i. In case of no argument, it is written as $Op \triangleq e$. A definition is just an abbreviation or syntactic sugar for an expression, and never changes its meaning.

The dynamic behavior of a system is expressed in TLA$^+$ as a transition system, with an initial state predicate, and *actions* to describe the transitions. An action formula describes the changes of state variables after a transition. In an action formula, x denotes the value of a variable x in the origin state, and x' denotes its value in the destination state. A prime is never used to distinguish symbols but always means “in the next state”. ENABLED A is a predicate which is true in a state iff the action A is feasible, i.e. there exists a next state such that A is true.

A specification of a system is written as $Init \wedge \Box[Next]_{vars} \wedge \mathcal{F}$, where $Init$ is a predicate specifying the initial states, $\Box$ is the temporal operator that asserts that the formula following it is always true, $Next$ is the transition relation, usually expressed as a disjunction of actions, $[Next]_{vars}$ is defined to equal $Next \vee vars' = vars$ ($Next$ with stuttering), and $\mathcal{F}$ expresses fairness conditions. Fairness is usually expressed as a conjunction of weak or strong fairness on actions $\text{WF}_{vars}(A_1) \wedge \text{WF}_{vars}(A_2) \ldots \wedge \text{SF}_{vars}(A_i) \ldots$. Weak fairness $\text{WF}_v(A)$ means that either infinitely many A steps occur or A is infinitely often disabled. In other words, an A step must eventually occur if A is continuously enabled. Strong fairness $\text{SF}_v(A)$ means that either infinitely many A steps occur or A is eventually disabled forever. In other words, an A step must eventually occur if A is repeatedly enabled.

System properties are specified using linear temporal logic (LTL). $\Box\phi$ means that ϕ holds in every suffix of the behavior. $\Diamond\phi$ is defined to equal $\neg\Box\neg\phi$ and means that ϕ eventually holds in a subsequent state. $\psi \rightsquigarrow \phi$ is defined to equal $\Box(\psi \Rightarrow \Diamond\phi)$ and means that, whenever ψ holds, then later ϕ holds.

3. Overview of the verification framework

The goal of the framework is to check the compatibility or the correction of a composition of peers under specific properties on the communication. The key feature is a strict separation of concerns between the specification of the peers and the specification of the communication properties. So, the distributed system consists in the product of two transition systems: the composition of peers and the communication model. Both are labeled by localized communication events and

are written in separate TLA$^+$ modules that are connected during the verification process carried out by model checking using TLC.

In this section, the initial presentation only considers point-to-point and multicast communication to avoid introducing too many concepts. Convergecast communication will be added later (Section 4.2) with minor changes.

3.1. Specification of a composition of peers

The specification of a composition of peers is a TLA$^+$ module that describes the state of each peer in the distributed system and specify their behavior according to transition predicates (actions). There is no restriction on the design of the specification of the composition as long as there is at most one communication action (send, receive or ignore) per transition. The actions in the composition usually consist in a conjunction of a communication action and a local state change. In practice, the state of the composition is usually a vector of every peer's state and the actions are local. During an action in the system, the state of a peer evolves either spontaneously or alongside a communication action. Nevertheless, the framework does not forbid that some state is shared by several peers, or that the peers evolves synchronously. It's up to the designer to decide if the only exchanges between peers occur with the available communication actions, or if some hidden communication channels are used.

3.1.1. Communication actions

The available communication actions provided by our framework follow.

Send. send(sender, receivers, channel, data) is enabled when the emission of a message by *sender* on channel *channel* is possible. We use the channel as an indirection on the notion of destination peer (point-to-point) or destination group (multicast). Besides, it makes it possible to specify systems where channels are not statically associated to a given sender and to a given group of receivers. *receivers* restrict the set of possible receivers for this message: it is usually the set of all peers since channels dynamically account for the destination or destination group but it may be used to narrow a possible set of receivers down or to send a message to an explicit destination. Eventually, the payload of the message is *data* without restriction on its type which can be adapted on a case-by-case basis. This payload is retrieved at delivery.

Receive. receive(receiver, channel, data) is enabled when the reception of a message by *receiver* on channel *channel* that contains *data* is possible. We assume peers cannot prevent a delivery based on the content $data$ of the message: the communication model imposes the message to be received and the content is only available afterwards. Therefore, in practice, a receive action in the specification of a composition has the form $\exists data \in DATATYPE : receive(\_, \_, data) \wedge P(data)$ where $P(data)$ is a transition predicate that covers all the possible values of *data* in *DATATYPE*. This means that the next state of the receiver may depend on data but the enabledness of the reception itself is independent of this value.

Ignore. ignore(peer, channels) is always enabled. It states that *peer* does not expect to receive messages from the channels in *channels* anymore. The channels a peer has not ignored is called the *interest* of this peer. Ignoring a channel cannot be reverted as this would otherwise breaks delivery orderings: ignoring a channel and later getting interested in it again would allow to temporally bypass the message dependencies. The interest is crucial to the specification of some communication properties including multicast communication as detailed later in Section 4.1.1.

3.1.2. Back to the reviewing system

The reviewing system has been described in the PlusCal Algorithm Language, which is translated by the TLA$^+$ tools to a TLA$^+$ specification. An excerpt[1] is given in Fig. 2.

The peers (`processes` in PlusCal language) are the authors, the chairs and the reviewers. Only the reviewers are described in the excerpt. The reviewers have two actions: they can either receive a message on channel `paper` or send a message on channel `review`. For a message to be received on the channel `paper`, the reviewer must not have more than 4 papers to review, and the paper is added to his list of papers to review. For a message to be sent on `review`, it must concern a paper the reviewer have to review, and the paper is removed from his list of papers to review.

3.2. Specification of a communication model

A communication model is responsible for collecting messages sent by the peers, and delivering them to the relevant peers. In our framework, it is a combination of instances of micromodels, each corresponding to a subset of channels of the system. We will first see the structure of the micromodels before explaining how the different micromodels interact to form the communication model.

[1] The complete files, of the example and of the used communication models, are available online [12].

```
MODULE reviewing
 ....
–algorithm reviewing
 ....
fair process Reviewer ∈ IdReviewers
variable
   readinglist = {};  – for each reviewer, the papers he has to review
begin
rl0:                  – listen only on channel "paper"
   await ignore(self, CHANNELS \ {"paper"});
rl1:
   while TRUE do
      either          – receive a paper to review
         await Cardinality(readinglist) ≤ 4;
         with paper ∈ IdPapers do
            await COM!receive(self, "paper", paper);
            readinglist := readinglist ∪ {paper};
         end with ;
      or              – send a review to the chairs
         with paper ∈ readinglist do
            await COM!send(self, IdChairs, "review", ⟨self, paper⟩);
            readinglist := readinglist \ {paper};
         end with ;
      end either ;
   end while ;
end process
end algorithm
```

Fig. 2. The peers (`processes` in PlusCal language) are the authors, the chairs and the reviewers. The reviewers have two actions: they can either receive a message on channel `paper` or send a message on channel `review`. The chairs and the authors (not shown here) have respectively five and two actions.

3.2.1. Micromodels of communication

As just stated, a communication model is a combination of communication properties we call micromodels. A micromodel has to answer the following two essential questions from which six other questions are derived:

q1) When is the emission of a message, on a given channel, by a given peer, possible?
q2) When is the delivery of a message, on a given channel, to a given peer, possible?

In order to address these questions, the specification of a micromodel, a TLA$^+$ module, relies on its current state.

q3) Which information must the state carry?

Besides, a micromodel can be parameterized by constants in the module. For example, a micromodel corresponding to the property "the number of messages in transit is capped" has a parameter: the bound, and its state is the set of messages in transit. An emission requires the cardinality of this set not to exceed the limit and a delivery is always possible. The sole purpose of this micromodel is to limit the number of messages in transit and it imposes no constraint on the deliveries: the basics of the communication such as "a message must have been sent before it is delivered" are part of another micromodel involved alongside. Micromodels are complementary with minimum overlap.

The remaining questions are then:

q4) What is the initial state?
q5) How does the state evolve after an emission?
q6) How does the state evolve after a delivery?
q7) How does the state evolve after some channels are ignored by a peer?

Since we aim at modeling both point-to-point and multicast communication, the answer to the last two questions is not trivial. Consider a micromodel that specifies either point-to-point or multicast communication and let us combine it with our example cap micromodel, characterized by a set of messages in transit. When performing a reception in this micromodel, the resulting state depends on the communication paradigm: the delivered message must be removed when the communication is point-to-point (the message is not in transit anymore) but the set may be left unchanged when the communication is multicast (the message remains in transit for further deliveries). We therefore distinguish two classes of micromodels: physical and non-physical. Physical micromodels specify when a message is removed from the communication model because it can no longer be received. Non-physical models specify predicates that control the sending and receiving

```
MODULE message_cap
EXTENDS Naturals, FiniteSets
CONSTANTS ID, PEERS, CHANNEL, BOUND  Maximum nb of messages in transit
PhysicalMicromodel ≜ FALSE  q8

The state consists of one field: the ids of the messages in transit.
TypeInvariant(s) ≜ s ∈ [idInTransit : SUBSET ID]  q3
Init ≜ [idInTransit ↦ {}]  q4

usedIds(s) ≜ s.idInTransit

preSend(s, id, from, to, channel, data) ≜  q1
    Cardinality(s.idInTransit) < BOUND
postSend(s, id, from, to, channel, data) ≜  q5
    [s EXCEPT !.idInTransit = s.idInTransit ∪ {id}]

preReceive(s, id, to, channel, data) ≜ TRUE  q2
postReceive(s, id, to, channel, data, remove) ≜  q6
    IF remove THEN [s EXCEPT !.idInTransit = @ \ {id}] ELSE  s

postIgnore(s, peer, chan_set, removedIds) ≜  q7
    [s EXCEPT !.idInTransit = s.idInTransit \ removedIds]
```

Fig. 3. TLA$^+$ module of a parameterized micromodel that caps the number of messages in transit. The annotations q1 to q8 indicate the answers to the questions a micromodel has to address.

of messages but are not concerned by the lifetime of a message. This information is fed to non-physical models by the physical models so they can evolve in a consistent way.

q8) Is the micromodel physical?

The specification of any micromodel, such as our example (message cap) whose TLA$^+$ specification is in Fig. 3, must answer each of the eight questions q1 to q8. The answer to q8 is a boolean $PhysicalMicromodel$; q1 and q2 are predicates $preSend$ and $preReceive$ that depend on the current state of the micromodel, the sender or receiver, the channel, and the data contained in the message; q3 is a type predicate $TypeInvariant$ depending on the current state s; q4 is the value $Init$ of the initial state; q5, q6, and q7 are the values $postSend$, $postReceive$, $postIgnore$ of the state after the operation. $postSend$ and $postReceive$ share the interface of $preSend$ and $preReceive$, $postIgnore$ depends on a peer and set of channels to ignore. Additionally, in the specification of non-physical micromodels, $postReceive$ has an additional boolean parameter $remove$ stating whether the received message should be removed or kept in transit, and $postIgnore$ has a set $removedIds$ of messages to remove.

3.2.2. Assembly of a communication model

The following details an example communication model whose structure is summed up in Fig. 4. This assembly states that, among channels a, b, c, d, e, and f, the communication has the property of a given micromodel on channels a, b, and c (say a message ordering property) and that another property (hence another instance of a micromodel such as the message cap micromodel) is associated to channels c and d. Overlaps are possible: communication on channel c has both the message ordering and the message cap properties. A micromodel can also be instantiated more than once: the first micromodel can be instantiated again on channels e and f which would mean messages on e and f are ordered, messages on a, b, and c are ordered, but there is no guarantee on the ordering of a message of the first group and a message of the second group.

As stated earlier, a physical micromodel dictates when a message no longer exists in the communication model (e.g. after the first delivery if the physical micromodel is point-to-point communication) and the information is used by the non-physical micromodels to update their local state. This implies that every channel must be associated to exactly one physical micromodel. Especially, the sets of channels of physical micromodels must not overlap. Otherwise, two physical micromodels could disagree on whether to remove a message on a shared channel. However, the restriction does not apply to non-physical micromodels: the sets of channels may overlap or extend beyond the domains of physical micromodels. Given a communication model that is part point-to-point, part multicast, it is possible to limit the number of messages in transit on the whole communication model with a message cap instance that encompasses the domains of both the point-to-point and multicast physical micromodels.

The architecture of the reviewing system is given in Fig. 5. As an author writes to all the chairs, the communication is multicast on channel `submission`. A chair forwards the papers it handles to all the reviewers, the communication is also multicast on channel `paper`. A chair waits for all the reviews before deciding on acceptance, so the communication is convergecast on channel `review`. The acceptance result is a message between a chair and an author so the communication is point-to-point on channel `acceptation`. In addition, ordering of delivery is required to ensure that the chairs get the

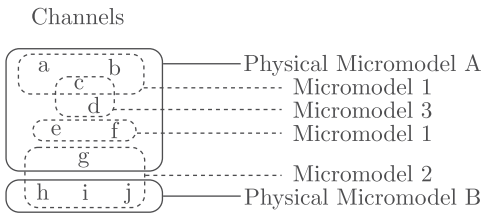

Fig. 4. A communication model built as a combination of micromodels. Each channel is associated to a unique physical micromodel.

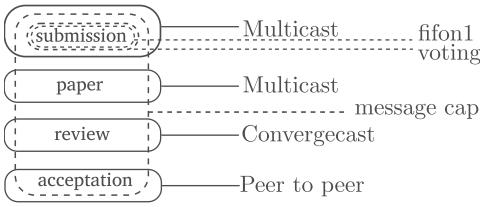

Fig. 5. Communication model for the reviewing system example, built as a combination of micromodels.

MODULE *reviewing*

CONSTANT $NbAuthors$, $NbChairs$, $NbReviewers$,
$NbMinReviews$, $NbMaxReviews$, $Capacity$

$CHANNELS \triangleq$ {"submission", "paper", "review", "acceptation"}
$COMMODELS \triangleq$ {
$[name \mapsto$ "multicast", $params \mapsto [chan \mapsto$ {"submission"}, $min \mapsto 1$, $max \mapsto NbChairs]]$,
$[name \mapsto$ "multicast", $params \mapsto [chan \mapsto$ {"paper"}, $min \mapsto NbMinReviews$, $max \mapsto NbMaxReviews]]$,
$[name \mapsto$ "convergecast", $params \mapsto [chan \mapsto$ {"review"}, $min \mapsto NbReviews$, $max \mapsto NbReviews]]$,
$[name \mapsto$ "p2p", $params \mapsto [chan \mapsto$ {"acceptation"}$]]$,
$[name \mapsto$ "fifon1", $params \mapsto [chan \mapsto$ {"submission"}$]]$,
$[name \mapsto$ "voting", $params \mapsto [chan \mapsto$ {"submission"}, $bound \mapsto 1]]$,
$[name \mapsto$ "message_cap", $params \mapsto [chan \mapsto CHANNELS$, $bound \mapsto Capacity]]$}

$COM \triangleq$ INSTANCE $multicom$ WITH
$PEERS \leftarrow IdAuthors \cup IdChairs \cup IdReviewers$,
$COM \leftarrow COMMODELS$,
$CHANNEL \leftarrow CHANNELS$
....

Fig. 6. An excerpt of the conference reviewing system. $COMMODELS$ specifies the properties of the channels (e.g. `submission` is a multicast 1-NbChairs channel).

papers in the same order and assign the same number to a paper (*FIFO n–1* on channel `submission`), and a constraint on sent messages exists to limit the number of submissions by an author (*voting* on channel `submission`). Eventually, to limit explosion when model checking, a message cap is set to all the channels. Note that this bounded model-checking technique is to be used only to find bugs, as it restricts the checked executions.

3.2.3. Interlinking of the micromodels

The TLA$^+$ module that exposes the three communication operations available for the specification of compositions of peers is called the "multicom". It is instantiated in the specification of the composition of peers with a parameter: the specification of the communication model. The module parameters the desired layout of micromodels of communication and instantiates the resulting communication model which then enables the peers to interact and exchange information. Fig. 6 gives the set up of the communication model of the reviewing example where a communication model COM is instantiated according to a layout of micromodels described in $COMMODELS$, in compliance with Fig. 5.

The multicom is a dispatcher that gathers the local states of the micromodels, checks whether an operation is possible (using the $pre\cdots$ predicates), and how the local states evolve (using the $post\cdots$ values). The multicom also generates and manages the message identifiers: a message has the same identifier across all the micromodels which makes it possible to maintain coherence. When an operation is to be performed, say a reception on channel c, the conjunction of all the $preReceive$ predicates of micromodels associated to c determines whether the reception is possible. If so, the new state of the physical micromodel of c is computed. By comparing it to the former state, the set of messages identifiers that are no longer in use (i.e. removed messages) is computed. It is provided to the non-physical micromodels whose state is updated afterwards.

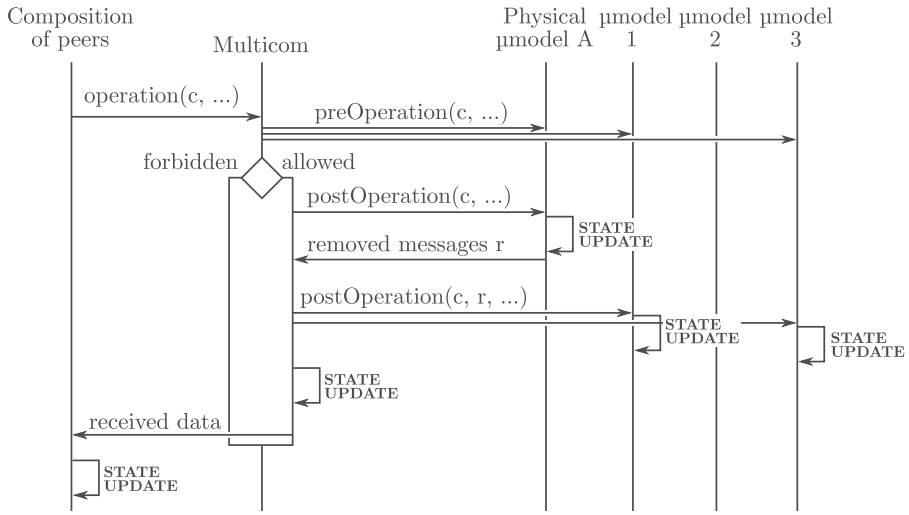

Fig. 7. Illustration of the dispatcher role of the multicom. An operation on channel c is initiated by a peer of a composition. It corresponds to a unique atomic TLA$^+$ action. The communication model is described in Fig. 4: for channel c, it does not involve micromodel 2. The conjunction of the guards on the operation determines whether the operation is possible. If so, it is applied on the physical micromodel first and then on the others with knowledge of the removed messages.

MODULE $multicom$

$receive(peer,\ chan,\ data) \triangleq$
 $\exists\, id \in id.in\_use :$ There exists a message,
  $\land\, \forall\, com \in COM :$ that is receivable in all the micromodels the channel is associated to.
   $chan \in com.params.chan \implies ComPreReceive(com,\ id,\ peer,\ chan,\ data)$
  $\land$ LET $physicalCom \triangleq$ Find the physical micromodel associated to the channel.
    CHOOSE $c \in COM : c \in \{com \in COM : chan \in com.params.chan \land ComPhysical(com)\}$
  IN Compute the next state of the physical micromodel.
   LET $postPhysicalState \triangleq ComPostReceivePhysical(physicalCom,\ id,\ peer,\ chan,\ data)$
   IN Check if the physical micromodel decides to remove the received message.
    LET $remove \triangleq (id \notin ComUsedIds(physicalCom,\ postPhysicalState))$ IN
     $\land\, s' = [com \in COM \mapsto$ Apply state update to all concerned micromodels.
      IF $chan \in com.params.chan$ THEN
       IF $com = physicalCom$ THEN $postPhysicalState$
       ELSE $ComPostReceiveNonPhysical(com,\ id,\ peer,\ chan,\ data,\ remove)$
      ELSE $s[com]]$
     Update the id state variable for id generation and reuse.
     $\land\, id' = [id$ EXCEPT $!.in\_use = ComAllUsedNext(AllModels)]$

Fig. 8. TLA$^+$ specification of the reception by a peer of a message on a channel. It finds a message that is receivable in all the micromodels the channel is associated to, and updates the state of these micromodels. The physical micromodel decides if the message is removed and the other micromodels get this information to purge their state.

Fig. 7 is a sequence diagram that gives insight into the process. Note that it is purely illustrative: it actually corresponds to a unique atomic TLA$^+$ action, that is a transition predicate involving the conjunctions of the micromodel-specific predicates. An extract of the *multicom* module is shown Fig. 8. It presents the receive action $receive(peer, chan, data)$ where $peer$ receives a message on the channel $chan$ with content $data$. It consists in finding a message that is receivable in all the micromodels the channel is associated to, and then updating the state of these micromodels. The physical micromodel decides if the message is removed (i.e. it is no longer available for another reception) and the other micromodels get this information to purge their state.

4. Micromodels in detail

In this section, several micromodels are detailed: first two physical micromodels for multicast communication and convergecast communication, and then non-physical micromodels that constrain the emission of messages or their delivery order.

4.1. A physical micromodel for multicast communication

A physical micromodel for asynchronous point-to-point communication can be modeled as a set of messages in transit, initially empty: the *network*. The *send* action is always enabled, and adds the message to the network. Delivering a message

requires it to be in the network and removes it. Obviously, a message is delivered at most once. In order to describe multicast communication that allows multiple deliveries of a message (at most one per peer), the lifespan of a message in transit must be extended to encompass multiple deliveries while making it possible to eventually suppress it.

4.1.1. Lifespan of messages in transit

Sending the messages over and over. A simple solution would send the message again once it has been received so it can be received another time by another peer. There are two problems. This solution does not specify when to stop sending messages again. Second, when considering message-ordered communication where the order of the emissions matters (e.g. messages must be received in their emission order), sending a message again might modify the ordering. For instance, send m_1 followed by m_2, then deliver m_1. The semantics of this solution implies that m_1 is put back in the network and the new ordering is $m_2 \cdot m_1$ instead of $m_1 \cdot m_2$ the actual order of the multicast emission.

Never removing the messages from the network. Were the messages to remain in the network forever, they could be received as many times as needed. Once again however, this might conflict with some ordering policies. Assume that messages *must be* received in their emission order, that is to say the network can be viewed as a global queue, and consider two messages in transit. Even after all the peers have received the first message, since it remains in transit forever, none of them will ever receive the second (not first in queue) and the system will deadlock.

Removing a message from the network once delivered to all the peers. The previous issue is overcome by removing a message from the network after it has been delivered to every peer. Still, this means that all the peers must be ready to receive all the messages in order not to block the system. This requirement is too strong to allow for the verification of interesting and realistic systems: the specification of a peer should not depend on the noise in the environment it takes part in.

Removing when no one is interested. In order not to impose the delivery of a message that a peer has nothing to do with, and never will, we rely on the concept of interest. A peer is interested in some channels only: it expects messages on these channels. Over time, the peer may lose interest in some or all of them: either the expected deliveries have occurred or the peer has ruled out the possibility of ever receiving the messages. Action $ignore$ of the communication model allows a peer to lose interest in a given set of channels, as described in the previous section. The interest of the peers is part of the state of the multicast micromodel. The most sensible behavior would be to remove a message from the network as soon as the last peer interested in the channel of this message receives or ignores it. However, a more generic approach is only a few tweaks away from this main rule.

4.1.2. A generic description for point-to-point, multicast, and one-to-all communication

The proposed operational specification of multicast communication is adapted to become generic and encompass, in particular, point-to-point communication. Consider two parameters of the communication denoted MIN and MAX.

- MIN is the minimal number of times a message must be received before it is removed from the network when no peer is interested;
- MAX is the maximal number of times a message can be received before it is removed from the network regardless of the interest.

Let N denote the number of peers in the system. Up until now, we have described *multicast(0,N)* communication: a message is removed from the network when the corresponding channel does not interest any peer.

Point-to-point communication corresponds to *multicast(1,1)*. Indeed, a message must be received at least once before it can be removed from the network and must not be received more than once. This means it is immediately removed from the network following the first reception, never before. Similarly, *multicast(1,N)* corresponds to multicast communication where at least one peer must receive a message before it is removed, and *multicast(N,N)* models one-to-all communication where a message must be received by all the peers (including the sender) before it is removed from the network, regardless of the interest. MIN and MAX can also take any other value between 0 and N.

Fig. 9 illustrates the differences between *multicast(0,N)*, *multicast(1,1)*, and *multicast(N,N)* with a common example scenario involving a global message-ordering policy (the network consists of a global common queue of messages). It shows the possible constraints and deadlocks that arise from combining two micromodels: a variant of multicast, and the global ordering policy.

The complete specification of the proposed generic micromodel is presented in Fig. 10 and consists of two state variables $network$ and $interest$. The first one is a set of messages in transit which expands after each new emission; the second one contains, for each peer, the set of channels that it has not ignored (i.e. its interest). A message is composed of metadata including its unique identifier provided by the multicom, the sender, channel, and a set $receivedBy$ of peers it has already been delivered to in order to prevent multiple deliveries to the same peer (see $preReceive$). After a delivery (see $postReceive$), the receiver is added to the message's $receivedBy$ set but the message remains in the $network$ unless MAX

	$interest$			$network$		
Operation	p_1	p_2	p_3	*(0,N)*	*(1,1)*	*(N,N)*
	$\{a, b\}$	$\{a, b\}$	$\{a, b\}$	$\emptyset$	$\emptyset$	$\emptyset$
p_1 i a	$\{b\}$	$\{a, b\}$	$\{a, b\}$	$\emptyset$	$\emptyset$	$\emptyset$
p_1 ! a	$\{b\}$	$\{a, b\}$	$\{a, b\}$	a	a	a
p_1 ! b	$\{b\}$	$\{a, b\}$	$\{a, b\}$	$a \cdot b$	$a \cdot b$	$a \cdot b$
p_2 ? a	$\{b\}$	$\{a, b\}$	$\{a, b\}$	$a \cdot b$	b	$a \cdot b$
p_2 i a	$\{b\}$	$\{b\}$	$\{a, b\}$	$a \cdot b$	b	$a \cdot b$
p_3 ? a	$\{b\}$	$\{b\}$	$\{a, b\}$	$a \cdot b$	$\perp$[1]	$a \cdot b$
p_3 i a	$\{b\}$	$\{b\}$	$\{b\}$	b		$a \cdot b$
p_1 ? b	$\{b\}$	$\{b\}$	$\{b\}$	b		$\perp$[2]

[1] The message is not in the network anymore ($MAX = 1$).
[2] The message on a is still in the network ($MAX = N$) and must be received first according to the current ordering policy.

Fig. 9. Evolution of the state of the communication according to different instances of *multicast(*,*)* with global message-ordering, channels a and b, and $N = 3$ peers $(p_i)_{i \in 1..N}$. The network is represented by a queue. ! means "send", ? means "receive", i means "ignore".

receptions have occurred (after the first delivery in point-to-point, i.e. *multicast(1,1)*). When channels are ignored by a peer (see $postIgnore$), the $interest$ is updated, and messages that no longer interest any peer are removed from the $network$ unless they have not been delivered at least MIN times yet.

4.2. A physical micromodel for convergecast communication

Convergecast communication, or N-to-1 communication, is the dual of multicast. Where multicast sends a message to a set of peers, convergecast consists in the reception of a set of messages in one action [31,27,21]. This communication primitive is interesting as a building block for more complex architectures (e.g. join in a fork-join schema). Contrary to the multicast micromodel, the convergecast micromodel could actually be simulated with point-to-point communication: to receive messages from a set of peers, a peer receives them individually until it has received all of them. However, having a dedicated micromodel is useful, to make explicit a convergence operation in an algorithm, and to reduce the interleaving in the reception. As the transition of multireception only occurs when all the messages are available, this reduces the $n!$ possible executions coming from the interleavings of individual receptions of each message to *one* transition. Since our framework uses the TLC model checker to verify the correctness, this point alone suffices to justify convergecast.

Like multicast, convergecast is parameterized by MIN and MAX. A multireception of n messages is enabled if $MIN \leq n \leq MAX$. Note that the n messages must come from n distinct peers: a multireception never delivers, in the same action, two messages issued by a same peer. Point-to-point communication exactly corresponds to *convergecast(1,1)*. *convergecast(N,N)* models all-to-one communication where a message must have been sent by all the peers (including the receiver) for the multireception to be enabled. *convergecast(1,N)* allows to receive an arbitrary subset of the messages in transit to the receiver.

In addition to the *send*, *receive* and *ignore* communication actions, a new action is added to the *multicom* module: *multireceive(receiver, channel, datas)*. This action is always disabled in the point-to-point and multicast micromodels. In the convergecast micromodel, it is enabled when the reception by peer *receiver* of a set of messages sent by distinct peers on channel *channel*, and such that the bag holding the content of the messages is *datas*, is possible. As for *receive*, we assume peers cannot prevent a delivery based on the content of the messages, and a reception action in the specification of a composition has the form $\exists datas \in SubBag(\ldots) : multireceive(\_, \_, datas) \wedge P(datas)$.

The specification of the convergecast is presented Fig. 11. The precondition of multireceive takes ids a set of message identifiers. It checks that the number of messages is in the bound of the convergecast micromodel, that each message is actually deliverable, that all the messages come from different peers, and builds $datas$ as the bag of all message contents. As in point-to-point communication, a received message is removed from the messages in transit as it can be received at most once.

As the point-to-point and multicast physical micromodels, the convergecast micromodel is expected to be used in combination with other micromodels, such as the cap micromodel or any of the ordering micromodels described in the next section. For instance, one can combine convergecast and *FIFO 1–1* (described below). In this way, successive multireceptions will get messages from each peer in their send order. In practice, convergecast is mainly used in a fork-join schema and is used without any ordering or with *FIFO 1–1*.

4.3. Constraining micromodels

We provide non-physical micromodels that limit the enabledness of sending or receiving a message. These models implement classical constraints of common communication models. First, we provide generic message-ordering models such as FIFO delivery. We also provide an applicative message-ordering model where priorities between channels can be expressed.

```
MODULE multicast
EXTENDS Naturals, FiniteSets
CONSTANTS ID, PEERS, CHANNEL, DATATYPE, MIN, MAX
PhysicalMicromodel ≜ TRUE
----------
LOCAL Message ≜ [
   id : ID,    Message identifier
   from : PEERS,    Sender
   to : SUBSET PEERS,    Possible receivers
   channel : CHANNEL,    Channel
   data : DATATYPE,    Payload
   receivedBy : SUBSET PEERS]    Peers it has already been delivered to
LOCAL Network ≜ SUBSET Message
LOCAL Interest ≜ [PEERS → SUBSET CHANNEL]
----------
TypeInvariant(s) ≜ s ∈ [network : Network, interest : Interest]
Init ≜ [network ↦ {}, interest ↦ [peer ∈ PEERS ↦ CHANNEL]]
usedIds(s) ≜ {m.id : m ∈ s.network}
----------
postIgnore(s, peer, chan_set) ≜
   LET new_peer_interest ≜ s.interest[peer] \ chan_set IN
   [s EXCEPT !.interest = [@ EXCEPT ![peer] = new_peer_interest],
           !.network = {m ∈ @ :
              ∨ m.channel  ∈ new_peer_interest
              ∨ Cardinality(m.receivedBy) < MIN    not received enough
              ∨ ∃ p ∈ PEERS \ {peer} : m.channel ∈ s.interest[p]}]    another peer is still interested


 Emission: the message is added to the nework
preSend(s, id, from, to, channel, data) ≜ TRUE
postSend(s, id, from, to, channel, data) ≜
   [s EXCEPT !.network = @ ∪ {[id ↦ id, from ↦ from, to ↦ to, channel ↦ channel,
                               data ↦ data, receivedBy ↦ {}]}]

preReceive(s, id, to, channel, data) ≜
   ∃ m ∈ s.network :    The metadata of a message in transit match.
      ∧ m.id = id ∧ to ∈ m.to ∧ m.channel = channel ∧ m.data = data ∧ channel ∈ s.interest[to]
      ∧ to ∉ m.receivedBy    The peer has not received it yet.

postReceive(s, id, to, channel, data) ≜
   LET m ≜ (CHOOSE x ∈ s.network : x.id = id) IN
   LET newm ≜ [m EXCEPT !.receivedBy = @ ∪ {to}] IN
   IF ∧ Cardinality(newm.receivedBy) < MAX
      ∧ ∨ Cardinality(newm.receivedBy) < MIN
        ∨ ∃ p ∈ PEERS : newm.channel ∈ s.interest[p]
      THEN [s EXCEPT !.network = (s.network \ {m}) ∪ {newm}]    keep the message
      ELSE  [s EXCEPT !.network = (s.network \ {m})]    drop it
```

Fig. 10. TLA$^+$ specification of the generic multicast physical micromodel. The parameters MIN and MAX make it possible to use different instances of this module to model multicast communication, one-to-all communication, point-to-point communication, or in-between variants.

The previously presented micromodel that caps the number of messages in transit is another constraining model. A dedicated micromodel for voting and bounding the number of sent messages is also specified. These various micromodels can all be combined together and their diversity demonstrates the power of our framework. New micromodels (e.g. for content filtering) are easily specified in a few lines.

4.3.1. Generic message-ordering micromodels

We provide non-physical micromodels for a large set of generic message-ordering policies. A detailed description, both axiomatic and operational, of classic point-to-point communication models is found in [11]. They include the following:

- *RSC* Realizable with Synchronous Communication [8,21]. The emission of a message is immediately followed by its delivery. Viewed atomically, it corresponds to synchronous communication.
- *FIFO n–n* Messages are globally ordered and are delivered in their emission order.
- *FIFO 1–n* Messages sent from a same peer are delivered in their emission order.
- *FIFO n–1* On a given peer, messages are received in their absolute emission order.

```
1  ──────────────────── MODULE convergecast ────────────────────
2  EXTENDS FiniteSets, Bags, Naturals
3  CONSTANTS ID, PEERS, CHANNEL, DATATYPE, MIN, MAX
4  PhysicalMicromodel ≜ TRUE
5  ──────────────────────────────────────────────────────────────
6  LOCAL Message ≜ [
7     id : ID,    Message identifier
8     from : PEERS,    Sender
9     to : SUBSET PEERS,    Possible receivers
10    channel : CHANNEL,    Channel
11    data : DATATYPE]    Payload
12 LOCAL Network ≜ SUBSET Message
13 ──────────────────────────────────────────────────────────────
14 TypeInvariant(s) ≜ s ∈ [network  : Network]
15 Init ≜ [network  ↦ {}]
16 usedIds(s) ≜ {m.id : m ∈ s.network}
17 ──────────────────────────────────────────────────────────────
18 postIgnore(s, peer, chan_set) ≜ s

20 Like in point-to-point / multicast
21 preSend(s, id, from, to, channel, data) ≜ TRUE
22 postSend(s, id, from, to, channel, data) ≜
23    [s EXCEPT !.network = @ ∪ {[id ↦ id, from ↦ from, to ↦ to, channel ↦ channel, data ↦ data]}]

25 preMultiReceive(s, ids, to, channel, datas) ≜
26    ∧ Cardinality(ids) ∈ MIN .. MAX
27    ∧ ∀ id ∈ ids :
28          ∃ m ∈ s.network :
29             ∧ m.id = id
30             ∧ to ∈ m.to
31             ∧ m.channel = channel
32    ∧ ∀ i, j ∈ ids : i ≠ j ⟹   the messages come from different peers
33          (CHOOSE m ∈ s.network : m.id = i).from ≠ (CHOOSE m ∈ s.network : m.id = j).from
34    ∧ datas = BagOfAll(LAMBDA m : m.data, SetToBag({m ∈ s.network : m.id ∈ ids}))

36 postMultiReceive(s, ids, to, channel, datas) ≜   the messages are removed
37       [s EXCEPT !.network = {m ∈ @ : m.id ∉ ids}]

39 preReceive(s, id, to, channel, data) ≜   a single receive is allowed if MIN = 1
40       preMultiReceive(s, {id}, to, channel, SetToBag({data}))
41 postReceive(s, id, to, channel, data) ≜
42       postMultiReceive(s, {id}, to, channel, SetToBag({data}))
43 ──────────────────────────────────────────────────────────────
```

Fig. 11. TLA$^+$ specification of the generic convergecast physical micromodel. The distinctive feature with regard to p2p/multicast is *preMultiReceive* which checks that the number of messages is correct, that they are in transit, that they come from different peers, and which constructs the bag of the message payloads.

- *FIFO 1–1* Messages between a couple of peers are delivered in their emission order. Messages from/to different peers are independently delivered.
- *causal* Messages are delivered according to the causality of their emission [22]. 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.

The communication models in [11] are only for point-to-point communication. Moreover they are standalone, including the management of the lifespan of messages in transit. They have been rewritten to obtain specifications of their ordering policies that follow the previous conventions as pluggable, multicast-ready and convergecast-ready micromodels that make use of the concept of interest and rely on message histories. The *FIFO n–n* micromodel is shown in Fig. 12.

4.3.2. Applicative message-ordering micromodel

We also provide a micromodel where priorities are assigned to channels, instead of ordering the deliveries with regard to the emission events. If a channel a has a higher priority than a channel b, then the existence of a message on a blocks the delivery of any message on b, for the same receiver. These messages will become deliverable only after the message on a has been received. A classic use can be found in abortion messages. If the communication model allows the system to take other messages over the abortion one, this results in a seemingly unresponsive behavior to abortion or presents security issues.

```
MODULE fifonn
CONSTANTS ID, PEERS, CHANNEL
PhysicalMicromodel ≜ FALSE
---
LOCAL Message ≜ [
            id :       ID,
            from :     PEERS,
            to :       SUBSET PEERS,
            channel :  CHANNEL,
            history  : SUBSET ID]
LOCAL Network ≜ SUBSET Message
---
TypeInvariant(s) ≜ s ∈ [network  : Network]
Init ≜ [network  ↦ {}]
usedIds(s) ≜ {m.id : m ∈ s.network}
---
postIgnore(s, peer, chan_set, removedIds) ≜ s

preSend(s, id, from, to, channel, data) ≜ TRUE
postSend(s, id, from, to, channel, data) ≜
    [s EXCEPT !.network = @ ∪ {[id ↦ id, from ↦ from, to ↦ to, channel ↦ channel,
                                history ↦ UNION {m.history ∪ {m.id} : m ∈ s.network}]}]

preReceive(s, id, to, channel, data) ≜
    ∃ m ∈ s.network :
        ∧ m.id = id ∧ to ∈ m.to ∧ m.channel = channel
        ∧ ¬∃ m2 ∈ s.network : m2.id ∈ m.history   there is no preceding message in transit
postReceive(s, id, to, channel, data, remove) ≜
    IF remove
    THEN [s EXCEPT !.network = {[mes EXCEPT !.history = @ \ {id}] : mes ∈ {mes2 ∈ s.network : mes2.id ≠ id}}]
    ELSE  s

preMultiReceive(s, ids, to, channel, datas) ≜
    ∀ id ∈ ids : ∃ m ∈ s.network :
        ∧ m.id = id ∧ to ∈ m.to ∧ m.channel  = channel
        ∧ ¬∃ m2 ∈ {mm ∈ s.network : mm.id ∉ ids} : m2.id ∈ m.history
postMultiReceive(s, ids, to, channel, datas, remove) ≜
    [s EXCEPT !.network = {[mes EXCEPT !.history = @ \ ids] : mes ∈ {mes2 ∈ s.network : mes2.id ∉ ids}}]
```

Fig. 12. TLA$^+$ Module of the *FIFO n–n* Micromodel. A message is deliverable (*preReceive*) if its history does not contain another message in transit, which must be delivered before.

An extract of the TLA$^+$ model is shown Fig. 13. Priorities are modelled by a set of channel pairs that parameterizes the micromodel (*BLOCKS* constant): $\langle a, b\rangle \in BLOCKS$ means that a has a higher priority than b. A reception of a message on $channel$ is enabled (*preReceive*) if there is no message for the same peer on another channel with a higher priority.

4.3.3. Message cap micromodel

This micromodel ensures that the number of messages in transit is capped by an upper bound. It was presented in Section 3 and its TLA$^+$ specification is in Fig. 3.

4.3.4. Voting micromodel

The voting micromodel limits the number of messages a peer can send on a set of channels during an execution. Once a peer has reached the limit, its send action on these channels is permanently disabled. While the *message cap* micromodel disables sending for all peers and by looking at the current number of messages in transit, the *voting* micromodel disables sending per peer and by taking into account its past actions. This model is especially useful to implement voting by setting the limit to 1: no peer can send a message (i.e. vote) more than once on the configured channels. Another use is to limit a cyclic behavior to occur a bounded number of times. This reduces the state space and accelerates model checking of the system. As said earlier, note that this bounded model-checking technique is to be used only to find bugs, and some liveness properties may become invalid on these finite executions.

The *voting* module is shown in Fig. 14. It consists in keeping a state that counts the number of sent messages per peer (field *sent*). This state is used to allow send (*preSend*) and is then updated (*postSend*).

```
MODULE priority
EXTENDS Naturals, FiniteSets
CONSTANTS ID, PEERS, CHANNEL, BLOCKS
  BLOCKS is a set of channel pairs: ⟨ca, cb⟩ means that a message on ca blocks the delivery of a message on cb.

LOCAL Message ≜ [id : ID, to : SUBSET PEERS, channel : CHANNEL]
Init ≜ [network ↦ {}]

...
preSend(s, id, from, to, channel, data) ≜ TRUE
postSend(s, id, from, to, channel, data) ≜
  [s EXCEPT !.network = @ ∪ {[id ↦ id, to ↦ to, channel ↦ channel]}]

preReceive(s, id, to, channel, data) ≜
    ∃ m ∈ s.network :
        ∧ m.id = id ∧ to ∈ m.to ∧ m.channel = channel
        ∧ ¬∃ m2 ∈ s.network :  there is no other message in transit for this peer with a higher priority
            ∧ to ∈ m2.to ∧ ⟨m2.channel, m.channel⟩ ∈ BLOCKS
postReceive(s, id, to, channel, data, remove) ≜
    IF remove THEN [s EXCEPT !.network = {mes ∈ @ : mes.id ≠ id}] ELSE  s
...
```

Fig. 13. TLA$^+$ Module of the *priority* micromodel (extract).

```
MODULE voting
EXTENDS Naturals, FiniteSets
CONSTANTS PEERS, CHANNEL, BOUND  Maximum number of sent messages per peer
PhysicalMicromodel ≜ FALSE

TypeInvariant(s) ≜ s ∈ [sent : [PEERS → Nat]]
Init ≜ [sent ↦ [peer ∈ PEERS ↦ 0]]

usedIds(s) ≜ {}
postIgnore(s, peer, chan_set, removedIds) ≜ s

preSend(s, id, from, to, channel, data) ≜
    s.sent[from] < BOUND
postSend(s, id, from, to, channel, data) ≜
    [s EXCEPT !.sent = [@ EXCEPT ![from] = @ + 1]]

preReceive(s, id, to, channel, data) ≜ TRUE
postReceive(s, id, to, channel, data, remove) ≜ s

preMultiReceive(s, ids, to, channel, datas) ≜ TRUE
postMultiReceive(s, ids, to, channel, datas, remove) ≜ s
```

Fig. 14. TLA$^+$ Module of the *voting* micromodel.

4.4. Analysis of the example

Getting back to our introductory example, let's consider in detail its description. It uses four channels: *submission* from authors to chairs (*multicast* to all, and *voting* with bound 1 to limit the number of submissions), *paper* from chairs to reviewers (*bounded multicast* based on the number of expected reviews), *review* from reviewers to chairs (*convergecast*), and *acceptation* from chairs to authors (*point-to-point*). Additionally, all chairs need to attribute the same number to a given paper, and without internal coordination between the PC chairs, the authors must use a totally ordered multicast so that the papers are delivered in the same order to all the chairs. As demonstrated in Section 5.3.3, this is achieved with the *FIFO n–1* ordering model on the *submission* channel.

The verified properties are both safety and liveness:

- Safety: one author is handled by exactly one chair:

$$\text{LET } handledAuthors(pres) \triangleq \{papers[pres][id].author : id \in \text{DOMAIN } papers[pres]\} \text{ IN}$$
$$\Box(\forall p1, p2 \in IdPresidents : p1 \neq p2 \Rightarrow handledAuthors(p1) \cap handledAuthors(p2) = \emptyset)$$

Table 1
Number of transitions & distinct states for the reviewing example.

	cap = 1	cap = 2	cap = 3	cap = 4
2 authors, 2 chairs, 2 reviewers	4151 / 1962	63481 / 24204	599625 / 166481	2498881 / 560994
2 authors, 2 chairs, 3 reviewers	26191 / 12820	694385 / 232776	6970035 / 1737452	
2 authors, 2 chairs, 4 reviewers	158289 / 68996	4726501 / 1394440	47312453 / 10664656	
3 authors, 2 chairs, 2 reviewers	86819 / 50200	9382271 / 3530626		

- Liveness: Any submission eventually gets an acceptation/rejection and the authors terminate:

$$AuthorsGetAnswer \triangleq \forall i \in IdAuthors : \Diamond(pc[i] = \text{“}Done\text{”})$$

This system exposes both strict ordering constraints (submissions sent to the chairs), and high interleaving (each reviewer is independently handling the papers it has received). During the development of the system, several bugs were found. For instance, the logic to split the papers among the chairs was faulty with an odd number of chairs and some authors were never receiving their acceptance result; in some cases, the same paper was sent twice to the reviewers, and an unfortunate (but legal) interleaving in the reception of the reviews led to two acceptance messages to the same author. This system, albeit simple, already experiences enough communication interactions to warrant formal verification.

In Table 1, we present some results obtained from running TLC, the TLA$^+$ model checker. The message cap on the number of messages in transit is instrumental to avoid state explosion as it ensures that messages are not delayed for too long.

5. Properties of multicast and convergecast communication

This section presents results that allow to compare models. First, these results are essential to substitutability, the ability to replace one model with another, without having to redo the proofs. We say that a communication model M_1 is stricter than a communication model M_2 if M_1 cannot deliver more messages than M_2, or conversely, if any message that M_1 delivers is also deliverable in M_2. Thus, for any system using asynchronous communication, a safety property which is proved with M_2 is necessarily true when substituting M_2 with M_1. Liveness properties are also preserved if the stricter model does not cause more blocking. For instance, *FIFO 1–1* does not block more than unordered communication and liveness properties are preserved, while *RSC* doesn't allow two consecutive send events without a receive event between them, and thus a system may deadlock with *RSC* while progressing with a more liberal model.

Secondly, these results help in differentiating the models. For instance *FIFO n–1* is a model where each peer has an input mailbox, and senders add messages to it. It is sometimes labelled plainly as asynchronous and confused with *FIFO 1–1*. It is actually stricter than *causal* communication. Moreover, it induces totally-ordered communication: two independent multicasts will be delivered in the same order to all the common peers, without any additional coordination. Its dual model *FIFO 1–n*, where each peer has an output mailbox where it puts messages for the senders to retrieve, is peculiar: it is incomparable to *causal* except in point-to-point communication, and it does not induce totally-ordered communication.

Providing hierarchies of message-ordering policies helps developers with substitutability and gives them a better intuition of their properties. In the following we recall the hierarchies for point-to-point communication and give three new hierarchies: for multicast communication, for totally-ordered communication and for convergecast communication.

5.1. Formal specification

To study the relations between the models, the set of executions of each model is formally defined.

5.1.1. Specification of executions

Consider a set of messages M and a set of peers P, let $E \triangleq \{s(p,m) \mid p \in P \wedge m \in M\} \cup \{r(p,m) \mid p \in P \wedge m \in M\} \cup \{mr(p,mm) \mid p \in P \wedge mm \in \mathbb{P}(M)\}$ (where $\mathbb{P}(M)$ is the power set of M) be the set of communication events: the disjoint union of the set of send, receive and multireceive events.

An execution σ is a finite or infinite sequence of events such that a message is sent at most once, no message is received more than once on the same peer, and a receive event of a message is preceded by a send event of this message:

$$\begin{aligned}
&\forall p \in P : \forall m \in M : \forall j,k \in dom(\sigma) : \\
&\qquad \forall p' \in P : \sigma_j = s(p,m) \wedge \sigma_k = s(p',m) \Rightarrow j = k && (a)\\
&\qquad \wedge\ \sigma_j = r(p,m) \wedge \sigma_k = r(p,m) \Rightarrow j = k && (b)\\
&\qquad \wedge\ \sigma_j = r(p,m) \Rightarrow \exists i \in dom(\sigma) : \exists p' \in P : \sigma_i = s(p',m) \wedge i < j && (c)
\end{aligned} \tag{1}$$

Additionally, if multireception (convergecast) is allowed, each of the received messages is received at most once, the multireception must be preceded by the emission events of each message, and for $mr(p,mm)$, all messages come from distinct peers:

$$\forall p \in P : \forall j, k \in dom(\sigma) : \tag{2}$$
$$\forall p' \in P : \forall mm, mm' \in \mathbb{P}(M) : \forall m \in M :$$
$$m \in mm \wedge m \in mm' \wedge \sigma_j = mr(p, mm) \wedge \sigma_k = mr(p', mm') \Rightarrow j = k \quad (a)$$
$$\wedge\ \forall mm \in \mathbb{P}(M) : \forall m \in mm : \sigma_j = mr(p, mm) \Rightarrow \exists i \in dom(\sigma) : \exists p' \in P : \sigma_i = s(p', m) \wedge i < j \quad (b)$$
$$\wedge\ \forall mm \in \mathbb{P}(M) : \sigma_j = mr(p, mm) \Rightarrow |\{p' \in P : \exists m \in mm, \exists i \in dom(\sigma) : \sigma_i = s(p', m)\}| = |mm| \quad (c)$$

Point-to-point communication adds an additional constraint on executions as specified by (1): no message is received more than once (whereas multicast communication imposes that no message is received more than once *on the same peer*):

$$\forall p, p' \in P : \forall m \in M : \forall j, k \in dom(\sigma) : \sigma_j = r(p, m) \wedge \sigma_k = r(p', m) \Rightarrow j = k \tag{3}$$

5.1.2. Specification of the generic ordering micromodels

Each communication model is characterized by the set of executions it allows to unfold. For instance, the set of executions of *FIFO n–n* contains all the executions such that if a reception happens before another, the two emissions of the messages must have happened in the same order. The generic message-ordering properties of the micromodels described in 4.3.1 are specified as additional constraints on executions (equations (1), (2) and (3)):

- *RSC* (Realizable with Synchronous Communication)

$$\begin{array}{l}\forall m \in M, \forall p_1, p_2 \in P, \forall i, j \in dom(\sigma) : \\ \quad \sigma_i = s(p_1, m) \wedge \sigma_j = r(p_2, m) \Rightarrow (j = i + 1)\end{array} \tag{4}$$

- *FIFO n–n*

$$\begin{array}{l}\forall m, m' \in M, \forall p_1, p_1', p_2, p_2' \in P, \forall i, j, k, l \in dom(\sigma) : \\ \quad \sigma_i = s(p_1, m) \wedge \sigma_j = s(p_1', m') \wedge \sigma_k = r(p_2, m) \wedge \sigma_l = r(p_2', m') \\ \quad \Rightarrow \big((i < j) \Leftrightarrow (k < l)\big)\end{array} \tag{5}$$

- *FIFO 1–n*

$$\begin{array}{l}\forall m, m' \in M, \forall p_1, p_2, p_2' \in P, \forall i, j, k, l \in dom(\sigma) : \\ \quad \sigma_i = s(p_1, m) \wedge \sigma_j = s(p_1, m') \wedge \sigma_k = r(p_2, m) \wedge \sigma_l = r(p_2', m') \\ \quad \Rightarrow \big((i < j) \Leftrightarrow (k < l)\big)\end{array} \tag{6}$$

- *FIFO n–1*

$$\begin{array}{l}\forall m, m' \in M, \forall p_1, p_1', p_2 \in P, \forall i, j, k, l \in dom(\sigma) : \\ \quad \sigma_i = s(p_1, m) \wedge \sigma_j = s(p_1', m') \wedge \sigma_k = r(p_2, m) \wedge \sigma_l = r(p_2, m') \\ \quad \Rightarrow \big((i < j) \Leftrightarrow (k < l)\big)\end{array} \tag{7}$$

- *FIFO 1–1*

$$\begin{array}{l}\forall m, m' \in M, \forall p_1, p_2 \in P, \forall i, j, k, l \in dom(\sigma) : \\ \quad \sigma_i = s(p_1, m) \wedge \sigma_j = s(p_1, m') \wedge \sigma_k = r(p_2, m) \wedge \sigma_l = r(p_2, m') \\ \quad \Rightarrow \big((i < j) \Leftrightarrow (k < l)\big)\end{array} \tag{8}$$

- *Causal.* This model is the most peculiar as it uses Lamport's well-known causal order [22], denoted $\prec$, and defined as the reflexive transitive closure of $s(p, m) \prec r(p', m)$ (reception is caused by emission) and local order on peer.

$$\begin{array}{l}\forall m, m' \in M, \forall p_1, p_1', p_2 \in P, \forall i, j, k, l \in dom(\sigma) : \\ \quad \sigma_i = s(p_1, m) \wedge \sigma_j = s(p_1', m') \wedge \sigma_k = r(p_2, m) \wedge \sigma_l = r(p_2, m') \\ \quad \Rightarrow \big((\sigma_i \prec \sigma_j) \Rightarrow (k < l)\big)\end{array} \tag{9}$$

5.2. Correlation with the micromodels of the framework

The communication models are described by the executions they unfold, and they are specified in TLA$^+$ in the framework. We now prove that both descriptions are equivalent, i.e. that the models in the framework are correct and complete with regard to the execution-based specifications. The correctness means that all the executions generated by the framework respect the constraints (1)+((4)-(9) depending of the model) (+(2) for convergecast, +(3) for point-to-point). The completeness means that, for a set of peers, the framework generates all the possible executions which conform to (1)+((4)-(9) depending on the model) (+(2) for convergecast, +(3) for point-to-point) and to the peers behaviors, without omitting any of them.

[11] proves the correctness and the completeness of fully integrated point-to-point communication models. Here, the equivalent models are built with two micromodels: the point-to-point part, and the ordering part. In both cases, the orderings are specified in TLA$^+$ using history of messages. [11] proves that these histories exactly encode the orders used in (4) to (9), and the proofs apply to the ordering micromodels presented here.

Nevertheless, we need to prove that the multicast micromodel is correct and complete with regard to (1) and that the convergecast micromodel is correct and complete with regard to (1) $\wedge$ (2).

5.2.1. Multicast communication

Proof (Correctness). To be correct, an execution generated by a micromodel must respect (1):

- $(1.a)$: the TLA$^+$ module "multicom" creates a new message identifier at every send event, so a send event is associated to a unique distinct message.
- $(1.b)$: the precondition of the reception check that the peer is not in the list of the peers that have received the message (line 38, Fig. 10); the post-condition of the reception either adds the receiver to the list of the peers that have received the message (lines 42 and 46), or definitively removes the message from the network (line 47); since a message is sent only once, a message cannot be received again by the same receiver.
- $(1.c)$: to be received, a message must be in the network (line 36) and a new message is added in the network solely by the post condition of a send event, so the message was sent before being received. □

Proof (Completeness). To prove that all the valid executions are generated, we prove that no emission or reception is wrongly disabled.

- *preSend* is always possible (line 30, Fig. 10).
- *postSend* adds the new message in the network (lines 32–33) with an empty set of receivers, which allows the message to be received later by any peer.
- *preReceive* is enabled if the message has not already been received by the peer (lines 36 and 38), so the only disabled receptions are the invalid ones.
- *postReceive* disables future receptions of the same message by the same peer (line 42), which only suppresses invalid executions. It removes the message from the network if:
  - it has been received more than MAX times (lines 43 and 47), which forbids an execution where a message would be received more than MAX times. Regarding formula (1), the micromodel is complete when $MIN = 0$ and $MAX = N$.
  - it has been received more than MIN times and no peer is interested (lines 44-45 and 47): as the interest decreases, no peer will ever be interested again by this message, and no valid execution is omitted.
- *postIgnore* removes a message from the network when no peer is interested (lines 24 and 26). As the interest decreases, no peer will ever be interested again by this message, and no valid execution is omitted. □

5.2.2. Convergecast communication

Proof (Correction). To be correct, an execution generated by a micromodel must respect (2) $\wedge$ (1):

- (1.a): the TLA$^+$ module "multicom" creates a new message identifier at every send event, so a send event is associated to a unique distinct message.
- (2.a): to be received, a message must be in the network (lines 27–28, Fig. 11). After the multireception, the messages are removed from the network (line 37), and since a message is sent only once, a message is only received at most once.
- (2.b): to be received, a message must be in the network (lines 27–28) and a new message is added in the network solely by the post condition of a send event, so the message was sent before being received.
- (2.c): for a multireception, all messages come from distinct peers: lines 32–33.
- (1.b) and (1.c): the reception of a single message is done through the multireception of this message (line 41). In that case, (2.a) yields (1.b) and (2.b) yields (1.c). □

Proof (Completeness). To prove that all the valid executions are generated, we prove that no emission or (multi)reception is wrongly disabled:

- *preSend* is always possible (line 21, Fig. 11).
- *postSend* adds the new message on the network (lines 22–23), which allows the message to be received later by any peer.
- *preReceive* is enabled if:
  - the expected messages are in the network (lines 27–31), which is mandatory for a valid execution;

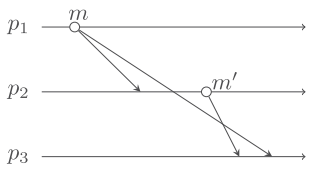

Fig. 15. A *FIFO 1–n* execution which is not *Causal*.

  - there are between MIN and MAX messages to receive (line 26). Regarding formula (2), the micromodel is complete when $MIN = 0$ and $MAX = N$;
  - no two messages are from the same peer (lines 32–32), which is the expected behavior, so no valid execution is forbidden.
- *postReceive* removes all the received messages from the network (line 38), forbidding the messages from being received again later, as expected in (2).
- *postIgnore* does not change the state. □

5.3. Hierarchy of communication models

A communication model X is stricter than a communication model Y when the valid executions for X are included in the valid executions for Y. This is noted $X \to Y$.

5.3.1. Point-to-point communication

Existing results with point-to-point communication reveal the following hierarchy from the strictest model to the most liberal.

Theorem 1 *(Hierarchy of point-to-point communication).*

- *RSC* $\longrightarrow$ *FIFO n–n* $\rightrightarrows$ *FIFO n–1* / *FIFO 1–n* $\rightrightarrows$ *causal* $\longrightarrow$ *FIFO 1–1* $\longrightarrow$ *no-ordering*
- *FIFO n–1 and FIFO 1–n are not comparable.*

These results are essentially proved by first-order logic implication of the constraining properties. The inclusions in *causal* are the only ones that need additional reasoning because of the causal partial order $\prec$. These results are explained and proved in [11] and we refer the reader to it for the details.

5.3.2. Multicast communication

We have extended these results to multicast communication. As it turns out, the multicast hierarchy is not the same as the point-to-point hierarchy: the *FIFO 1–n* $\to$ *causal* inclusion no longer stands. Observe that for all constraints (4), (5), (7), (6), (8), i.e. all except *causal*, the addition of the point-to-point constraint (3) has no influence. Intuitively, the difference between multicast and point-to-point lies in the possibility of receiving a message, which creates a causal link on a peer, while this same message is still in transit for another peer. This can happen only if the message has at least two receptions (on different peers), which is impossible in point-to-point. The resulting hierarchy is the following:

Theorem 2 *(Hierarchy of multicast communication).*

- *RSC* $\longrightarrow$ *FIFO n–n* $\rightrightarrows$ *FIFO n–1* $\longrightarrow$ *causal* $\rightrightarrows$ *FIFO 1–1* $\longrightarrow$ *no-ordering* (*FIFO 1–n* $\rightrightarrows$ *causal*)
- *FIFO 1–n is not comparable to FIFO n–1 and causal.*

Proof.
- *RSC* $\to$ *FIFO n–n*, *FIFO n–n* $\to$ *FIFO n–1*, *FIFO n–n* $\to$ *FIFO 1–n* and *FIFO 1–n* $\to$ *FIFO 1–1*: (1) $\wedge$ (4) $\Rightarrow$ (1) $\wedge$ (5), (1) $\wedge$ (5) $\Rightarrow$ (1) $\wedge$ (6), (1) $\wedge$ (5) $\Rightarrow$ (1) $\wedge$ (7), (1) $\wedge$ (6) $\Rightarrow$ (1) $\wedge$ (8) are first-order logic formulae and easily checked.
- *FIFO n–1* $\to$ *causal*: in (9), $\sigma_i \prec \sigma_j \Rightarrow i < j$ (if one event causally precedes another, then it occurred before in absolute time), thus (1) $\wedge$ (7) $\Rightarrow$ (1) $\wedge$ (9).
- *FIFO 1–n* and *FIFO n–1* are not comparable in point-to-point communication, and point-to-point communication is a special case of multicast communication, thus they are still not comparable.
- To show that *FIFO 1–n* $\not\to$ *causal*, we exhibit a counter-example. Consider three peers p_1, p_2, p_3, two messages m and m' and the execution $s(p_1, m) \cdot r(p_2, m) \cdot s(p_2, m') \cdot r(p_3, m') \cdot r(p_3, m)$ (see Fig. 15). This execution trivially ensures that messages sent from a same peer are delivered in their emission order (no peer sends more than one message) and it's *FIFO 1–n*. However it's not *causal*: the message m' causally depends on m (m' emission on p_2 occurs after m reception on that peer), and m' should not be received on p_3 before m.
- *causal* $\to$ *FIFO 1–1*: the causal order $\prec$ contains the local order of events on a peer, and thus (1) $\wedge$ (9) $\Rightarrow$ (1) $\wedge$ (8). □

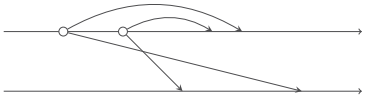

Fig. 16. Totally-ordered but not *FIFO 1–1*.

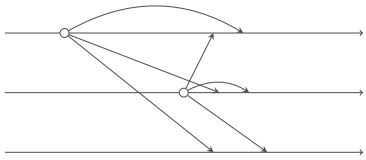

Fig. 17. *Causal* & *FIFO 1–n* but not totally-ordered.

5.3.3. Totally-ordered communication

Some distributed systems feature duplicated peers that are supposed to serve the same purpose and make the overall system more robust. A message that would be sent to a single peer in point-to-point communication is sent, in multicast communication, to all the duplicates. In such cases, it is interesting to guarantee that the messages are delivered in the same order to all the duplicates. This way, the receptions may be viewed as atomic, as if the duplicates were abstracted by a single peer that receives the message in question. This property is called totally ordered multicast and is independent from other ordering policies.

Definition 3 *(Total ordering).* An execution σ is totally ordered if messages are received (when they are received) in the same order on all the peers.

$$\begin{aligned} &\forall m_1, m_2 \in M, \forall p_1, p_2 \in P, \forall i, j, k, l \in dom(\sigma) : \\ &\quad \sigma_i = r(p_1, m_1) \wedge \sigma_j = r(p_1, m_2) \wedge \sigma_k = r(p_2, m_1) \wedge \sigma_l = r(p_2, m_2) \\ &\quad \Rightarrow (i < j) \Leftrightarrow (k < l) \end{aligned} \tag{10}$$

Although we do not propose a micromodel of totally-ordered multicast communication, we have been able to identify the more liberal and already available micromodel that provides the property. In the example described in the following section, we make use of this knowledge in a use case involving both *multicast(0,N)* and *FIFO n–1*.

Theorem 4 *(Totally-ordered communication).*

- *FIFO n–1* $\rightarrow$ *totally-ordered multicast*
- *FIFO n–n* $\rightarrow$ *totally-ordered multicast*
- *RSC* $\rightarrow$ *totally-ordered multicast*
- *Causal, FIFO 1–n, FIFO 1–1 are incomparable to totally-ordered multicast.*

Proof. *FIFO n–1* imposes that on a given peer, messages are received in their absolute emission order (equation (7)). Consider two messages m and m', their send events $\sigma_i = s(\_, m)$ and $\sigma_j = s(\_, m')$, and the receptions of both messages on two peers p_1 and p_2: $\sigma_k = r(p_1, m)$, $\sigma_l = r(p_1, m')$, $\sigma_m = r(p_2, m)$, $\sigma_n = r(p_2, m')$. We show that the messages are received in the same order on p_1 and p_2, i.e. $k < l \Leftrightarrow m < n$. Without loss of generality, assume $k < l$. As the communication is *FIFO n–1* and σ_k and σ_l are on the same peer, (7) gives us $i < j$. Thus, again from (7) on p_2, $m < n$.

The second and third inclusions are a consequence of Theorem 2.

For the last results, it suffices to give a counter-example of a totally-ordered execution which is not *FIFO 1–1* (Fig. 16), and a counter-example of an execution which is *causal* and *FIFO n–1*, but not totally-ordered (Fig. 17). □

The results for multicast communication are illustrated Fig. 18. $Exec_X$ are the sets of execution with ordering X (where *FIFO x–y* is abbreviated *xy*). **ExecT** is the set of totally ordered executions.

5.3.4. Convergecast communication

Regarding message ordering, as all messages in one multireception come from distinct peers and each message is received at most once, convergecast behaves as point-to-point.

Theorem 5 *(Hierarchy of convergecast communication).*

- *RSC* $\longrightarrow$ *FIFO n–n* $\rightrightarrows$ *FIFO n–1* / *FIFO 1–n* $\rightrightarrows$ *causal* $\longrightarrow$ *FIFO 1–1* $\longrightarrow$ *no-ordering*
- *FIFO n–1 and FIFO 1–n are not comparable.*

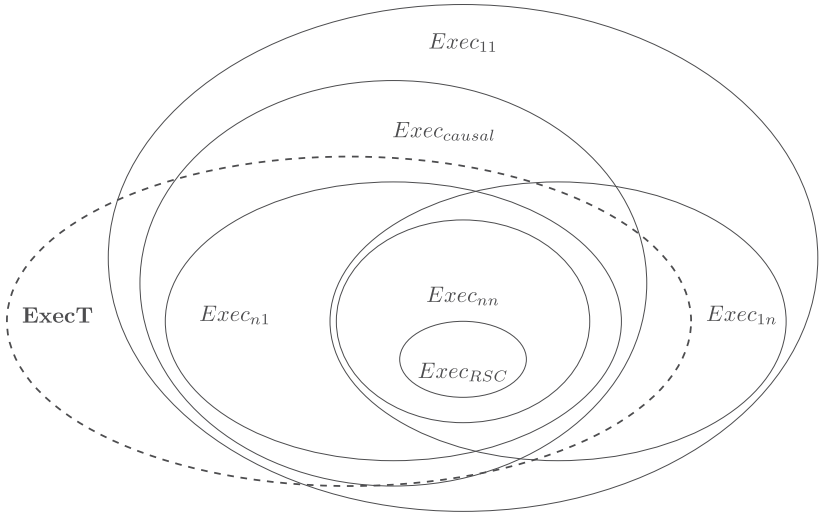

Fig. 18. Inclusion of the sets of executions for multicast communication. The dashed line is totally-ordered communication. All inclusions are strict.

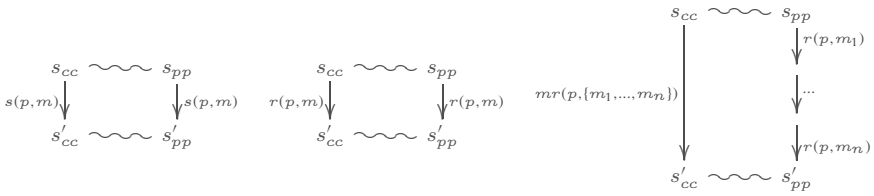

Fig. 19. Simulation of convergecast executions by point-to-point executions.

Proof. The preservation of the hierarchy is proved by giving a bisimulation between point-to-point executions ($s(p, m)$ and $r(p, m)$ events) and convergecast executions ($s(p, m)$, $r(p, m)$ and $mr(p, mm)$ events). An execution is defined as a sequence of transitions (events). Informally, s_i, the state reached in σ after i transitions, is the set of messages in transit and their relations. Formally, all considered transitions are deterministic and a state is defined by the prefix of σ up to i. We note s_{pp} (resp. s_{cc}) for a state in a point-to-point (resp. convergecast) execution.

- Initial states are the same, with no message in transit.
- Point-to-point to convergecast: Assume $s_{pp} \sim s_{cc}$ and $m \in M, p \in P$. Trivially $s_{pp} \cdot s(p, m) \sim s_{cc} \cdot s(p, m)$ and $s_{pp} \cdot r(p, m) \sim s_{cc} \cdot r(p, m)$.
- Convergecast to point-to-point: assume $s_{cc} \sim s_{pp}$. Trivially $\forall m \in M, p \in P : s_{cc} \cdot s(p, m) \sim s_{pp} \cdot s(p, m)$ and $s_{cc} \cdot r(p, m) \sim s_{pp} \cdot r(p, m)$. For the multireception of a set of messages, we show it is the same as a sequence of receptions of these messages (see Fig. 19). Observe that all the seven ordering models are defined using a total or partial order on messages. Thus, in any finite set S of messages, if the transition $mr(\_, S)$ is enabled, then there is at least one receivable message by itself.[2] Let m be this message. Then the states $s_{cc} \cdot mr(p, S)$ and $s_{cc} \cdot r(p, m) \cdot mr(p, S \setminus \{m\})$ are the same. Inductively, this means there is (at least) one sequence $m_1, \ldots, m_n$ such that $s_{cc} \cdot mr(p, S)$ and $s_{cc} \cdot r(p, m_1) \cdots r(p, m_n)$ are identical. As $s_{cc} \sim s_{pp}$, then $s_{cc} \cdot mr(p, S) \sim s_{pp} \cdot r(p, m_1) \cdots r(p, m_n)$.

To conclude, as there is a bisimulation between point-to-point executions and convergecast executions, the inclusions of Theorem 1 are preserved. □

6. Related work

6.1. Communication models

Tel's textbook [32] describes a distributed system as a "collection of processes and a communication subsystem". Each process is a transition system, and the transition system induced under asynchronous communication is built with the product of the process transition systems extended with a collection of messages in transit, and two rules for send and receive.

[2] This is not as trivial as it seems: consider *FIFO n–n* and the execution $s(p_1, m) \cdot s(p_2, m')$. In the last state, the multireception $mr(p_3, \{m, m'\})$ is enabled, as well as the reception $r(p_3, m)$, but $r(p_3, m')$ is impossible while m is in transit.

His formal definition considers synchronous and fully asynchronous (unordered) point-to-point communication whereas we explicitly describe the communication subsystem with a conjunction of transition systems, consider several communication properties, including multicast and message-ordering policies, compare them, and offer a mechanized framework for checking compositions of peers.

Micro-protocols have been used in Horus [30] and Ensemble [29]. The developer arranges a stack of micro-protocols to obtain precisely the desired properties. Each micro-protocol layer handles some small aspect of these properties. For instance, one layer might deal with message loss, one with encryption, one with group membership, and another one with multicast ordering. One notable point of Ensemble was the use of NuPrl for provably rewriting the stack and generating optimized implementations [25], and of I/O automata for formalizing, specifying, and verifying the Ensemble implementation [18]. The main differences with our work is the hierarchical structure of the stack, and that the objectives of Horus and Ensemble was to provide efficient implementations of a group-communication infrastructure and was not concerned with the verification of the applications themselves.

Regarding ordering of message reception, generic ordering, such as FIFO or causal delivery, has been studied in the context of distributed algorithms. Asynchronous communication models in distributed systems have been studied in [21] (notion of ordering paradigm), [8] (notion of distributed computation classes), [16] (hierarchy of communication models for message sequence charts in a point-to-point setting), or [11] (formal description and hierarchy of several asynchronous point-to-point models).

Message Sequence Charts are convenient diagrams that allow to describe the desired interactions between components that exchange messages in a system. Communication is asynchronous but the Message Sequence Charts do not assume any particular communication model. There are two common ways to formalize the semantics of MSC: with a process algebra [28]; with partially ordered sets of events [2]. The second formalisation considers the local ordering of events on each peer and the possible ordering of couples of send and receive events. This corresponds to our generic message-ordering micromodels (Section 4.3.1), and this also allows [16] to build a hierarchy of communication models.

Priorities, in the context of verification, are usually introduced on specific actions. For instance [13] presents a priority operator *prisum* for CCS, allowing for the expression of preference between two actions. The behavior of the ALT construct in the Occam programming language [20] lets its users give a list of channels to receive from, establishing a priority relation between them according to their location in the list. In [3], priorities are introduced in a process calculus for trust. In a choice $a.A \bar{+} b.B$, one of the alternatives is preferred, according to the trust given to the other interacting process. Each peer has a trust table, and a $\bar{+}$ is given a minimal threshold for the first alternative to be allowed (the labels have no priority associated). As we have done here, [4] assign priorities to labels to provide an interrupt mechanism in process algebra. Its semantics is defined by rules such as $a + b = a$ if $a > b$. A thorough exploration of priority in process algebras with synchronous communication is done in [10]. Priorities are associated to labels, and are used only in a synchronous communication event.

In some architectural description languages (ADL), it is possible to explicitly describe the communication between the components. For example, Wright [1] is an ADL whose goal is to describe the interaction between components. Those interactions are described thanks to connectors whose behavior is described with CSP. It is then possible to describe different types of connectors like pipes, broadcast or shared variables. C2 [33] is specialized in message based-architecture with asynchronous communication like in this paper, but it is designed for describing GUI architectures. Since it is GUI architecture oriented, there is a hierarchy of components and the communication must respect this hierarchy i.e. a component cannot send a message to all the other ones. The communication is done through connectors whose goal is to route and broadcast messages. Several policies for filtering messages are provided: no filtering (broadcast to all the linked peers), notification filtering (equivalent to our notion of interest), prioritized (in C2, the priority is between the receivers, whereas our priorities are on messages) and message sink (ignore all messages). Acme [17] is an ADL whose goal is to unify the various existing ADLs. It is based on an ontology that describes seven type of elements. Communication is administrated through connectors, ports and roles. Connectors represent interactions among components, such as pipes, procedure calls, or event broadcasts. The multiplicity of the communication is given through roles. For example, for point-to-point communication a connector has two roles: caller / callee (RPC connector), reading / writing (pipe), sender / receiver (message passing). For broadcast, the connector has a unique role as an event announcer but an arbitrary number of event receiver roles. Note that as the ADL name says, its main concern is to describe an architecture, while we are interested in operational models in order to propose a framework for verification.

6.2. Verification of distributed systems

Formal verification of distributed algorithms has 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, [26] 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.

Promela (Process Meta Language) [19] is used to specify state transition systems that may describe distributed systems and asynchronous interactions. The associated model checker, SPIN, performs efficiently on these specifications. However, Promela only provides FIFO message channels to model the communication whereas our work requires an approach that encompasses the variety of asynchronous communication properties and the deriving communication models.

Input/output automata [27] provide a generic way to describe components that interact with each other with input and output actions. Components can describe processes as well as communication channels, and in that sense, I/O automata provide a flexible framework to describe distributed systems. However, few automatic tools have been developed to make use of I/O automata and perform modeling and property checking.

Compatibility of services or software components has largely been studied, especially with regard to collaborations and choreographies. Usually the interaction model is fixed and global for all the interactions. The majority of the approaches consider synchronous communication (e.g. [6,15]), even if a few works consider asynchronous communication with variations of FIFO ordering (e.g. [7,5]). To the best of our knowledge, no work has considered multicast communication or composed communication models.

Given the implementation of a distributed system, it is easier to prove it correct under a simple reliable environment than under a more realistic fault model. Wicox et al. propose Verdi [34], a framework that makes it possible to transform an implementation and proofs of safety in Coq established under an ideal environment into an implementation that is fault-tolerant under a more hostile environment. "Verified System Transformers" may enrish the system in a modular way with fault models for crashes, message loss, duplication or reordering. A question remains whether, with this approach, it would be possible to verify a system with a strict communication model (e.g. *FIFO n–n*, low message cap…) and then transform the system and its proof for a more liberal model (e.g. *FIFO 1–1*, no message cap…).

PSync [14] is a domain specific language based on the Heard-Of m odel [9], which imposes that the algorithms are structured in rounds. Distributed algorithms are described in a partially synchronous context, and then translated in Scala to run on their runtime environment. The Heard-Of machine is semi-automatically verified by stating inductive invariants which are checked by SMT provers.

7. Conclusion

This paper proposes an approach to the verification of asynchronous distributed systems that considers the influence of each individual property of the communication medium on the peers of the system. The first contribution is a verification framework in TLA$^+$ that offers to build communication models by combining individual communication properties we call micromodels. It benefits from the T LA$^+$ tools, and especially the model checker TLC. This makes it possible to cover a wide range of possible asynchronous communication variants while existing verification tools usually stick to a few particular cases and seldom offer much control over the features of the communication medium. Each specification of a micromodel follows a simple yet generic template allowing for easy expansion. Among them, we distinguish physical micromodels that specify when a message is removed from the whole communication model, and non-physical micromodels that provide additional properties among message ordering, cap on the number of messages in transit, or applicative priorities. The second contribution is a physical micromodel that encompasses both point-to-point and multicast communication thanks to a notion called the interest. The interest is an indirection on the usual notion of destination group in multicast communication: a message is proposed for delivery as long as some peers are or may later be interested in receiving it. By tweaking this rule with two parameters MIN and MAX that respectively prevent or force the removal of a message from the communication model depending on the current number of deliveries, we describe the whole spectrum of multicast communication spanning from point-to-point to one-to-all communication. This is complemented by a physical micromodel for convergecast communication, which acts as a dual of multicast communication. The last contribution is a comparison of several message-ordering models in multicast and convergecast settings as well as a study of totally-ordered multicast.

Ongoing work aims at specifying a micromodel for totally-ordered multicast communication. This ordering policy happens not to integrate as easily as other classic ordering policies with the notion of interest. Knowledge on the future behavior of the peers is necessary to check whether a delivery violates the ordering. Further thinking is thus required. In the meantime, this paper identifies the weakest classic ordering policy that provides totally-ordered multicast in a ready-to-use micromodel. Considering fault models among message loss, duplication, or crash of a peer, is another perspective. More generally, we do not specify the behavior of classic ordering policies after a loss or duplication of a message, or any other failure. The two challenges involve adapting existing micromodels and studying how the very notion of fault models can be integrated in the framework: it may require small adaptations or deeper refactoring.

References

[1] Robert Allen, David Garlan, Formalizing architectural connection, in: 16th International Conference on Software Engineering, ICSE '94, IEEE Computer Society Press, 1994, pp. 71–80.

[2] Rajeev Alur, Gerard J. Holzmann, Doron Peled, An analyzer for message sequence charts, in: International Workshop on Tools and Algorithms for the Construction and Analysis of Systems, in: LNCS, vol. 1055, Springer, 1996, pp. 35–48.

[3] Alessandro Aldini, Modeling and verification of trust and reputation systems, Secur. Commun. Netw. 8 (16) (2015) 2933–2946.

[4] Jos C.M. Baeten, Jan A. Bergstra, Jan Willem Klop, Syntax and defining equations for an interrupt mechanism in process algebra, Fundam. Inform. IX (1986) 127–168.

[5] Samik Basu, Tevfik Bultan, Meriem Ouederni, Deciding choreography realizability, in: 39th Symposium on Principles of Programming Languages, POPL '12, ACM, 2012, pp. 191–202.
[6] Antonio Brogi, Carlos Canal, Ernesto Pimentel, Antonio Vallecillo, Formalizing web service choreographies, Electron. Notes Theor. Comput. Sci. 105 (December 2004) 73–94.
[7] Daniel Brand, Pitro Zafiropulo, On communicating finite-state machines, J. ACM 30 (2) (April 1983) 323–342.
[8] Bernadette Charron-Bost, Friedemann Mattern, Gerard Tel, Synchronous, asynchronous, and causally ordered communication, Distrib. Comput. 9 (4) (February 1996) 173–191.
[9] Bernadette Charron-Bost, André Schiper, The heard-of model: computing in distributed systems with benign faults, Distrib. Comput. 22 (1) (April 2009) 49–71.
[10] Rance Cleaveland, Matthew Hennessy, Priorities in process algebras, Inf. Comput. 87 (1/2) (1990) 58–77.
[11] Florent Chevrou, Aurélie Hurault, Philippe Quéinnec, On the diversity of asynchronous communication, Form. Asp. Comput. 28 (5) (September 2016) 847–879.
[12] Florent Chevrou, Aurélie Hurault, Philippe Quéinnec, TLA$^+$ modules for a modular framework for verifying versatile distributed systems, http://vacs.enseeiht.fr/vacs2/, 2019.
[13] Juanito Camilleri, Glynn Winskel, CCS with priority choice, Inf. Comput. 116 (1) (1995) 26–37.
[14] Cezara Drăgoi, Thomas A. Henzinger, Damien Zufferey Psync, A partially synchronous language for fault-tolerant distributed algorithms, in: 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '16, ACM, 2016, pp. 400–415.
[15] Francisco Durán, Meriem Ouederni, Gwen Salaün, A generic framework for n-protocol compatibility checking, Sci. Comput. Program. 77 (7–8) (July 2012) 870–886.
[16] André Engels, Sjouke Mauw, Michel A. Reniers, A hierarchy of communication models for message sequence charts, Sci. Comput. Program. 44 (3) (2002) 253–292.
[17] David Garlan, Robert Monroe, David Wile Acme, An architecture description interchange language, in: CASCON'97, 1997, pp. 169–183.
[18] Jason J. Hickey, Nancy Lynch, Robbert van Renesse, Specifications and proofs for Ensemble layers, in: Fifth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99), in: LNCS, vol. 1579, Springer-Verlag, 1999, pp. 119–133.
[19] Gerard J. Holzmann, The Spin Model Checker: Primer and Reference Manual, Addison-Wesley, 2004.
[20] M. Elizabeth C. Hull, Occam - a programming language for multiprocessor systems, Comput. Lang. 12 (1) (1987) 27–37.
[21] Ajay D. Kshemkalyani, Mukesh Singhal, Distributed Computing: Principles, Algorithms, and Systems, Cambridge University Press, March 2011.
[22] Leslie Lamport, Time, clocks and the ordering of events in a distributed system, Commun. ACM 21 (7) (July 1978) 558–565.
[23] Leslie Lamport, Specifying Systems, Addison Wesley, 2002.
[24] Leslie Lamport, The PlusCal algorithm language, in: Theoretical Aspects of Computing - ICTAC 2009, 6th International Colloquium, in: Lecture Notes in Computer Science, vol. 5684, Springer, 2009, pp. 36–60.
[25] Xiaoming Liu, Christoph Kreitz, Robbert van Renesse, Jason J. Hickey, Mark Hayden, Kenneth Birman, Robert Constable, Building reliable, high-performance communication systems from components, in: 17th ACM Symposium on Operating Systems Principles (SOSP'99), in: Operating Systems Review, vol. 33(5), ACM Press, December 1999, pp. 80–92.
[26] Xiaozhou Li, Jayadev Misra, C. Greg Plaxton, Active and concurrent topology maintenance, in: Distributed Computing, 18th International Conference, in: Lecture Notes in Computer Science, vol. 3274, Springer, 2004, pp. 320–334.
[27] Nancy A. Lynch, Distributed Algorithms, Morgan Kaufmann Publishers Inc., 1996.
[28] Sjouke Mauw, Michel A. Reniers, An algebraic semantics of basic message sequence charts, Comput. J. 37 (4) (1994) 269–277.
[29] Robbert van Renesse, Kenneth P. Birman, Mark Hayden, Alexey Vaysburd, David Karr, Building adaptive systems using Ensemble, Softw. Pract. Exp. 28 (9) (August 1998) 963–979.
[30] Robbert van Renesse, Kenneth P. Birman, Silvano Maffeis Horus, A flexible group communications system, Commun. ACM 39 (4) (April 1996) 76–83.
[31] A. Segall, Distributed network protocols, IEEE Trans. Inf. Theory 29 (1) (1983) 23–35.
[32] Gerard Tel, Introduction to Distributed Algorithms, second edition, Cambridge University Press, 2000.
[33] R.N. Taylor, N. Medvidovic, K.M. Anderson, E.J. Whitehead, J.E. Robbins, K.A. Nies, P. Oreizy, D.L. Dubrow, A component- and message-based architectural style for GUI software, IEEE Trans. Softw. Eng. 22 (6) (June 1996) 390–406.
[34] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, Thomas Anderson Verdi, A framework for implementing and formally verifying distributed system, in: 36th ACM Conference on Programming Language Design and Implementation, June 2015, pp. 357–368.