Asynchronous Communication with Channel Priorities

Nathanaël Sensfelder[1], Aurélie Hurault[1], and Philippe Quéinnec[1]

1 **IRIT - Université de Toulouse, 2 rue Camichel, F-31000 Toulouse, France**
`http://www.irit.fr`

Abstract

In distributed systems, the order in which the messages are received by the processes is crucial to ensure the expected behavior. This paper presents a communication model which allows for restrictions on the deliveries of a channel depending on the availability of messages in other channels. This corresponds to prioritizing some channels over others. It relies on a framework able to verify if a given system satisfies a user defined LTL property with different configurations of the communication model. We also propose an algorithm to automatically infer the channel priorities so that the system does not infringe on this temporal property. Examples show that application priorities can ensure correct executions of a system where classic generic ordering mechanisms, such as FIFO or causal delivery, are insufficient.

1998 ACM Subject Classification D.2.4 Software/Program Verification

Keywords and phrases asynchronous communication; constraints inference; channel priorities; formal verification;

1 Introduction

In a purely asynchronous environment, the delivery of process messages is not assumed to be done in any order, as a reception's sole requirement is for the message to be in transit. The nondeterministic nature of those deliveries is bound to cause complications, notably with software testing, making the possibility of a formal verification of the system's correctness highly desirable. Many communication models imposing additional constraints on the availability of a message have been established, such as First-In-First-Out and causal. We can partition those communication models into two categories: the generic models, which are defined independently of the system that uses them; and the applicative models, whose constraints refer to the system's components (such as, in our case, its channels). By essence, the latter requires the communication model to be updated whenever the definition of the system is changed. In effect, this makes it an impractical solution for non-trivial systems if the updates are not automatically engineered. This difficulty is solved by the inferring capabilities of the approach presented in this paper. The definition of a correct system is very much dependent on the problem it is meant to solve, this is why our framework lets its users describe the target property using LTL (Linear Temporal Logic).

Consider a distributed system composed of peer exchanging messages over predefined channels. Following the Calculus of Communicating Systems (CCS) notation, the sending of a message on a channel is indicated by a '!', the reception by a '?', and a peer internal action by 'τ'. Using a purely asynchronous communication model in the system described in Figure 1 fails to prevent the last peer from going into a forbidden state (noted $\bot$), for instance via the sequence of events: $a! \cdot b! \cdot b? \cdot c! \cdot c?$. Note that the same issue occurs using point-to-point communication with FIFO ordering, whereas using a causal communication model successfully removes the problematic executions. However, using that causal communication model comes


Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

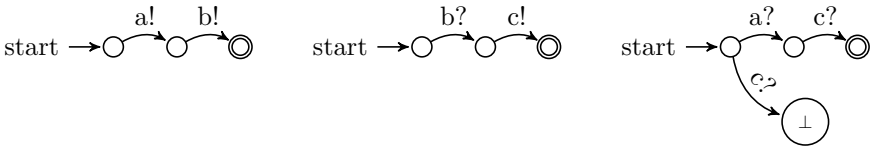

Figure 1 Three Peers Interacting With Three Channels a, b, c. $a!$ is a send event, $a?$ a receive event, communication is asynchronous.

at a cost, making the use of a more specialized (i.e. applicative) solution preferable. The issue comes from a process receiving from channel c instead of channel a, despite both being available. Clearly, in this case, channel a should be preferred over c. This is exactly what channel priorities are intended to describe.

Another use of the channel priorities can be found when trying to reduce the nondeterminism of a system, even when all possible executions are valid, should certain executions be preferable to other. A classic example of this 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.

The outline of this paper is the following. Section 2 presents the framework for the verification of asynchronous communication and precisely defines what priorities mean. Section 3 describes two inference algorithms which use the framework to discover the necessary priorities in order to ensure the correct behavior of a system. Section 4 illustrates our approach with several examples. Section 5 gives an overview of other approaches for ordering communication interactions and Section 6 provides perspectives and final remarks.

2 A Framework for the Verification of Asynchronous Communication with Priorities

Our objective is to tell if a system, composed of peers and of a communication model, verifies a correctness property given by the designer. The peers asynchronously interact through channels, and the communication model decides the delivery of messages (e.g. in what order the messages are available). Following [11], we have built a framework and a ready-to-use automated toolchain, based on TLA$^+$ [22] and its tools, that enables to check a LTL property on a system. As the communication models are TLA$^+$ modules which are composed with the peers, the framework allows to easily verify how a set of peers interacts with several models, or if the specified parameters are sufficient to validate the expected property.

2.1 Modeling the System

A system is composed of a set of peers and a communication model which controls the interactions between the peers. Additionally, we are given a property to verify on this system.

In our framework, peers are described using labeled transition systems. As we are only concerned with the interactions between the peers, this description only takes into account the network related actions of the peer, having a single label for all internal actions (τ). The peers communicate through channels and the communication actions are sending a message to a channel or receiving a message from a channel. Channels do not have explicit sender and receiver and are not limited to one sender/one receiver. They are nevertheless a point-to-point communication abstraction: a given message has exactly one sender and is received only once. For user-friendliness purposes, the peers can be described using terms of

the Calculus of Communicating Systems language, as long as their corresponding labeled transition systems (ignoring the synchronous communication rule of CCS) are finite. Our toolchain handles such conversions, making use of the Edinburgh Concurrency Workbench.

A communication model enforces rules on the communication and controls the message deliveries. In an asynchronous view of distributed systems, the communication model pushes messages up to the peers. The peers are limited to specifying which channels they will attempt to receive from, and they cannot choose their delivery order. For instance, the communication model can enforce an order of message receptions in relation to the order of their emissions, e.g. a FIFO communication model ensures that two consecutive messages sent from one peer to another peer are delivered in their send order. The communication model also controls sending. For instance, a bounded communication model disallows sending when its buffers are full. In this paper, channels may be preferred over other ones and the communication model delivers messages based on these priorities.

The expected property to check the system for is an LTL formula, including safety and liveness property in TLA$^+$ notation. Examples of useful properties are termination (all peers end in their terminal states), no unexpected reception (a peer receives a message in a state where it cannot deal with it), no deadlock (waiting forever for a message), emptiness of the network at termination (no pending messages)...

2.2 TLA$^+$ Specifications

The framework includes a toolchain that converts the system's model into TLA$^+$ modules. A communication model is specified by a module. It defines three predicates: *send(peer, channel)*, which asserts that the communication model has added a new message in the corresponding channel; *receive(peer, channel)*, indicating that the peer receives a message from the specified channel; and *tau(peer)*, that states no change is made to the network.

The peers are represented by a program counter tuple, keeping track of their current state. All of their transitions are converted to TLA$^+$ predicates, requiring the program counter tuple to be appropriately updated and the communication model predicate to be validated. In effect, the resulting system is a synchronized product of the transitions of the peers and the transitions of the communication models, each of its transitions affecting both the network and one peer. The executions are thus an interleaving of communication actions and of internal actions.

As stated previously, the control of the message deliveries' order is done by the communication model. Hence, whenever a peer has a reception transition, it must also have one for each of the channels it may receive from in the future. To avoid burden on the designer, those transitions are automatically added during the conversion process from the peer description (be it as a labeled transition system or a CCS term) in TLA$^+$.

The framework is accessible online at `http://vacs.enseeiht.fr/`. It can be used both to check if a system is correct (for a selection of predefined LTL properties), and to discover the necessary priorities to make it correct (if possible). The verification is done using TLC, the TLA$^+$ model checker, and is limited to finite (and small enough) systems.

2.3 A Communication Model with Channel Priorities

We present a communication model with channel priorities. It is parameterized by a set of BLOCKS constraints. Constraint (A BLOCKS b) means: if any of the channels in the set A contains a message, reception on channel b is disabled. Note that the BLOCKS constraints do not necessarily form a partial order on channels.

The internal predicate *is_blocked* specifies if reception on a channel is currently disabled. Additionally, a function *uid* is assumed to exist, and is used to obtain a new unique identifier to distinguish messages on a channel. A message in transit is then defined by a couple $\langle id, channel\rangle$. $Network$ is a variable which holds the messages in transit. In a TLA$^+$ action formula, x denotes the value of a variable x in the origin state, and x' denotes its value in the destination state.

$send(peer, channel) \triangleq Network' = Network \cup \{\langle uid(), channel\rangle\}$
$is\_blocked(channel) \triangleq$
$\quad \exists (B, c) \in \text{BLOCKS} : (channel = c) \wedge (\exists \langle id, chan\rangle \in Network : chan \in B)$
$receive(peer, channel) \triangleq$
$\quad \neg is\_blocked(channel)$
$\quad \wedge \exists \langle id, chan\rangle \in Network : (chan = channel) \wedge (Network' = Network \setminus \{\langle id, chan\rangle\})$
$tau(peer) \triangleq \text{UNCHANGED } Network$

2.4 Formalization

This section presents the formal definitions upon which the framework is built, and gives the precise definition of the BLOCKS constraint. Classically, the system resulting of the composition of peers and a communication model is defined as a labelled transition system, and an execution as a sequence of events (plus label). *Net* is an abstraction of the messages in transit, *disabled* captures the relevant parts of the peers. The order of delivery by the communication model is based on the channel priorities defined by BLOCKS, and peer identification is not involved.

▶ **Definition 1** (Composed System). A system is a quintuplet $(States, Init, Labels, Relation, Channels)$ where

- $Labels \subseteq (Channels \times \{\text{“?”}, \text{“!”}\}) \cup \{\tau\}$. $c!$ is interpreted as the sending of a message on channel c, and $c?$ as the reception of a message from channel c, and τ is an internal action;
- $Relation \subseteq (States \times Labels \times States)$ is the transition relation.
- $Init \subseteq States$ are the initial states.

▶ **Definition 2** (Execution). $\mathcal{EXEC}$, the set of all possible executions, is the set of all finite or infinite sequences of couples (state,label), conforming to the transition relation and such that a reception is preceded by a send:

$$\mathcal{SEQ} \triangleq (States \times Labels)^* \cup (States \times Labels)^\omega$$

$$\mathcal{EXEC} \triangleq \left\{ \sigma \in \mathcal{SEQ} \;\middle|\; \begin{array}{l} \quad \sigma_0 = (s_0, \_) \text{ with } s_0 \in Init \\ \wedge\ \forall i \in dom(\sigma) \setminus \{0\} : (s_{i-1}, l_i, s_i) \in Relation \\ \qquad\qquad \text{where } (s_i, l_i) = \sigma_i \text{ and } (s_{i-1}, l_{i-1}) = \sigma_{i-1} \\ \wedge\ \forall i \in dom(\sigma) : \forall c \in Channels : \\ \quad |\{j \in dom(\sigma) \wedge j \leq i : l_j = c!\}| \geq |\{j \in dom(\sigma) \wedge j \leq i : l_j = c?\}| \end{array} \right\}$$

▶ **Definition 3** (Network). At any point of an execution, *Net* is the set of channels where at least one message is in transit (a message has been sent and not received):

$$\forall \sigma \in \mathcal{EXEC} : \forall i \in dom(\sigma) :$$
$$\quad Net_i = \{c \in Channels : |\{j \in dom(\sigma) \wedge j \leq i : l_j = c!\}| > |\{j \in dom(\sigma) \wedge j \leq i : l_j = c?\}|\}$$

▶ **Definition 4** (Disabled reception). Reception on a channel c is *disabled* at a given point of an execution if it does not occur at that point:

$$\forall \sigma \in \mathcal{EXEC} : \forall i \in dom(\sigma) : disabled_i(c?) \triangleq i + 1 \notin dom(\sigma) \vee l \neq c? \text{ with } \sigma_{i+1} = (s, l)$$

Channel priorities are a set of static constraints that forces specific transitions to be *disabled*, depending on the values of *Net* and the *Channels* those transitions relate to. With regard to BLOCKS constraints, only reception transitions can be disabled. (B BLOCKS c) means that whenever at least one message is present on any channel of B, then the reception on c is *disabled*.

▶ **Definition 5** (BLOCKS)**.** A system Sys parameterized by the C BLOCKS constraints respects:

$$Sys, C \models \forall B \subseteq Channels, \forall c \in Channels,$$
$$(B \text{ BLOCKS } c) \in C \implies (\forall b \in B, \Box((b \in Net) \implies disabled(c?)))$$

3 Inferring Channel Priorities

When inferring the channel priorities, the objective is, given a system Sys and a property P, to find all the BLOCKS constraint sets C such that $Sys, C \models P$. To do so, we introduce analyzers which ask the framework whether a set of constraints applied to Sys satisfies P. The analyzers infer new constraints using the counter-example given by the framework when the property is not verified with the proposed constraint set. This yields to the generation of new sets of constraints, called solution-candidates, built by adding new constraints to the current candidates. The analyzers are thus able to find multiple satisfactory values for C.

3.1 Introducing New Constraint Types

Our framework lets us exhibit the states in which channel priorities are capable of influencing which transitions are *disabled*. We therefore limit our decisions to the reachable states where more than one receptions are possible. Those states are exposed in an order that is identical to the execution they originate from. Because of this, we can progressively build up our candidates for C. To do so, we have to make sure that adding a constraint to a solution-candidate does not prevent us from reaching the state that justified said constraint. This problem is solved by introducing a new type of constraint during the inference process, the ALLOWED constraint.

(a ALLOWED b) lets the analyzer remember that, at some point before reaching the state it is currently working on, channel a was available to block a reception on channel b, yet that reception did occur, meaning that a *allowed* the reception on b. Adding, at any point during the inference process, a new constraint implying that (a BLOCKS b) would potentially prevent the current state from being reachable and is therefore forbidden. We enforce:

For any set of constraints C,
$$\forall B \subseteq Channels, \forall c \in Channels, (B \text{ BLOCKS } c) \in C \implies \forall b \in B, (b \text{ ALLOWED } c) \notin C$$

On the other hand, keeping this reachability is made difficult when using BLOCKS constraints. As an example, suppose the analyzer needs to block reception on channel a and has at its disposal channels $\{b, c, d\}$. Adding ($\{b, c, d\}$ BLOCKS a) would work, but is overly restrictive. Indeed, if we later require (b ALLOWED a), we will be stuck due to our previous choice. Actually, any non-empty subset of $\{b, c, d\}$ achieves the same local effect of blocking a, but each add distinct restrictions on the future choices.

There are two ways to deal with this: fork the inferring process to explore each subset of $\{b, c, d\}$, which is extremely inefficient, or create another type of constraint. Obviously, the latter is preferable. This new constraint structure, named BLOCKED, is semantically

close to BLOCKS, but allows for elements of the blocking set to be removed. We just want to remember that $\{b, c, d\}$ *blocked* a but not precisely which one(s) did.

$$Sys, C \models \forall B \subseteq Channels, \forall c \in Channels,$$
$$(B \text{ BLOCKED } c) \in C \implies \exists b \in B, \Box(b \in Net \implies disabled(c?))$$

Observe that the definition of BLOCKED is similar to the one of BLOCKS, with the exception of a $\forall$ being replaced by an $\exists$. For the same reason as with BLOCKS constraints, we enforce:

For any set of constraints C,
$$\forall B \subseteq Channels, \forall c \in Channels, (B \text{ BLOCKS } c) \in C \implies \exists b \in B, (b \text{ ALLOWED } c) \notin C$$

Going back to our previous example, adding the constraint $(\{b, c, d\} \text{ BLOCKED } a)$ would keep a blocked at the relevant state, yet would make no commitment on which channel(s) blocked it. Therefore, we can later add that $(b \text{ ALLOWED } a)$ constraint if needed, as c and d are still capable of honoring the BLOCKED constraint.

3.2 Framework Alterations

Using BLOCKED and ALLOWED constraints instead of BLOCKS ones means that we need to alter our framework. As we don't actually know which element(s) of B in $(B \text{ BLOCKED } c)$ are blocking c, there are now three possibilities when a peer attempts reception on c. Either c is BLOCKED by at least one channel in Net, or c is ALLOWED by all of the channels in Net, or the situation is *ambiguous*. Whenever a peer of the system makes a reception, its *Ambiguity* set is calculated.

▶ **Definition 6** (Ambiguity). Considering that the *available_channels* of a state is the set of all channels that can be received from by the peer at that state, and that a channel is *active* if there is a message on it, the *Ambiguity* set is made of the elements of *available_channels* which are not ALLOWED by at least one of channels holding at least one message (i.e. Net).

$$available\_channels(peer) \triangleq \{channel \in Channels \mid \text{ENABLED}^1\ receive(peer, channel)\}$$
$$Ambiguity(peer) \triangleq$$
$$\left\{ chan \in Channels \;\middle|\; \begin{array}{c} chan \in available\_channels(peer) \\ \wedge\ \exists\ ch \in Net, (chan \neq ch) \wedge \neg(ch \text{ ALLOWED } chan) \end{array} \right\}$$

Because we are no longer using BLOCKS constraints, we have to employ another communication model, used only for the inference. It is identical to the channel priorities one, except for the function that decides whether a channel is blocked or not, which becomes:

$$is\_blocked(channel) \triangleq$$
$$\exists (B, c) \in blocked : (channel = c) \wedge (\forall b \in B, \exists \langle id, chan \rangle \in Network, (chan = b))$$

Thus, $(B \text{ BLOCKED } c)$ leads to c being blocked if, and only if, all the channels of B have a message in transit. Note that the ambiguous situations never lead to c being blocked.

Using *Ambiguity*, the framework constructs its report(s) to the analyzers, which contain:

- $Report_{net} \triangleq Net_i$.
- $Report_{invalid} \triangleq$ the elements of *Ambiguity* for which the framework knows that enabling a reception causes the expected property of the system to be invalidated (e.g. $\perp$ states).

[1] In TLA$^+$, ENABLED *Action* is true in a state if the action is possible, meaning there is a successive state reachable with *Action*.

- $Report_{valid} \triangleq Ambiguity \setminus Report_{invalid}$.

This report is returned to the analyzer whenever the framework finds an execution which invalidates the requested property. If no such execution is found, the analyzer is simply informed that the property is verified.

At the end of the inference process, the constraint sets are converted so they only use BLOCKS constraints. Given a constraint set using solely BLOCKED and ALLOWED constraints, the analyzer creates, for every channel c being BLOCKED by at least one set of channels, a single constraint (B BLOCKS c) such that B is the union of all sets that BLOCKED c, then removes from B all channels that ALLOWED c.

$$convert(solution) \triangleq \left\{ (B \text{ BLOCKS } c) \;\middle|\; \begin{array}{l} c \in Channels, \\ B = \left\{ a \in Channels \;\middle|\; \begin{array}{l} \exists (A \text{ BLOCKED } c) \in solution, \\ a \in A \wedge (a \text{ ALLOWED } c) \notin solution \end{array} \right\} \end{array} \right\}$$

3.3 The Analyzers

The analyzers handle sets of solution-candidates, each of which is a set of ALLOWED and BLOCKED constraints. They start with a single solution-candidate without any constraint. Information on the system is gathered by choosing a solution-candidate, setting it as the constraint set of the communication model, and then asking the framework to report. The framework either gives its report(s) of *ambiguous* states, as explained previously, or declares that the targeted property is validated. If *ambiguous* states are reported, the analyzers replace the solution-candidate by its children, each of which is generated using the following function where *chosen* is a subset of $Report_{valid}$:

$$\begin{array}{l} update(Candidate, Report, chosen) \triangleq \\ \quad Candidate \\ \cup \{((Report_{net} \setminus \{i\}) \text{ BLOCKED } i) \mid i \in Report_{invalid}\} \\ \cup \{((Report_{net} \setminus \{v\}) \text{ BLOCKED } v) \mid v \in Report_{valid} \setminus chosen\} \\ \cup \{(c \text{ ALLOWED } v) \mid v \in chosen \wedge c \in Report_{net}\} \end{array}$$

This function ensures that the undesirable channels from $Report_{invalid}$ are BLOCKED by the communication model. Every subset of $Report_{valid}$ generates its own child (including $\emptyset$). The ALLOWED constraints make sure that the elements of *chosen* are not BLOCKED. The channels of $Report_{valid}$ that are not *chosen* are also BLOCKED, so there is no longer any ambiguity. Obviously, inconsistent children (e.g. a child with both ({a} BLOCKED b) and (a ALLOWED b)) are discarded.

Whenever the framework reports that the expected property is verified, the analyzer adds the solution-candidate to its list of solutions. This process is continued until there are no more solution-candidates.

3.3.1 Pessimistic Priority Finder

The 'pessimistic' approach requests an additional property to be verified by the system: there must not be ambiguities. This causes the framework to signal an error and report whenever *Ambiguity* is not empty, even if the designer property is validated. The drawback of this approach shows when looking at the number of framework calls that such a thorough search implies. Indeed, even if the only required BLOCKS constraint is due to the last explored state of the system, the analyzer still has to make decisions and stops the framework at *all* the

```
function pessimistic_analyser(Sys,Property) → Solutions {
   candidates ← {∅};
   solutions  ← ∅;
   while (candidates ≠ ∅) {
     candidate ← pop(candidates);
     report ← framework_check(Sys,candidate,Property);
     if (report = accepted) {
        solutions ← {candidate} ∪ solutions;
     } else {
        foreach (powerset(report_valid) as S) {
           candidates ← {update(candidate, report, S)} ∪ candidates;
        }
     }
  }
  return solutions;
}
```

Figure 2 Pessimistic Priority Finder

ambiguous states. The analyzer is quite straightforward, since it only handles a single report per framework call, and is given Figure 2.

3.3.2 Optimistic Priority Finder

The 'optimistic priority finder' is a variation of its 'pessimistic' counterpart, sacrificing exhaustiveness for performance (Figure 3). Instead of being restarted at every ambiguous state, the framework continues its exploration, considering that channels in *Ambiguity* are not blocked, as expressed in the definition of *is_blocked*. Instead, it simply memorizes what the report would have been. Only when a counter-example is found does the framework give its reports to the analyzer, preserving the order in which they appeared. This means that if no counter-example is found, i.e. the system satisfies the requested property, the reports are not transmitted to the analyzer. This prevents the exploration of solution-candidates that would have needlessly block channels.

The optimistic priority finder handles multiple reports at once. Each of the ambiguous states in a counter-example has its own report. The first ambiguous state's report is handled like in the previous algorithm and generates new solution-candidates. All of those are added to the list of solution-candidates that have to be tested by the framework, with the exception of the solution-candidate coming from $chosen = report_{valid}$. Indeed, whenever $report_{invalid}$ is empty, this solution-candidate is the only one that is ensured to reach the next ambiguous state, should such a state exist. If there is no next report, this solution-candidate leads to the same invalid execution, it is thus discarded. However, if it does exist, the next report is treated as if it has been generated by the $chosen = report_{valid}$ solution-candidate. Whenever $report_{invalid}$ is not empty, meaning the next ambiguous state may not be reachable using this solution-candidate, the candidate is added to the solution-candidates to be tested by the framework. As the generation of the $chosen = report_{valid}$ solution-candidate is done using the *update* function, which adds the blocking of all the elements in $report_{invalid}$, testing this solution-candidate with the framework will not result in the same report being generated.

```
function optimistic_analyser(Sys,Property) → Solutions {
   candidates ← {∅};
   solutions  ← ∅;
   while (candidates ≠ ∅) {
      candidate ← pop(candidates);
      reports ← framework_check(Sys,candidate,Property);
      if (reports = accepted) {
         solutions ← {candidate} ∪ solutions;
      } else {
         candidates ← handle_report(candidate,0,report)∪candidates;
      }
   }
   return solutions;
}
function handle_report(candidate,i,report) → Candidates {
   candidates ← ∅;
   if (i ≠ |reports|) {
      foreach (powerset(report^i_valid) as chosen) {
         new_candidate ← update(candidate,report^i,chosen);
         if (chosen = report^i_valid) {
            if (report^i_invalid ≠ ∅) {
               candidates ← new_candidate ∪ candidates;
            } else {
               candidates ← handle_report(new_candidate,i+1,report)
                  ↪ ∪ candidates;
            }
         } else {
            candidates ← new_candidate ∪ candidates;
         }
      }
   }
   return candidates;
}
```

Figure 3 Optimistic Priority Finder

3.4 Heuristic

Consider the system in Figure 4. All peers end up in accepting states, even in a completely asynchronous environment. Using the pessimistic approach of the inference process, we find the following solutions: no constraints, ($\{a\}$ BLOCKS b), or ($\{b\}$ BLOCKS a). However, using the ($\{a\}$ BLOCKS b) constraint means that part of the system is cut off. The "no constraints" solution is very much likely to be preferred, which means that not all solutions are of equal interest. Note that there may not always be an optimal solution, as illustrated in Section 4.3.

Our proposed heuristic sorts the solutions according to a sum of the $Report_{valid}$ elements blocked ratio. This means that blocking a channel reception without being sure it needs to be (since it's not in $Report_{invalid}$) results in the solution-candidate lowering its score.

$$h(\emptyset) \triangleq 0$$

$$h(sc) \triangleq h(parent(sc)) + \begin{cases} 0 & \text{if } Report_{valid} = \emptyset \\ \frac{cardinal(Report_{valid} \setminus chosen)}{cardinal(Report_{valid})} & \text{otherwise} \end{cases}$$

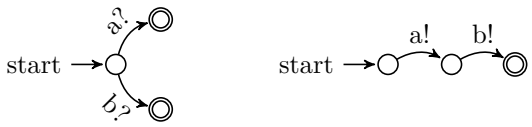

Figure 4 A System to Illustrate "Good" and "Bad" Solutions. Constraint ($\{a\}$ BLOCKS b) works but cuts off valid executions.

Figure 5 Comparison of the Optimistic/Pessimistic Approaches. This trivial system is immediately found valid by the optimistic approach, whereas the pessimistic one requires 5 runs of the framework to conclude.

4 Examples

This section presents several examples to show both the interest and the limits of priority channels. In the first cases, the necessary priorities are rather obvious and could be manually discovered. By contrast, the third and four examples justify the automatic inference of priorities to ensure that all solutions are found and offered to the designer, as no best solution can be identified. The reported number of **Distinct states** is the number of distinct states that were explored to verify that the system is valid with the proposed constraints. This number is not influenced by the additional variables from the inferring process.

4.1 Comparison of the Optimistic/Pessimistic Approaches

The example in Figure 5 shows the difference between the pessimistic and the optimistic approaches. The expected property is that all processes terminate in an accepting state. Instead of converting the solutions so they use BLOCKS constraints, we kept the inference process constraint types, as it helps to understand what is going on. The pessimistic priority finder requires 5 framework runs to find all the possible values of C. The first solution is found in 2 runs.

1. (b ALLOWED a), (a ALLOWED b)
2. (b ALLOWED a), ($\{a\}$ BLOCKED b)
3. (a ALLOWED b), ($\{b\}$ BLOCKED a)

As the system is correct without any BLOCKS constraints, the optimistic priority finder stops after a single framework call and answers that no constraints are necessary.

4.2 Additional LTL Constraints

In the example of Figure 6, in addition that all peers reach their terminating states, an additional expected property is that the network is eventually empty and stable ($\Diamond\Box(Network = \emptyset)$). Arbitrary designer's LTL formulae are allowed in the inference process. The inference finds the only constraint which works: ($\{a\}$ BLOCKS b). FIFO delivery would also ensure termination with an empty network. However, this forces the messages on a to

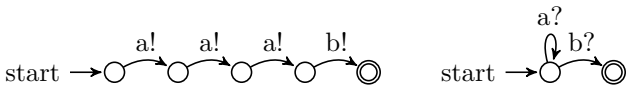

Figure 6 Arbitrary LTL Properties. No constraint is necessary to ensure termination of the two peers. However constraint ($\{a\}$ BLOCKS b) is required for the peers to terminate with an empty network.

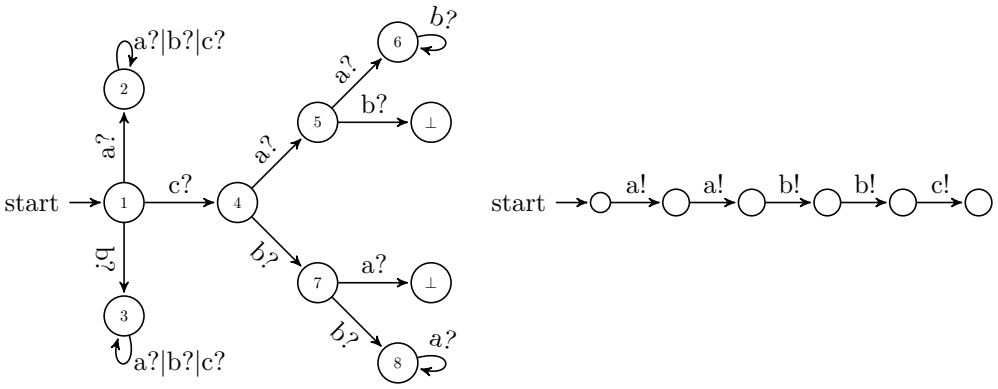

Figure 7 Incomparable Solutions and Unexpected Cuts. Four possible sets of constraints avoid a $\bot$ state ($\{(\{a\}$ BLOCKS $c)\}$, $\{(\{b\}$ BLOCKS $c)\}$, $\{(\{a\}$ BLOCKS $b)\}$, $\{(\{b\}$ BLOCKS $a)\}$), yet none can be said to be the best.

be received in their send order, whereas with our solution based on priority, the messages on a can be received in any order, allowing more non-deterministic executions.

4.3 Incomparable Solutions and Unexpected Cuts

In the example shown in Figure 7, the goal is to avoid the $\bot$ states. If the first received message is from a or b, no order is imposed on the next receptions; if the first received message is from c, then the system expects to consecutively receive either the two messages from a or the two messages from b. This example is solvable with four incomparable solutions, and only the designer knows which one is the best. Below are the minimal inferred constraints found by the pessimistic analyzer (more restrictive solutions exist, such as $(\{a, b\}$ BLOCKS $c)$ which contains $(\{a\}$ BLOCKS $c)$ and $(\{b\}$ BLOCKS $c)$, but they are filtered out) :

1. ($\{a\}$ BLOCKS c)
   Distinct states: 327
2. ($\{a\}$ BLOCKS b)
   Distinct states: 145
3. ($\{b\}$ BLOCKS c)
   Distinct states: 312
4. ($\{b\}$ BLOCKS a)
   Distinct states: 184

This system can be made correct by ensuring that messages are received in their send order (i.e. FIFO delivery). However, only states 1 and 2 would then be visited, and this is more restrictive than what allows some valid solutions with channel priorities. Nevertheless, all four solutions induce non-trivial cut offs (i.e. cut offs of states that are not $\bot$). For instance, the solution ($\{a\}$ BLOCKS b) is equivalent to cutting the south-east branch of state 4 (states 7 and 8) as well as state 3.

The solutions ($\{a\}$ BLOCKS b) and ($\{b\}$ BLOCKS a) do not have the same amount of distinct states, as the a messages are always sent before the b ones, meaning that ($\{b\}$ BLOCKS a)

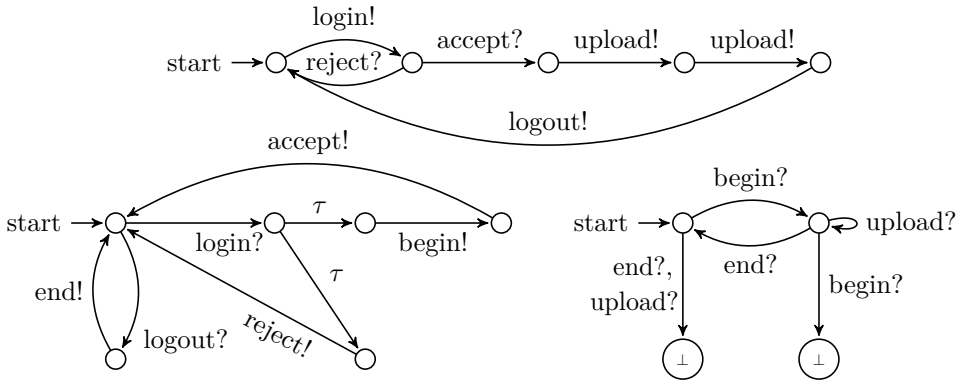

Figure 8 A Client-Server System. The Server is split into a controller and an application. The controller accepts or rejects the client, and begins/ends the application when needed.

cannot ensure that the system does not start by consuming the a messages (before the messages are sent on b), whereas ($\{a\}$ BLOCKS b) prevents b messages from being read first.

Interestingly enough, the solutions ($\{a\}$ BLOCKS c) and ($\{b\}$ BLOCKS c) have the highest number of distinct states. However, it means cutting off the c? branch of state 1, what appears to be more than half of the transition system of the first peer, and, for a designer, what is probably the interesting part of the system. This high number of states only comes from the allowed interleaving of the send and receive actions.

4.4 A Client-Controller-Application System

A system is composed of three peers: a client, a controller and an application (see Figure 8). The client interacts with the controller to get the authorization to access the application, then interacts with the application which has been started when needed by the controller. More precisely, the client sends a *login* to the controller which can *accept* of *reject* the demand. If accepted, the client can send several *upload* messages to the application. This controller starts the application (message *begin*) when it accepts a client, and signals it to *end* when the client *logout*s.

Several problems occur if the messages are randomly delivered. Among others, the application must consume message *begin* before processing the messages *upload* whereas they come from different peers; message *end* must not be consumed when there are pending *upload*s; when the client *logout*s then *login*s again, message *begin* must not overtake any of the messages from the previous round (notably *upload* or *end*)... To avoid deadlock and $\perp$ states, specifying and manually verifying a correct ordering of messages is not an easy task. Our framework automatizes the verification and the inference algorithms help in discovering some or all the solutions. Seven solutions exist, among others:

($\{begin\}$ BLOCKS $upload$)	($\{begin\}$ BLOCKS $accept$)	($\{begin\}$ BLOCKS $accept$)
($\{upload\}$ BLOCKS end)	($\{upload\}$ BLOCKS end)	($\{upload\}$ BLOCKS $logout$)
($\{logout, end\}$ BLOCKS $login$)	($\{logout\}$ BLOCKS $login$)	($\{logout, end\}$ BLOCKS $login$)
Distinct states: 298	($\{end\}$ BLOCKS $begin$)	**Distinct states:** 99
	Distinct states: 465	

Observe that the solutions present a large scattering in the number of states, meaning that some solutions allow more executions than others. Note also that the client is not waiting for the application to progress. The message priorities ensure that the application

does not lag behind too much.

5 Related Work

Generic ordering Generic ordered delivery, such as FIFO or causal delivery, has been studied in the context of distributed algorithms. Asynchronous communication models in distributed systems have been studied and compared in [21] (notion of ordering paradigm), [10] (notion of distributed computation classes), or [12] (formal description and hierarchy of several asynchronous models). Implementations of the basic communication models using histories or clocks are explained in classic textbooks [3, 16, 26, 21, 25], and the minimum required information to realize these orders has been studied [23, 20]. These works deal with generic orderings, solely based on the behavior of the system (e.g. relative order of the events) but they are not application-defined.

Applicative Priority on Specific Transitions In [9], a priority operator, "prisum", is added to CCS, allowing for the expression of preference between two actions. Its semantic is similar to the (extended) BLOCKS constraints presented in section 6, as an actor may only do an action when it can not do one that has a strictly higher priority. However, there are two major differences. First, the prisum relation is defined at state level, making it possible for preference between two actions to change over the course of the execution, something that cannot happen with BLOCKS constraints. On the other hand, expressing $(\{a\}$ BLOCKS $b), (\{b\}$ BLOCKS $a)$ using prisum does not appear to be possible.

The behavior of the "ALT" construct in the Occam programming language [18] lets its users give a list of channels to receive from, establishing a priority relation between them according to their location in the list (the first element having a higher priority). While this can easily be translated to BLOCKS constraints, with channel being blocked by all of those that surpass its priority in the ALT construct, we again face priorities that are established for (and at) a specific state.

In [1], 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.

π-calculus have been extended with attributed processes [19] which allow to express local and global interaction constraints, based on the values of the attributes. First, communication in π-calculus is extended with priority, where the dynamic priority is defined at send events ($x:r!z$ is a send on channel x, with priority r and parameter z), and only the highest priority synchronous communication are enabled. Then, it is extended to allow priorities to be λ-calculus expressions. Priorities are determined by the sender (which defines a function to compute the priority) and the receiver (which defines the arguments of the function).

Priorities have also been introduced in Petri Nets. In an initial work [8], priorities are statically associated to couples of transitions. These priorities are an arbitrarily relation and do not necessarily form a partial order. The interpretation of these priorities is simple with interleaving semantics: a transition is enabled at a marking only when no transition with higher priority is enabled at that marking. However, concurrent semantics is less clear and cannot be naturally defined with causal partial order. This paper analyses the different choices, among which it proposes rules to translate a system with priorities into another system without priorities. Dynamic priorities are introduced in [6]. Priorities are a relation between couples of transitions, for each possible marking of the net. The goal is to use

priorities to reduce the number of enabled transitions at a given marking, thus hoping to reduce the size of the reachability graph without affecting the truth of the property under consideration.

Applicative Priorities on Labels Another point of view is to assign priorities to labels or message identities, as we have done in this paper. It appeared in [4, 5] to provide an interrupt mechanism in process algebra. Priority are associated to labels and form a partial order. Semantics is defined by rules such that $a+b = a$ if $a > b$. However, this is not as easy as stated when labels can be masked and composability is lost if care is not taken. Cleaveland and Hennessy have done a thorough exploration of priority in process algebras with synchronous communication [14, 15]. They distinguish operational semantics and behavioral congruences (for compositional reasoning). Priorities are associated to labels, and are used only in a synchronous communication event.

Further extensions have followed. For instance, [24] presents a broadcasting calculus with priority. Priorities are associated to processes and to send/receive events. A process can only receive messages with a higher priority than its own. [17] introduces priority in a chemical calculus. Priorities are statically associated to actions (chemical reactions) which describe molecule transformations. It gives an operational semantics where an action can occur only if no other action of higher priority can be played in the current state.

Synchronous (zero-delay) calculus [7] studies preemption in concurrent synchronous real-time calculi. Its goal is to provide preemption to handle interrupt (trap), suspension and abortion. It distinguishes between "may preemption" (in a time independent calculus) and "must preemption" (in a synchronous or zero-delay calculus). It argues that time independence and may preemption are sufficient for distributed computations, but real-time or reactive systems require must preemption, where the preemption is guaranteed to occur in a bounded delay, or even instantaneously. It also defends that preemption primitives should be orthogonal to other primitives of the calculus, including concurrency and communication, in order to ease composition.

Controller synthesis One difference with all the previous work is that our work not only defines priorities on channels and offers a framework to check if a temporal property is verified by the system, but we also provide an inference process to find all the solutions (if any) which guarantee that a given propriety is satisfied. This is reminiscent of controller synthesis [13, 2]. We differ from this approach as we are not building a synchronizer based on the temporal property of interest, and several incomparable solutions are possible (see example 4.3).

6 Generalization and Conclusion

The logical continuation of reception being controlled by channel priorities is to extend those constraints to the other action types: *send*, *receive*, and *tau* all having priorities over one another, depending on the channel. This can be achieved by making the channel priority constraint types work with *Actions* instead of *Channels*, an *Action* being $c?$ (receive from channel c), $c!$ (send on c) or τ. Interestingly enough, this does not cause any change to the channel priority inferring analyzers and the framework can easily be altered to take those new constraints into account.

Figure 9 ({$a?$} BLOCKS τ) avoids getting stuck in infinite stuttering.

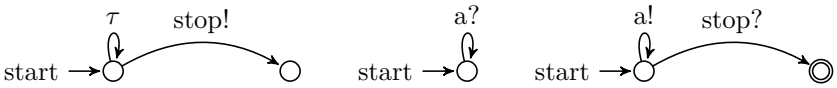

Figure 10 ({$stop?$} BLOCKS $a!$) ensures cancellation of emissions on a.

This extension allows to set a constraint ({$tick?$} BLOCKS $alarm!$), to make a peer report an alarm if and only if it is unable to receive an expected message. In the example of Figure 9, the constraint ({$a?$} BLOCKS τ) ensures that the peer does not get stuck in a τ loop if a reception on a is possible. This makes it comparable to a fairness constraint. In the example of Figure 10, ({$stop?$} BLOCKS $a!$) ensures that the third peer terminates as soon as the first peer sends *stop*. This allows a cancellation or abortion action to be easily described.

References

1 Alessandro Aldini. Modeling and verification of trust and reputation systems. *Security and Communication Networks*, 8(16):2933–2946, 2015. Available from: `http://dx.doi.org/10.1002/sec.1220`.

2 Paul C. Attie, Anish Arora, and E. Allen Emerson. Synthesis of fault-tolerant concurrent programs. *ACM Transactions on Programming Languages and Systems*, 26(1):125–185, 2004. Available from: `http://doi.acm.org/10.1145/963778.963782`.

3 Özalp Babaoğlu and Keith Marzullo. Consistent global states of distributed systems : Fundamental concepts and mechanisms. In Sape J. Mullender, editor, *Distributed Systems*, pages 55–96. ACM Press Frontier Series, second edition, 1993.

4 Jos C. M. Baeten, Jan A. Bergstra, and Jan Willem Klop. Syntax and defining equations for an interrupt mechanism in process algebra. *Fundamenta Informaticae*, IX:127–168, 1986.

5 Jos C. M. Baeten, Jan A. Bergstra, and Jan Willem Klop. Ready-trace semantics for concrete process algebra with the priority operator. *The Computer Journal*, 30(6):498–506, 1987. Available from: `http://dx.doi.org/10.1093/comjnl/30.6.498`.

6 Falko Bause. Analysis of petri nets with a dynamic priority method. In Pierre Azéma and Gianfranco Balbo, editors, *18th International Conference on Application and Theory of Petri Nets ICATPN'97*, volume 1248 of *Lecture Notes in Computer Science*, pages 215–234. Springer, 1997. Available from: `http://dx.doi.org/10.1007/3-540-63139-9_38`.

7 Gérard Berry. Preemption in concurrent systems. In R. K. Shyamasundar, editor, *13th Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 761 of *Lecture Notes in Computer Science*, pages 72–93. Springer, 1993. Available from: `http://dx.doi.org/10.1007/3-540-57529-4_44`.

8 Eike Best and Maciej Koutny. Petri net semantics of priority systems. *Theoretical Computer Science*, 96(1):175–174, 1992. Available from: `http://dx.doi.org/10.1016/0304-3975(92)90184-H`.

9 Juanito Camilleri and Glynn Winskel. CCS with priority choice. *Information and Computation*, 116(1):26–37, 1995. Available from: `http://dx.doi.org/10.1006/inco.1995.1003`.

10 Bernadette Charron-Bost, Friedemann Mattern, and Gerard Tel. Synchronous, asynchronous, and causally ordered communication. *Distributed Computing*, 9(4):173–191, February 1996. Available from: http://dx.doi.org/10.1007/s004460050018.

11 Florent Chevrou, Aurélie Hurault, and Philippe Quéinnec. Automated verification of asynchronous communicating systems with TLA+. *Electronic Communications of the EASST*, 72, 2015. Available from: http://journal.ub.tu-berlin.de/eceasst/article/view/1019.

12 Florent Chevrou, Aurélie Hurault, and Philippe Quéinnec. On the diversity of asynchronous communication. *Formal Aspects of Computing*, 2016. Available from: http://link.springer.com/article/10.1007/s00165-016-0379-x.

13 Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In Dexter Kozen, editor, *Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer, 1981. Available from: http://dx.doi.org/10.1007/BFb0025774.

14 Rance Cleaveland and Matthew Hennessy. Priorities in process algebras. *Information and Computation*, 87(1/2):58–77, 1990. Available from: http://dx.doi.org/10.1016/0890-5401(90)90059-Q.

15 Rance Cleaveland, Gerald Lüttgen, and V. Natarajan. Priority and abstraction in process algebra. *Information and Computation*, 205(9):1426–1458, 2007. Available from: http://dx.doi.org/10.1016/j.ic.2007.05.001.

16 George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems: concepts and design*. Addison Wesley, second edition, 1994.

17 Maxime Folschette, Loïc Paulevé, Morgan Magnin, and Olivier F. Roux. Under-approximation of reachability in multivalued asynchronous networks. *Electronic Notes in Theoretical Computer Science*, 299:33–51, 2013. Available from: http://dx.doi.org/10.1016/j.entcs.2013.11.004.

18 M. Elizabeth C. Hull. Occam - A programming language for multiprocessor systems. *Computer Languages*, 12(1):27–37, 1987. Available from: http://dx.doi.org/10.1016/0096-0551(87)90010-5.

19 Mathias John, Cédric Lhoussaine, Joachim Niehren, and Adelinde Uhrmacher. The attributed Pi calculus with priorities. *Transactions on Computational Systems Biology*, XII(5945):13–76, February 2010. Available from: https://hal.inria.fr/inria-00422969.

20 Ajay D. Kshemkalyani and Mukesh Singhal. Necessary and sufficient conditions on information for causal message ordering and their optimal implementation. *Distributed Computing*, 11(2):91–111, 1998.

21 Ajay D. Kshemkalyani and Mukesh Singhal. *Distributed Computing: Principles, Algorithms, and Systems*. Cambridge University Press, March 2011.

22 Leslie Lamport. *Specifying Systems*. Addison Wesley, 2003.

23 Larry L. Peterson, Nick C. Buchholz, and Richard D. Schlichting. Preserving and using context information in interprocess communication. *ACM Transactions on Computer Systems*, 7(3):217–246, 1989.

24 K. V. S. Prasad. A calculus of broadcasting systems. *Science of Computer Programming*, 25(2-3):285–327, 1995. Available from: http://dx.doi.org/10.1016/0167-6423(95)00017-8.

25 Michel Raynal. *Distributed Algorithms for Message-Passing Systems*. Springer, 2013.

26 Gerard Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, second edition, 2000.