Inference of Channel Priorities for Asynchronous Communication

Nathanael Sensfelder, Aurélie Hurault, Philippe Quéinnec

Inference of Channel Priorities for Asynchronous Communication

Nathanaël Sensfelder[1], Aurélie Hurault[1] and Philippe Quéinnec[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 (Linear Temporal Logic) property with different priorities. We also propose to automatically infer the channel priorities so that the system does not infringe on this temporal property.

1 Introduction

In an asynchronous environment, the delivery of messages is by essence nondeterministic. This 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 impose additional constraints on the delivery of a message, such as First-In-First-Out or causally ordered communication. Those communication models belong in two classes: 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). The latter require the communication model to be updated whenever the definition of the system is changed. This makes it impractical for non-trivial systems if the constraints are not automatically inferred.

Consider a distributed system composed of peer exchanging messages over predefined channels. Following the Calculus of Communicating Systems (CCS) syntax, 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 (unlike CCS) 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: $a! \cdot b! \cdot b? \cdot c! \cdot c?$. The same issue occurs using point-to-point communication with FIFO ordering (first in first out, i.e. a queue between each

[1] This work was partially supported by project PARDI ANR-16-CE25-0006. An extended version of the paper is available at http://vacs.enseeiht.fr/dcai17-long.pdf.

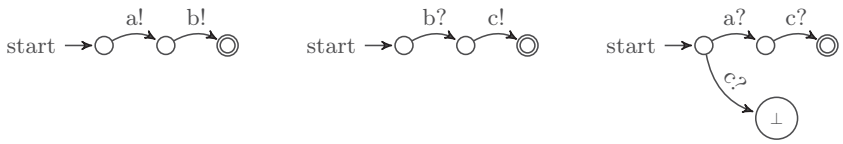

Fig. 1. Three Peers Interacting With Three Channels a, b, c. $a!$ is a send event, $a?$ a receive event, communication is asynchronous.

couple of peers), whereas using a causal communication model would avoid the problematic executions. However, using that causal communication model comes at a cost, making the use of an applicative solution worth considering. The issue comes from a process receiving from channel c instead of channel a, despite both being available. In this case, channel a should have a higher priority than c.

Another use of the channel priorities is 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 is 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 the inference algorithm which uses the framework to discover the necessary priorities in order to ensure the correct behavior of a system. Section 4 illustrates our approach with an example. Section 5 gives an overview of other approaches for ordering communication interactions and Section 6 provides perspectives and final remarks.

2 Verification of Asynchronous Communication with Priorities

2.1 The Framework

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 [6], we have built a framework and an automated toolchain, based on TLA$^+$ [13] and its tools, that enables to check an LTL (Linear Temporal Logic) 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.

In the present case, the communication model with channel priorities is specified 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 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).

2.2 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 states. *Net* is an abstraction of the messages in transit. The order of delivery by the communication model is based on the channel priorities defined by BLOCKS.

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). *The set of all possible executions is the set of all finite or infinite sequences of states where consecutive states conform to the transition relation, and such that a message is received at most once and this reception is preceded by a send.*

Definition 3 (Network). *At any point of an execution, Net is the set of channels where at least one message is in transit.*

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.*

Channel priorities are a set of static constraints that forces specific receptions to be *disabled*, depending on the values of *Net* and the *Channels* those transitions relate to. (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:*

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

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 $Sys, C \models P$. We define an analyzer which asks the framework whether a set of constraints applied to Sys satisfies P. The analyzer infers 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 candidates, built by adding new constraints to the current candidate.

3.1 Introducing New Constraint Types

The candidates are built in a incremental manner. Because of this, we have to ensure that any constraint added to a candidate does not invalidate the need for the existing ones. Indeed, adding a new BLOCKS constraint may cause a previously taken reception transition to become *disabled* without notice. This is resolved by the use of two other constraints types, exclusively used during the inference process: the ALLOWED constraints, and the BLOCKED constraints.

The (a ALLOWED b) constraint indicates that the building of the candidate has allowed the exploration of states that would be unreachable should channel a BLOCKS channel b. A constraint (a BLOCKS b) is thus prevented from being added to this candidate.

The (B BLOCKED c) constraint, on the other hand, is used to convey that at least one channel in the B set BLOCKS the channel c. Not specifying which channel does the actual blocking, like we would have to when using BLOCKS constraints, lets the incremental inferring process add (d ALLOWED c) with $d \in B$ to the candidate as long as, for any set of constraints C:

$$\forall B \subseteq Channels, \forall c \in Channels, (B \text{ BLOCKED } c) \in C \Rightarrow \\ \exists b \in B, (b \text{ ALLOWED } c) \notin C$$

As we don't know which element(s) of B in (B BLOCKED c) are blocking c, there are now three possibilities when a peer attempts reception on c. Either c is BLOCKED by all the channels in the blocking set and the reception is disabled, or c is ALLOWED by all of the channels in Net and the reception is possible, or the situation is *ambiguous* and it is yet unknown if the reception is possible.

Definition 6 (Ambiguity). *At a given state,* available *is the set of all channels that can be received from by the peer at that state.* Ambiguity *is the subset of* available *channels which are not* ALLOWED *by at least one of channels holding at least one message.*

$$available(peer) \triangleq \{channel \in Channels \mid \text{ENABLED}^1\ receive(peer, channel)\}$$
$$Ambiguity(peer) \triangleq \\ \{c \in available(peer) \mid \exists\ ch \in Net, (c \neq ch) \wedge \neg(ch \text{ ALLOWED } c)\}$$

[1] In TLA$^+$, ENABLED *Action* is true in a state if the action is possible, meaning there is a successive state reachable with *Action*. The action $receive(p, c)$ is enabled in a state if c is not blocked by a BLOCKED constraint at that point.

3.2 The Analyzer

The analyzer handles sets of candidates, each of which is a set of ALLOWED and BLOCKED constraints. It starts with a single candidate without any constraint. Information on the system is gathered by choosing a candidate, setting it as the constraint set of the communication model, and then asking the framework to report. The framework either declares that the targeted property is validated, or gives its report of *ambiguous* states. This report is $\langle net, channels \rangle$, the current value of Net and the *Ambiguity* set. When the framework reports that the expected property is verified, the candidate is added to the solutions. Otherwise, if there is no *ambiguous* state, the candidate is rejected; if *ambiguous* states are reported, the analyzer replaces the candidate by its children, each of which is generated using the following function where *chosen* is any subset of *channels*:

$$\begin{array}{l} update(Candidate, \langle net, channels \rangle, chosen) \triangleq \\ \quad Candidate \\ \cup \{((net \setminus \{v\}) \text{ BLOCKED } v) \mid v \in channels \setminus chosen\} \\ \cup \{(c \text{ ALLOWED } v) \mid v \in chosen \wedge c \in net\} \end{array}$$

Every subset of *channels* generates its own child (including $\emptyset$). The ALLOWED constraints make sure that the elements of *chosen* are not BLOCKED. The elements of *channels* that are not *chosen* are 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.

The inference process offers two variations of the analyzer. The Pessimistic Priority Finder find all the possible solutions. When studying a candidate, the framework stops and reports at each ambiguous state, and the analyzer explores all the children of this candidate. The Optimistic Priority Finder sacrifices exhaustiveness for performance. In that case, the framework pursues the verification until it finds an invalidation of the desired property (in which case another candidate is generated), or it validates the candidate.

4 Example: A Client-Controller-Application System

A system is composed of three peers: a client, a controller and an application (Figure 2). 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* it. If accepted, the client can send *upload* messages to the application. This controller starts the application (message *start*) when it accepts a client, and signals it to *end* when the client *logout*s.

Several problems occur if the messages are arbitrarily delivered. Among others, the application must consume *start* before processing the messages *upload* whereas they come from different peers; *end* must not be consumed when there are pending *upload*s; when the client *logout*s then *login*s again, *start* must not overtake any of the messages from the previous round (notably *upload* or *end*)...

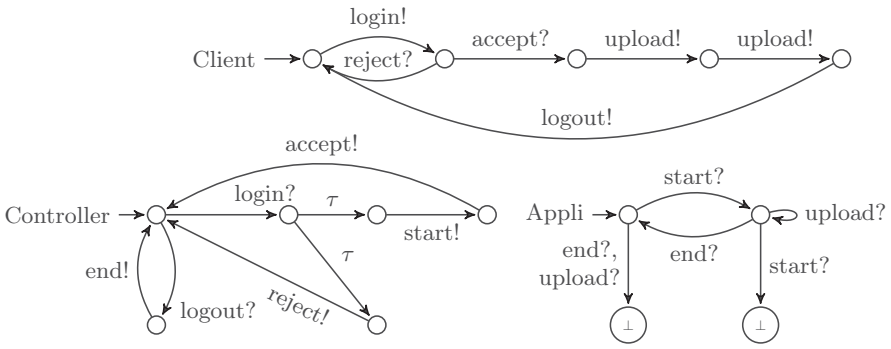

Fig. 2. A Client-Server System. The server is split into a controller (bottom left) and an application (bottom right). The controller accepts or rejects the client (top), and starts/ends the application when needed.

To avoid deadlock and $\perp$ states, specifying and manually verifying a correct ordering of messages is not easy. Our framework automatizes the verification and the inference algorithm discovers the seven possible solutions, among others:

($\{start\}$ BLOCKS $upload$)	($\{start\}$ BLOCKS $accept$)	($\{start\}$ 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 $start$)	**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 explicitly 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 in [12] (notion of ordering paradigm), [5] (notion of distributed computation classes), or [7] (formal description and hierarchy of several asynchronous models). Implementations of the basic models using histories or clocks are explained in classic textbooks [10, 12, 15]. 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 [4], a priority operator *prisum* is added to CCS, allowing for the expression of preference between two actions. Its semantics is similar to the extended constraints presented in section 6, as an enabled action forbids any action with a strictly lower priority. A difference is that 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. Conversely, 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 [11] lets its users give a list of channels to receive from, establishing a priority relation between them according to their location in the list. 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.

Priorities have been introduced in Petri Nets. In [3], priorities are statically associated to couples of transitions. In interleaving semantics, a transition is enabled at a marking when no transition with higher priority is enabled at that marking. However, concurrent semantics cannot be naturally defined with causal partial order. Dynamic priorities are introduced in [2]. Priorities are a relation between couples of transitions, for each possible marking. The objective is to reduce the number of enabled transitions at a marking, in order to reduce the size of the reachability graph without affecting the truth of the studied property.

Applicative Priorities on Labels. Another point of view is to assign priorities to labels or message identities, as we have done in this paper. [1] uses this to provide an interrupt mechanism in process algebra. Priorities are associated to labels and form a partial order. Its 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. A thorough exploration of priority in process algebras with synchronous communication is done in [9]. The authors distinguish operational semantics and behavioral congruences (for compositional reasoning). Priorities are associated to labels, and are used only in a synchronous communication event. [14] is an extension to 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.

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 [8]. 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.

6 Generalization and Conclusion

On the whole, channel priorities are easy to use when the priorities are automatically inferred. Thanks to inference, fine grain interactions do not have to be specified when developing a system, and application-specific protocols are derived from the system requirements. This allows to focus on the architecture of the system, and to postpone protocol considerations until deployment.

The next step is to extend the constraints to all the action types: *send*, *receive*, and *tau*, all having priorities over one another. 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 inference analyzer and the framework is easily altered to take those new constraints into account.

This extension allows to set a constraint such as ($\{tick?\}$ BLOCKS $alarm!$), to make a peer report an alarm if and only if it is unable to receive an expected message. It also allows to express fairness constraint: for instance a ($\{a?\}$ BLOCKS τ) with τ-stuttering ensures that the peer does not get stuck in a τ loop if a reception on a is possible. This also allows a cancellation or abortion action to be easily described for any kind of action.

References

1. 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.
2. Falko Bause. Analysis of Petri nets with a dynamic priority method. In *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.
3. Eike Best and Maciej Koutny. Petri net semantics of priority systems. *Theoretical Computer Science*, 96(1):175–174, 1992.
4. Juanito Camilleri and Glynn Winskel. CCS with priority choice. *Information and Computation*, 116(1):26–37, 1995.
5. Bernadette Charron-Bost, Friedemann Mattern, and Gerard Tel. Synchronous, asynchronous, and causally ordered communication. *Distributed Computing*, 9(4):173–191, February 1996.
6. Florent Chevrou, Aurélie Hurault, and Philippe Quéinnec. Automated verification of asynchronous communicating systems with TLA+. *Electronic Communications of the EASST (special issue AVOCS'15)*, 72:1–15, 2015.
7. Florent Chevrou, Aurélie Hurault, and Philippe Quéinnec. On the diversity of asynchronous communication. *Formal Aspects of Computing*, 28(5):847–879, 2016.
8. Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer, 1981.
9. Rance Cleaveland and Matthew Hennessy. Priorities in process algebras. *Information and Computation*, 87(1/2):58–77, 1990.
10. George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems: concepts and design*. Addison Wesley, second edition, 1994.
11. M. Elizabeth C. Hull. Occam - A programming language for multiprocessor systems. *Computer Languages*, 12(1):27–37, 1987.
12. Ajay D. Kshemkalyani and Mukesh Singhal. *Distributed Computing: Principles, Algorithms, and Systems*. Cambridge University Press, March 2011.
13. Leslie Lamport. *Specifying Systems*. Addison Wesley, 2003.
14. K. V. S. Prasad. A calculus of broadcasting systems. *Science of Computer Programming*, 25(2-3):285–327, 1995.
15. Michel Raynal. *Distributed Algorithms for Message-Passing Systems*. Springer, 2013.