Modeling Communication Semantics for Distributed Systems in Event-B*

Fernando Luís Dotti [1] **, Leila Ribeiro** [2]

[1]Faculdade de Informática - PUCRS - Porto Alegre - Brazil
fernando.dotti@pucrs.br

[2]Instituto de Informática - UFRGS - Porto Alegre - Brazil
leila@inf.ufrgs.br

Abstract. *During the development of algorithms for distributed systems, one has to adopt clear assumptions about the semantics offered by the underlying communication platform in order to show that the algorithms under construction fulfill the expected liveness and safety properties. In this paper we propose a library of reusable formal specifications defining several classic communication semantics. The specification of each communication semantics is presented along with the proofs of the expected main properties of each model. The library was build using Event-B and properties were shown using the theorem proving approach with the Rodin system. While modeling a distributed application one can reuse models from the proposed library (by refinement or extension) without having to redo all the proofs related to the communication platform. Moreover, existing proofs can be used to show desired properties of the application.*

1. Introduction

Building dependable systems is not a trivial task and several means are employed to enhance system dependability. Fault-prevention [Laprie et al. 2004] through the use of formal specification and analysis is one of those means.

When specifying and reasoning about distributed algorithms the assumptions about the semantics of the communication platform should be clearly defined such that (i) the desired properties of the algorithm are proven considering the appropriate communication semantics; (ii) the kind of target environment for the algorithm becomes explicit. The use of formal specification and analysis methods for distributed systems is not recent. Most commonly, specification languages support classic notions of processes and synchronization. However, there are several possible communication models available to build distributed algorithms. These models emerge from the combination of basic aspects of the message passing mechanism such as one-to-one or group communication, reliable or lossy, and several possible delivery orders (e.g.: unordered, FIFO, absolute, total and causal). The specification of a distributed algorithm requires the modeling of the underlying communication model, which is a non trivial task, as well as the distributed algorithm itself. Therefore, a library of reusable formal specifications of relevant communication models is highly desirable. This is the contribution of this paper, that provides formal definitions of several communication semantics in the asynchronous model of computation, both for point-to-point as well as for group communications. More specifically, we present 3 models for point-to-point communication: unordered, unordered lossy, and ordered message delivery; and 5 for group communication: unordered, fifo, total, absolute

*This work is partially supported by FAPERGS and CNPq.

and causal orders of message delivery. The specification of each model is characterized by its main properties, that are then proven using the theorem proving approach.

Most of the approaches to formalize and reason about distributed systems are based on model-checking, which is usually more amenable to non-theoreticians if compared to theorem proving. One of the main advantages of model checking is that the process of proving properties is fully automatic. However, there are some drawbacks of this approach. One of them is the impossibility to handle infinite (or very large systems). Another relates to how model checkers are typically used. The common approach is to translate a specification/program to the input language of the model checker and then verify the properties using the tool. However, in most of the cases, there is no proof of correctness of this translation. Such proofs are usually very involved (see e.g. [Ribeiro et al. 2011]), but are absolutely necessary to ensure that the properties shown to hold by the model checker will actually hold in the actual specification or implementation.

In the approach proposed in this paper, we use the theorem proving approach, relying on the Event-B method [Abrial 2010]. Event-B is based on B [Abrial 2005], but adds the concepts of machine and events, that make the language very well-suited to the specification of distributed and concurrent systems. While analysis through theorem proving is not fully automatic, it allows to consider infinite state systems or systems with arbitrary size. The refinement technique supported by Event-B allows stepwise developing of systems, by gradually introducing implementation details. This way, the transformation step from a very concrete Event-B specification (after a series of refinements) to code in a programming language is easier to perform (and consequently less error prone). One of the potential reasons for this increase of use of Event-B is the current tool support for specification and theorem proving offered by the RODIN platform[DEPLOY Project].

This paper is organized as follows: in Section 2 we present informally the several communication models defined in the library; in Section 3 we review the Event-B formal method; in Section 4 the library of communication models is presented, starting with basic data structures, followed by models for point-to-point and group communication. Section 5 presents a discussion on how to use the proposed library, and Section 6 comments on related work, future directions and conclusions.

2. Communication Models

In this contribution we focus on message passing communication. Within this scope, we can classify communication according to the number of processes involved and the several delivery orders. In the first dimension we have point-to-point[1] or group communication; and in the second dimension we have: unordered, fifo, causal and total (the last two applicable only to group communication) [Birman 1996]. Combining these dimensions, the following communication models for point-to-point communication were modeled:

- unordered, lossy channel: communication is among a producer and a consumer part and messages can be delivered[2] out of order as well as be lost;
- unordered, reliable: same as above, but all messages are delivered;
- FIFO: all messages are delivered in the order they are created.

[1] In this text also referred to as one-to-one.

[2] In the Distributed Systems literature message “delivery” means the arrival of a message at the destination and that the message can be consumed according to the communication criteria in use.

Note that we enumerate communication models according also to their non-determinism, the former models exhibit a reacher behavior than the next ones. Regarding group communication, the following orders were modeled:

- unordered: messages are delivered at the group members in any order;
- FIFO: any two messages from the same source are delivered at each member in the order they were sent by the source. However, messages from different sources may be delivered in different orders at different members;
- causal: message delivery is according to the "happened-before" (or $\rightsquigarrow$) relationship by Lamport [Lamport 1978]; Any message m delivered at process p_s before it sends m' has to be delivered at process p_r before m' is delivered. The causal relation is transitive (if $m \rightsquigarrow m' \rightsquigarrow m''$ then $m \rightsquigarrow m''$) and partial.
- total: messages are delivered in the same order within a group. This order is not restricted to the order in which the messages were created, the important fact is that delivery is identical in all processes belonging to the same group.

3. Event-B

Event-B [Abrial 2010] is a state-based formalism closely related to Classical B [Abrial 2005].

Definition 1 (Event-B Model, Event) *An Event-B Model is defined by a tuple* $EBModel = (c, s, P, v, I, R_I, E)$ *where* c *are constants and* s *are sets;* v *are the model variables*[3]*;* $P(c, s)$ *is a collection of axioms constraining* c *and* s*;* $I(c, s, v)$ *is a model invariant limiting the possible states of* v *s.t.* $\exists c, s, v \cdot P(c, s) \wedge I(c, s, v)$ *- i.e.* P *and* I *characterise a non-empty set of model states;* $R_I(c, s, v')$ *is an initialisation action computing initial values for the model variables; and* E *is a set of model* events.

Given states v, v' *an event is a tuple* $e = (H, S)$ *where* $H(c, s, v)$ *is the guard and* $S(c, s, v, v')$ *is the before-after predicate that defines a relation between current and next states. We also denote an event guard by* $H(v)$*, the before-after predicate by* $S(v, v')$ *and the initialization action by* $R_I(v')$*.*

A model has a set of possible initial states, defined by the initialization action, that by construction satisfy the model's invariant I. Given a state s satisfying I, an enabled event on s leads to another state s' which also satisfies I. The existence of such an enabled event and the invariant non-violation are assured by construction trough a series of proof obligations on each event of the model. The behavior of the model is obtained by the application of every possible event to every valid state, starting from the set of states valid according to the initialization action, giving rise to a state transition system.

To refine model M we construct a new model M' that is behaviorally related to the old one. In Event-B, this is achieved by constructing a refinement mapping between M' and M and by discharging a number of refinement proof obligations.

4. Formalizing Communication Semantics using Event-B

Now we present the library of formal specifications that represent the several communication semantics listed in Section 2. They were developed in Event-B, using a refinement

[3]For convenience, as in [Abrial 2005], no distinction is made between a set of variables and a state of a system.

approach. The invariants and variants are used to characterize and prove the main properties of each of these models.

The most abstract specification, $M0$, is the less restricted one in terms of possible computations, and models basic elements of unordered reception in a lossy channel. $M1$ refines $M0$ introducing the sender part and restricting the behavior to reliable channels. $M1loss$ refines $M0$ analogously to $M1$, but allowing message loss. $M2$ refines $M1$ including an order in message generation. In this model the order is not respected in the reception, $M2$ models reliable unordered reception. $M2ordered$ refines $M2$ restricting its computations to ordered reception of messages, modeling FIFO communication.

The most abstract group communication specification, $M3$, refines $M2$ introducing: multiple processes with unique identifications; the concept of process group; message addressing using groups; and multiple message reception (at each process in the destination group). $M3$ thus presents all basic elements to model unordered group communication. $M3fifo$, $M3total$ and $M3causal$ each refine $M3$ to restrict possible computations representing, respectively FIFO, total, and causal orders in group communication.

Section 4.1 describes the data structures of the models; Sections 4.2 and 4.3 present the point-to-point and group communication models, respectively.

4.1. Basic Data Structures

The first step to construct a model using Event-B is to define the main types of elements that play important roles in the reality being modeled. These types are defined as $Sets$ in an Event-B context. Special elements of these types as well as basic operations to manipulate them can be defined as $Constants$, the behavior of the operations is restricted by stating suitable $Axioms$.

In the following, we present the main types used in our description of communication models. First, we present general structures that will be used both in point-to-point as well as in group communication, and then we list the structures that are specific to group communication. To ease understanding and save space, we will not use strictly the Event-B syntax here. We will rather list the main sets, operations and axioms together with informal explanations[4]. We also list some auxiliary theorems that were proven.

4.1.1. General Communication Structures

The basic data type in our communication models is a set of messages ($Msgs$). The intuition is that this set should contain all messages that should be delivered in a system in some period of time. We assume that this set is finite because in a any (finite) time interval, it is not possible that an infinite number of messages is delivered. The number of messages in this set is arbitrary, this is not stated in the model. We use two operations on messages: $idmsg$, that returns the message identifier (modeled by a natural number), and $next$, that, given a set of messages, returns the one with the smallest identifier. These structures can be seen in Figure 1. The operation $idmsg$ is a total injective function, guaranteeing that all messages have unique identifiers, whereas $next$ is a total function mapping each non-empty subset $msgs$ of $Msgs$ to one of its messages, this function is

[4]In Event-B, names of operations are declared as constants and their types as axioms.

defined in axiom **ax1** by using the inverse $idmsg^{-1}$ over the number that is the minimum of all numbers in the range of a function obtained by restricting $idmsg$ to those pairs whose domain is in the set $msgs$. Theorem **the1** states that the identifier of the $next$ of a group of messages is really the least one in this set. The fact that this theorem could be proven assures that our definition of the $next$ operation is correct.

SETS
$Msgs$

OPERATIONS
$idmsg$: $Msgs \rightarrowtail \mathbb{N}$
$next$: $\mathbb{P}_1(Msgs) \rightarrow Msgs$

AXIOMS
ax1 : $\forall msgs \cdot msgs \subseteq Msgs \wedge msgs \neq \varnothing \Rightarrow next(msgs) = idmsg^{-1}(min(ran(msgs \lhd idmsg)))$

THEOREMS
the1 : $\forall msgs \cdot \forall m \cdot msgs \subseteq Msgs \wedge m \in msgs \Rightarrow (m \neq next(msgs) \Rightarrow idmsg(m) > idmsg(next(msgs)))$

Figure 1. Basic Data Structures

4.1.2. Group Communication Structures

To specify group communication, additionally to the set of messages we define the sets of processes ($Proc$) and groups ($Groups$). These sets are also required to be finite. We define 4 operations: $idproc$, that assigns a unique identifier to each process (a natural number); $source$, that assigns a source process to each message; $target$, that assigns a target group to each message; and $members$, that assigns a set of processes to each group. These structures can be visualized in Figure 2.

SETS
$Procs$, $Groups$

OPERATIONS
$idproc$: $Procs \rightarrowtail \mathbb{N}$
$target$: $Msgs \rightarrow Groups$
$source$: $Msgs \rightarrow Procs$
$members$: $Groups \rightarrow \mathbb{P}_1(Procs)$

Figure 2. Group Data Structures

4.2. Point-to-Point Communication

The behavior of each communication model will be described by corresponding Event-B machines. A machine is defined by a set of variables, describing states; a set of invariants, used to define the types of variables and state properties that should remain the same throughout execution; a variant, that is an expression used to prove termination; and a set of events, that actually define which state changes may occur. In the following, we start with a very abstract machine, representing a system that might behave well, but might also lose messages, and refine it further until we get a model in which we can guarantee that all messages arrive in the same order they were generated.

Again, we will not follow strictly the Event-B syntax. We will describe the set of variables of each machine, together with the corresponding types (instead of declaring the types using invariants), the relevant invariants/variant and all events. The initialization event will be omited because it is typically trivial.

4.2.1. Model M0

We start with a very simple machine with one variable ($deliveredMsgs$) that describes the set of messages that have been successfully delivered. This set is of course a subset of the set of all messages that should be delivered ($Msgs$). The guard of the *RECEIVEor-LOSE* event assures that it may only happen if there is a message m that has not yet been delivered. The action is to update the set of delivered messages. However, at this abstract level, the action is a non-deterministic assignment modeling the fact that this message may either be delivered or not. Figure 3 presents this model.

MACHINE *M0*
VARIABLES
$deliveredMsgs \subseteq Msgs$

EVENTS
Event *RECEIVEorLOSE* $\widehat{=}$
any m
where
grd1 : $m \notin deliveredMsgs$
grd2 : $m \in Msgs$
then
actDeliv : $deliveredMsgs :\in \{deliveredMsgs, (deliveredMsgs \cup \{m\})\}$

Figure 3. Machine Model 0

4.2.2. Model M1

In the previous model, it was not possible to guarantee that all messages would eventually be delivered. Actually, it was possible that messages get lost (or that the event deliver do nothing). Now, we will add a set of messages that have been generated in the network and have not yet been delivered (called $currentMsgs$). The main invariant is that there is no message that can be currently in the network and have been delivered (**invMsgs**). We included an event that generates messages (event *SEND*, it just picks a message that has not yet been generated and includes it in the $currentMsgs$ set. Invariant, **invEnable** assures that the occurrence of event *SEND* enables the occurrence of event *RECEIVE* (it makes the guard of *RECEIVE* true). The event *RECEIVE* was then refined to actually remove the message from the $currentMsgs$ set and include it in the $deliveredMsgs$ set. The variant assures that, if we consider just the deliver event, the set of delivered messages always grows (formally, this is represent by the fact that the number of elements of set $Msgs$ minus set $deliveredMsgs$ decreases). Since the set of all messages that should be delivered is finite, this variant guarantees that eventually all messages will reach their destinations. The status $convergent$ in the description of an event means that this event should be considered to prove the variant. Since all occurrences of event *SEND* enable an occurrence of *RECEIVE*, the former can be seen as an intermediate step to deliver

messages. This model is shown in Figure 4. Note that, since this model is a refinement of M0, we only present in Figure 4 the newly introduced variables and invariants, as well as new guards and new/modified actions in the events.

MACHINE *M1*
REFINES M0
VARIABLES
$currentMsgs \subseteq Msgs$

INVARIANTS
invMsgs : $deliveredMsgs \cap currentMsgs = \varnothing$
invEnable : $(\exists m \cdot m \in currentMsgs \Rightarrow (m \in Msgs \wedge m \notin deliveredMsgs))$
VARIANT
$Msgs \setminus deliveredMsgs$

EVENTS
Event *SEND* $\widehat{=}$
any m
where
grd1 : $m \in Msgs$
grd2 : $m \notin currentMsgs$
grd3 : $m \notin deliveredMsgs$
then
actCurrent : $currentMsgs := currentMsgs \cup \{m\}$
Event *RECEIVE1* $\widehat{=}$ **refines** *RECEIVE*
status CONVERGENT
any m
where
grd3 : $m \in currentMsgs$
then
actDeliv : $deliveredMsgs := deliveredMsgs \cup \{m\}$
actCurr : $currentMsgs := currentMsgs \setminus \{m\}$

Figure 4. Machine Model M1

4.2.3. Model M1loss

Alternatively, *RECEIVE* event could be refined in two different events, one delivering a message and another losing it (removing from $currentMsgs$ and not including it in $deliveredMsgs$). In this case, it is possible to prove that eventually all messages would either be delivered or lost. Due to space limitations, this model will not be shown here.

4.2.4. Model M2

To be able to show that messages arrive in the same order in which they were generated, the first step is to assume a generation order. This will be done in this model, called M2, that refines model M1. We used an auxiliary variable called $generatedMsgs$, that is just the union of the sets of current and delivered messages. To generate messages in some order, we assume that each message in the set of messages to be delivered (set $Msgs$) has a unique message identifier (given by function $idmsg$). This identifier is a natural number, and the order in which messages are generated will obey the $less$ order between

natural numbers (that is a total order). The invariant that guarantees the generation order (**invGenOrder**) states that all messages that have been generated in the network must have identifiers that are less than identifiers of non-generated messages. This model is depicted in Figure 5. Note that, with respect to model M1, only event *SEND* is refined, and therefore the variant remains true (assuring that all messages will be eventually delivered). In the *SEND2* event we require, additionally to the guards of *SEND1*, that the message that was chosen to be generated has the least identifier among all non-generated messages.

MACHINE *M2*
REFINES M1
VARIABLES
$generatedMsgs \subseteq Msgs$

INVARIANTS
invGen : $deliveredMsgs \cup currentMsgs = generatedMsgs$
invGenOrder: $\forall mc \cdot \forall md \cdot mc \in generatedMsgs \wedge md \in Msgs \setminus (generatedMsgs) \Rightarrow idmsg(md) > idmsg(mc)$

EVENTS
Event *SEND2* $\widehat{=}$ **refines** *SEND*
any m
where
grd4: $m = next(Msgs \setminus generatedMsgs)$
then
actGen: $generatedMsgs := generatedMsgs \cup \{m\}$
Event *RECEIVE2* $\widehat{=}$ *RECEIVE1*

Figure 5. Machine Model M2

4.2.5. Model M2ordered

To be able to prove that messages are delivered in the same order in which they were generated, we need, additionally to the invariant **invGenOrder** of the previous model, to guarantee that any message that have been delivered has an identifier that is less than all identifiers of messages currently in the network. This is stated by invariant **invOrder**, that is satisfied by model M2ordered, that is a refinement of model M2.

MACHINE *M2ordered*
REFINES M2
INVARIANTS
invOrder : $\forall mc \cdot \forall md \cdot mc \in currentMsgs \wedge md \in deliveredMsgs \Rightarrow idmsg(mc) > idmsg(md)$

EVENTS
Event *SEND2order* $\widehat{=}$ *SEND2*
Event *RECEIVE2order* $\widehat{=}$ **refines** *RECEIVE2*
status CONVERGENT
any m
where
grd4 : $m = next(currentMsgs)$

Figure 6. Machine Model M2ordered

4.3. Group Communication

To model group communication, we use both the general as well as the group data structures (Figures 1 and 2). Now, each message in the $Msgs$ set has the information about its source process and its target group. Therefore, when one message from this set is chosen to be delivered, one copy is generated for each of the processes belonging to the target group, building the $currentGMsgs$ set. When messages are delivered to one of the processes of the group, they are transferred to the $deliveredGMsgs$ set. Only when the last copy of the same message is delivered, the message is transferred from the $currentMsgs$ to the $deliveredMsgs$ (refining the event *RECEIVE* from the previous models). The invariants that describe the different group communication models are defined over the $currentGMsgs$ and $deliveredGMsgs$ sets. In the following we present the basic model for group communication (model M3), and then refine it to generate 3 different models, corresponding to FIFO order, causal order and total order of message delivery.

4.3.1. Model M3

As discussed above, the variables introduced in this model are $currentGMsgs$ and $deliveredGMsgs$ sets, that are similar to their counterparts in the point-to-point communication, but have several copies of the same message, one for each of its target processes. Instead of just messages, these sets contain pairs of message and target process. This model is constructed as a refinement of model M2, assuring that all messages are eventually delivered. A number of invariants were included that guarantee the consistency between the existing and the new variables. For example, invariant **invCons1** assures that only copies of messages that are in the $currentMsgs$ set can be in the $currenGMsgs$ set. Invariant **invCons2** guarantees that for any message m that is in $currentMsgs$, all copies of m (described by restricting the set $currentGMsgs$ to the domain $\{m\}$) are actually addressed to targets of m. Analogous invariants exist for delivered messages. Finally, invariant **invCon6** states that for any message m that was generated in the network, there is one copy for each of its target processes (this copy can be either in the $currentGMsgs$ set or in the $deliveredGMsgs$ set). Moreover, there can be no copy of this message to any process that does not belong to the members of the target group. The variant assures that the set of copies of messages that should be delivered is always becoming smaller (and thus eventually all messages will be delivered). There is still an event that describes the complete delivery of one message (the delivery of the last of the copies), and this event refines the one of the more abstract level. We include a new event that model the delivery of messages that are not the last one. The guards are practically the same, except of guard **grd7**, that checks whether the copy is the last one or not. The behavior of the event, considering the sets $currentGMsgs$ and $deliveredGMsgs$ is the same, but the partial receive event does not update other variables (and therefore, is not observable at the abstract level). The next three models are refinements of this model.

4.3.2. Model M3fifo

To model FIFO communication, we defined invariant **invFIFO**. This invariant states that whenever there are two different messages mc and md having the same process p as

MACHINE *M3*
REFINES M2
VARIABLES
$currentGMsgs \subseteq (currentMsgs \times Procs)$

$deliveredGMsgs \subseteq (generatedMsgs \times Procs)$

INVARIANTS
invDelCurr : $deliveredGMsgs \cap currentGMsgs = \varnothing$
invCons1 : $currentMsgs = dom(currentGMsgs)$
invCons2 : $\forall m \cdot m \in currentMsgs \Rightarrow \{m\} \lhd currentGMsgs \subseteq (\{m\} \times members(target(m)))$
invCons3 : $deliveredMsgs \subseteq dom(deliveredGMsgs)$
invCons4 : $\forall m \cdot m \in deliveredMsgs \Rightarrow m \notin dom(currentGMsgs)$
invCons5 : $\forall m \cdot m \in deliveredMsgs \Rightarrow \{m\} \times members(target(m)) \subseteq deliveredGMsgs$
invCons6 : $\forall m \cdot m \in generatedMsgs \Rightarrow (\{m\} \lhd currentGMsgs) \cup (\{m\} \lhd deliveredGMsgs) = (\{m\} \times members(target(m)))$

EVENTS
Event *SEND3* $\widehat{=}$ **refines** *SEND2*
any m
then
actCurrentG : $currentGMsgs := currentGMsgs \cup (\{m\} \times members(target(m)))$
Event *RECEIVE3* $\widehat{=}$ **refines** *RECEIVE2*
status CONVERGENT
any m, p
where
grd4 : $p \in Procs$
grd5 : $p \in members(target(msg))$
grd6 : $p \notin ran(\{m\} \lhd deliveredGMsgs)$
grd7 : $ran(\{m\} \lhd deliveredGMsgs) \cup \{p\} = members(target(msg))$
then
actCurrentG : $currentGMsgs := currentGMsgs \setminus \{(m \mapsto p)\}$
actDeliverG : $deliveredGMsgs := deliveredGMsgs \cup \{(m \mapsto p)\}$
Event *PART-RECEIVE3* $\widehat{=}$
status CONVERGENT
any m, p
where
grd1 : $msg \notin deliveredMsgs$
grd2 : $msg \in Msgs$
grd3 : $msg \in currentMsgs$
grd4 : $p \in Procs$
grd5 : $p \in members(target(msg))$
grd6 : $p \notin ran(\{m\} \lhd deliveredGMsgs)$
grd7 : $ran(\{m\} \lhd deliveredGMsgs) \cup \{p\} \subset members(target(msg))$
then
actCurrentG : $currentGMsgs := currentGMsgs \setminus \{(m \mapsto p)\}$
actDeliverG : $deliveredGMsgs := deliveredGMsgs \cup \{(m \mapsto p)\}$

Figure 7. Machine Model M3

target, one of them currently in the network (mc) and the other already delivered (md), and the source of these two messages is the same process, the identifier of message mc must be greater than the one of md (remind that messages with smaller identifier are always generated before messages with greater ones). This assures that a process can not receive messages from another process in an order that was different from the one they were generated. Note that this invariant does not restrict the order in which messages from different processes are received. To implement this model, we just need to include one extra guard in the receipt events (guard **grd8**), that assures that the message being chosen to be delivered (m) has the least identifier among all messages from the same source process that are currently in the network. Due to lack of space, we will only depict event *RECEIVE3fifo*, event *PART-RECEIVE3fifo* is analogous.

MACHINE *M3fifo*
REFINES M3
INVARIANTS
invFIFO: $\forall p \cdot \forall mc \cdot \forall md \cdot p \in Procs \land (mc \mapsto p) \in currentGMsgs \land (md \mapsto p) \in deliveredGMsgs \land source(mc) = source(md) \Rightarrow idmsg(mc) > idmsg(md)$

EVENTS
Event *SEND3fifo* $\widehat{=}$ *SEND3*

Event *RECEIVE3fifo* $\widehat{=}$ **refines** *RECEIVE3*
status CONVERGENT
any m
where
grd8: $\forall msg \cdot msg \in currentMsgs \land source(m) = source(msg) \Rightarrow idmsg(m) < idmsg(msg)$

Figure 8. Machine Model M3fifo

4.3.3. Model M3total

Total order is achieved by invariant **invTOTAL** requiring that if a message $m1$ has been delivered for process $p1$ and there is another message $m2$ for $p1$ currently in the network, if $m2$ has already been delivered for any other process $p2$ that is also in the target group of message $p1$, than $m1$ must have also been delivered for $p2$. This assures that messages to the same targets arrive in the same order. The implementation of this behavior occurs in the receive events, that must have an additional guard (**grd8**) guaranteeing that, whenever message m is chosen to be delivered to process p, p has already received all messages that all other processes that have also m as target have received.

4.3.4. Model M3causal

Finally, to model causal order we included a new variable called $order$, that records the causal order in which messages are delivered. This order is built incrementally during execution: once a message is generated, it causally depends on all messages that the source process of this message have received. This is described by action **actOrder** in the event *SEND3causal*. The invariant that describes causal order is **invCAUSAL**, that states that if a process p received a message $m1$ and did not receive $m2$ yet, than $m2$ can not be a

MACHINE *M3total*
REFINES M3
INVARIANTS
invTOTAL: $\forall p1 \cdot \forall p2 \cdot \forall m1 \cdot \forall m2 \cdot$
$((m1 \mapsto p1) \in deliveredGMsgs \wedge (m2 \mapsto p1) \in currentGMsgs) \Rightarrow$
$((m2 \mapsto p2) \in deliveredGMsgs \Rightarrow (m1 \mapsto p2) \in deliveredGMsgs)$

EVENTS
Event *SEND3total* $\widehat{=}$ *SEND3*

Event *RECEIVE3total* $\widehat{=}$ **refines** *RECEIVE3*
status CONVERGENT
any m
where
grd8: $\forall p1 \cdot p1 \in members(target(msg)) \Rightarrow (\exists m1 \cdot (m1 \mapsto p1) \in deliveredGMsgs \wedge p \in members(target(m1)) \Rightarrow (m1 \mapsto p) \in deliveredGMsgs)$

Figure 9. Machine Model M3total

cause of $m1$. Upon reception of a message, we have to make sure that all possible causes of this message have already been received (guard **grd8**). In Figure 10 we omitted some invariants about the causal order (assuring that is is irreflexive, antisymmetric, transitive, and that the causal order is compatible with the order of identifiers).

MACHINE *M3causal*
REFINES M3
VARIABLES
$order \subseteq generatedMsgs \times generatedMsgs$

INVARIANTS
invCAUSAL: $\forall p \cdot \forall m1 \cdot \forall m2 \cdot ((m1 \mapsto p) \in deliveredGMsgs \wedge (m2 \mapsto p) \in currentGMsgs) \Rightarrow (m2 \mapsto m1 \notin order)$

EVENTS
Event *SEND3causal* $\widehat{=}$ **refines** *SEND3*
any m
then
actOrder : $order := order \cup ((dom(deliveredGMsgs \rhd \{source(m)\}) \cap dom(target \rhd \{target(m)\})) \times \{m\})$
Event *RECEIVE3total* $\widehat{=}$ **refines** *RECEIVE3*
status CONVERGENT
any m, p
where
grd8: $\forall msg \cdot msg \mapsto p \in currentGMsgs \wedge m \neq msg \Rightarrow msg \mapsto m \notin order$

Figure 10. Machine Model M3causal

5. Using the Library of Communication Models

With the proposed library of communication models, a developer of a distributed algorithm would first choose the appropriate communication model specification, matching the assumptions of the algorithm in terms of communication type (group or point-to-point) and message ordering, and then would refine that specification to represent the specific aspects of the distributed algorithm.

As an example, we have modeled a mutual exclusion distributed algorithm based on the one proposed by [Ricart and Agrawala 1981]. That algorithm defines how a group of processes should collaborate to assure deadlock- and starvation-free mutual exclusion. A process that wants to enter the critical section sends a request to all group members. Each member has to eventually answer the request giving permission to the originator. When the originator receives permissions from all the processes, then it is allowed to enter the critical section. The access is assured to be linearized because a process in the critical section or processes that compete for the critical section, and have priority over others, do not send back permissions until they leave their critical session. Priority is decided homogeneously using a message sequence number scheme.

In our exercise, instead of dealing with sequence numbers, we choose the group communication system to offer the total order delivery and build a mutual exclusion protocol based on that. To entry the critical session the procedure is the same as above. Now, relying on the total ordering of messages, when the originator process receives its own request it delays answers to requests by others until it leaves its own critical section. Any request by other processes that arrives prior to its own request is answered with permission. This example is interesting because the fundamental properties of the algorithm are built relying on the communication semantics which is explicitly stated and proved in the library. Due to space restrictions a detailed presentation of the example is not possible.

6. Final Considerations

As mentioned before, formal specification and analysis of distributed algorithms is not recent. Most commonly, one can observe the use of specification languages such as Pi-calculus [Milner 1999], CSP [Hoare 1985], I/O Automata [Lynch and Tuttle 1989], PROMELA [Holzmann 1997], that support notions of processes and communication channels. PROMELA, for instance, supports channels suited to model one to one communication. I/O Automata and some process calculi languages allow the synchronization of multiple processes in a channel. While this can be used as an abstraction to group communication, it does not represent important details in an asynchronous distributed setting where arbitrary communication delays may happen between any two pairs of processes and give rise to several possible delivery orders.

Efforts for a formal representation of middleware and some available communication mechanisms can be witnessed in the literature. For instance, in [Bryans et al. 2009] the authors model middleware behavior to assess its impact in the design of distributed applications. As a part of that, some aspects of the communication platform are modeled. However, only a restricted subset of communication models is considered. In [Hoang et al. 2009] the authors propose the use of specification patterns as means to foster the reuse of specifications and proofs. As example, they model some one-to-one communication mechanisms as patterns. A system designer could then adopt one such pattern in a guided refinement step, what would ease the integration of communication mechanisms to the design of a system. Despite these contributions, we could not find support for a more comprehensive set of communication models in a way that would enable reuse of specifications and proofs when designing and reasoning about distributed algorithms.

Differently of building a specific distributed algorithm, in this paper we have proposed a library of specifications, for several communication semantics, with their main

properties proven. Approximately 170 proof obligations were generated by the platform and proved interactively. A developer of a distributed algorithm can reuse the specification that matches the assumptions of the algorithm in terms of type, ordering and reliability of the communication mechanism. This reuse includes the reuse of all existing proofs.

This was the first step towards the construction of a formal framework for the verification of distributed algorithms using theorem proving. Ongoing work is the modeling and verification of distributed algorithms as case studies. Future work includes extending modeling abstractions to other features of group communication.

References

Abrial, J. R. (2005). *The B-Book: Assigning Programs to Meanings*. Cambridge University Press.

Abrial, J.-R. (2010). *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, New York, NY, USA, 1st edition.

Birman, K. P. (1996). *Building secure and reliable network applications*. Manning Publications, USA.

Bryans, J., Fitzgerald, J., Romanovsky, A., and Roth, A. (2009). Formal modelling and analysis of business information applications with fault tolerant middleware. In *Proceedings of the 2009 14th IEEE International Conference on Engineering of Complex Computer Systems*, ICECCS '09, pages 68–77. IEEE Computer Society.

DEPLOY Project. Event-b and the rodin platform. http://www.event-b.org/ (last accessed April 2nd 2012).

Hoang, T. S., Fürst, A., and Abrial, J.-R. (2009). Event-B patterns and their tool support. In Hung, D. V. and Krishnan, P., editors, *SEFM*, pages 210–219. IEEE Computer Society.

Hoare, C. A. R. (1985). *Communicating Sequential Processes*. Prentice Hall.

Holzmann, G. J. (1997). The model checker SPIN. *IEEE Trans. on Soft. Eng.*, 23(5):279–295.

Lamport, L. (1978). Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565.

Laprie, J.-C., Randell, B., Avizienis, A., and Landwehr, C. (2004). Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Secur. Comput.*, 1(1):11–33.

Lynch, N. and Tuttle, M. (1989). An introduction to input/output automata. *CWI-Quarterly*, 2(3):219–246.

Milner, R. (1999). *Communicating and mobile systems: the π-calculus*. Cambridge University Press, USA.

Ribeiro, L., dos Santos, O. M., Dotti, F. L., and Foss, L. (2011). Correct transformation: From object-based graph grammars to promela. *Science of Computer Programming*, In Press, Corrected Proof:–.

Ricart, G. and Agrawala, A. K. (1981). An optimal algorithm for mutual exclusion in computer networks. *Commun. ACM*, 24(1):9–17.