An Abstract Channel Specification and an Algorithm Implementing It Using Java Sockets*

Chryssis Georgiou
Department of Computer Science
University of Cyprus, 1678 Nicosia, Cyprus
chryssis@cs.ucy.ac.cy

Peter M. Musial
Computer Science Department
Naval Postgraduate School, Monterey, CA 93943, USA
pmmusial@gmail.com

Alexander A. Shvartsman
Department of Computer Science & Engineering
University of Connecticut, Storrs, CT 06269, USA
aas@cse.uconn.edu

Elaine L. Sonderegger
Department of Computer Science & Engineering
University of Connecticut, Storrs, CT 06269, USA
esonderegger@engr.uconn.edu

Abstract

Abstract models and specifications can be used in the design of distributed applications to formally reason about their safety properties. However, the benefits of using formal methods are often negated by the ad hoc process of mapping the semantics of an abstract specification to algorithms designed to be executed on target distributed platforms. The challenge of formally specifying communication channels and correctly implementing them as algorithms that use realistic distributed system services is the focus of this paper. This work provides an original formal specification of an abstract asynchronous communication channel with support for dynamic creation and tear down of links between participating network nodes, and its implementation as an algorithm using Java sockets. The specification and the algorithm are expressed using the Input/Output Automata formalism, and it is proved that the algorithm correctly implements the specification, viz. that any externally observable behavior (trace) of the algorithm has a corresponding behavior of the specification. The approach presented here can be used to implement algorithms for dynamic systems, where communicating nodes may join, leave, and experience delays. The result is also of direct benefit to automated code generation, such as that implemented within the Input/Output Automata Toolkit at MIT.

*This work is supported in part by the NSF Grants 0121277, 0311368, and 0702670, and by VEROMODO Inc. through AFOSR Contracts F9550-05-C-0178 and FA9550-07-C-0114.

1. Introduction

The increasing complexity of distributed software systems makes reasoning about their behavior evermore challenging. Abstract specifications of distributed systems simplify formal reasoning about their correctness, and several formal systems have been used for this purpose, e.g., [2, 9, 1, 10, 11, 14, 17]. Using such systems enables end-to-end algorithm design in which correctness properties are preserved from initial specification to final executable.

Translation of abstract specifications into executable code for target environments is particularly challenging in the case of communication channels. Distributed systems are designed for a specific communication model, where the correctness (safety) properties of the communication channels used by the system directly impact the safety guarantees of the overall system. Common practice often foregoes the rigorous correctness arguments about the channel implementation and its interaction with the system components. Hence, it is not clear whether the resulting communication service is correct with respect to its high-level specification.

The key contribution of this work is the first specification of an abstract asynchronous communication channel with explicit support of dynamic creation and tear down of communication links between the network nodes, and its correct implementation as an algorithm using Java sockets [21]. For simplicity, our algorithm associates a unique socket with each communication link between a pair of nodes; the solution can be naturally extended to incorporate multiple, con-

978-0-7695-3192-2/08 $25.00 © 2008 IEEE
DOI 10.1109/NCA.2008.12

current, point-to-point socket connections. We prove that the algorithm correctly implements the specification, i.e., it preserves the safety guarantees of the specification. Hence, if our channel representation is used in a system that relies on such channels, then the reasoning presented here can be used in proving correctness of the system.

We use the Input/Output Automata model [14] to specify and reason about the behavior of distributed algorithms. A plethora of algorithms have been described using this model [12]. We refer to the language used to describe systems in this model as IOA [4]. Suites of tools [3, 23] have been developed to support system specification and development in IOA, including code generation tools [22]. These tools have been used to develop a number of distributed algorithms (e.g, [6, 5, 8]). Our work can be used to extend these tools to support the implementation of distributed algorithms with *dynamic* node participation.

Document structure. In Section 2 we present prior work and Java support for TCP sockets. In Section 3 we review the IOA model and state our assumptions. In Section 4 we present our communication channel, its implementation, and proof of correctness outline. We conclude in Section 5.

2. Background

Prior Work. Tauber [22] presents an IOA compiler for a target programming framework consisting of Java [21] and MPI [16]. The compiler design is proved correct to ensure that the safety guarantees of the source specification are preserved in the resulting Java/MPI implementation. The choice of MPI limits the domain of systems to those that do not encounter failures and arbitrary message latency, and where nodes do not join and leave during execution. In our work we use Java sockets and TCP [20], thus extending the domain of discourse to include systems supporting dynamic behaviors and failures.

Java Environment. Java [21] provides several classes to establish point-to-point connections via TCP sockets. A receiving node indicates its willingness to communicate with other nodes by creating an instance of the `ServerSocket` class and invoking its `accept()` method. A sending node attempts to connect with an accepting receiver by creating a `Socket` with the address and port of the receiver. If successful, the sender and receiver create data streams for the socket with the `getInputStream()` and `getOutputStream()` methods of `Socket`. Messages are received and sent over the connection with the `read` methods of the `InputStream` class and the `write` methods of the `OutputStream` class. Either the sender or the receiver may close the connection with the `close()` method.

The Java mechanism for reporting an error, such as a network timeout, is an `Exception`. For instance, when performing a `write` method, a node may discover by an exception that the destination node has initiated a close sequence or that the link between the nodes is no longer functioning. The methods used to establish communication and exchange messages are blocking; however, these methods can be parameterized to timeout if the desired event does not occur in some predetermined amount of time.

3. Model, Definitions, and Data Types

Input/Output Automata. Specifications in this work are done in terms of the Input/Output Automata model of Lynch and Tuttle [14, 12]. It is a labeled transition system model for specifying components in asynchronous distributed systems. Each automaton consists of a set of *actions* π (classified as input, output, or internal), a set of *states* s that includes *start* states, and a set of *transitions* or *steps* of the form (s, π, s') that specify the effects of the automaton's actions. An action is *enabled* if its preconditions are satisfied. Input actions are always enabled. The (parallel) *composition* operator allows an output action of one automaton to be identified with input actions in other automata.

The behaviors of an Input/Output Automaton are described by its *executions* and *traces*. An execution is a sequence of alternating states and actions starting with an initial state, e.g., $s_0, \pi_1, s_1, \pi_2, s_2 \ldots$. A trace is the sequence of input and output actions occurring in an execution. An automaton is said to *implement* another automaton if any trace of the former is a trace of the latter.

Channels and Nodes. As in [22], we assume algorithms given in the *node-channel form*, meaning the system is a collection of n asynchronous nodes executing application automata connected by up to $n(n-1)$ asynchronous channel automata. Each node is labeled with a unique identifier. We assume that channels do not corrupt and do not spontaneously create messages.

We concentrate on the high-level behavior and the interface with sockets via the Java libraries; we do not model TCP or the Java Virtual Machine (JVM) environment. We restrict our attention to unidirectional communication over a single socket connection between any pair of nodes.

Data Types. Throughout the paper, the set of unique location (node) identifiers is denoted as I. The set of all messages is denoted as M. $Streams$ is a set of all Java streams

used to read and write messages to and from TCP sockets. For the transition parameters, we have $i, j \in I$, $m \in M$, and $s \in Streams$.

4. Communication Channel with Graceful Comings and Goings

We present a model for an asynchronous communication channel connecting applications running on any number of networked nodes. A sender node may create connections with any number of receiver nodes, and any node may gracefully close the connection. Messages may be lost, delayed, and delivered out of order.

The current model supports a single socket connection between any two nodes. Thus, once a connection between two nodes is established and subsequently closed, it cannot be reopened (unless it can be determined that the socket can be reused). Allowing multiple, concurrent, socket connections between two nodes is a straightforward extension to this model, accomplished by adding the socket number as another dimension to each of the state variable arrays.

We first define an automaton, called AbsCh, modeling the behavior of a many-to-many, asynchronous communication channel that allows nodes to spontaneously connect and gracefully disconnect. Next, we present an automaton, called JvmCh, that models the behavior of the Java interface to a communication channel using TCP. Following Tauber's approach [22], we then establish a mediation between the sending application, the communication channel, and the destination application. The mediating automata are mapped to the nodes of the corresponding application automata, as illustrated in Figure 1. We refer to the composition of the JvmCh automaton with the mediating automata as the CompCh automaton. We then show that CompCh implements AbsCh, hence preserving the properties of our abstract asynchronous channel.

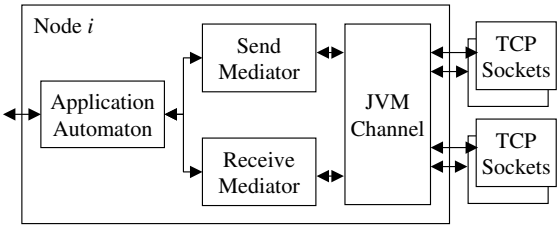

Figure 1. Node automata.

4.1. Abstract Channel Automaton

We present an abstract communication channel automaton, called AbsCh, that allows individual connections for nodes in I to be created. Messages then can be sent and received over the connections. The connections are closed in a graceful way, ensuring that messages that are in-transit are delivered before the connection is closed. The state and transitions of AbsCh are depicted in Figure 2.

The state of the automaton consists of four state variables, $messages$, $listening$, $status$, and $emptying$. The state variable $messages$ is a set of triples of the form $\langle m, i, j \rangle$, where m is a message sent by node i but not yet delivered to its destination at node j. A set called $listening$ is used to record the identifiers of all the receiver nodes that are listening for incoming connections. The $status$ state variable is an array indexed by $I \times I$. For each i and j from I, $status(i, j)$ contains the state of the channel from i to j, which may be any one of the following states:

- closed, the unidirectional channel from i to j is closed, and i cannot send messages to j.
- connecting, i is attempting to connect with j, but the connection is not yet fully established.
- connected, the channel between i and j is open, and messages sent by i will be transmitted to j.

The last state variable $emptying$ is a Boolean array, indexed by $I \times I$, where $emptying(i, j)$ is true if sender i is attempting to close the connection with receiver j; it remains true, and $status(i, j)$ is not set to closed, until all the messages in $messages$ from i to j are delivered.

In order to establish a connection from i to j, the receiving node, j, invokes the receiverListening(j) action that indicates its willingness to communicate. The receiver then awaits a connection request from i, which is a senderOpen(i, j) action. The respReceiverListening(i, j) action notifies j that a connection with node i has been created. Node i is then allowed to send messages to node j, and node j is capable of receiving messages sent by i. If a receiver j, after invoking a receiverListening(j) action, decides it is no longer willing to communicate, it indicates this with a receiverStopListening(j) action.

Messages may be sent at any time after a senderOpen(i, j) action, i.e., when $status(j)$ is connecting or connected. A message is deposited into the channel via the action send(m, i, j), where m is the message, i is the source node, and j is the destination node. The tuple $\langle m, i, j \rangle$ is added to the set $messages$, and the message is considered to be in-transit.

A message m from i may be received by j if $messages$ contains $\langle m, i, j\rangle$ as a result of an earlier send(m, i, j) action. The effect of receive(m, i, j) is the removal of m from the channel and its delivery to j. Messages can be delivered out of order. The action lose(m) models the loss of a message (e.g., due to a buffer overflow or network failure).

State:
$messages$, subset of $M \times I \times I$, initially $\emptyset$
$listening$, subset of I, initially $\emptyset$
$status : I \times I \rightarrow \{$closed, connecting, connected$\}$, initially all closed
$emptying : I \times I \rightarrow Boolean$, initially all false

Transitions:
input send(m, i, j)
Effect:
  if $status(i, j) \neq$ closed $\wedge \neg emptying(i, j)$ then
    $messages \leftarrow messages \cup \{\langle m, i, j\rangle\}$

input receiverListening(j)
Effect:
  $listening \leftarrow listening \cup \{j\}$

input senderOpen(i, j)
Effect:
  $status(i, j) \leftarrow$ connecting

input receiverStopListening(j)
Effect:
  $listening \leftarrow listening - \{j\}$

input receiverClose(i, j)
Effect:
  $messages \leftarrow messages - \{\langle m, s, r\rangle \in messages | s = i \wedge r = j\}$
  $status(i, j) \leftarrow$ closed

input senderClose(i, j)
Effect:
  $emptying(i, j) \leftarrow$ true

output receive(m, i, j)
Precondition:
  $\langle m, i, j\rangle \in messages$
  $status(i, j) =$ connected
Effect:
  $messages \leftarrow messages - \{\langle m, i, j\rangle\}$

output respReceiverListening(i, j)
Precondition:
  $status(i, j) =$ connecting
  $j \in listening$
Effect:
  $status(i, j) \leftarrow$ connected

internal senderClosing(i, j)
Precondition:
  $emptying(i, j)$
  $\forall\langle m, s, r\rangle \in messages, s \neq i \wedge r \neq j$
Effect:
  $status(i, j) \leftarrow$ closed
  $emptying(i, j) \leftarrow$ false

internal lose(m)
Precondition:
  $\langle m, i, j\rangle \in messages$
Effect:
  $messages \leftarrow messages - \{\langle m, i, j\rangle\}$

Figure 2. State and transitions of the abstract many-to-many automaton, ABSCH.

A connection may be closed by either the sender or the receiver. A sender node i initiates the closing of a connection with a senderClose(i, j) action. As a result $emptying(i, j)$ is set to true, and no new messages can be sent. However, the messages that are already in the channel must be delivered. Therefore, action senderClosing(i, j), which puts the connection in the closed state, occurs only after all the messages from i to j have been delivered (or lost). Alternatively, the receiver may close the connection with a receiverClose(i, j) action, in which case all in-transit messages to j from i are dropped, and the connection between i and j enters the *closed* state. Once the connection from i to j is closed, any messages that node i attempts to send to j are dropped.

4.2. JVM-TCP Channel Automaton

The state variables and transitions of the automaton JVMCH is defined in Figures 3 and 4. The set $jvmBuffer$ contains all messages $\langle m, stream\rangle$ in transit, where $stream$ defines the destination. The set $writeErrors$ records any messages written to streams that have been closed by their destinations. All requests to read a message from a stream are placed in $reading$, a subset of $Streams$. Identifiers of nodes that are in the accepting mode (i.e., Java servers) are maintained in the set $accepting$. The status of a connection between any two nodes is recorded in $jvmStatus$, whose values may be closed, notAccepting, connecting, sConnected, or connected (initially closed). The streams dedicated to each connection are stored in $jvmStream$. The Boolean array $jvmEmptying$ indicates if any messages remain to be sent to the destination after the sender has closed the connection.

Prior to exchanging messages, the following steps are taken. First, the receiver indicates its readiness to accept messages, which is done via action accept(j), where j is the receiving node. This corresponds to invoking the Java `accept()` method, and results in j being added to the set $accepting$. Second, sender i tries to create a connection with receiver j. This is accomplished using createStream(i, j), which combines constructing a `Socket` and invoking its `getOutputStream()` method in Java. There are two possible outcomes for the createStream(i, j) action. If a previous accept(j) action occurred at j, and thus j is in $accepting$, the operation is successful, and respCreateStream(i, j, s) will return the new stream dedicated to the connection. This outcome is indicated by removing j from $accepting$, assigning $jvmStatus(i, j)$ to connecting and then sConnected, and

updating $jvmStream(i, j)$ with the new stream. The receiver is then notified of the successful connection and assigned stream with a respAccept(i, j, s) action, which combines the return from the receiver's `accept()` invocation with `getInputStream()` in Java. Finally, $jvmStatus(i, j)$ is set to connected. On the other hand, if a previous accept(j) action has not occurred at j, and thus j is not in $accepting$, the connection cannot be established. This is indicated to the sender with a createStreamError(i, j) action (a Java exception), and $jvmStatus$ is assigned to notAccepting and then closed.

Once the sender obtains an output stream for the connection, a message m may be written to the stream s via write(m, s). The message is sent only if $jvmStatus$ is connected or sConnected, indicating that the destination has not closed the connection. If these preconditions are satisfied, then $jvmBuffer$ is updated with the tuple $\langle m, s\rangle$; otherwise, $\langle m, s\rangle$ is added to $writeErrors$. A subsequent action writeError(m, s) will indicate to the sender that the write operation was unsuccessful, corresponding to an exception on the `write` invocation in Java.

State:
$jvmStatus : I \times I \rightarrow$ {closed, notAccepting, connecting, sConnected, connected}, initially all closed
$jvmBuffer$, subset of $M \times Streams$, initially $\emptyset$
$writeErrors$, subset of $M \times Streams$, initially $\emptyset$
$reading$, subset of $Streams$, initially $\emptyset$
$accepting$, subset of I, initially $\emptyset$
$jvmStream : I \times I \rightarrow Streams$, initially all undefined.
$jvmEmptying : I \times I \rightarrow Boolean$, initially all false

Transitions:

input write(m, s)
Effect:
  if $s = jvmStream(i, j) \wedge (jvmStatus(i, j) =$ connected$\vee$
      $jvmStatus(i, j) =$ sConnected) then
    $jvmBuffer \leftarrow jvmBuffer \cup \{\langle m, s\rangle\}$
  else $writeErrors \leftarrow writeErrors \cup \{\langle m, s\rangle\}$

input read(s)
Effect:
  $reading \leftarrow reading \cup \{s\}$

input accept(j)
Effect:
  $accepting \leftarrow accepting \cup \{j\}$

input createStream(i, j)
Effect:
  if $j \in accepting$ then
    $jvmStatus(i, j) \leftarrow$ connecting
    $accepting \leftarrow accepting - \{j\}$
  else $jvmStatus(i, j) \leftarrow$ notAccepting

input stopAccepting(j)
Effect:
  $accepting \leftarrow accepting - \{j\}$

input senderCloseStream(i, j)
Effect:
  if $(jvmStatus(i, j) =$ connected $\vee\ jvmStatus(i, j) =$ sConnected) then
    $jvmEmptying(i, j) \leftarrow$ true
  else $jvmStatus(i, j) \leftarrow$ closed

input receiverCloseStream(i, j)
Effect:
  $jvmBuffer \leftarrow jvmBuffer -$
      $\{\langle m, s\rangle \in jvmBuffer | s = jvmStream(i, j)\}$
  if $\neg jvmEmptying(i, j)$ then $jvmStatus(i, j) \leftarrow$ closed

Figure 3. State and input transitions of the many-to-many automaton JvmCh.

output respRead(m, s)
Precondition:
  $\langle m, s\rangle \in jvmBuffer$
  $s \in reading$
Effect:
  $jvmBuffer \leftarrow jvmBuffer - \{\langle m, s\rangle\}$
  $reading \leftarrow reading - \{s\}$

output writeError(m, s)
Precondition:
  $\langle m, s\rangle \in writeErrors$
Effect:
  $writeErrors \leftarrow writeErrors -$
      $\{\langle m, stream\rangle \in writeErrors | stream = s\}$

output readError(s)
Precondition:
  $s \in reading$
  $s = jvmStream(i, j) \wedge jvmStatus(i, j) =$ closed
Effect:
  $reading \leftarrow reading - \{s\}$

output respAccept(i, j, s)
Precondition:
  $jvmStatus(i, j) =$ sConnected
  $s = jvmStream(i, j)$
Effect:
  $jvmStatus(i, j) \leftarrow$ connected

output respCreateStream(i, j, s)
Precondition:
  $jvmStatus(i, j) =$ connecting
  $\forall\, i, j \in I,\ s \neq jvmStream(i, j)$
Effect:
  $jvmStatus(i, j) \leftarrow$ sConnected
  $jvmStream(i, j) \leftarrow s$

output createStreamError(i, j)
Precondition:
  $jvmStatus(i, j) =$ notAccepting
Effect:
  $jvmStatus(i, j) \leftarrow$ closed

internal senderClosingStream(i, j)
Precondition:
  $jvmEmptying(i, j)$
  $\forall \langle m, s\rangle \in jvmBuffer, s \neq jvmStream(i, j)$
Effect:
  $jvmStatus(i, j) \leftarrow$ closed
  $jvmEmptying(i, j) \leftarrow$ false

internal jvmLose(m)
Precondition:
  $\langle m, s\rangle \in jvmBuffer$
Effect:
  $jvmBuffer \leftarrow jvmBuffer - \{\langle m, s\rangle\}$

Figure 4. Output and internal transitions of the many-to-many automaton JvmCh.

A receiver initiates a read request from an assigned stream s with a read(s) action. If there is a message for s in $jvmBuffer$, the message is removed from $jvmBuffer$ and returned to the receiver with a respRead(m, s) action. However, if the sender has closed the connection and all previously sent messages have been received, the receiver is notified of this closure with a readError(s) action. The respRead(m, s) corresponds to a normal return on the `read` invocation in Java, and readError(s) signals an exception.

A stream may be closed by the sender or the receiver. If the receiver closes the connection with a receiverCloseStream(i, j) action, all messages in transit to the receiver are dropped, and the connection is closed. If the sender closes the connection with a senderCloseStream(i, j) action, all messages in transit are delivered to the receiver (or lost) before the connection is closed. This is indicated by setting $jvmEmptying$ to true.

Finally, the jvmLose(m) action drops a message m as a result of a network failure such as a broken or reset TCP connection [21].

4.3. Send Mediator Automaton

The state and transitions of the SENDMED automaton are given in Figure 5. The set $sendBuffer$ tracks messages sent by the application automaton, but not yet forwarded to JVMCH, where for each tuple $\langle m, i\rangle$ in the set, m is the message and i is an identifier of the destination. The variables $sendStatus$ and $sendStream$ keep track of the status and the stream associated with each connection established by the sender. Finally, $sendEmptying$ is an array of Boolean values used during the process of closing a connection to ensure that all previously sent messages are delivered.

The signature of SENDMED defines the interface between the application automaton and JVMCH. Before an application at node i can communicate with another at node j, it must create a connection with a senderOpen(i, j) action. This action sets $sendStatus$ to opening. Next, SENDMED attempts to create a stream for this connection, via the action createStream(i, j). There are two possible outcomes. If JVMCH is able to negotiate a connection between i and j and a stream is created, it indicates this with the action respCreateStream(i, j, s) that stores s in $sendStream$ and sets $sendStatus$ to connected. Alternatively, if a connection or stream cannot be created, JVMCH generates the action createStreamError(i, j). However, SENDMED will keep trying to establish the connection, and hence maintain $sendStatus$ as opening. The application may request that the connection be gracefully closed using action senderClose(i, j). However, the stream is not closed until all messages sent to j are transmitted to the channel, at which time a senderCloseStream(i, j) action is generated and $sendStatus$ is set to closed.

State:
$sendBuffer_i$, a subset of $M \times I$, initially $\emptyset$
$sendStatus_i : I \to \{$closed, opening, connecting, notAccepting, connected$\}$, initially all closed
$sendStream_i : I \to Streams$, initially all undefined
$sendEmptying_i : I \to Boolean$, initially all false

Transitions:
input send(m, i, j)
Effect:
  if $sendStatus_i(j) \neq$ closed $\wedge \neg sendEmptying_i(j)$ then
    $sendBuffer_i \leftarrow sendBuffer_i \cup \{\langle m, j\rangle\}$

input writeError(m, s)
Effect:
  if $s = sendStreams_i(j)$ then
    $sendBuffer_i \leftarrow sendBuffer_i -$
      $\{\langle m, receiver\rangle \in sendBuffer_i | receiver = i\}$
    $sendStatus_i(j) \leftarrow$ closed

input senderOpen(i, j)
Effect:
  $sendStatus_i(j) \leftarrow$ opening

input respCreateStream(i, j, s)
Effect:
  $sendStatus_i(j) \leftarrow$ connected
  $sendStream_i(j) \leftarrow s$

input createStreamError(i, j)
Effect:
  $sendStatus_i(j) \leftarrow$ opening

input senderClose(i, j)
Effect:
  $sendEmptying_i(j) \leftarrow$ true

output write(m, s)
Precondition:
  $\langle m, j\rangle \in sendBuffer_i$
  $sendStatus_i(j) =$ connected
  $s = sendStream_i(j)$
Effect:
  $sendBuffer_i \leftarrow sendBuffer_i - \{\langle m, j\rangle\}$

output createStream(i, j)
Precondition:
  $sendStatus_i(j) =$ opening
Effect:
  $sendStatus_i(j) \leftarrow$ connecting

output senderCloseStream(i, j)
Precondition:
  $sendEmptying_i(j)$
  $\forall \langle m, receiver\rangle \in sendBuffer_i, receiver \neq j$
Effect:
  $sendStatus_i(j) \leftarrow$ closed
  $sendEmptying_i(j) \leftarrow$ false

internal senderLose(m)
Precondition:
  $\langle m, j\rangle \in sendBuffer_i$
Effect:
  $sendBuffer_i \leftarrow sendBuffer_i - \{\langle m, j\rangle\}$

Figure 5. State and transitions of the one-to-many automaton SENDMED at node i.

Finally, the action senderLose(m) models the loss of a message due to an overflow of $sendBuffer$.

4.4. Receiver Mediator Automaton

The state and transitions of the automaton RECVMED are given in Figures 6 and 7. The state is composed of five variables: $acceptStatus$, representing the status of the receiver; $receiveBuffer$, the set of all messages received but not yet delivered to the application automaton; $receiveStatus$, representing the status of the connections with each node in I; $receiveStream$, the active streams – one for each node connected to the receiver automaton; and $receiveEmptying$, an array of Boolean flags used for processing sender-initiated closings of connections.

The application automaton at node j announces its readiness to accept connections from other nodes via the action receiverListening(j). As a result, RECVMED invokes an accept(j) action, notifying JVMCH of its willingness to wait for a connection to be established. When a sender node i makes a connection, RECVMED is handed the assigned stream for the incoming connection via the action respAccept(i, j, s). RECVMED then issues the action respReceiverListening(i, j), signaling the completion of the connection setup.

Once a stream s between the sender and receiver is created, RECVMED attempts to read a message via the action

State:
$acceptStatus_j \in$ {idle, accepting, waiting, stopping}, initially idle
$receiveBuffer_j$, a subset of $M \times I$, initially $\emptyset$
$receiveStatus_j : I \rightarrow$ {closed, connecting, connected, reading, rClosing}, initially all closed
$receiveStream_j : I \rightarrow Streams$, initially all undefined
$receiveEmptying_j : I \rightarrow Boolean$, initially all false

Transitions:
input respRead(m, s)
Effect:
  if $s = receiveStream_j(i) \wedge receiveStatus_j(i) =$ reading then
    $receiveBuffer_j \leftarrow receiveBuffer_j \cup \{\langle m, i\rangle\}$
    $receiveStatus_j(i) \leftarrow$ connected

input readError(s)
Effect:
  if $s = receiveStream_j(i)$ then
    if $receiveStatus_j(i) \neq$ closed then
      $receiveEmptying_j(i) \leftarrow$ true

input receiverListening(j)
Effect:
  $acceptStatus_j \leftarrow$ accepting

input respAccept(i, j, s)
Effect:
  $receiveStatus_j(i) \leftarrow$ connecting
  $reciveStream_j(i) \leftarrow s$
  if $acceptStatus_j =$ waiting then
    $acceptStatus_j \leftarrow$ accepting

input receiverStopListening(j)
Effect:
  if $acceptStatus_j \neq$ idle then
    $acceptStatus_j \leftarrow$ stopping

input receiverClose(i, j)
Effect:
  $receiveBuffer_j \leftarrow receiveBuffer_j -$
    $\{\langle m, sender\rangle \in receiveBuffer_j | sender = i\}$
  $receiveStatus_j(i) \leftarrow$ rClosing

Figure 6. State and input transitions of the many-to-one automaton RECVMED at node j.

output receive(m, i, j)
Precondition:
  $\langle m, i\rangle \in receiveBuffer_j$
Effect:
  $receiveBuffer_j \leftarrow receiveBuffer_j - \{\langle m, i\rangle\}$

output accept(j)
Precondition:
  $acceptStatus_j =$ accepting
Effect:
  $acceptStatus_j \leftarrow$ waiting

output read(s)
Precondition:
  $s = receiveStream_j(i)$
  $receiveStatus_j(i) =$ connected
Effect:
  $receiveStatus_j(i) \leftarrow$ reading

output respReceiverListening(i, j)
Precondition:
  $receiveStatus_j(i) =$ connecting
  $acceptStatus_j =$ accepting $\vee$ $acceptStatus_j =$ waiting
Effect:
  $receiveStatus_j(i) \leftarrow$ connected

output stopAccepting(j)
Precondition:
  $acceptStatus_j =$ stopping
Effect:
  $acceptStatus_j \leftarrow$ idle

output receiverCloseStream(i, j)
Precondition:
  $receiveStatus_j(i) =$ rClosing
  $\neg receiveEmptying_j(i)$
Effect:
  $receiveStatus_j(i) \leftarrow$ closed

internal senderClosing(i, j)
Precondition:
  $receiveEmptying_j(i)$
  $\forall \langle m, sender\rangle \in receiveBuffer_j, sender \neq i$
Effect:
  $receiveStatus_j(i) \leftarrow$ closed
  $receiveEmptying_j(i) \leftarrow$ false

internal receiverLose(m)
Precondition:
  $\langle m, i\rangle \in receiveBuffer_j$
Effect:
  $receiveBuffer_j \leftarrow receiveBuffer_j - \{\langle m, i\rangle\}$

Figure 7. Output & internal transitions of the many-to-one automaton RECVMED at node j.

read(s). If there is a message, JVMCH responds to the action read(s) with the action respRead(m, s) containing the message. However, if the sender has closed the connection and all previous messages from the sender have been delivered to RECVMED, JVMCH responds with the action readError(s). A message that is successfully received by RECVMED is delivered to the application automaton in the action receive(m, i, j).

The application automaton may stop listening for connections from other nodes at any time by indicating this with the action receiverStopListening(j). As a result, the action stopAccepting(j) follows, ensuring that no more connections will be accepted. Also, the application automaton may wish to close a connection with the action receiverClose(i, j). Once a connection is closed, all messages destined to the application automaton from that sender are purged from $receiveBuffer$.

The two remaining actions are internal. The action senderClosing(i, j) indicates that all messages sent prior to a sender-initiated closing of the connection have been delivered, lost, or purged. The action receiverLose(m) indicates that a message has been lost (modeling buffer overflow).

4.5. Proof of Correctness

Forward simulation [12, 15] is used to prove that COMPCH, comprised of JVMCH composed with the send and receive mediators at each node, implements ABSCH, the abstract asynchronous channel in Figure 2. A well-formedness condition on the behaviors of the application automata is required for this forward simulation, namely neither a sender nor a receiver can issue more than one request to close a connection. From our assumption that there is only a single socket connection between any two nodes, we also can conclude that a new connection from a sender to a receiver cannot be opened until the previous connection, if any, from that sender to that receiver is completely closed.

Theorem 4.1 *Any trace of* COMPCH *is a trace of* ABSCH.

The complete proof is contained in [7]. We begin by presenting a mapping from the states of COMPCH to the states of ABSCH. For example, consider the mapping for the set $messages$ in ABSCH. It is not difficult to see that a message in $messages$ must be in one of the sets $sendBuffer_i$, $jvmBuffer$, and $receiveBuffer_j$ in COMPCH. However, when the receiver initiates a closing procedure, all messages in $messages$ are purged, but some messages may remain in $sendBuffer_i$ and $jvmBuffer$ for a short period of time until they also are purged. Thus, $messages$ in ABSCH is mapped to the union of the following three sets in COMPCH:

$$
\begin{aligned}
messages \equiv\ & \{\langle m,i,j\rangle | \langle m,i\rangle \in receiveBuffer_j\} \cup \\
& \{\langle m,i,j\rangle | \langle m,s\rangle \in jvmBuffer \wedge s = jvmStream(i,j) \wedge \\
& \quad receiveStatus_j(i) \neq \mathsf{rClosing}\} \ \cup \\
& \{\langle m,i,j\rangle | \langle m,j\rangle \in sendBuffer_i \wedge receiveStatus_j(i) \neq \mathsf{rClosing} \wedge \\
& \quad \neg[sendStatus_i(j) = \mathsf{connected} \wedge jvmStatus(i,j) = \mathsf{closed} \ \wedge \\
& \quad receiveStatus_j(i) = \mathsf{closed}]\}
\end{aligned}
$$

As another example of the simulation relation mapping, the variable $emptying(i, j)$ in ABSCH becomes true when the sender desires to close the connection and remains true until all previously sent messages are delivered to the receiver. In COMPCH, the previously sent messages may appear in $sendBuffer_i$, $jvmBuffer$, and $receiveBuffer_j$. While these messages remain in $sendBuffer_i$, $sendEmptying_i(j) =$ true. Then, while messages remain in $jvmBuffer$, $jvmEmptying(i, j) =$ true. Finally, as soon as RECVMED$_j$ determines that the sender has closed the connection, it sets $receiveEmptying_j(i)$ to true; $receiveEmptying_j(i)$ remains true as long as messages remain in $receiveBuffer_j$. Therefore, we have:

$$
\begin{aligned}
emptying(i,j) \equiv\ & sendEmptying_i(j) \vee jvmEmptying(i,j) \vee \\
& receiveEmptying_j(i) \vee \ [sendStatus_i(j) = \mathsf{closed} \wedge \\
& jvmStatus(i,j) = \mathsf{closed} \wedge receiveStatus_j(i) \neq \mathsf{closed}]
\end{aligned}
$$

The proof consists of two parts. First, we show that every initial state of COMPCH maps to an initial state of ABSCH. Next, we show that for every reachable state cc of COMPCH, mapping to state ac of ABSCH, and for every transition π of COMPCH enabled in state cc and resulting in state cc', there is a (possibly empty) sequence of transitions α of ABSCH that results in state ac', where cc' maps to ac' and α has the same trace as π. The proof proceeds by case analysis on the transitions π of COMPCH.

4.6. Implementation

The mediator automata presented in Sections 4.3 and 4.4 have been manually implemented in Java [19]. Although the automata are non-deterministic, where all enabled actions in a given state have the same probability of being chosen for execution and where there are arbitrary delays between state transitions, our implementation restricts this non-determinism in order to ensure efficient execution. The functionality of the implementation has been tested in a LAN setting using three Windows machines (two XP and

one Vista). A driver program was used to perform various tests. We are currently integrating our implementation in the RAMBO service [13, 18] and the IOA Toolkit [3].

5. Discussion

The work presented in this paper is the first formal presentation of an abstract asynchronous communication channel with graceful comings and goings. Many algorithm implementations rely on such abstract channels without providing a proof of correctness for the composition of the source algorithm and the communication channel. Therefore, our solution can be used to claim that such implementations are, in fact, correct.

The importance of proving the correctness of the composite automaton (Theorem 4.1) cannot be overemphasized. It was only by going through the proof process that several subtle errors in the design of the component automata were discovered and corrected.

We intend to use this work in formally reasoning about the correctness of dynamic distributed data-sharing applications. We also plan to use our proposed solution in exploring automated code generation for dynamic networked applications. Future extensions to the model will include support for bidirectional communication over socket pairs, multiple connections between pairs of nodes, and timing considerations.

References

[1] B. Bernardo and R. Gorrieri. A theory of concurrent processes with nondeterminism, priorities, probabilities and time. *Theoretical Computer Science*, 202(1–2):1–54, 1998.

[2] F. Cristian and C. Fetzer. The timed asynchronous distributed system model. *IEEE Trans. Parallel Distrib. Syst.*, 10(6):642–657, 1999.

[3] S. Garland and N. Lynch. The IOA language and toolset: Support for designing, analyzing, and building distributed systems. Technical Report MIT/LCS/TR-762, LCS, MIT, August 1998.

[4] S. Garland, N. Lynch, and M. Vaziri. *IOA: A language for specifying, programming, and validating distributed systems*, 2001.

[5] C. Georgiou, N. Lynch, P. Mavrommatis, and J. A. Tauber. Automated implementation of complex distributed algorithms specified in the IOA language. In *Proc. of the 18th International Conference on Parallel and Distributed Computing Systems*, pages 128–134, 2005.

[6] C. Georgiou, P. Mavrommatis, and J. A. Tauber. Implementing asynchronous distributed systems using the IOA toolkit. Technical Report MIT-LCS-TR-966, CSAIL, MIT, 2004.

[7] C. Georgiou, P. Musial, A. Shvartsman, and E. Sonderegger. A formal treatment of an abstract channel implementation using java sockets. Technical Report BECAT/CSE-TR-06-10, University of Connecticut, 2006.

[8] P. Hatziprocopiou. Automated implementation of the Paxos algorithm using the IOA toolkit. Bachelor Thesis, Department of Computer Science, University of Cyprus, June, 2006.

[9] A. P. J. A. Bergstra and S. A. Smolka. *Handbook of Process Algebra*. North-Holland, 2001.

[10] F. Kroger. *Temporal Logic of Programs*. Springer-Verlag, 1987.

[11] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.

[12] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.

[13] N. Lynch and A. Shvartsman. RAMBO: A reconfigurable atomic memory service for dynamic networks. In *Proc. of 16th International Symposium on Distributed Computing*, pages 173–190, 2002.

[14] N. Lynch and M. Tuttle. An introduction to Input/Output Automata. *CWI-Quarterly*, 2(3):219–246, 1989.

[15] N. Lynch and F. Vaandragen. Forward and backward simulations, Part I: Untimed systems. *Information and Computation*, 121(2), 1995.

[16] Message Passing Interface Forum. MPI: A Message-Passing Interface standard. *The International Journal of Supercomputer Applications and High Performance Computing*, 8, 1994.

[17] Microsoft Corporation. System definition model overview white paper. Microsoft Windows Server System – SDM, April 2004.

[18] P. Musial and A. Shvartsman. Implementing a reconfigurable atomic memory service for dynamic networks. In *Proc. of 18'th International Parallel and Distributed Symposium — FTPDS WS*, page 208b, 2004.

[19] Peter M. Musial. Lossy channel implementation in java. `http://www.cse.uconn.edu/~piotr/channel.zip`.

[20] J. Postel. DARPA Internet program specification (internet standard stc-007). Internet RFC-793, September 1981.

[21] Sun Microsystems. Java^{TM} platform SE 6. `http://java.sun.com/javase/6/docs/api/`.

[22] J. Tauber. *Verifiable Compilation of I/O Automata without Global Synchronization*. PhD thesis, MIT, September 2004.

[23] VeroModo Inc. Tempo toolset. `http://www.veromodo.com`.