Specification and Refinement of a Message Router

H. Conrad Cunningham and Yinxiu Cai
Department of Computer and Information Science
University of Mississippi
University, Mississippi 38677 USA

Abstract

This paper considers a variant of the message router problem discussed during the Concurrency and Distribution sessions of IWSSD-6. First, it presents a high-level specification of the router as a reactive system expressed in the UNITY logic. Second, it refines the interface of the router using a new approach called the reactive envelope heuristic. Third, it decomposes the router into a grid of switches. In closing, the paper analyzes the specification and refinement techniques used in this study and proposes future research.

1 Introduction

A reactive program is a program that maintains an on-going interaction with its environment throughout execution (e.g., an operating system) [8]. Because of the complexity of the interactions of the concurrent components of the program and its environment, a reactive program is notoriously difficult to "get right." At first glance a problem may seem easy to comprehend, but a closer examination reveals many challenges. This paper considers one such problem, the specification and refinement of a message router [6].

First, consider a message router with M sender and N receiver channels. One sender is connected to each sender channel and one receiver to each receiver channel. From time to time a sender may send a message to one of the receivers via the router—perhaps a different receiver for each message. A sender presents one message at a time to the router. Likewise, the router presents one message at a time to a receiver. A message consists of two fields accessible to the router—the receiver channel address and the data. The router must not change the value of the data field but may modify the address field during routing of the message.

Second, consider the following refinement. A message now consists of a finite sequence of tokens. A sender presents one token at a time to the router. Likewise, the router presents one token at a time to a receiver. Tokens are of three types. A header token marks the beginning of a message and identifies the intended receiver's channel address. A data token carries data associated with the message. A trailer token marks the end of message. A message begins with a header token, continues with zero or more data tokens, and ends with a trailer token. Neither data nor trailer tokens carry any information that identifies the receiver of the message. The router must not change the value of the data. However, the router may modify the address field of the header token during routing of the message. The order of the tokens within a message must be preserved by the router. Hence, the tokens of multiple messages cannot be interleaved on the channels connected to the senders and receivers.

Third, consider the following refinement of the token-level router. The router consists of an $M \times N$ grid of switches. Each sender sends its messages to a different row of the grid and each receiver receives its messages from a different column. The tokens of a message travel along a row until the destination column is encountered then travel along the column toward the receiver.

The goal of this case study is to devise a specification for the message router as an open system. That is, it seeks a formulation that states the properties of the router in terms of its interactions with its environment. When the correct functioning of the router depends upon certain characteristics of the environment, the specification must identify these assumptions and state them precisely. A specification of this nature should be easy to compose with other specifications.

The router problem poses several challenges that this case study must address:

- devising a specification that is as "weak" as is practical (i.e., allowing the designer to choose among many possible implementation methods),
- stating the complex properties of the reactive system in a concise and modular way,
- identifying the subtle assumptions the router specification makes about its environment,

- refining the atomicity of the router's interface,
- refining the router internally into the grid-like composition of switching elements.

Before proceeding with the development of the specification and its refinements, we take a look at the specification model and notation.

2 Specification model and logic

This case study adopts Chandy and Misra's UNITY model [4] as the notation and logic for specification. A UNITY program is, in essence, a nondeterministic program in Dijkstra's Guarded Commands notation [7] with the form

initialize variables ;
do $g_0 \rightarrow a_0 \; [] \; g_1 \rightarrow a_1 \; [] \cdots [] \; g_{n-1} \rightarrow a_{n-1}$ **od**

where

- n is a finite constant,
- a_i (for $0 \leq i < n$) denotes an atomic, terminating, deterministic, multiple-assignment command that accesses a fixed set of variables,
- the execution is *fair* in the sense that, if a guard g_i holds at some point in an infinite computation, then there exists a later point at which either a_i is executed or g_i no longer holds.

The union of two UNITY programs consists of a program in which the initialization is formed by the union of the two initializations and the **do** loop is formed by the union of the two sets of guarded commands.

UNITY's operational model represents a program as the set of all maximal execution sequences of the corresponding **do** program. A maximal execution sequence records the sequence of states corresponding to a possible execution of the program. The transition from one state to the next corresponds to the execution of an atomic action. These execution sequences are either infinite or end in a state in which all guards of the **do** program are false. This operational model allows program properties to be stated in terms of temporal logic [8].

This case study uses UNITY's simple subset of temporal logic [4] to specify the properties that a program must be constructed to satisfy. For the purposes here, we consider the logical relations **initially**, **stable**, **invariant** (abbreviated as **inv**), and $\longmapsto$ (read "leads-to"). Informally, for arbitrary predicates p and q on program states:

- **initially** p means that p must hold for the initial state of every execution sequence. (The predicate p must hold between the initialization and the beginning of the **do** loop.)
- **stable** p means that, for any execution sequence, if p holds for some state, then p must continue to hold for all succeeding states of the sequence. (A stable predicate is preserved by the actions in the body of the **do** loop.)
- **invariant** p means that p must hold for all states of all execution sequences. That is, both **initially** p and **stable** p hold. (UNITY invariants are loop invariants of the **do** loop.)
- $p \longmapsto q$ means that, if p holds for any state of any execution sequence, q must also hold within a finite number of steps in the execution sequence.

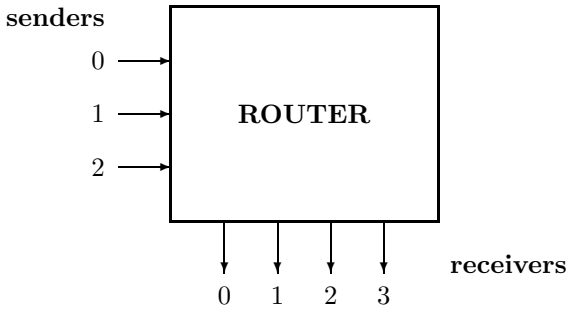
The annotation *prop* **in** P denotes that *prop* is a property of program P considered in isolation and $P \; [] \; Q$ denotes the union of programs P and Q. A UNITY conditional property specified a property of a reactive program that is dependent upon properties of the program's environment. The specification

Hypothesis: *Property_List_1*
Conclusion: *Property_List_2*

means that the program must satisfy *Property_List_2* whenever *Property_List_1* holds.

3 Message-level router specification

The message router must transmit each message appearing on any of the M sender channels to the message's destination on one of the N receiver channels. For example, we can picture a router with three sender channels and four receiver channels as follows:



In the following, program R represents the router program (the inside of the box), program E represents an arbitrary environment for the router (the outside of the box), and shared variables represent the interface

between R and E (the boundary of the box). The actual senders and receivers of messages are part of E and are, hence, not explicitly specified.

The program variables $send.i$, for $0 \leq i < M$, and $recv.j$, for $0 \leq j < N$, denote the channels on which sender i may send a message and receiver j may receive a message, respectively. Each $send.i$ and $recv.j$ holds a finite sequence of messages, with each message having the format $(s, r, (a, v))$ such that:

- s and r are auxiliary fields denoting the sender and receiver, respectively, for the message—an implementation cannot access these fields.
- (a, v) is the actual message available to the implementation—a is the intended receiver and v is the data value carried by the message.

The $send.i$ and $recv.j$ channels are the only program variables shared between programs R and E. Program E may append one message at a time to the tail of a $send.i$ sequence and program R may remove one message at a time from the head. Similarly, program R may append one message at a time to the tail of a $recv.j$ sequence and program E may remove one message at a time from the head. Program R can initialize all sender and receiver channels. No other references or updates are allowed.

As the router executes, it removes messages from the $send.i$ channels and eventually appends them to $recv.j$ channels. This message transmission must done in a safe way, without misdelivering, corrupting, duplicating, losing, forging, or reordering the messages. These requirements constrain the relationship between a message and other messages that have been sent or received on some channel. Similarly, the router's progress at some point can be measured in terms of the messages sent and delivered up to that point.

Because of the constraints on access to the $send.i$ and $recv.j$ variables, it is convenient to express the specifications of the router in terms of the histories of the changes to these variables. Thus, we introduce the auxiliary variables $\overline{send.i}$ and $\overline{recv.j}$. Initially, $\overline{send.i} = send.i$ and $\overline{recv.j} = recv.j$. Whenever a message is appended to a $send.i$ or $recv.j$ sequence the same message is appended to the corresponding $\overline{send.i}$ or $\overline{recv.j}$ sequence. Removal of messages from the head of $send.i$ or $recv.j$ does not change the associated $\overline{send.i}$ or $\overline{recv.j}$. Hence, the values of $\overline{send.i}$ and $\overline{recv.j}$ never decrease in length.

In addition to the access constraints, there are four classes of properties in the router specification: initializations, delivery invariants, prefix invariants, and eventual delivery properties. Each refinement has one or more properties from each of these classes.

Initializations specify the initial values for the channels and other variables.

Delivery invariants state the structural properties of the sequences of delivered messages. In general, any message delivered on a receiver channel j must have originated on some valid sender channel and must be destined for receiver j.

Prefix invariants state the structural relationship between the sender and receiver channels. In general, any two messages with different data fields sent from sender i to receiver j must be delivered in the same relative order as sent. Messages must not be lost, duplicated, or spontaneously generated by the router. That is, the sequence of messages delivered to receiver j from sender i must be a prefix of the sequence of messages sent by sender i to receiver j.

Eventual delivery properties state that any message sent will eventually be delivered to its destination. That is, progress eventually occurs.

Now we can turn our attention to the high-level specification itself. The Message Router Initialization property (shown below) states that program R must initialize all sender and receiver channels to be empty, denoted here by ϵ. We use the convention that all free variables occurring in the statement of a property are universally quantified over all possible values. For convenience, we restrict the quantifications of the sender channels i and the receiver channels j to be over the ranges $0 \leq i < M$ and $0 \leq j < N$, respectively. The scope of this implicit quantification is the entire property being stated, including both the hypothesis and conclusion of conditional properties.

Property 1 (Message Router Initialization)
initially $send.i = \epsilon \wedge recv.j = \epsilon$ **in** R

Clearly, if the router is to deliver messages correctly, it must make some assumptions about the activities on its sender channels. If the assumptions are violated, then the router may not be able to perform its task correctly. For this case study, we opt for simplicity and assume that only complete messages with valid data fields and destination addresses may be sent on any sender channel. The auxiliary sender and receiver fields of the messages must also be set appropriately, of course. We introduce the predicate $send\_ok.i$ to

formalize this condition; it holds when the sequence of messages $\overline{send.i}$ is a valid sequence of messages for sender i to send. This predicate can be defined recursively as a function on the message sequences.

The remaining properties of the message-level router depend upon $send\_ok.i$ holding for all sender channels. The UNITY logic's conditional properties provide a formal means for making this dependence explicit. In the hypothesis we state the assumed properties of the environment E and in the conclusion the properties of the overall system $R \,[]\, E$ that the router must implement. For compactness in presentation, we assume that the following properties are all conditionals with the given conclusions and the hypothesis:

$$\langle \forall\, k : 0 \leq k < M :: \textbf{ stable } send\_ok.k \textbf{ in } E \rangle$$

We also implicitly include the access constraints on the $send.i$ and $recv.j$ variables in the hypothesis.

The Message Delivery Invariant requires that, if every sender sends a valid sequence of messages, then the router must deliver a valid sequence of messages to every receiver. This is a safety property; it does not require the router to deliver any messages, but it does require that any messages delivered be delivered correctly. We formalize this property with the $recv\_ok.j$ predicate. Like $send\_ok.i$, predicate $recv\_ok.j$ holds when the sequence of messages $\overline{recv.j}$ is a valid sequence of messages for receiver j to receive.

Property 2 (Message Delivery Invariant)
inv $recv\_ok.j$ **in** $R \,[]\, E$

Another safety property is the Message Prefix Invariant. This property requires that, if every sender sends a valid sequence of messages, then the sequence of messages that each receiver j receives from each sender i corresponds exactly to the sequence that was sent by i to j except that a few messages may be in transit. This relationship can be expressed by the sequence prefix relation $\sqsubseteq$. In stating this property, we also introduce the sequence filtering function $route.i.j.x$ defined on sequences of messages x that have the format $(s, r, (a, v))$. Function $route.i.j.x$ returns the subsequence of x containing all messages whose source field s and destination field r match i and j, respectively. Since the address field a of a message can change as it moves from a $send.i$ to a $recv.j$, we also define $route$ so that it suppresses differences among values for the field a.

Property 3 (Message Prefix Invariant)
inv $route.i.j.\overline{recv.j} \sqsubseteq route.i.j.\overline{send.i}$ **in** $R \,[]\, E$

Finally, we must ensure that the router eventually does the work desired. The Eventual Message Delivery property requires that, if every sender sends a valid sequence of messages, all messages sent by sender i and intended for receiver j will be delivered to receiver j eventually.

Property 4 (Eventual Message Delivery)
$route.i.j.\overline{send.i} = x \;\longmapsto\; route.i.j.\overline{recv.j} = x$
in $R \,[]\, E$

To prove that a program or refined specification satisfies the specification for R, we may assume that the environmental access constraints and hypothesis hold. Then, given the program or refined specification's properties, we must then prove that the Conclusions of R's properties and R's variable access constraints are satisfied.

4 Token-level router specification

The task is now to specify the token-level router so that it satisfies the message-level specification. This presents an interesting challenge. In the message-level specification a message is a single entity that can be written to (or read from) the interface variables as one atomic operation. However, in the token-level refinement a message becomes a sequence of entities, each of which must be sent or received as a separate operation. The refinement of the atomicity of the interface requires that actions in both the router and in its environment be broken up into sequences of actions. Because we seek to specify the router as a reactive system, we scrupulously avoided specifying any progress properties for the environment at the message level. But now we must pay more attention to the part of the environment that interacts with the router.

To specify the token-level refinement, we apply the *reactive envelope* heuristic. We specify a token-level router TR with the functionality given in the problem description and enclose TR in another program, a reactive envelope, that makes TR "look like" message-level program R to the message-level environment E. The reactive envelope captures the changed environmental assumptions required by the token-level router and provides a framework for proving that the token-level router satisfies the message-level specification. For convenience, we divide the reactive envelope into a sending envelope SE and a receiving envelope RE.

Thus the task becomes refinement of the message-level program R into the union $SE \,[]\, TR \,[]\, RE$. We assign the $send.i$ variables to the interface between E

and SE and the $recv.j$ variables to the interface between RE and E; reactive envelope programs SE and RE retain message-level program R's access to these variables. We postulate new variables $tsend.i$ to represent the token send channels and assign them to the SE-to-TR interface with access constraints like those upon the $send.i$. Similarly, we postulate new variables $trecv.j$ to represent the token receive channels and assign them to the TR-to-RE interface with access constraints like those upon the $recv.j$. Program TR can initialize all $\overline{tsend.i}$ and $\overline{trecv.j}$ channels. As at the message level, $\overline{tsend.i}$ and $\overline{trecv.j}$ represent the history sequences for $tsend.i$ and $trecv.j$ respectively.

The new variables $tsend.i$ and $trecv.j$ hold sequences of tokens of the format $(s, r, (d, v))$ where

- s and r are auxiliary fields denoting the sender and receiver channels, respectively, for the message of which the token is a part,
- (d, v) is the actual token available to the implementation—d is the designator for the token (one of the constants HDR, DAT, or TRL) and v is the value carried by the token.

The data component v of a header token encodes the intended receiver for the message; the data component of a trailer token has an arbitrary value.

First, let us examine the requirements for the token-level router program TR. As at the message level, we give an initialization, a delivery invariant, a prefix invariant, and an eventual delivery property. The Token Router Initialization is straightforward.

Property 5 (Token Router Initialization)
initially $tsend.i = \epsilon \wedge trecv.j = \epsilon$ **in** TR

Although the specification of TR must be stated in terms of sequences that have different characteristics, the delivery and prefix invariants for TR are almost identical to those of the message-level router R. The different characteristics of the sequences can be hidden in the definitions of the operators on those sequences.

The message-level specification assumes that only complete messages with valid destination addresses and data fields will be sent on any sender channel; at the token level a "complete message" becomes a sequence of tokens. Thus we define predicate $tsend\_ok.i$ to hold when the sequence of tokens $\overline{tsend.i}$ is well-formed: all tokens originate from sender i, tokens in the sequence appear in the prescribed header-data-trailer pattern, all header tokens give valid destination addresses, and the auxiliary fields are set properly for that channel and message destination. The message at the tail may not yet be complete. Similarly, we define predicate $trecv\_ok.j$ to hold when $\overline{trecv.j}$ is well-formed: all tokens are destined for receiver j, tokens in the sequence appear in the prescribed header-data-trailer pattern, and the auxiliary fields are set properly. These predicates can be defined recursively as functions on the token sequences.

Again for compactness in presentation, we assume that the following properties of TR are all conditionals with the given conclusions and the hypothesis:

$$\langle \forall\, k : 0 \le k < M :: \ \textbf{stable}\ tsend\_ok.k\ \textbf{in}\ F \rangle$$

F represents the environment of program TR. In this system $F = SE \,[]\, RE \,[]\, E$.

The Token Delivery Invariant requires that, if every sender sends a valid sequence of tokens, then the router must deliver a valid sequence of tokens to every receiver. Here valid includes, of course, that the sequences of tokens delivered corresponds to a structurally correct sequence of messages. That is, there is no interleaving of the tokens of the messages delivered.

Property 6 (Token Delivery Invariant)
inv $trecv\_ok.j$ **in** $TR \,[]\, F$

The Token Prefix Invariant requires that, if every sender sends a valid sequence of tokens, then the sequence of tokens that each receiver j receives from each sender i corresponds exactly to the sequence that was sent from i to j except that a few tokens may be in transit. To aid in specifying this property, we define the sequence filtering function $route'.i.j.x$. Function $route'.i.j.x$ returns the subsequence of x containing all tokens whose source field s and destination field r match i and j, respectively. Like $route$, $route'$ suppresses differences among the values of the address fields of the header tokens.

Property 7 (Token Prefix Invariant)
inv $route'.i.j.\overline{trecv.j} \sqsubseteq route'.i.j.\overline{tsend.i}$ **in** $TR \,[]\, F$

In expressing TR's eventual delivery property, we must again consider the consequences of the atomicity refinement. In the message-level router, a message is a single entity sent or received as an atomic action. Neither is the case at the token level. The $tsend\_ok.i$ hypothesis of the above safety properties ensures that the message is broken up into tokens correctly, but getting the progress properties of the token router correct may require, in addition, that we constrain how the send actions are broken up.

Suppose that a message's header token is sent by the environment but the corresponding trailier token

is never sent—either there is an infinite sequence of data tokens or there is an infinite pause in sending tokens. What consequence does this have for TR? Since TR cannot take back tokens once they are delivered, clearly it must not deliver any of the tokens of a message that will never be completed. The delivery of any of the tokens would cause the receiver channel to be blocked forever. Because the tokens of messages cannot be interleaved, it would not be possible to deliver messages to that receiver from any other sender.

To prevent such a blockage from occurring, we either need to constrain the behavior of TR or of its environment. Two approaches seem feasible:

- have the token router wait until a message's trailer has been accepted before it starts to deliver any of the message's tokens,
- assume that the environment will not attempt to send incomplete messages.

Which approach is better? Both approaches seem to lead to a refinement of the message-level specification. Both approaches require the assumption that all messages consist of a finite number of tokens. The first approach, however, requires that the token router buffer tokens until an entire message has been sent. Thus the router must have a buffer for each sender channel that is long enough to handle the longest message to be sent on that channel. In a realistic implementation this means we would be required to place an explicit bound on the size of messages. Although the second approach does not seem to require as much internal buffering, it does require that the senders (part of the environment) be constrained from sending any incomplete messages—the send of a header will eventually be followed by the send of a trailer. Because it seems to lead to a simple design that can handle messages of unbounded length, we choose the second approach. The reactive envelope programs must then capture what this decision means for the environment.

Accordingly, we make the Eventual Token Delivery property conditional upon a second environmental hypothesis. We state this hypothesis in terms of the function *partial* on sequences like $tsend.i$ and $trecv.j$. Function $partial.x$ returns the sequence of tokens corresponding to the incomplete message, if any, at the tail of sequence x. The additional hypothesis is

$$\langle \forall\, k, l : 0 \le k < M \wedge 0 \le l < N :: \\ partial.(route'.k.l.\overline{tsend.k}) \neq \epsilon \;\longmapsto \\ partial.(route'.k.l.\overline{tsend.k}) = \epsilon \textbf{ in } TR \,[]\, F \,\rangle$$

The Eventual Token Delivery property requires that, if every sender sends a sequence of tokens corresponding to a valid sequence of messages, all tokens sent by sender i and intended for receiver j will be delivered to receiver j eventually.

Property 8 (Eventual Token Delivery)

$$route'.i.j.\overline{tsend.i} = x \;\longmapsto \\ route'.i.j.\overline{trecv.j} = x \textbf{ in } TR \,[]\, F$$

With the specification of TR complete, we turn our attention to the reactive envelope programs SE and RE. The purpose of the reactive envelope is to make TR "behave like" the message-level router program R in interactions with its environment E and vice versa. The reactive envelope should allow us to prove that $SE \,[]\, TR \,[]\, RE$ satisfies the specification for R.

The sending envelope program SE accepts valid sequences of messages from E and delivers the corresponding sequences of message tokens to TR. This is straightforward to state given a function *unpack* with an appropriate recursive definition. Function $unpack.x$ must map each element of message sequence x to a token sequence beginning with a header token to carry the destination address, followed by a finite number of data tokens to carry the message's value, and ending with a trailer token. We assume that every message's value can be represented as a finite sequence of data tokens.

The specification for SE has one property from each of the four classes identified above. Again for compactness in presentation, we assume that the following properties (except initialization) are all conditionals with the given conclusions and the hypothesis:

stable $send\_ok.i$ **in** G, **initially** $tsend.i = \epsilon$ **in** G

Let G represent the environment of program SE. In this system $G = TR \,[]\, RE \,[]\, E$.

Property 9 (Sending Envelope Initialization)

initially $send.i = \epsilon$ **in** SE

Property 10 (Message Sending Invariant)

inv $tsend\_ok.i$ **in** $SE \,[]\, G$

Property 11 (Message Sending Prefix Inv)

inv $\overline{tsend.i} \;\sqsubseteq\; unpack.\overline{send.i}$ **in** $SE \,[]\, G$

Property 12 (Eventual Message Sending)

$$\overline{send.i} = x \;\longmapsto\; \overline{tsend.i} = unpack.x \textbf{ in } SE \,[]\, G$$

The specification for program RE is symmetrical to that of SE. We assume that the following properties (except initialization) are all conditionals with the given conclusions and the hypothesis:

inv *trecv_ok.j* **in** *H*

Here we let H represent the environment of program *RE*. That is, $H = SE \,[]\, TR \,[]\, E$.

Property 13 (Receiving Envelope Init)
initially $recv.j = \epsilon$ **in** *RE*

Property 14 (Message Reception Invariant)
inv *recv_ok.j* **in** $RE \,[]\, H$

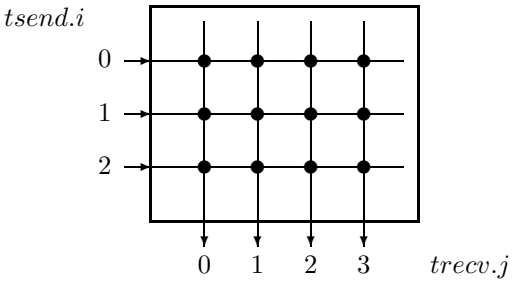
Property 15 (Message Reception Prefix Inv)
inv $\overline{unpack.recv.j} \sqsubseteq \overline{trecv.j}$ **in** $RE \,[]\, H$

Property 16 (Eventual Message Reception)
$\overline{trecv.j} = unpack.x \longmapsto \overline{recv.j} = x$ **in** $RE \,[]\, H$

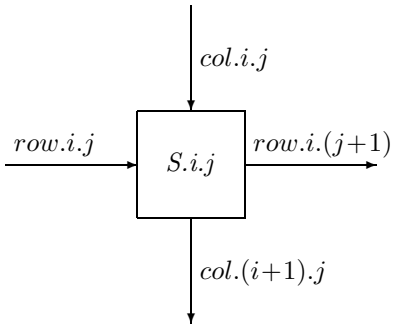
The proof of refinement requires proof that the union $SE \,[]\, TR \,[]\, RE$ satisfies the specification for R. That is, if R's environmental hypotheses and the specifications for *SE*, *TR*, and *RE* hold, then one must prove that the Conclusions of R's properties hold. This proof may use the conditional properties of the token-level specification as inference rules. Because the proofs are lengthy, they are not presented here.

5 Grid-level router specification

The task now is to refine the token-level router program *TR* into a grid of switch programs with M rows and N columns. We picture the senders, receivers, switches, and channels as being laid out as shown in the grid below. Switch *S.i.j* (with $0 \leq i < M$ and $0 \leq j < N$) is located at the intersection of the row from sender i and the column to receiver j.



We postulate new two-dimensional arrays of program variables *row* and *col* to represent the channels connecting the switches along the rows and columns, respectively. These variables hold sequences of tokens having the same format as the *tsend.i* and $\overline{trecv.j}$ sequences. As above, the auxiliary variables $\overline{row.i.j}$ and $\overline{col.i.j}$ represent the history sequences for *row.i.j* and *col.i.j*, respectively. To construct the grid we assign the variables *row.i.j*, $row.i.(j+1)$, *col.i.j*, and $col.(i+1).j$ to the interface of switch program *S.i.j* as shown in the diagram below.



To connect the grid with its environment, we assume that the following Grid Refinement Invariant holds for any i and j such that $0 \leq i < M$ and $0 \leq j < N$. This refinement invariant relates the channel variables along the left side and bottom of the grid with variables in the token-level specification.

Property 17 (Grid Refinement Invariant)
inv $row.i.0 = tsend.i \wedge col.M.j = trecv.j$

In the token-level specification, program *TR* initializes both the sender and receiver channels. Thus the grid router must initialize those channels as well as the other channels of the grid. We require that each switch program *S.i.j* initialize its output channels $row.i.(j+1)$ and $col.(i+1).j$ as specified by the Switch Initialization property below. This takes care of the receiver channels and the unused channels along the right boundary. To initialize the sender channels and the unused channels along the top boundary, we introduce the Grid Router Initialization program *Init* that does nothing but initialize those channels.

Property 18 (Grid Router Initialization)
initially $row.i.0 = \epsilon \wedge col.0.j = \epsilon$ **in** *Init*

Property 19 (Switch Initialization)
initially $row.i.(j+1) = \epsilon \wedge col.(i+1).j = \epsilon$ **in** *S.i.j*

Like the interface variables in the message- and token-level routers, we constrain access to the *row* and *col* variables. Switch *S.i.j* may remove one token at time from the head of input sequence *row.i.j* or *col.i.j* and append one token at a time to the tail of output sequence $row.i.(j+1)$ or $col.(i+1).j$. In accordance with the Grid Refinement Invariant and the access constraints at the token level, program *SE* may append elements to the $row.i.0$ variables and program *RE* may remove the head elements from the $col.M.j$ variables. No other accesses or updates are allowed.

Before proceeding with a specification of the grid-level router's properties, we must first consider how

the router moves tokens through the grid toward their designated receiver channels. The informal description of the grid requires that tokens of a message travel along a row until the destination column is encountered and then travel along the column toward the receiver. The only destination information available to the router is the address field of the message's header token. Upon entry into the grid (on a $tsend.i$), this field specifies the destination for the message (i.e., some receiver j). The router must deliver the message to its destination, but it is allowed to modify the address field as the header token moves through the grid.

Thus there must be a method for switch program $S.i.j$ to determine a message's destination given the address field of the header token. Since we do not want to constrain the switch's designers unnecessarily by prematurely specifying details, we do not yet wish to commit to a specific method. However, we do need to characterize a class of methods that are acceptable. First, the address calculation must invariantly yield the actual destination of the message. Second, it should be possible to compute the destination from the header token's address value and its position within the grid (i.e., which channel it is in). In particular, the address calculation for a message must not be sensitive to the state of any switch or of any other message. (Perhaps the second requirement is not absolutely necessary, but it is a constraint that seems to lead to simple methods.)

Therefore, we postulate the existence of a function $\boldsymbol{addr}$ such that $\boldsymbol{addr}.p.i.j.v$ maps the header token's current location within the grid (denoted by $p \in \{ROW, COL\}$, $0 \leq i < M$, and $0 \leq j < N$) and the address field of the header token (denoted by v) into the destination for the message. The actual definition depends upon the routing strategy chosen.

Note: There are at least two reasonable definitions for function $\boldsymbol{addr}$. With the definition

$$\boldsymbol{addr}.p.i.j.v = v$$

the value of the address field of the header token of a message does not change within the router. To determine how to route a message, switch $S.i.j$ simply compares the address field in the header against its own column number j. The switch must "know" in which column it is located. With the definition

$$\boldsymbol{addr}.p.i.j.v = v + j$$

the value of the address field of the header token of a message is decremented by 1 each time a switch forwards it along a row. Switch $S.i.j$ can compare the address field of a header token against 0 to determine how to route a message. The switch does not need to "know" its column number.

As in the higher-level specifications, we assume that no invalid tokens will arrive at a switch. But here we must be concerned with two input channels, $row.i.j$ and $col.i.j$. For a switch $S.i.j$, the history sequence $\overline{row.i.j}$ must be a valid sequence of message tokens from sender i to either receiver j or some receiver to the right of j in the grid. Similarly, the history sequence $\overline{col.i.j}$ must be a valid sequence of message tokens from a sender above row i in the grid to receiver j. Also, for all header tokens $(s, r, (HDR, v))$ in $row.i.j$ and $col.i.j$, $\boldsymbol{addr}.ROW.i.j.v = r$ and $\boldsymbol{addr}.COL.i.j.v = r$, respectively. Thus we define predicates $row\_ok.i.j$ and $col\_ok.i.j$ to hold when the sequences of message tokens $\overline{row.i.j}$ and $\overline{col.i.j}$, respectively, are sequences with these characteristics.

Switch $S.i.j$ has two output channels, $row.i.(j+1)$ and $col.(i+1).j$. Thus its specification must include a delivery invariant for each, the Row Invariant and Column Invariant below. These properties state that, if the switch receives valid input sequences, then it is required to generate valid output sequences. Because token sequences must be merged from the row and column inputs onto the column output, the Column Invariant must be conditional upon receiving valid input on both input channels.

Assume that the following properties are all conditionals with the given conclusions and the hypothesis:

inv $row\_ok.i.j$ **in** $D.i.j$, **inv** $col\_ok.i.j$ **in** $D.i.j$

Let $D.i.j$ represent the environment of program $S.i.j$. In this system, $D.i.j = SE \,[]\, Init \,[]\, \langle []k, l : 0 \leq k < M \wedge 0 \leq l < N \wedge (k,l) \neq (i,j) :: S.k.l \rangle$. Note that i and j are parameters of program $S.i.j$ rather than free variables.

Property 20 (Row Invariant)
inv $row\_ok.i.(j+1)$ **in** $S.i.j \,[]\, D.i.j$

Property 21 (Column Invariant)
inv $col\_ok.(i+1).j$ **in** $S.i.j \,[]\, D.i.j$

Tokens arriving at switch $S.i.j$ on channel $col.i.j$ must be forwarded downward on $col.(i+1).j$. However, tokens arriving at switch $S.i.j$ on channel $row.i.j$ must be forwarded to the right on channel $row.i.(j+1)$ or downward on channel $col.(i+1).j$ depending upon their destinations. Thus we need a prefix invariant for each of the three possible token movements through the switch. Together, the following three invariants specify how a sequence of message tokens arriving from a sender p and destined for a receiver q may flow

through the switch—without tokens being corrupted, duplicated, lost, forged, or reordered. Below p is universally quantified over all rows and q is universally quantified over all columns.

Property 22 (Row Prefix Invariant)
inv $j < q \Rightarrow route'.i.q.\overline{row.i.(j+1)} \sqsubseteq route'.i.q.\overline{row.i.j}$ **in** $S.i.j \,[]\, D.i.j$

Property 23 (Direction Switch Prefix Inv)
inv $route'.i.j.\overline{col.(i+1).j} \sqsubseteq route'.i.j.\overline{row.i.j}$ **in** $S.i.j \,[]\, D.i.j$

Property 24 (Column Prefix Invariant)
inv $p < i \Rightarrow route'.p.j.\overline{col.(i+1).j} \sqsubseteq route'.p.j.\overline{col.i.j}$ **in** $S.i.j \,[]\, D.i.j$

As with the prefix invariants, the grid-level router specification requires three eventual delivery properties. Together the following three properties specify that a token arriving from a sender p and destined for a receiver q must eventually move through the switch toward its destination. As with the Eventual Token Delivery property, we must prevent the output channels from being blocked by the attempted delivery of an incomplete message. Hence, we make these eventual delivery properties conditional upon all input messages eventually being complete. The additional hypotheses are:

$\langle \forall\, l : j \le l < N :: partial.(route'.i.l.\overline{row.i.j}) \neq \epsilon \longmapsto partial.(route'.i.l.\overline{row.i.j}) = \epsilon$ **in** $S.i.j \,[]\, D.i.j \,\rangle$

$\langle \forall\, k : 0 \le k < i :: partial.(route'.k.j.\overline{col.i.j}) \neq \epsilon \longmapsto partial.(route'.k.j.\overline{col.i.j}) = \epsilon$ **in** $S.i.j \,[]\, D.i.j \,\rangle$

Property 25 (Row Forwarding)
$j < q \wedge route'.i.q.\overline{row.i.j} = x \longmapsto route'.i.q.\overline{row.i.(j+1)} = x$ **in** $S.i.j \,[]\, D.i.j$

Property 26 (Direction Switching)
$route'.i.j.\overline{row.i.j} = x \longmapsto route'.i.j.\overline{col.(i+1).j} = x$ **in** $S.i.j \,[]\, D.i.j$

Property 27 (Column Forwarding)
$p < i \wedge route'.p.j.\overline{col.i.j} = x \longmapsto route'.p.j.\overline{col.(i+1).j} = x$ **in** $S.i.j \,[]\, D.i.j$

Unlike the choices for the Message Sending Prefix Invariant and Eventual Message Sending properties, here we choose weak formalizations of the Row Prefix and Row Forwarding properties. The properties allow $S.i.j$ to change the relative order of messages destined for different receivers from sender i. Likewise, we choose weak formalizations of the Column Prefix and Column Forwarding properties. The properties allow $S.i.j$ to change the relative order of messages destined for receiver j from different senders.

The proof of refinement requires proof that the composite program $GR = Init \,[]\, \langle []\, k, l : 0 \le k < M \wedge 0 \le l < N :: S.k.l \,\rangle$ satisfies the specification for TR. That is, we assume that TR's environmental hypotheses and the specifications for $Init$ and $S.i.j$ all hold for the grid and then prove that the Conclusions of TR's properties hold. The proofs of these properties will rely upon inductive proofs over the rows and columns of the grid. Since the proofs are lengthy, they are not presented here.

6 Discussion

This case study began with an intriguing problem, a message router, and devised a high-level specification and a series of refinements using the UNITY model [4]. Others who have studied this problem have approached it as a closed system [5]. That is, they chose to specify both the router and its environment as active agents. This case study took a different approach. It developed the router specification as an open system. That is, it characterized the router in terms of its interactions with an arbitrary environment. The resulting specification can be readily composed with any environment that satisfies a few simple constraints.

The decision to specify the router as an open system meant that we had to give careful attention to its interface to the environment. By constraining this interface, we simplified both the statement of the specification and subsequent proofs of properties. The nature of the router enabled us to represent its interface by a group of shared variables, each of which can only be written by the environment and read by the router, or vice versa. This allowed us to decouple the internal operation of the router from its environment. (Future research should reexamine the interface specification in light of recent work on monotonic variables and program composition [2, 3].)

Another critical early decision was the choice of a method for stating the properties. Given the constraints placed upon the interface to the router, the use of auxiliary history sequences seemed appropriate. That is, we chose to express the specification in terms of the sequences of values written to the shared variables from the beginning of the computation up to the "current" point. The history sequences enabled us to constrain the allowed behaviors of the router by stating invariant properties of the individual sequences

or of the relationship among two or more sequences. The history sequences also enabled us to express the required behaviors of the router in a straightforward way—as increases in the lengths of the appropriate sequences. By defining special predicates and operations on the history sequences similar to those used in functional programming [1], we were also able to state the router's properties concisely.

In addition to conciseness, the specifications needed to be consistent, both within and between levels. Analysis of the problem revealed four classes of properties that capture the essence of the router: initializations, delivery invariants, prefix invariants, and eventual delivery properties. These classes guided the derivation of the specific properties for the programs within each level. They helped make the specification modular and consistent. The grid-level refinement highlighted the need for an additional class, refinement invariants. These invariants relate data entities in a refined specification to entities in the higher level specification. Other classes can be added as needed. For example, if we need bounds on the sizes of the channel buffers, then we can add a new class of invariants to constrain the lengths of the values of the channel variables (e.g., *tsend.i*).

Clearly, the message-level and token-level routers have different interfaces. In the former, a sender transmits a message as a single atomic operation. In the latter, a sender transmits a message as a sequence of atomic operations, one for each token of the message. To handle this situation, we proposed the reactive envelope heuristic. Applying this heuristic, we refined the message-level router into a token-level router that is encapsulated within a reactive envelope. The reactive envelope makes the interfaces of the token-level router and message-level environment conform to each other. The reactive envelope, in essence, captures the changes that need to be made to the message-level environment for it to interact with the token-level router. The reactive envelope heuristic was an effective technique for this case study. However, future research should formalize this heuristic and study its applicability to other problems.

The use of formal notations forced us to confront the important aspects of the problem early in the development process. For example, this case study had to handle the potential blockage caused by an "infinite pause" in the sending of the tokens of a message. This problem might be easy to overlook and its discovery late in the design process could be costly. Unfortunately, this case study still relied upon operational reasoning to help identify and resolve such issues. Future research should explore ways of using calculational techniques more extensively.

Overall, this case study met the challenges it faced. It developed the router as an open system, assuming very little about the environment. For each program, it devised a very general specification, avoiding unnecessary constraints upon the designers. It also stated the program properties in a consistent and modular way. By struggling to meet these challenges, this research yielded a better understanding of the process of specification and refinement and uncovered opportunities for further research.

Acknowledgements

The National Science Foundation supported the authors' work under Grant CCR-9210342. The authors thank the anonymous referees and several colleagues for their suggestions concerning this work.

References

[1] R. Bird and P. Wadler. *Introduction to Functional Programming.* Prentice Hall International, 1988.

[2] K. M. Chandy. Using triples to reason about concurrent programs. Technical Report 93-02, Dept. of Computer Science, California Institute of Technology, Pasadena, CA, January 1993.

[3] K. M. Chandy and C. Kesselman. The derivation of compositional programs. Technical Report 92-18, Dept. of Computer Science, California Institute of Technology, Pasadena, CA, July 1992.

[4] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.

[5] C. Creveuil and G.-C. Roman. Formal specification and design of a message router. Technical Report 92–44, Dept. of Computer Science, Washington University, St. Louis, MO, Dec. 1992.

[6] H. C. Cunningham and J. T. Udding. "Succeedings" of the Sixth International Workshop on Software Specification and Design: Concurrency and Distribution. *ACM SIGSOFT Software Engineering Notes*, 17(1):46–47, Jan. 1992.

[7] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[8] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer-Verlag, 1992.