Formal Specification and Design of a Message Router

CHRISTIAN CREVEUIL and GRUIA-CATALIN ROMAN
Washington University in Saint Louis

Formal derivation refers to a family of design techniques that entail the development of programs which are guaranteed to be correct by construction. Only limited industrial use of such techniques (e.g., UNITY-style specification refinement) has been reported in the literature, and there is a great need for methodological developments aimed at facilitating their application to complex problems. This article examines the formal specification and design of a message router in an attempt to identify those methodological elements that are likely to contribute to successful industrial uses of program derivation. Although the message router cannot be characterized as being industrial grade, it is a sophisticated problem that poses significant specification and design challenges—its apparent simplicity is rather deceiving. The main body of the article consists of a complete formal specification of the router and a series of successive refinements that eventually lead to an immediate construction of a correct UNITY program. Each refinement is accompanied by its design rationale and is explained in a manner accessible to a broad audience. We use this example to make the case that program derivation provides a good basis for introducing rigor in the design strategy, regardless of the degrees of formality one is willing to consider.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming—*distributed programming*; *parallel programming*; D.2.1 [**Software Engineering**]: Requirements/Specification—*methodologies*; D.2.4 [**Software Engineering**]: Program Verification—*correctness proofs*; D.2.10 [**Software Engineering**]: Design—*methodologies*

General Terms: Design, Verification

Additional Key Words and Phrases: Formal methods, program derivation, specification refinement, UNITY

1. INTRODUCTION

Increasing demands for reliable performance provide a strong impetus for the software engineering community to evaluate and adopt formal methods.

The first author was supported by the Institut National de la Recherche en Informatique et en Automatique under a postdoctoral grant. The second author was supported in part by the National Science Foundation under grants CCR-9015677 and CCR-9217751. The U.S. Government has certain rights in this material.

Authors' address: Department of Computer Science, Washington University, Campus Box 1045, One Brookings Drive, Saint Louis, MO 63130-4899; email: roman@cs.wustl.edu. All correspondence should be directed to the second author.

Formal notations led to the development of specification languages; formal verification contributed to the application of mechanical theorem provers to program checking; and formal derivation—a class of techniques that ensure correctness by construction—has the potential to reshape the way software will be developed in the future. Program derivation is less costly than *post factum* verification, is incremental in nature, and can be applied with varying degrees of rigor in conjunction with or completely apart from program verification. More significantly, while verification is tied to analysis and support tools, program derivation deals with the very essence of the design process, the way one thinks about problems and constructs solutions.

In sequential programming, formal derivation enjoys a long-standing and prestigious tradition [Dijkstra 1976; Dijkstra and Fiejen 1988; Gries 1981; Gries and Prins 1985; Morgan 1988; Morris 1989]. By contrast, derivation is a relatively new concern in concurrent programming. Although a clean and comprehensive characterization of the field is difficult to make and is beyond the scope of this article, three general directions seem to have emerged in the concurrency arena. *Constructivist approaches* start with simple components having known properties and combine them into larger ones whose properties may be computed. CSP-related efforts [Ebergen and Hoogerwoord 1990; Hoogerwoord 1990; Lengauer 1982; Lengauer and Hehner 1982] appear to favor this approach in part due to the algebraic mindset that characterizes the work on abstract CSP. *Specification refinement* has been advocated strongly in the work on UNITY [Chandy and Misra 1988; Knapp 1990; Staskauskas 1988]. An initial highly abstract specification is gradually refined up to the point when it contains so much detail that writing a correct program becomes trivial. *Program refinement* uses a correct program as starting point and alters it until a new program satisfying some additional desired properties is produced. In some of the work on action systems [Back and Sere 1990], for instance, sequential programs are transformed into concurrent or distributed ones. Mixed specification and program refinement [Roman and Wilcox 1994; Roman et al. 1993] has been used in conjunction with the Swarm model [Roman and Cunningham 1990] and its proof logic [Cunningham and Roman 1990; Roman and Cunningham 1992].

Encouraged by these recent developments, we say that it is reasonable to pose the question whether program derivation is a viable substitute for current, mostly ad hoc, methods employed by concurrent system designers. With this aim in mind, our research group has embarked on a number of case studies whose immediate objective is to develop an understanding of how program derivation may be applied to industrial-grade problems. The emphasis is not on tool development but on identifying a design style and associated skills that can be taught effectively and applied productively. We want to show that, given the right model and heuristics, program derivation can be made simple to the point that it can be explained even to the nonspecialist.

The model we chose for our investigation is UNITY and its proof logic. The versatility of the UNITY-style formal derivation strategy has been demonstrated through its application to numerous examples including one industrial-grade problem: the I/O subsystem of the GCOS operating system

[Staskauskas 1993]. These exercises led to the development of a large repertoire of heuristics which we hope to incorporate in a practical design methodology. It is the focus on developing a design methodology which can scale up to large problems and can be applied using various degrees of formality that distinguishes our efforts from previous published work on UNITY. This article reports on some of the lessons learned from one of the program derivation exercises we carried out recently: the design of a message router.

We were introduced to the message router problem through Josephs et al. [1992] where a CSP solution is proposed and proved to be correct by calculating the behavior of the router from that of its components. Our problem formulation is somewhat more abstract—we treat messages as consisting of packets rather than bits—and our design strategy relies on specification refinement. Furthermore, our goal is to study the refinement process rather than to produce a design having the least number of circuit components.

The body of this article consists of a formal specification of the router problem introduced in Section 2 and a UNITY-style specification refinement process developed in Section 3. The UNITY program is derived from the final specification in Section 4, and Section 5 summarizes the lessons we have learned from this exercise. An Appendix provides technical details regarding the proof logic.

2. SPECIFICATION OF THE ROUTER PROBLEM

In this section we give a brief overview of the UNITY logic, an informal description of the router problem, a formal specification, and an analysis of the methodological choices made in constructing the initial specification.

2.1 UNITY Logic

Before presenting the logic, we need to say a few words about UNITY programs, and especially the way they are executed. A typical UNITY program consists of three sections: a *declare* section, which contains Pascal-style declarations of variables; an *initially* section, where all or some of the variables are initialized; and an *assign* section, which is a set of multiple assignment statements. The execution of a UNITY program consists simply in repeating forever the following sequence: select at random a statement in the assign section, and execute it. The only constraint on the selection process is that each statement is chosen infinitely often. The semantics of a UNITY program can then be defined as a set of execution sequences. Each sequence begins with the initial state, as defined in the initially section, and each of the following states is obtained from the previous one by executing a statement.

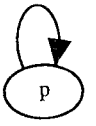
The UNITY logic is used both as a specification language and as a proof logic. The description we give here is intentionally informal and intuitive, in order to allow nonspecialist readers to understand the router specification and design. A more formal description may be found in the Appendix. In the simplest terms, the UNITY logic is composed of several unary and binary operators, which allow one to define global properties over the execution

sequences. Let p and q be two predicates. The list of properties, with their intuitive definition, is given below. Let us note that the program we refer to in the remaining of the paragraph is either the program we want to derive, if the logic is used as a specification language, or the program we want to verify, if the logic is used as a proof logic.

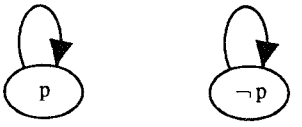
—p **unless** q: this property means that whenever predicate p holds for a program state, p continues to hold in the execution sequence at least up to the point when q is established. In other words, the execution can either stay in a state satisfying $p \wedge \neg q$, or move to a state satisfying q.

Note that p ***unless*** q does not require q to hold eventually; in such a case, p must hold forever.

—**stable** p: predicate p is a *stable* predicate if it remains true forever once it becomes true. This property is equivalent to p ***unless*** *false*.



—**const** p: predicate p is *constant* if both p and $\neg p$ are stable predicates. In other words, p remains true forever if it is initially true, and false forever if it is initially false.



—**inv** p: predicate p is *invariant* if it holds in the initial state, and is stable along any execution sequence.

The four properties described above are *safety* properties, in the sense that they prevent the occurrence of certain state transitions. For instance p **unless** q disallows the transition from $p \wedge \neg q$ to $\neg p \wedge \neg q$. However, to specify problems, we must also be able to state that some *progress* is made, i.e., that certain predicates hold at some point in the future. For this, the UNITY logic offers the following properties:

—$p \mapsto q$: predicate p leads to predicate q if, from a state in which p holds, a state in which q holds is eventually reached.

—p **until** q: this property is slightly stronger than $p \mapsto q$, since it guarantees also that p holds at least until q holds. Its definition is: (p **unless** q) $\wedge$ ($p \mapsto q$).

—p **ensures** q: this last property is strongly related to the text of the program. It states that, whenever p holds, it must hold at least until q

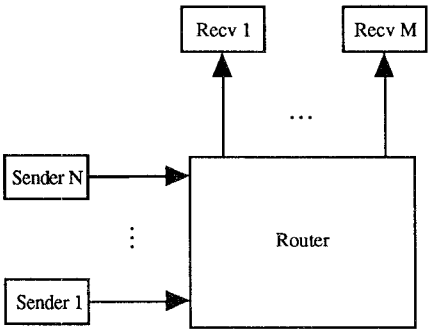


Fig. 1. Structure of the system.

holds (p **unless** q), and that there must exist a statement in the program that establishes the truth of q. The **ensures** property is thus stronger than **until**, since the truth of q in the **until** case can be established by the execution of more than one statement.

2.2 Description of the Problem

We consider a communication network that connects N senders of messages to M receivers via a message router. Each sender is connected to one of the input ports of the router, and each receiver to one of the output ports (Figure 1).

Each message is composed of a finite number of packets that can be of three different types: *header*, *body*, and *tail*. The header, which is the first packet of the message, contains the port address of the message destination. Each header is followed by one or more body packets which contain the actual data. Finally, the tail packet marks the end of the message.

The behavior of the router is defined by the following requirements:

(R1) The value of the body packets must not be modified, but, for control purposes, the router may modify the value of the header and tail packets.

(R2) Packet ordering within a message must be preserved.

(R3) Messages from the same source going to the same destination must not be reordered (at the receiver).

(R4) Messages from different sources going to the same destination must not be interleaved (at the receiver).

(R5) Each packet that is sent must eventually be delivered to the intended receiver.

2.3 Formal Specification

The system we want to specify is composed of three interacting entities: an input environment (the N senders), an output environment (the M receivers), and the router. The UNITY formalism offers the possibility to specify those three parts separately, and then to compose the three specifications. However, this implies the use of conditional properties (for instance, assuming

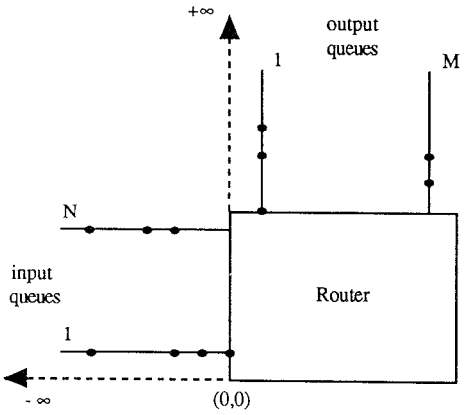


Fig 2. Abstract structure of the system

that property (P1) is true in the input environment; property (P2) is true in the router), which make reasoning and understanding more difficult. We believe it is easier to deal with a unique specification describing the behavior of the system in its entirety, without any interaction with the outside world.

To do so, we abstract the senders and the receivers as infinite I/O queues (Figure 2). Initially the input queues contain all the packets that have to be sent, and the output queues, as well as the router, are empty. Each packet in the input and output environments resides at some distinct location, which is a pair of coordinates (*row*, *column*). For instance, packets in the input queues have a row coordinate that ranges from 1 to N, and a column coordinate that ranges from 0 to $-\infty$.

Let us define some notation. Packets are designated by notation $\tau[v]$, where τ, the packet type, is equal to h, b, or t—header, body, or tail—and v, the packet value, belongs to some set V. Each packet is augmented with four auxiliary variables: n, m, i, and j. These auxiliary variables have constant values which may not be used by the program we want to derive, their only purpose is to make the specification and design process easier. Variables n and m, ranging respectively from 1 to N and from 1 to M, are the addresses of the sender and receiver of the packet (or, in other words, the numbers identifying the source input queue and destination output queue). Variable i is the number of the message the packet belongs to. We define the number of the first message sent by each sender to be equal to 1, the number of the second message to be equal to 2, and so on. Finally, variable j is the packet number, that is, the position of the packet within the message. A packet augmented with the auxiliary variables is called a *logical* packet, as opposed to the *physical* packets carried by the network. We use notation $\tau(n, m, i, j)[v]$ to designate a logical packet. Note that the pair (n, i)—(sender number, message number)—identifies uniquely a message in the system, and that the triple (n, i, j)—(sender number, message number, packet number)—identifies uniquely a packet.

In the specification, the notation $\tau(n, m, i, j)[v]$ is never used. Rather we designate logical packets by Greek letters (α, β, or δ), and access the packet

attributes through several access functions. Function *type*, applied to a packet, returns the type of the packet (τ). Functions *src* and *dest* return the identity of the source input queue (n) and of the destination output queue (m). The message number (i) and packet number (j) are given by functions *mnr* and *pnr*. Finally, we use the functions *mid* and *pid* to access the message identifier (i.e., the pair (n, i)) and the packet identifier (i.e., the triple (n, i, j))—the lexicographical order is assumed whenever identifiers are compared in logical expressions such as $pid.\beta < pid.\alpha$. When it is necessary to deal explicitly with the value of a packet, we use the notation $\sigma[v]$, where σ represents $\tau(n, m, i, j)$.

The location of packets in the system is given by the function Π. Let Λ be the set of all logical packets in the router and its environment. Let E be the set of locations in the environment

$$E = \langle \textbf{set } p, q : (1 \le p \le N \wedge q \le 0) \vee (p \ge N + 1 \wedge 1 \le q \le M) :: (p, q) \rangle^{1}$$

where negative values of q are associated with packets which have not yet entered the router while values of p exceeding N are associated with delivered packets. Let R be the set of locations in the router (the value of R is unknown at this time). Function Π is then defined as

$$\Pi : \Lambda \to E \cup R.$$

To make the specification easier to read, we use the notation $\alpha@\lambda$ to mean that packet α is at location λ—i.e., $\Pi.\alpha = \lambda$ (throughout the article, the operator "." is used to denote function application). We sometimes add a type designator (h, b, or t) to the variables representing packets. For instance, $\alpha^h@\lambda$ means that a header packet α is at location $\lambda : \alpha@\lambda \wedge type.\alpha = h$. To state the no-reordering constraint, we define a relation $\sqsubset$ on packets. Its definition is

$$\begin{aligned}
&\alpha \sqsubset \beta \Leftrightarrow \\
&\quad \langle \exists p, q, q' : q < q' \le 0 :: \alpha@(p, q) \wedge \beta@(p, q') \rangle \\
&\quad \vee \langle \exists p, p', q : N + 1 \le p < p' :: \alpha@(p, q) \wedge \beta@(p', q) \wedge \mathrm{src}.\alpha = \mathrm{src}.\beta \rangle \\
&\quad \vee \langle \exists p, p', q, q' : q' \le 0 \wedge p' \ge N + 1 :: \alpha@(p, q') \wedge \\
&\qquad \beta@(p', q) \wedge \mathrm{src}.\alpha = \mathrm{src}.\beta \rangle.
\end{aligned}$$

That is, β is ahead of α according to $\sqsubset$ iff β is in front of α in the same input queue, or β is in front of α in the same output queue and both packets

[1] This is an example of a *constructor*, a syntactic element which occurs frequently in UNITY notation. The general form of the constructor is:

$$\langle \textbf{op } \text{dummy\_variables} : \text{range\_constraint} :: \text{expression} \rangle$$

where **op** is typically a binary, associative, and commutative operator (such as $+$, $*$, $\wedge$, $\vee$, written Σ, Π, $\forall$, $\exists$, respectively). Logically, the constructor creates a multiset of values $\{v_1, v_2, \ldots, v_n\}$ by evaluating the *expression* for every possible instantiation of the *dummy_variables* satisfying the *range_constraint*. The final value of the constructor is obtained by evaluating the expression v_1 **op** v_2 **op** ... **op** v_n. If the range is empty the zero-element for the operator is returned. Other frequently used operators are *min*, *max*, and *set*, having the obvious interpretations.

come from the same source, or β is in the output environment, α is in the input environment, and both packets have the same source.

The top-level specification S_0 is given below. (While every informal requirement can be traced to one or more logical assertions, the formal specification does not follow the exact structure of the informal specification.) Any free variable appearing in the UNITY assertions is assumed to be universally quantified over all the elements of its domain. Explanations follow the specification.

Message Representation

$$\textbf{inv}\ \langle \exists \lambda :: \alpha@\lambda \rangle \wedge k = \langle \Sigma\ \beta,\ \lambda : \beta@\lambda \wedge \mathrm{mid}.\beta = \mathrm{mid}.\alpha :: 1 \rangle \quad \text{(P1)}$$
$$\Rightarrow\ k \geq 3 \wedge (\mathrm{pnr}.\alpha = 1 \Leftrightarrow \mathrm{type}.\alpha = h) \wedge$$
$$(1 < \mathrm{pnr}.\alpha < k \Leftrightarrow \mathrm{type}.\alpha = b) \wedge (\mathrm{pnr}.\alpha = k \Leftrightarrow \mathrm{type}.\alpha = t)$$
$$\textbf{const}\ \langle \exists v, \lambda :: \sigma[v]^h@\lambda \rangle \quad \text{(P2)}$$
$$\textbf{const}\ \langle \exists \lambda :: \sigma[v]^b@\lambda \rangle \quad \text{(P3)}$$
$$\textbf{const}\ \langle \exists v, \lambda :: \sigma[v]^t@\lambda \rangle \quad \text{(P4)}$$

Message Location in the Environment

$$\textbf{inv}\ \alpha@(p, q) \wedge (p \geq N + 1 \vee q \leq 0) \wedge$$
$$\beta@(p', q') \wedge (p' \geq N + 1 \vee q' \leq 0)$$
$$\Rightarrow (\mathrm{pid}.\alpha = \mathrm{pid}.\beta \Leftrightarrow (p, q) = (p', q')) \quad \text{(P5)}$$
$$\textbf{inv}\ (\alpha@(p, q) \wedge q \leq 0 \Rightarrow p = \mathrm{src}.\alpha) \wedge$$
$$(\alpha@(p, q) \wedge p \geq N + 1 \Rightarrow q = \mathrm{dest}.\alpha) \quad \text{(P6)}$$

Queue Properties

$$\langle \exists q, q' : q < q' \leq 0 :: \alpha@(p, q) \wedge \beta@(p, q') \rangle$$
$$\textbf{unless}\ \neg \langle \exists q' : q' \leq 0 :: \beta@(p, q') \rangle \quad \text{(P7)}$$
$$\textbf{stable}\ \langle \exists p : p \geq N + 1 :: \alpha@(p, q) \wedge \neg \langle \exists p' : p' > p :: \beta@(p', q) \rangle \rangle \quad \text{(P8)}$$

No-Reordering Property

$$\textbf{inv}\ \alpha \sqsubset \beta \Rightarrow \mathrm{pid}.\beta < \mathrm{pid}.\alpha \quad \text{(P9)}$$

Noninterleaving Property

$$\textbf{inv}\ \alpha@(p, q) \wedge \beta@(p', q) \wedge \delta@(p'', q) \wedge$$
$$N + 1 \leq p < p' < p'' \wedge \mathrm{mid}.\alpha = \mathrm{mid}.\delta$$
$$\Rightarrow \mathrm{mid}.\beta = \mathrm{mid}.\alpha \quad \text{(P10)}$$

Packet Movement

$$\sigma[v]@(p, q) \wedge q \leq 0 \mapsto \langle \exists v', p', q' : p' \geq N + 1 :: \sigma[v']@(p', q') \rangle \quad \text{(P11)}$$

The first four properties are related to the representation of messages. Property (P1) states that messages are properly structured. If α is a packet at some location in the system, and k is the number of packets in the message α belongs to, then

—k is greater than or equal to 3, i.e., each message is composed of at least three packets;

—the packet number of α is equal to 1 iff α is a header, ranges strictly from 1 to k iff α is a body, and is equal to k iff α is a tail.

Properties (P2) to (P4) state that packets are neither created nor destroyed, and that the value of body packets may not be modified. Let us for instance explain property (P2). Going back to the definition of **const**, (P2) means that:

—If the header packet $\sigma[v]$ exists initially in the system, it will exist forever, but not necessarily with the same value (existential quantification on v).
—If, for any value v, $\sigma[v]$ does not initially exist in the system, then it will never exist.

Property (P4) is the symmetric of (P2) for tail packets. Additionally, property (P3) prevents the value of the body packets to be changed.

Properties (P5) and (P6) specify packet locations in the environment. (P5) states that each packet has a unique location, and that each location contains at most one packet. Property (P6) expresses the requirement that each packet in the input environment resides in its source input queue, and each packet in the output environment resides in its destination output queue.

Properties (P7) and (P8) specify that the input rows and output columns are queues. (P7) states that packets are sent in the order they are queued, i.e., a packet cannot be sent if there are packets in front of it in the queue: considering packet β in its input queue, and packet α behind β in the same queue, α must stay behind β as long as β is in the input queue. Property (P8) states that packets received in the output queues can only appear behind the packets already present in the queues. One way to specify this is to say that, considering two packets α and β, if α is in its output queue and β is not ahead of α in the queue, then it will never be.

The no-reordering and noninterleaving constraints are expressed by properties (P9) and (P10). Initially, the packets in the input environment are queued in the order they have to be sent—i.e., the packets with the lexicographically smallest identifiers are in the first positions of the queues. This means that if packet β is ahead of α ($\alpha \sqsubset \beta$), β's pid is smaller than α's. To express the no-reordering constraint, we have to state that this relation is invariant all along the execution, as expressed by property (P9). The noninterleaving requirement is specified by property (P10). Considering three packets α, β, and δ such that δ is ahead of β which is ahead of α, if α and δ belong to the same message, β must also belong to the same message.

The first ten properties are all safety properties. The only progress requirement is stated by property (P11). Each packet $\sigma[v]$ in the input environment must eventually reach the output environment, but not necessarily with the same value.

2.4 Discussion

The development of the initial specification requires a careful balancing act. One must seek simplicity, generality, and convenience. Simplicity is achieved through a solid understanding of the problem which facilitates elegant modeling of the environment and proper choice of notation. The key decisions we took regarding modeling the environment were to treat the router and its environment as a closed system and to view the senders and receivers as

unbounded queues. The closed-system assumption is motivated by our desire to avoid dealing with conditional properties which add a certain degree of complexity to the specification and verification processes. As long as the assumptions about the environment are clearly distinguishable from the properties of the router and the environment is not unduly restricted, the design of the router is made simpler without loss of generality. The price we pay for this becomes evident only if we compose the router with some other device and attempt to prove properties of the composite, an equivalent open-system specification may need to be developed. This potential penalty may be acceptable if the design of the router is made simpler by the closed-system assumption—our experience to date leads us to believe that this is actually the case.

Modeling the senders and receivers as unbounded queues is a direct consequence of the fact that any formal characterization of the router behavior must have some representation for its input and output histories. Each input queue simply reflects the future inputs from that particular sender while each output queue captures the past outputs directed to the particular receiver. This characterization imposes no constraints on what messages a sender supplies to the router except for the fact that they are independent of its outputs and other inputs. Moreover, the size of the specification is reduced by having certain environmental obligations automatically satisfied, e.g., the guarantee that a sender eventually transmits the entire message. One technical detail we had to consider carefully was the use of unbounded structures inside of a UNITY specification, particularly in light of the property (P11) which requires each input message eventually to be delivered. Since for any given message, the number of messages ahead of it is finite and messages cannot pass each other, the specification is technically correct. Finally, one should not underestimate the power of mathematical notation as a complexity control mechanism. The packet location function (Π) and the packet ordering relation ($\sqsubset$) are indicative of level of compactness one can achieve by employing proper mathematical notation and of the manner in which single symbols may be associated with concepts which, though intuitive in nature, have complex formal definitions.

Generality means avoiding design biases in the initial specification. We accomplished this by simply saying nothing about the router itself aside from the fact that it exists. The only reference to the router is the set R of router locations. Since R is left undefined, no conceivable design solution is ruled out. For instance, by allowing the elements of R to be sets of nodes in a network one could design a router that makes use of message duplication to achieve reliable delivery. Packets could be interleaved, combined, or reordered freely by the router as long as their delivery to the output ports is in accordance with the stated specification. Finally, we must note that the specification does not even require for the router and the output queues to be initially empty.

Convenience is concerned with the ability to use effectively the notation introduced in the initial specification throughout the remainder of the design process, even though one does not know what the ultimate outcome of the

design will be. Experience can be called upon to anticipate the general direction the design will assume, and preliminary design exercises can provide hints that help one choose among alternatives. Our choice for numbering the input and output ports, while natural for any router, will turn out to be particularly well suited for the specific router topology we will adopt in the next section. A related decision, that of representing the input and output queues as unbounded arrays rather than sequences, led to a uniform representation of both the environment and the router—here insight in the design direction we planned to pursue influenced our choice of representation. However, one cannot accomplish with notation that which the problem itself does not allow; even though both the inputs and the outputs are represented as queues, their distinct nature (no enqueuing permitted on the input and no dequeuing permitted on the output) did not allow us to unify their treatment, and we had to include in the specification two distinct properties, (P7) and (P8). Reasonable sacrifices with respect to generality buy often a significant level of convenience. We chose, for instance, to center the specification process around the packet concept rather than develop first a specification which is independent of the message structure. Such an unnecessarily general specification would have added a significant level of complexity to the design by increasing the size of the specification and the amount of notation required to accommodate it.

3. ROUTER DESIGN

The type of router specified in Section 2 could be found in the design of a multiprocessor interconnection network, an area of much research and commercial interest in recent years. The reader who desires to learn more about such networks may want to use Ni and McKinley [1993] as a starting point. This article, however, is not about network topologies or routing algorithms. The router is simply a vehicle to help us develop a better understanding about formal design methods and the obstacles one faces in their application to industrial problems. For this reason, we make no attempt to explore a large design space but focus our attention on how a specific design idea shapes the specification refinement process. The final result is a mesh-connected network and a deadlock-free, deterministic routing algorithm. More specifically, we employ wormhole routing [Dally and Seitz 1987] by using the header packet to construct a path from the sender to the receiver. The path is followed blindly by all subsequent packets belonging to the same message. The tail message frees the path for use by other messages.

The details of the solution emerge through a design process entailing multiple refinement steps. The objective of these steps is to gradually give a more and more detailed description of the router, up to the point where a UNITY program can be derived trivially from the specification. To be correct, each refined specification must imply the specification of the previous step. We start with a brief overview of the refinement steps.

Refinement 1. In the first refinement, we define the general topology of the router as a grid of $N \times M$ switches. The N input lines and M output

lines are thus extended inside the router. Each switch can receive packets from its left neighbor on the row or its bottom neighbor on the column, and can route them either to its right neighbor on the row or to its upper neighbor on the column, depending on the destination of the packets. So, to move from its source row to its destination column, each packet first travels along the row (one switch at a time) until it reaches the destination column, and then just moves up the column (also one switch at a time).

Refinement 2. The second refinement provides additional details about the behavior of each single switch, by defining the mechanism that prevents messages from being interleaved along the columns. We will see that this can be done by associating two mutually exclusive signals—*turn* and *up*—with each switch. Signal *turn* prevents messages to move through the switch along the column when a message is currently passing through the switch from the row to the column, and the other way around for signal *up*.

Refinement 3. In the third refinement, we specify further the behavior of the switches by introducing a strong fairness constraint, i.e., the existence of a constant upper bound on the number of messages that can block a particular message from passing through each switch. We choose a design in which a message waits for at most one other message to pass through the switch before it can proceed.

Refinement 4. At this point in the design, each switch on the rows makes the decision to route messages either to the next switch on the row, or to the next switch on the column, by comparing the message destination to the number of the column it is located at. This implies that each switch has to know its location. The purpose of the fourth refinement is to eliminate this knowledge by using the value of the header packets. We will make the value of each header packet decrease by one each time the packet passes through a switch along the row. Since the value is initially equal to the destination column, this implies that a message will have to take a turn when the value is equal to 1.

Refinement 5. The fifth refinement deals with the execution control we impose on the switches. A possible choice is to have the switches running asynchronously; another choice is to have them working in a synchronous way. We chose the more realistic asynchronous behavior. Since each location can contain at most one packet, this implies that a packet will not be able to move to the next location, unless it is empty.

Refinement 6. Finally, the last refinement step is the transformation of the **leads-to** properties into **ensures** properties. As we said earlier in the paper, **ensures** properties are strongly related to the text of the program, which implies that we can easily derive assignment statements from them.

3.1 Refinement 1: Definition of the Router Topology

In this section we introduce a router design consisting of an $N \times M$ grid of switches. As an example, a three-sender four-receiver router is depicted in

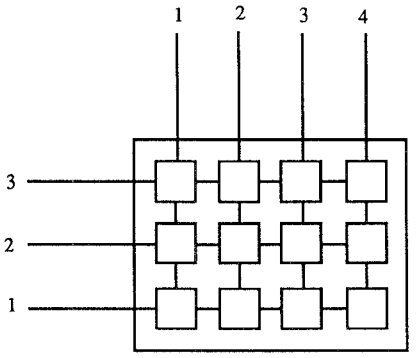


Fig. 3. Internal structure of the three-sender four-receiver router.

Figure 3. The location of each switch in the grid is given by the pair (p, q) where p is the row number, and q the column number. Each switch (p, q) contains two registers—a row and a column register—and an arbitration element (Figure 4). We identify the location of the row register by the triple $(p, q, 0)$, and the location of the column register by $(p, q, 1)$. The function of the arbitration element is to handle the movement of the packets showing up in the row and column registers, in order to avoid interleaving of messages on the columns. Its design will be the subject of refinement 2.

The packet movement inside the router is defined as follows: from the row register of switch (p, q)—i.e., location $(p, q, 0)$—a packet moves to the row register of switch $(p, q + 1)$—i.e., location $(p, q + 1, 0)$—if q is not the destination column, or to the column register of switch $(p + 1, q)$—i.e., location $(p + 1, q, 1)$—if q is the destination column. Once it has reached its destination column, a packet just goes up, one location at a time.

Let us now give some definitions and notation. To make the location spaces inside and outside the router uniform, and thus make the specification simpler, we redefine environment locations (p, q) to be triples $(p, q, 0)$ in the input queues, and triples $(p, q, 1)$ in the output queues. The previous set E of environment locations is thus refined into

$$E' = \langle \textbf{set } p, q, r : (1 \le p \le N \wedge q \le 0 \wedge r = 0) \vee (p \ge N + 1 \wedge 1 \le q \le M \wedge r = 1) :: (p, q, r) \rangle.$$

The definition of R, the set of locations inside the router, is

$$R \equiv \langle \textbf{set } p, q, r : 1 \le p \le N \wedge 1 \le q \le M \wedge 0 \le r \le 1 :: (p, q, r) \rangle.$$

With this definition of R, the function Π returns pairs when that message is outside the router and triples when the message is inside. This lack of uniformity is rectified by refining the location function Π into a new function Π':

$$\Pi' : \Lambda \to E' \cup R.$$

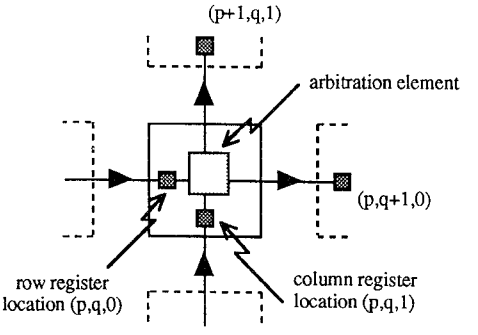


Fig. 4. Internal structure of switch (p, q).

The relation between Π and Π' is given by the following coupling invariant:

$$\begin{aligned}\langle \forall \alpha, p, q, r :: \\ p \le N \wedge q \ge 1 \wedge 0 \le r \le 1 \\ \Rightarrow [\Pi.\alpha = (p, q, r) \Leftrightarrow \Pi'.\alpha = (p, q, r)] \\ \wedge\ q \le 0 \Rightarrow [\Pi.\alpha = (p, q) \Leftrightarrow \Pi'.\alpha = (p, q, 0)] \\ \wedge\ p \ge N + 1 \Rightarrow [\Pi.\alpha = (p, q) \Leftrightarrow \Pi'.\alpha = (p, q, 1)]\rangle.\end{aligned}$$

We now use $\alpha@(p, q, r)$ to mean $\Pi'.\alpha = (p, q, r)$, and we introduce the notation $(p, q, r) \gg (p', q', r')$ to mean

$$(p' = p \wedge q' = q + 1 \wedge r = r' = 0) \vee (p' = p + 1 \wedge q' = q \wedge r' = 1).$$

The operator "$\gg$" relates valid pairs of consecutive locations along legal paths through the router, i.e., from a row register to the one to its right, from a row register to the column register above, or from a column register to the one above. Location $(p, q, 0)$ is thus in relation with $(p, q + 1, 0)$ and $(p + 1, q, 1)$, and location $(p, q, 1)$ is in relation with $(p + 1, q, 1)$. The relation $\sqsubset$ is refined into the relation $\prec$. Its definition is

$$\begin{aligned}\alpha \prec \beta \Leftrightarrow\ & \langle \exists p, q, q' : q < q' :: \alpha@(p, q, 0) \wedge \beta@(p, q', 0)\rangle \\ & \vee \langle \exists p, p', q : p < p' :: \alpha@(p, q, 1) \wedge \beta@(p', q, 1) \wedge \mathrm{src}.\alpha = \mathrm{src}.\beta\rangle \\ & \vee \langle \exists p, p', q, q' : q' \le q :: \alpha@(p, q', 0) \wedge \beta@(p', q, 1) \wedge \mathrm{src}.\alpha = \mathrm{src}.\beta\rangle.\end{aligned}$$

That is, β is ahead of α according to $\prec$ iff β is in front of α on the same row, or β is in front of α on the same column and both packets come from the same source, or β is on its column, α is on its row at the left of β, and both packets have the same source. The refined specification S_1 is the following:

Message Representation
Properties (P1), (P2), (P3), and (P4).

Message Location

$$\begin{aligned}&\textbf{inv}\ \alpha@(p, q, r) \wedge \beta@(p', q', r') \\ &\quad \Rightarrow (\mathrm{pid}.\alpha = \mathrm{pid}.\beta \Leftrightarrow (p, q, r) = (p', q', r')) \qquad \text{(P5.1)} \\ &\textbf{inv}\ \alpha@(p, q, r) \Rightarrow (p = \mathrm{src}.\alpha \wedge q \le \mathrm{dest}.\alpha \wedge r = 0) \vee \\ &\quad (p > \mathrm{src}.\alpha \wedge q = \mathrm{dest}.\alpha \wedge r = 1) \qquad \text{(P6.1)}\end{aligned}$$

No-Reordering Property
$$\textbf{inv}\ \alpha \prec \beta \Rightarrow \mathrm{pid}.\beta < \mathrm{pid}.\alpha \tag{P9.1}$$

Noninterleaving Property
$$\textbf{inv}\ \alpha@(p, q) \wedge \beta@(p', q) \wedge \delta@(p'', q) \wedge N + 1 \leq p < p' < p'' \wedge \mathrm{mid}.\alpha = \mathrm{mid}.\delta \Rightarrow \mathrm{mid}.\beta = \mathrm{mid}.\alpha \tag{P10.1}$$

Packet Movement in the Environment
$$\sigma[v]@(p, q, r) \wedge (p \geq N + 1 \vee q \leq 0)\ \textbf{until}\ \langle \exists p', q', r' : (p, q, r) \gg (p', q', r') :: \sigma[v]@(p', q', r') \rangle \tag{P11.1}$$

Packet Movement Inside the Router
$$\sigma[v]@(p, q, r) \wedge p \leq N \wedge q \geq 1\ \textbf{until}\ \langle \exists v', p', q', r' : (p, q, r) \gg (p', q', r') :: \sigma[v']@(p', q', r') \rangle \tag{P11.2}$$

Properties (P5), (P6), (P9), (P10), and (P11) of specification S_0, which define properties of the packets in the environment, have been extended to the entire system. Property (P5.1) specifies that each packet in the environment or the router has a unique location, and that each location contains at most one packet. Property (P6.1) defines the set of locations packets are allowed to be at. Each packet can only be located in the source input row or the destination output column, but cannot move beyond the destination column on the row, and below the input row on the column. Property (P9.1) states that packets remain ordered according to their identifiers, and (P10.1) specifies that messages are not interleaved. Finally, property (P11), which states that every packet in the input eventually moves to the output, has been refined into two progress properties, one for the environment, and the other one for the router. Property (P11.1) states that eventually each packet in the environment moves to the next location, as defined by the "$\gg$" operator. Since it is an **until** property, (P11.1) also specifies that packets cannot move anywhere but to the next location. Property (P11.2) is quasisymmetrical to (P11.1) for packets inside the router. The only difference is that packet values are allowed to change.

PROOF OBLIGATIONS. We need to show that each property of S_0 is implied by S_1. Note that even though properties (P1) to (P4) are not syntactically modified, the definition of the function @ has changed, which implies that these properties also need to be verified.

PROOF OUTLINE. The formal proof of each of the refinement steps are omitted. In the proof outline sections, we just give an intuitive explanation in order to convince the reader of the validity of the refinements.

The proof of properties (P1)–(P6) and (P9)–(P10) is straightforward. Each property is directly implied by the corresponding property in S_1 and the coupling invariant.

The proof that packets are sent in the order they are queued (property (P7)) follows from the fact that packets in the input queues move one location at a

time (property (P11.1)), and that each location can contain at most one packet (property (P5.1)). This implies that packets cannot pass each other, and hence that they can only be sent in the order they are queued.

The proof of (P8)—packets in the output queues can only appear behind the packets already present in the queues—is a bit more complicated. We need to show that, considering two packets α and β, if α is in its output queue and β is not ahead of α in the queue, then it will never be. Three cases are possible. If β does not exist in the system, then it will never be ahead of α since packets cannot be created (properties (P2)–(P4)). If β exists in the system, but its destination column is different from the destination column of α, then it will also never be ahead of α since a packet in the output environment can only reside in its destination column. The final case is β existing in the system with the same destination column as α. In this case, the only way for β to move ahead of α is to overtake it, which is not possible, as stated above.

Finally, the fact that packets in the input eventually move to the output (property (P11)) can be proved from (P11.1) and (P11.2) by a double application of the induction principle. We can first show that a packet in the input environment moves eventually to its destination column, by using the distance between the destination column and the position of the packet in the row as the metric. This distance decreases by one with each move. We can also prove that a packet on the destination column inside the router moves eventually to the output environment. The metric in this case is the distance between row $N + 1$ and the position of the packet in the column; it also decreases by one with each move. The truth of (P11) follows from the transitivity of **leads-to**.

Discussion. One of the fundamental precepts of stepwise refinement is the notion that design decisions ought to be postponed for as long as possible in order to avoid premature commitment to a particular solution path. While attempting to adhere to this principle, we sometimes found that the cost associated with maximizing the generality of the specifications at each step was excessive and did not serve well the goal of generating a dependable design in an easily understood and cost-effective manner. Refinement 1, for instance, involves not one but three important design decisions regarding the router topology, the message paths, and the buffer sizes. Since no useful refinement is possible without considering the router topology, the commitment to a cross-bar switch is a decision that clearly belongs in the first refinement. The selection of legal message paths and buffer sizes could have been postponed for subsequent refinements. Yet, in a practical setting these decisions are immediate and obvious concerns on the part of the designer because they deal with basic defining parameters of the design solution being considered. On the formal side, the result is a simple mathematical formulation—tailored to this type of design—and a major reduction in the complexity of the refinements and associated proofs.

In retrospect, the decisions taken in the first refinement are precisely those that enable the designer to move from one level of abstraction (router level) to

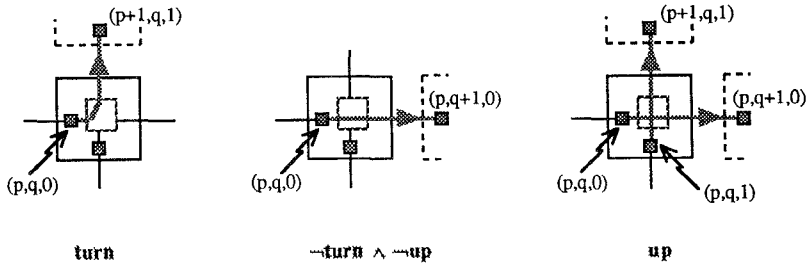


Fig. 5. Possible states of the arbitration element.

the next (switch level) in the router organization. Refinement 1 establishes the internal structure of the router and provides the requirements for its next level components. Each switch can (logically) buffer at most two packets and is not allowed to block indefinitely the passage of an individual message. The precise manner in which these requirements are realized is left undefined.

3.2 Refinement 2: Arbitration Element Refinement

The role of the arbitration element is to route out of the switch the packets showing up in the row and column registers. The main problem is to avoid the interleaving of messages passing through the switch on the column and from the row to the column. To solve this problem, we introduce two mutually exclusive signals—*turn* and *up*—whose purpose is to regulate the movement of packets at the intersection of the row and the column (Figure 5). When the signal *turn* is on—which implies that *up* is off—the paths from location $(p, q, 0)$ to location $(p, q + 1, 0)$, and from $(p, q, 1)$ to $(p + 1, q, 1)$ are broken. Packets can only move from location $(p, q, 0)$ on the row to location $(p + 1, q, 1)$ on the column. When the signals *turn* and *up* are off, packets can only move through the switch along the row. Finally, when the signal *up* is on—which implies that *turn* is off—packets can simultaneously move through the switch along the row and along the column.

The *turn* and *up* signals are initially false, and are triggered by the arrival of header packets in the row and column registers. The *turn* signal is eventually set when a header packet whose destination column is equal to q shows up at location $(p, q, 0)$. The signal remains on as long as the tail of the same message has not showed up, and is turned off as the tail moves through the switch. The policy associated with the *up* signal is symmetrical. Let us mention that we do not impose any priority between the row and column registers. That is, when two headers (going to the same location $(p + 1, q, 1)$) are in the row and column registers at the same time, the choice of which signal to be turned on is nondeterministic.

We define the predicate $turn(p, q)$ to mean that the *turn* signal in switch (p, q) is on, and $up(p, q)$ to mean that the *up* signal in switch (p, q) is on. We also define a two-parameter function *message*. The expression

message.$\lambda.\lambda'$ is interpreted to mean that there is a message in the system whose head is at location λ, and tail at location λ'. Formally

$$\text{message}.\lambda.\lambda' \equiv \langle \exists \alpha, \beta : \text{mid}.\alpha = \text{mid}.\beta :: \alpha^h@\lambda \wedge \beta^t@\lambda' \rangle.$$

The refined specification S_2 is the following:

Message Representation
Properties (P1), (P2), (P3), and (P4).

Message Location
Properties (P5.1) and (P6.1).

No-Reordering and Noninterleaving Properties
Properties (P9.1) and (P10.1).

Signal Properties

inv $\neg(\text{turn}(p, q) \wedge \text{up}(p, q))$ (P12)

inv $q \geq 1 \wedge \langle \exists p', q', r', q'' : q'' \leq q < q' ::$
$\text{message}.(p', q', r').(p, q'', 0)\rangle \Rightarrow \neg\text{turn}(p, q)$ (P13)

inv $q \geq 1 \wedge \langle \exists p', q' : p' > p \wedge q' \leq q ::$
$\text{message}.(p', q, 1).(p, q', 0)\rangle \Rightarrow \text{turn}(p, q)$ (P14)

inv $p \leq N \wedge \langle \exists p', p'', q'', r'' : p' > p \wedge$
$(p'' < p \vee (p'' = p \wedge r'' = 1)) ::$
$\text{message}.(p', q, 1).(p'', q'', r'')\rangle \Rightarrow \text{up}(p, q)$ (P15)

inv $\text{turn}(p, q) \Rightarrow \langle \exists p', q', r' : ((r' = 0 \wedge p' = p) \vee$
$(r' = 1 \wedge p' > p)) \wedge q' \leq q ::$
$\text{message}.(p', q, r').(p, q', 0)\rangle$ (P16)

inv $\text{up}(p, q) \Rightarrow \langle \exists p', p'', q'', r'' : p' \geq p \wedge$
$(\text{p}'' < \text{p} \vee (\text{p}'' = \text{p} \wedge \text{r}'' = 1))$
$:: \text{message}.(p', q, 1).(p'', q'', r'')\rangle$ (P17)

$\alpha^h@(p, q, 0) \wedge q = \text{dest}.\alpha \wedge \text{turn}(p, q)$ **unless** $\neg\, \alpha@(p, q, 0)$ (P18)

$\alpha^h@(p, q, 1) \wedge p \leq N \wedge \text{up}(p, q)$ **unless** $\neg\, \alpha@(p, q, 1)$ (P19)

$\alpha^t@(p, q, 0) \wedge q \geq 1 \wedge \text{turn}(p, q)$
unless $\neg\, \alpha@(p, q, 0) \wedge \neg\text{turn}(p, q)$ (P20)

$\alpha^t@(p, q, 1) \wedge p \leq N \wedge \text{up}(p, q)$
unless $\neg\, \alpha@(p, q, 1) \wedge \neg\text{up}(p, q)$ (P21)

Packet Movement in the Environment
Property (P11.1).

Packet Movement Inside the Router

$\sigma[v]@(p, q, r) \wedge p \leq N \wedge q \geq 1$
unless $\langle \exists v', p', q', r' : (p, q, r) \gg (p', q', r') :: \sigma[v']@(p', q', r')\rangle$ (P11.2.1)

$\sigma[v]^h@(p, q, 0) \wedge q \geq 1 \wedge q \neq \text{dest}.\sigma[v]$
$\mapsto \langle \exists v' :: \sigma[v']@(p, q + 1, 0)\rangle$ (P11.2.2)

$\sigma[v]^\tau@(p, q, 0) \wedge \tau \neq h \wedge q \geq 1 \wedge \neg\text{turn}(p, q)$
$\mapsto \langle \exists v' :: \sigma[v']@(p, q + 1, 0)\rangle$ (P11.2.3)

$\sigma[v]^h@(p, q, 0) \wedge q = \text{dest}.\alpha \mapsto \sigma[v]@(p, q, 0) \wedge \text{turn}(p, q)$ (P11.2.4)

$\sigma[v]@(p, q, 0) \wedge q \geq 1 \wedge \text{turn}(p, q)$
$\mapsto \langle \exists v' :: \sigma[v']@(p + 1, q, 1)\rangle$ (P11.2.5)

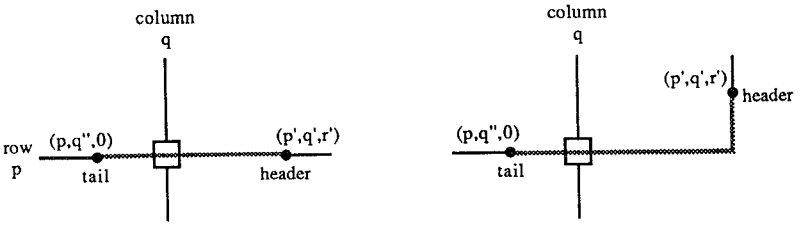


Fig. 6. The *turn* signal is off when a message is in transit through the switch along the row.

$$\sigma[v]^h@(p, q, 1) \wedge p \leq N \mapsto \sigma[v]@(p, q, 1) \wedge \mathrm{up}(p, q) \tag{P11.2.6}$$

$$\sigma[v]@(p, q, 1) \wedge p \leq N \wedge \mathrm{up}(p, q) \mapsto \langle \exists v' :: \sigma[v']@(p + 1, q, 1) \rangle \tag{P11.2.7}$$

The safety properties of the switch signals are described by (P12)–(P21). The fact that the two signals are mutually exclusive is expressed by (P12). The meaning of invariant (P13) is illustrated in Figure 6. If a message is in transit through the switch (p, q) along the row—i.e., the header is past the switch, either still on the row, or already on the destination column, and the tail has not passed the switch yet—then the turn signal is off.

As depicted in Figure 7, invariant (P14) states that, if a message is in transit through the switch (p, q) from the row to the column—i.e., the header is past the switch on the column, and the tail is still on the row—then the *turn* signal is on.

Invariant (P15) is symmetric to (P14) for the *up* signal. It states that, if a message is in transit through the switch (p, q) along the column—i.e., the header is past the switch on the column, and the tail is either on the column and has not moved through the switch yet, or is still on the source row—then the *up* signal is on, as shown in Figure 8.

Invariants (P16) and (P17) further specify that when the *turn* signal (*up* signal) is on, there must exist a message in transit through the switch from the row to the column (on the column). Note that, as opposed to (P14) and (P15), properties (P16) and (P17) provide for the case where the header of the message is still in the row register (column register) of the switch. The reason is that, when a header is in the row or column register, the proper signal is first turned on, and only in a subsequent step the packet is allowed to move.

Properties (P18) and (P19) state that once a signal is turned on, it remains on until the header moves. This prevents signals from being turned on and off repeatedly while the header of the message is still waiting in the row or in the column register. Finally, properties (P20) and (P21) express the fact that signals are turned off at the same time the tail of the message moves through the switch.

The movement of the packets inside the router is now described by properties (P11.2.1)–(P11.2.7). As in the previous specification, packets can only move to the next location (property (P11.2.1)). Properties (P11.2.2) and (P11.2.3) specify the movement of the packets along the rows. The former

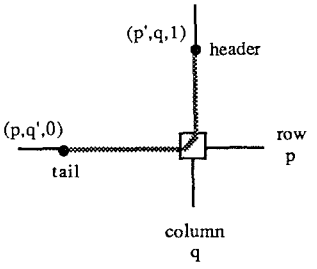


Fig. 7. The *turn* is on when a message is in transit from the row to the column.

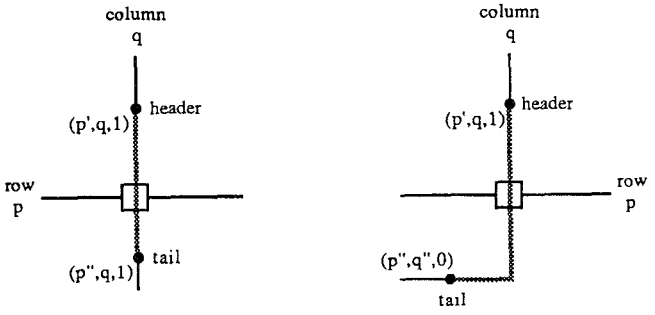


Fig. 8. The *up* signal is on when a message is in transit through the switch along the column.

states that a header packet at location $(p, q, 0)$, with q different from the destination column, eventually moves to the next location on the row. The latter states that the body and tail packets showing up at location $(p, q, 0)$ eventually move to the next location on the row if the signal *turn* is off, i.e., if it has not been turned on by the header of the message. Properties (P11.2.4) and (P11.2.5) deal with the movement of the packets from the row to the column. The *turn* signal in switch (p, q) is eventually turned on after the arrival of a header packet whose destination column is q (property (P11.2.4)). Once the signal has been turned on, packets are allowed to move to location $(p + 1, q, 1)$ on the column (property (P11.2.5)). Properties (P11.2.6) and (P11.2.7) are symmetric to (P11.2.4) and (P11.2.5) for the *up* signal.

PROOF OBLIGATIONS. The only change brought to specification S_1 is the refinement of (P11.2) into (P11.2.1)–(P11.2.7). So we only need to show that S_2 implies (P11.2).

PROOF OUTLINE. The **unless** part of (P11.2) is equivalent to (P11.2.1). To prove the **leads-to** part, we need to show that every packet inside the router moves eventually to the next location. The movement on the rows is implied by properties (P11.2.2), (P11.2.3), and invariant (P13). Property (P11.2.2) states directly that the header of each message will eventually move to the next location. The movement of the remaining packets of the message is guaranteed by (P11.2.3) when the *turn* signal is off, which is implied by invariant (P13). The movement from the row to the column is implied by

properties (P11.2.4), (P11.2.5), and invariant (P14). The movement of the header packets follows from the transitivity of **leads-to** applied to (P11.2.4) and (P11.2.5). The movement of the nonheader packets is guaranteed by (P11.2.5) when the *turn* signal is on, which is implied by (P14). The proof of the movement on the columns follows similarly from (P11.2.6), (P11.2.7), and (P15).

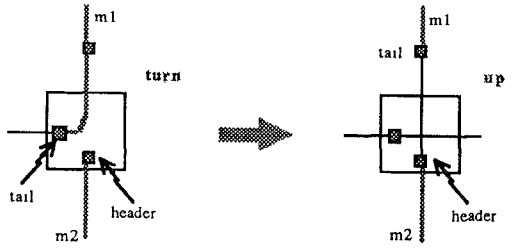
Discussion. This refinement moves the design process to the next logical level in the router's structure, the internal logic of the individual switches. Since the switch behavior is actually controlled by the flow of packets, it comes to no surprise that the refinement consists mainly of safety conditions which establish what the switch must do upon the arrival and departure of various kinds of packets. These new properties are simply added on to the earlier specification. The requirement that messages move along toward their destination remains unaltered, but knowledge of the switch logic permits us to break down the packet movement requirement into distinct cases that differentiate among packet types, their intended destinations, and their location in the two switch registers. While this latter aspect of Refinement 2 could be postponed for a later step, its presence here is motivated by the need to understand (and explain to others) the implications of the switch logic and to involve it in a meaningful verification step capable of revealing possible logical errors.

3.3 Refinement 3: Introduction of a Fairness Constraint

At this point in the design, packets are guaranteed to move through the switches without being interleaved on the columns, but no property in the specification constrains the arbitration elements to behave fairly. Consider for instance the case where n consecutive messages going to the same destination q have been sent by the same sender p. Suppose that when the first of these messages arrives at switch (p, q), another message is waiting on the column for moving up through the switch. Then a possible scenario is to have the n messages on the row moving through the switch up to the column, before the message on the column is allowed to do so, thus blocking all the messages behind it. A symmetric problem occurs when a message is waiting on the row, while several messages are passing through the switch along the column.

We want to prevent such undesirable behaviors by imposing a strong fairness constraint on the arbitration elements. No more than one message must pass through a switch from the row to the column (along the column), while another message is waiting on the column (on the row). Figures 9 and 10 describe a simple mechanism implementing this requirement. In the first case—a message m_2 is waiting on the column while another message m_1 is moving from the row to the column (Figure 9)—the solution is to turn the *up* signal on as the tail of m_1 is passing through the switch. Since the specification guarantees that the *up* signal cannot be turned off before the head of m_2 moves (property (P21)), no other message will be able to move from the row to the column.

Fig. 9 No more than one message can move from the row to the column, while another one is waiting on the column.



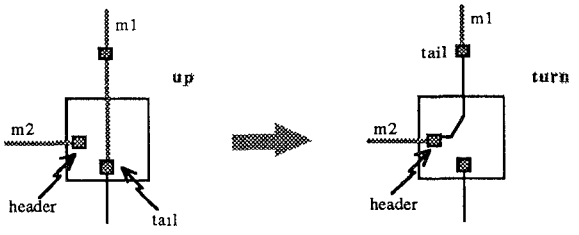


Fig. 10. No more than one message can move along the column, while another one is waiting.

In the second case—a message m_2 is waiting on the row while another message m_1 is moving along the column (Figure 10)—the solution is to turn the *turn* signal on as the tail of m_1 is passing through the switch.

Formally, this refinement entails the addition of two new properties describing the preceding behaviors. The refined specification S_3 is:

Message Representation
Properties (P1)–(P4).

Message Location
Properties (P5.1) and (P6.1).

No-Reordering and Noninterleaving Properties
Properties (P9.1) and (P10.1).

Signal Properties
Properties (P12)–(P21).

$$\alpha^t@(p, q, 0) \wedge q \geq 1 \wedge \text{turn}(p, q) \wedge \beta^h@(p, q, 1) \quad \textbf{unless} \quad \neg\, \alpha^t@(p, q, 0) \wedge up(p, q) \tag{P22}$$

$$\alpha^t@(p, q, 1) \wedge p \leq N \wedge \text{up}(p, q) \wedge \beta^h@(p, q, 0) \wedge q = \text{dest}.\beta \quad \textbf{unless} \quad \neg\, \alpha^t@(p, q, 1) \wedge \text{turn}(p, q) \tag{P23}$$

Packet Movement in the Environment
Property (P11.1).

Packet Movement Inside the Router
Properties (P11.2.1)–(P11.2.7).

Discussion. This refinement encapsulates a subtle technical and specialized issue which is better kept separate from other design decisions. Earlier

specifications ruled out any possibility of starvation but never stated how long a message may have to wait at a switch. Refinement 3 places a tight bound on the blocking time, the time it takes for all the packets of the blocking message to pass through. Because of the noninterleaving requirement no tighter bound is possible. Of course, since this requirement only adds constraints to the existing specification no proof of correctness is necessary.

3.4 Refinement 4: Refinement of the Header Values

A shortcoming of the design at this point is that switches need to know their location in the grid to route the packets. This knowledge appears in properties (ℙ11.2.2) and (P11.2.4), which state that when a header packet shows up in a row register, the switch either routes the packet to the next location on the row if *the number of the column* is different from the packet destination ((P11.2.2)), or sets the *turn* signal if *the number of the column* is equal to the packet destination ((P11.2.4)).

The purpose of the fourth refinement is to eliminate this knowledge by making the value of each header decrease by one, each time the packet moves through a switch along the row—this is possible only because we assumed that destinations are designated by column numbers. Since the value is initially equal to the number of the destination column, switches can now make the decision to route the header packets to the next location on the row when the header value is different from 1, and to set the *turn* signal when the header value is equal to 1. The location knowledge is not needed anymore. The refined specification S_4 is:

Message Representation
Properties (P1)–(P4).

Message Location
Properties (P5.1) and (P6.1).

No-Reordering and Noninterleaving Properties
Properties (P9.1) and (P10.1).

Signal Properties
Properties (P12)–(P23).

Header Value Invariant
$$\textbf{inv}\ \sigma[v]^h@(p, q, r) \wedge q \geq 1 \Rightarrow v = \text{dest}.\sigma[v] - q + 1 \qquad \text{(P24)}$$

Packet Movement in the Environment
Property (P11.1).

Packet Movement Inside the Router
Property (P11.2.1).

$$\sigma[v]^h@(p, q, 0) \wedge q \geq 1 \wedge v \neq 1 \mapsto \sigma[v - 1]@(p, q + 1, 0) \qquad \text{(P11.2.2.1)}$$

$$\sigma[v]^\tau@(p, q, 0) \wedge \tau \neq h \wedge q \geq 1 \wedge \neg\text{turn}(p, q) \mapsto \sigma[v]@(p, q + 1, 0) \qquad \text{(P11.2.3.1)}$$

$$\sigma[v]^h@(p, q, 0) \wedge v = 1 \mapsto \sigma[v]@(p, q, 0) \wedge \text{turn}(p, q) \qquad \text{(P11.2.4.1)}$$

$$\sigma[v]@(p, q, 0) \wedge q \geq 1 \wedge \text{turn}(p, q) \mapsto \sigma[v]@(p + 1, q, 1) \quad \text{(P11.2.5.1)}$$
$$\sigma[v]^h@(p, q, 1) \wedge p \leq N \mapsto \sigma[v]@(p, q, 1) \wedge \text{up}(p, q) \quad \text{(P11.2.6.1)}$$
$$\sigma[v]@(p, q, 1) \wedge p \leq N \wedge \text{up}(p, q) \mapsto \sigma[v]@(p + 1, q, 1) \quad \text{(P11.2.7.1)}$$

The value of the header packets is constrained by the invariant (P24) which states that, along the rows inside the router, the value is decreased by one with each move, and that it remains constant once the packet has reached the destination column. The movement of the packets inside the router is refined by properties (P11.2.2.1)–(P11.2.7.1). The fact that we know now how the values in the header and tail packets are being used results in the removal of the earlier existential quantification in the right-hand side.

PROOF OBLIGATIONS. We need to show that properties (P11.2.1)–(P11.2.7) are implied by S_4.

PROOF OUTLINE. The proof is straightforward. Properties (P11.2.3), (P11.2.5), (P11.2.6), and (P11.2.7) are directly implied by the corresponding refined properties (P11.2.3.1), (P11.2.5.1), (P11.2.6.1), and (P11.2.7.1). To prove that (P11.2.2) and (P11.2.4) are implied by (P11.2.2.1) and (P11.2.4.1), we only need to show

$$\textbf{inv}\ \sigma[v]^h@(p, q, 0) \wedge q \geq 1 \Rightarrow (q = \text{dest}.\sigma[v] \Leftrightarrow v = 1)$$

which is implied by the invariant (P24).

Discussion. This refinement is motivated by a strong engineering consideration: the desire to simplify the complexity of the router's circuitry by eliminating the need for the individual switches to store information about their position in the router. Such knowledge would require additional storage and would increase manufacturing costs. It is not difficult to argue that this is the proper time to address this issue—having established the pattern of movement of packets through the switch, it is only natural to reexamine the logic used to determine that a message arrived at the destination column. The idea of modifying the header is well known among switch designers, and it was natural for us to use it. What may be surprising to some of the readers is the fact that after several fairly detailed refinements the option to employ this design strategy was still present in the specification. Design experience suggested to us from the very start that changes in the header and tail packets ought to be permitted, and this notion made its way into the initial specification. In addition, careful attention to separation of concerns and a modular design philosophy kept considerations regarding the arrival to the destination column separate from the other aspects of the routing logic during the refinement process.

3.5 Refinement 5: Switches Work in an Asynchronous Way

The last design decision concerns the execution control we impose on the switches. Between the two possible choices—either synchronous behavior or asynchronous behavior—we chose the more realistic asynchronous behavior. Since each location can contain at most one packet, this means that, before routing a packet to the next location, a switch must first check whether this location is empty. Note that, in the case of a synchronous behavior, this

constraint would not have been needed, since two consecutive packets have to move synchronously in this case.

Let us define the predicate *empty*@(p, q, r) to mean that location (p, q, r) does not contain any packet

$$\text{empty@}(p, q, r) \equiv \langle \forall \alpha :: \neg\, \alpha@(p, q, r) \rangle.$$

A way to specify the asynchronous behavior is to state that each packet inside the router must leave an empty location behind it when it moves

$$\alpha@(p, q, r) \wedge p \leq N \wedge q \geq 1 \textbf{ unless } \text{empty@}(p, q, r).$$

This forces the switches to wait for the next location to be emptied, before routing a packet. The refined specification S_5 contains all the properties of S_4 plus the property above. The proof of the refinement is therefore straightforward.

Message Representation
Properties (P1)–(P4).

Message Location
Properties (P5.1) and (P6.1).

No-Reordering and Noninterleaving Properties
Properties (P9.1) and (P10.1).

Signal Properties
Properties (P12)–(P23).

Header Value Invariant
Property (P24).

Packet Movement in the Environment
Property (P11.1).

Packet Movement Inside the Router
Property (P11.2.1).
Properties (P11.2.2.1)–(P11.2.7.1).
$\alpha@(p, q, r) \wedge p \leq N \wedge q \geq 1$ **unless** empty@(p, q, r). (P25)

Discussion. The decision to adopt an asynchronous packet movement strategy is again motivated by the desire to keep the circuitry simple. A single bit of information is needed to signal that the next location can receive a new packet. A router capable of moving packets synchronously along a single path would entail a complicated design since knowledge that the header packet can advance would have to be transmitted to all the packets that follow it. It should be noted, however, that asynchrony applies only to the movement of packets while the underlying hardware can still be synchronous in its behavior.

3.6 Refinement 6: Transformation of the Leads-to Properties into Ensures Properties

This last refinement is not motivated by a design decision. The motivation here is to transform the specification into a form that can be directly

translated into program text, i.e., introduce **ensures** properties expressing atomic transformations. The progress properties are property (P11.1) specifying the packet movement in the environment, and properties (P11.2.2.1)–(P11.2.7.1) expressing the packet movement inside the router. Since we are only interested in the design of the message router, we will not transform the environment property (P11.1) into an **ensures** property. We will just make sure that it is satisfied when deriving the program. However, to make it more explicit, we can split it into two properties, one for the input queues and the other one for the output queues

$$\sigma[v]@(p, q, 0) \wedge q \leq 0 \textbf{ until } \sigma[v]@(p, q+1, 0) \quad \text{(P11.1.1)}$$
$$\sigma[v]@(p, q, 1) \wedge p \geq N+1 \textbf{ until } \sigma[v]@(p+1, q, 1). \quad \text{(P11.1.2)}$$

Let us now focus on properties (P11.2.2.1)–(P11.2.7.1). The simplest way to transform a **leads-to** property into an **ensures** property is to directly replace $\mapsto$ by **ensures**, and to see if the resulted property can be satisfied in an atomic transformation. By applying this method on (P11.2.2.1), we get

$$\sigma[v]^h@(p, q, 0) \wedge q \geq 1 \wedge v \neq 1 \textbf{ ensures } \sigma[v-1]@(p, q+1, 0).$$

This property cannot be satisfied in an atomic transformation since the next location may be busy at the time. This suggests the following property:

$$\sigma[v]^h@(p, q, 0) \wedge q \geq 1 \wedge v \neq 1 \wedge \text{empty}@(p, q+1, 0)$$
$$\textbf{ensures } \sigma[v-1]@(p, q+1, 0) \quad \text{(P11.2.2.1.1)}$$

which can easily be satisfied in an atomic step. In the same way, we get from (P11.2.3.1)

$$\sigma[v]^\tau@(p, q, 0) \wedge \tau \neq h \wedge q \geq 1 \wedge \neg\text{turn}(p, q) \wedge \text{empty}@(p, q+1, 0)$$
$$\textbf{ensures } \sigma[v]@(p, q+1, 0). \quad \text{(P11.2.3.1.1)}$$

The transformation of the properties (P11.2.5.1) and (P11.2.7.1) is slightly more complicated. We need to separate the cases where packets stay inside the router ($p < N$) or move from the router to the output environment ($p = N$). In the former case, we can transform the properties in the same way we did before. We get

$$\sigma[v]@(p, q, 0) \wedge q \geq 1 \wedge p < N \wedge \text{turn}(p, q) \wedge \text{empty}@(p+1, q, 1)$$
$$\textbf{ensures } \sigma[v]@(p+1, q, 1) \quad \text{(P11.2.5.1.1)}$$

and

$$\sigma[v]@(p, q, 1) \wedge p < N \wedge \text{up}(p, q) \wedge \text{empty}@(p+1, q, 1)$$
$$\textbf{ensures } \sigma[v]@(p+1, q, 1). \quad \text{(P11.2.7.1.1)}$$

Since we do not require that locations outside the router be emptied before packets move, properties (P11.2.5.1) and (P11.2.7.1) with p equal to N, can simply be transformed into

$$\sigma[v]@(N, q, 0) \wedge q \geq 1 \wedge \text{turn}(N, q)$$
$$\textbf{ensures } \sigma[v]@(N+1, q, 1) \quad \text{(P11.2.5.1.2)}$$

and

$$\sigma[v]@(N, q, 1) \wedge \mathrm{up}(N, q) \textbf{ ensures } \sigma[v]@(N+1, q, 1). \qquad \text{(P11.2.7.1.2)}$$

This implies that packets moving out of the router are allowed to push forward simultaneously all the packets in the output queues.

Let us now concentrate on property (P11.2.4.1). A direct transformation into an **ensures** property

$$\sigma[v]^h@(p, q, 0) \wedge v = 1 \textbf{ ensures } \sigma[v]@(p, q, 0) \wedge \mathrm{turn}(p, q)$$

is not possible because the *turn* signal cannot be set when a message is currently passing through the switch along the column, i.e., when the *up* signal is on. This suggests the following transformation

$$\sigma[v]^h@(p, q, 0) \wedge v = 1 \wedge \neg \mathrm{up}(p, q)$$
$$\textbf{ensures } \sigma[v]@(p, q, 0) \wedge \mathrm{turn}(p, q).$$

This is however not yet satisfactory. Consider the case: where a header packet with value 1 is in the row register; the *up* and *turn* signals are off; and another header packet is in the column register. It is easy to see that the *up* signal cannot be turned on without invalidating the **unless** part of the previous property. This means that we need to find a mechanism for preventing the *up* signal to be turned on in this case. Even though it is possible to do it, we do not want to establish any priority between the two signals, in order to have a nondeterministic behavior. The solution to this problem is to transform (P11.2.4.1) in the following way:

$$\sigma[v]^h@(p, q, 0) \wedge v = 1 \wedge \neg \mathrm{up}(p, q)$$
$$\textbf{ensures } (\sigma[v]@(p, q, 0) \wedge \mathrm{turn}(p, q)) \vee \mathrm{up}(p, q) \qquad \text{(P11.2.4.1.1)}$$

which allows the *up* signal to be turned on. A similar transformation on (P11.2.6.1) leads to

$$\sigma[v]^h@(p, q, 1) \wedge p \leq N \wedge \neg \mathrm{turn}(p, q)$$
$$\textbf{ensures } (\sigma[v]@(p, q, 1) \wedge \mathrm{up}(p, q)) \vee \mathrm{turn}(p, q). \qquad \text{(P11.2.6.1.1)}$$

The final and complete specification S_6 is:

Message Representation

$$\textbf{inv } \langle \exists \lambda :: \alpha@\lambda \rangle \wedge k = \langle \Sigma\, \beta, \lambda : \beta@\lambda \wedge \mathrm{mid}.\beta = \mathrm{mid}.\alpha :: 1 \rangle$$
$$\Rightarrow k \geq 3 \wedge (\mathrm{pnr}.\alpha = 1 \Leftrightarrow \mathrm{type}.\alpha = h) \wedge \qquad \text{(P1)}$$
$$(1 < \mathrm{pnr}.\alpha < k \Leftrightarrow \mathrm{type}.\alpha = b) \wedge (\mathrm{pnr}.\alpha = k \Leftrightarrow \mathrm{type}.\alpha = t)$$
$$\textbf{const } \langle \exists v, \lambda :: \sigma[v]^h@\lambda \rangle \qquad \text{(P2)}$$
$$\textbf{const } \langle \exists \lambda :: \sigma[v]^b@\lambda \rangle \qquad \text{(P3)}$$
$$\textbf{const } \langle \exists v, \lambda :: \sigma[v]^t@\lambda \rangle \qquad \text{(P4)}$$

Message Location

$$\textbf{inv } \alpha@(p, q, r) \wedge \beta@(p', q', r')$$
$$\Rightarrow (\mathrm{pid}.\alpha = \mathrm{pid}.\beta \Leftrightarrow (p, q, r) = (p', q', r')) \qquad \text{(P5.1)}$$
$$\textbf{inv } \alpha@(p, q, r) \Rightarrow (p = \mathrm{src}.\alpha \wedge q \leq \mathrm{dest}.\alpha \wedge r = 0) \vee$$
$$(p > \mathrm{src}.\alpha \wedge q = \mathrm{dest}.\alpha \wedge r = 1) \qquad \text{(P6.1)}$$

No-Reordering Property
inv $\alpha \prec \beta \Rightarrow \text{pid}.\beta < \text{pid}.\alpha$ (P9.1)

Noninterleaving Property
inv $\alpha@(p, q) \wedge \beta@(p', q) \wedge \delta@(p'', q) \wedge$
$N + 1 \leq p < p' < p'' \wedge \text{mid}.\alpha = \text{mid}.\delta$
$\Rightarrow \text{mid}.\beta = \text{mid}.\alpha$ (P10.1)

Signal Properties
inv $\neg(\text{turn}(p, q) \wedge \text{up}(p, q))$ (P12)
inv $q \geq 1 \wedge \langle \exists p', q', r', q'' : q'' \leq q < q' ::$
$\text{message}.(p', q', r').(p, q'', 0)\rangle \Rightarrow \neg\text{turn}(p, q)$ (P13)
inv $q \geq 1 \wedge \langle \exists p', q' : p' > p \wedge q' \leq q ::$
$\text{message}.(p', q, 1).(p, q', 0)\rangle \Rightarrow \text{turn}(p, q)$ (P14)
inv $p \leq N \wedge \langle \exists p', p'', q'', r'' : p' > p \wedge$
$(p'' < p \vee (p'' = p \wedge r'' = 1)) ::$
$\text{message}.(p', q, 1).(p'', q'', r'')\rangle \Rightarrow \text{up}(p, q)$ (P15)
inv $\text{turn}(p, q) \Rightarrow$
$\langle \exists p', q', r' : ((r' = 0 \wedge p' = p) \vee (r' = 1 \wedge p' > p)) \wedge q' \leq q ::$
$\text{message}.(p', q, r').(p, q', 0)\rangle$ (P16)
inv $\text{up}(p, q) \Rightarrow$
$\langle \exists p', p'', q'', r'' : p' \geq p \wedge (p'' < p \vee (p'' = p \wedge r'' = 1)) ::$
$\text{message}.(p', q, 1).(p'', q'', r'')\rangle$ (P17)
$\alpha^h@(p, q, 0) \wedge q = \text{dest}.\alpha \wedge \text{turn}(p, q)$ **unless** $\neg\alpha@(p, q, 0)$ (P18)
$\alpha^h@(p, q, 1) \wedge p \leq N \wedge \text{up}(p, q)$ **unless** $\neg\alpha@(p, q, 1)$ (P19)
$\alpha^t@(p, q, 0) \wedge q \geq 1 \wedge \text{turn}(p, q)$
unless $\neg\alpha@(p, q, 0) \wedge \neg\text{turn}(p, q)$ (P20)
$\alpha^t@(p, q, 1) \wedge p \leq N \wedge \text{up}(p, q)$
R**unless** $\neg\alpha@(p, q, 1) \wedge \neg\text{up}(p, q)$ (P21)
$\alpha^t@(p, q, 0) \wedge q \geq 1 \wedge \text{turn}(p, q) \wedge \beta^h@(p, q, 1)$
unless $\neg\alpha@(p, q, 0) \wedge \text{up}(p, q)$ (P22)
$\alpha^t@(p, q, 1) \wedge p \leq N \wedge \text{up}(p, q) \wedge \beta^h@(p, q, 0) \wedge q = \text{dest}.\beta$
unless $\neg\alpha@(p, q, 1) \wedge \text{turn}(p, q)$ (23)

Header Value Invariant
inv $\sigma[v]^h@(p, q, r) \wedge q \geq 1 \Rightarrow v = \text{dest}.\sigma[v] - q + 1$ (P24)

Packet Movement in the Environment
$\sigma[v]@(p, q, 0) \wedge q \leq 0$ **until** $\sigma[v]@(p, q + 1, 0)$ (P11.1.1)
$\sigma[v]@(p, q, 1) \wedge p \geq N + 1$ **until** $\sigma[v]@(p + 1, q, 1)$. (P11.1.2)

Packet Movement Inside the Router
$\sigma[v]@(p, q, r) \wedge p \leq N \wedge q \geq 1$
unless $\langle \exists v', p', q', r' : (p, q, r) \gg (p', q', r') ::$
$\sigma[v']@(p', q', r')\rangle$ (P11.2.1)
$\sigma[v]@(p, q, r) \wedge p \leq N \wedge q \geq 1$ **unless** $\text{empty}@(p, q, r)$ (P25)
$\sigma[v]^h@(p, q, 0) \wedge q \geq 1 \wedge v \neq 1 \wedge \text{empty}@(p, q + 1, 0)$
ensures $\sigma[v]@(p, q + 1, 0)$ (P11.2.2.1.1)

$\alpha[v]^{\tau}@(p, q, 0) \wedge \tau \neq h \wedge q \geq 1 \wedge$
$\neg \text{turn}(p, q) \wedge \text{empty}@(p, q + 1, 0)$
ensures $\sigma[v]@(p, q + 1, 0)$ (P11.2.3.1.1)

$\sigma[v]^{h}@(p, q, 0) \wedge v = 1 \wedge \neg \text{up}(p, q)$
ensures $(\sigma[v]@(p, q, 0) \wedge \text{turn}(p, q)) \vee \text{up}(p, q)$ (P11.2.4.1.1)

$\sigma[v]@(p, q, 0) \wedge q \geq 1 \wedge p < N \wedge \text{turn}(p, q) \wedge \text{empty}@(p + 1, q, 1)$
ensures $\sigma[v]@(p + 1, q, 1)$ (P11.2.5.1.1)

$\sigma[v]@(N, q, 0) \wedge q \geq 1 \wedge \text{turn}(N, q)$
ensures $\sigma[v]@(N + 1, q, 1)$ (P11.2.5.1.2)

$\sigma[v]^{h}@(p, q, 1) \wedge p \leq N \wedge \neg \text{turn}(p, q)$
ensures $(\sigma[v]@(p, q, 1) \wedge \text{up}(p, q)) \vee \text{turn}(p, q)$ (P11.2.6.1.1)

$\sigma[v]@(p, q, 1) \wedge p < N \wedge \text{up}(p, q) \wedge \text{empty}@(p + 1, q, 1)$
ensures $\sigma[v]@(p + 1, q, 1)$ (P11.2.7.1.1)

$\sigma[v]@(N, q, 1) \wedge \text{up}(N, q)$ **ensures** $\sigma[v]@(N + 1, q, 1)$ (P11.2.7.1.2)

PROOF OBLIGATIONS. We need to show that properties (P11.2.1.1)–(P11.2.7.1) are implied by S_6.

PROOF OUTLINE. The proof that packets inside the router move eventually to the next location—properties (P11.2.2.1), (P11.2.3.1), (P11.2.5.1), and (P11.2.7.1)—can be decomposed into three cases: packet movement along the columns, from the rows to the columns, and finally along the rows. The movement along the columns can be proved by induction on the row number. The base case is established by the property (P11.2.7.1.2) which directly implies the packet movement from the upper locations $(N, q, 1)$ to the output queues. Since a packet leaves an empty location behind it when it moves, the base case and the property (P11.2.5.1.1) allows us to prove that packets at locations $(N - 1, q, 1)$ will move to locations $(N, q, 1)$, and so on, down to row 1.

Packet movement from the row N to the output environment is directly implied by (P11.2.5.1.2). For p between 1 and $N - 1$, we have just proved that packets along the columns are guaranteed to move to the next location. This implies that locations along the columns will eventually be emptied, and thus, according to (P11.2.5.1.1), that packets at their turning switches will eventually move to their destination columns.

The packet movement along the rows can be proved by induction on the column number. We know that packets at the right end of the rows (locations $(p, M, 0)$) are necessarily at their turning switches. They will thus move to the columns and leave empty locations behind them. This will allow packets at locations $(p, M - 1, 0)$ to move to locations $(p, M, 0)$ (properties (P11.2.2.1.1) and (P11.2.3.1.1)), and so on, down to column 1.

Finally, we need to prove that the proper signals are eventually turned on when header packets show up in the registers (properties (P11.2.4.1) and (P11.2.6.1)). From (P11.2.4.1.1) we know that the *turn* signal will eventually be turned on, unless the *up* signal is turned on. In this case, the fairness property (P23) assures that the *turn* signal will also be turned on, as the tail of the message moving along the column will pass through the switch. The proof of (P11.2.6.1) is symmetrical.

Discussion. All previous refinements involved design activities which were germane to the router design and not specific to the formal method being applied. Of course, the manner in which the design decisions were formally captured was UNITY logic specific. Refinement 6 is the first one not to be tied directly to the traditional engineering process. The refinement is needed for technical reasons specific to the formal method we are employing. While this step cannot be eliminated, the overhead it adds is minimal and limited to a specific point in the application of the method. In some sense, this refinement is a test to verify that individual progress conditions do actually have statement-level counterparts. If this were not the case, additional refinements would be required. The reason progress conditions associated with the environment are not subject to the same scrutiny is because they are not to be implemented as such—they merely reflect obligations the environment must meet.

4. DERIVATION OF A PROGRAM FROM THE SPECIFICATION

The final step of the design consists of writing the program text. We first define the types and data structures used in the program, and then derive the program statements from the progress properties of the final specification.

4.1 Types and Data Structures of the Program

We define the type *packet* to be the Cartesian product between $\{h, b, t\}$ and the set V of all the possible packet values. If p is a variable of type *packet*, we use the notation $p.1$ to access the type, and $p.2$ to access the value. The grid of $N \times M$ switches is implemented by a three-dimensional array

$$\text{switch} : \text{array}[1..N, 1..M, 0..1] \text{ of packet} \cup \{\perp\}$$

such that $switch[p, q, 0]$ represents the row register of switch (p, q) and $switch[p, q, 1]$ the column register. When a register does not contain any packet, we assume that its value is equal to $\perp$. We define functions *header*, *body*, *tail*, *empty*, and *val* in the following way:

$$\begin{aligned}
\text{header}.(p, q, r) &\equiv (\text{switch}[p, q, r].1 = h),\\
\text{body}.(p, q, r) &\equiv (\text{switch}[p, q, r].1 = b),\\
\text{tail}.(p, q, r) &\equiv (\text{switch}[p, q, r].1 = t),\\
\text{empty}.(p, q, r) &\equiv (\text{switch}[p, q, r] = \perp),\\
\text{val}.(p, q, r) &\equiv \text{switch}[p, q, r].2.
\end{aligned}$$

We also define the function *dec_val* which accepts a header packet (h, v) as argument and returns the packet $(h, v - 1)$. The switch signals *turn* and *up* are implemented by two Boolean arrays

$$\text{turn}, \text{up} : \text{array}[1..N, 1..M] \text{ of Boolean},$$

and the N input queues and M output queues are represented by two arrays of sequences of packets

$$\text{input} : \text{array}[1..N] \text{ of sequence of packet},$$
$$\text{output} : \text{array}[1..M] \text{ of sequence of packet}.$$

We suppose that the output queues are initially empty, and that the input queues contain all the packets that have to be sent. Finally, we use the following operations on sequences:

$$hd.s \equiv \text{head element of the sequence } s,$$
$$tl.s \equiv \text{tail sequence of the sequence } s,$$
$$(s;\, x) \equiv \text{sequence obtained by appending the element } x \text{ at the end of the sequence } s,$$
$$nil.s \equiv s \text{ is an empty sequence.}$$

4.2 Program Text

Let us first briefly describe the UNITY program notation. As we stated at the beginning of the article, a typical UNITY program consists of a *declare* section, which contains Pascal-style declarations of variables; an *initially* section, where all or some of the variables are initialized; and an *assign* section, which is a set of multiple assignment statements separated by the operator ▯. The statements may be of the form

$$\text{var\_list} := \text{exp\_list}$$

or may be conditional

$$\text{var\_list}_1 := \text{exp\_list}_1 \textbf{ if } \text{bexp}_1 \sim \text{exp\_list}_2 \textbf{ if } \text{bexp}_2 \sim \ldots$$

Several statements may be composed with the parallel bar to form a bigger statement

$$\text{statement}_1 \| \text{statement}_2 \| \ldots$$

Finally, it is possible to generate a list of statements by using the following constructor

$$\langle ▯\ \text{dummy\_variables} : \text{range\_constraint} :: \text{statement} \rangle.$$

The program derived from the specification is as follows (we omitted the *declare* section):

Program Message Router
initially
$\langle ▯\ p, q : 1 \le p \le N \wedge 1 \le q \le M ::$
switch$[p, q, 0]$, switch$[p, q, 1]$, turn$[p, q]$, up$[p, q] :=$
$\perp, \perp$, false, false$\rangle$
assign
{*Packet movement from the input to the router: property* (P11.1.1)}
$\langle ▯\ p : 1 \le p \le N :: \text{switch}[p, 1, 0], \text{input}[p] :=$
$hd.\text{input}[p], tl.\text{input}[p]$
if empty.$(p, 1, 0) \wedge \neg\, nil.\text{input}[p]\rangle$

```
[] {Packet movement along the rows: properties (P11.2.2.1.1) and
   (P11.2.3.1.1)}
   ⟨[] p, q: 1 ≤ p ≤ N ∧ 1 ≤ q < M ::
       switch[p, q, 0], switch[p, q + 1, 0] :=
          ⊥ , dec_val.switch[p, q, 0]
          if header.(p, q, 0) ∧ val.(p, q, 0) ≠ 1 ∧ empty.(p, q + 1, 0) ~
          ⊥ , switch[p, q, 0]
          if (body.(p, q, 0) ∨ tail.(p, q, 0)) ∧
          ¬turn[p, q] ∧ empty.(p, q + 1, 0)⟩
[] {Packet movement from the rows (< N) to the columns: properties
   (P11.2.5.1.1), (P20), and (P22)}
   ⟨[] p, q : 1 ≤ p < N ∧ 1 ≤ q ≤ M ::
       switch[p, q, 0], switch[p + 1, q, 1] :=
       ⊥ , switch[p, q, 0]
       if turn[p, q] ∧ ¬empty.(p, q, 0) ∧ empty.(p + 1, q, 1)
    || turn[p, q] := false
       if turn[p, q] ∧ tail.(p, q, 0) ∧ empty.(p + 1, q, 1)
    || up[p, q] := true
       if turn[p, q] ∧ tail.(p, q, 0) ∧ empty.(p + 1, q, 1) ∧
       header.(p, q, 1)⟩
[] {Packet movement from row N to the output: properties (P11.2.5.1.2),
   (P20), and (P22)}
   {[] q : 1 ≤ q ≤ M ::
       switch[N, q, 0], output[q] := ⊥ , (output[q]; switch[N, q, 0])
       if turn[N, q] ∧ ¬empty(N, q, 0)
    || turn[N, q] := false if turn[N, q] ∧ tail.(N, q, 0)
    || up[N, q] := true
       if turn[N, q] ∧ tail.(N, q, 0) ∧ header.(N, q, 1)⟩
[] {Packet movement along the columns inside the router: properties
   (P11.2.7.1.1), (P21), and (P23)}
   {[] p, q : 1 ≤ p < N ∧ 1 ≤ q ≤ M ::
       switch[p, q, 1], switch[p + 1, q, 1] := ⊥ , switch[p, q, 1]
       if up[p, q] ∧ ¬empty.(p, q, 1) ∧ empty.(p + 1, q, 1)
    || up[p, q] := false
       if up[p, q] ∧ tail,(p, q, 1) ∧ empty.(p + 1, q, 1)
    || turn[p, q] := true
       if up[p, q] ∧tail.(p, q, 1) ∧ empty.(p + 1, q, 1) ∧
       header.(p, q, 0) ∧ val(p, q, 0) = 1⟩
[] {Packet movement from the columns to the output: properties
   (P11.2.7.1.2), (P21), and (P23))}
   ⟨[] q : 1 ≤ q ≤ M ::
       switch[N, q, 1], output[q] := ⊥ , (output[q]; switch[N, q, 1]
       if up[N, q] ∧ ¬empty.(N, q, 1)
    || up[N, q] := false if up[N, q] ∧ tail.(N, q, 1)
    || turn[N, q] := true
       if up[N, q] ∧ tail.(N, q, 1) ∧ header.(N, q, 0) ∧
       val.(N, q, 0) = 1⟩
```

```
[]  {Signal changes: Properties (P11.2.4.1.1) and (P11.2.6.1.1)]
    ⟨[] p, q : 1 ≤ p ≤ N ∧ 1 ≤ q ≤ M :: turn[ p, q ] := true
      if ¬turn[ p, q ] ∧ ¬up[ p, q ] ∧ header.( p, q, 0) ∧ val.( p, q, 0) = 1⟩
[]  ⟨[] p, q : 1 ≤ p ≤ N ∧ 1 ≤ q ≤ M ::
      up[ p, q ] := true
      if ¬up[ p, q ] ∧ ¬turn[ p, q ] ∧ header.( p, q, 1)⟩
end
```

Initially, all the row and column registers are empty, and the *turn* and *up* signals are off. The packet movement in the input queues (property (P11.1.1)) is implemented by the first statement of the *assign* section. As long as there exists an input queue that is not empty, its head element is removed and assigned to the first register on the row, if it is empty. This implies that all the packets in the queue simultaneously move forward one position.

The second statement takes care of the packet movement along the rows. It was trivially derived from the properties (P11.2.2.1.1) and (P11.2.3.1.1).

The movement from the lower rows (below row N) to the columns is realized by the third statement, which consists of three components. The first component was trivially derived from the progress property (P11.2.5.1.1). The second component makes sure that the *turn* signal is turned off when the tail of the message is passing through the switch. It was suggested by the safety property (P20), or more precisely by the conjunction of (P20) and (P11.2.5.1.1). The third component, which was derived from the conjunction of (P22) and (P11.2.5.1.1), makes sure that the *up* signal is turned on when the tail of the message is passing through the switch and a header packet is waiting in the column register. The packet movement along the columns inside the router (fifth statement) was derived from the properties (P11.2.7.1.1), (P21), and (P23) in a similar way.

The movement of the packets from the upper locations inside the router—either on the rows (fourth statement) or on the columns (sixth statement)—is achieved by just appending the outgoing packet at the end of the output queue. Note that this implies that packets move forward simultaneously one position in the output queue, and thus satisfies the property (P11.1.2).

Finally, the properties (P11.2.4.1.1) and (P11.2.6.1.1), specifying that the *turn* or *up* signals are turned off when a header packet shows up in the row or column registers, suggested the last two statements. To be complete, we also need to show that each statement preserves the safety and invariant properties of the specification. The proofs can be verified using the program text.

Discussion. One interesting feature of the UNITY program we generated is the fact that all the obligations imposed upon the environment are satisfied by simply treating the router's inputs and outputs as unbounded queues. As long as the input queues contain full messages, the program is correct. Moreover, correct execution is guaranteed in any environment which can be

modeled by unbounded queues. These kind of assumptions are common in the communications research and reinforce our view that adopting a closed-system model for the specifications led to no loss of generality.

For purposes of this article the derivation is complete in the sense that the basic structure and logic of the router is made explicit in the program structure and behavior. It is conceivable, however, that one may want to take the program one step further to the level of registers and wires. This can be achieved in two ways. One can either return to the specification resulting from the Refinement 5 or attempt to apply correctness-preserving transformations to the final program. The latter approach seems preferable to us. Simple mechanical transformations can be employed at this point because the program is sufficiently close to the circuit-level representation.

5. CONCLUSION

This article presents the formal derivation of a message router. The refinement method is that of UNITY. Although UNITY-style formal derivations have appeared in print before, the questions addressed by this case study are different, and the lessons learned bear careful scrutiny and further investigation. Others have been concerned with the issue of whether formal derivation can be applied successfully to sophisticated problems, such as the router. We are intrigued by the possibility that careful management of the refinement process may render formal derivation capable of supporting industrial-grade applications. Most of our effort went not into the router derivation per se but, in fine tuning the derivation process. In earlier sections we described the ultimate outcome of this study: a series of refinements and their motivation. Now we turn our attention to those elements of the derivation process that have been instrumental in shaping the specification style and the derivation strategy.

Early on we observed that it is better to specify and reason about a closed system—i.e., a system and its environment—rather than an open one. In the latter case, conditional properties make specifications more complex and proofs more difficult. In the router example, we have been able to specify a closed system by representing the input and output environments as infinite input and output queues, with the input queues initially containing all the packets that had to be sent. The unified formal treatment of the system and its environment reduced complexity without compromising the desire to separate concerns; the two types of properties were simply identified as addressing distinct issues. Naturally, only system properties were refined while the environmental properties were left unchanged. This would not be the case if one were free to alter the relation between the system and its environment. One may also conceive of situations in which the complexity of the task is such that multiple environment specifications, at different levels of abstraction, may prove to be helpful.

Another critical element is the formulation of the top-level specification. On one hand, it should be as general as possible so as not to restrict prematurely the range of possible designs; on the other hand, it ought to make the

refinements simple. The former is achievable by focusing the top-level specification on I/O properties alone while the latter is facilitated by selecting the "right" notation. For instance, by identifying packet locations in the environment as pairs of coordinates (row, column) the refinement of the router as a grid of switches became very natural. One does not stumble upon appropriate notation. Experience, exploration, and some looking ahead can provide the required insight. Derivation papers are often criticized for fortuitous early decisions which seem to indicate that the authors already have seen the final solution. Such objections are valid only if one claims that the nature of the specification dictates (in some mechanistic way) the next refinement step. True design, however, never makes such pretense. Looking ahead and backtracking are part of the method. Insights gained from considering various alternatives ultimately led—relatively early in the process—to the notation we adopted and to the choice of auxiliary variables: n (source row), m (destination column), i (message number), and j (packet number). The selection of auxiliary variables was one of the key decisions we had to make; it enabled simple formulation of many central properties such as those involving message location, no-reordering, and noninterleaving. Note, for instance, that using the destination column variable in the early stages of the design allowed us to deal with the value of the header packets only in the fourth refinement.

The scope of the individual refinements is another issue affecting the ease with which the derivation can be explained to others and verified formally. As expected, small refinements proved helpful in both respects. They also seem to help the designer avoid premature commitments. The refinement process can be viewed as a tree, where internal nodes represent specifications, and leafs represent programs. The more internal nodes, the more leafs at the bottom of the true. This conservative strategy leads to a broader exploration of alternatives and a decrease in the likelihood that not all implications of each design decision are well understood and considered.

Finally, we discovered that it was relatively easy to separate the formal treatment of the proofs from the refinement process itself. We spent a lot of investigation time on choosing the right top-level specification, the right notation and auxiliary variables, and the right sequence of refinements. During all this time, we came up with different solutions, but we hardly ever felt the need to verify formally the correctness of the refinements we generated—only at the end of the design, when all the design decisions were set, we generated the required formal proofs for each refinement step. This made us conclude that the refinement process is sufficiently intuitive to allow a rigorous design methodology to proceed without the burden of unnecessary and cumbersome proofs. This also means that design and verification can actually be carried out by different people. People with synthetic skills could focus their energy on the design while people with strong analytical skills could deal mostly with the proofs, often without even requiring an understanding of the design details. These observations strengthen our belief that formal methods may soon play an important role in the development of industrial-grade software systems.

APPENDIX

FORMAL DEFINITION OF THE UNITY LOGIC OPERATORS

The definition of the UNITY operators is based on the Hoare triple

$$\{p\}\, s\, \{q\}$$

which means that, starting in a state satisfying the precondition p, the execution of the statement s ends in a state satisfying the postcondition q. Let F be a UNITY program, *F.assign* be the assign the section of F, and *INIT* a predicate denoting the initial state of F. The UNITY operators are formally defined in the following way:

—p **unless** $q \equiv \langle \forall s : s \in F.\text{assign} :: \{p \wedge \neg q\}\, s\, \{p \vee q\} \rangle$.
Whenever the predicate p holds for a program state, it continues to hold at least until q holds.

—**stable** $p \equiv \langle \forall s : s \in F.\text{assign} :: \{p\}\, s\, \{p\} \rangle \equiv p$ **unless** false.
The predicate p is a *stable* predicate if it remains true forever once it becomes true.

—**const** $p \equiv$ **stable** $p \wedge$ **stable** $\neg p$.
The predicate p is *constant* if it remains true forever if it is initially true, and false forever if it is initially false.

—**inv** $p \equiv (\text{INIT} \Rightarrow p) \wedge$ **stable** p.
The predicate p is *invariant* if it holds in the initial state, and is stable all along the execution.

—p **ensures** $q \equiv p$ **unless** $q \wedge \langle \exists s : s \in F.\text{assign} :: \{p \wedge \neg q\}\, s\, \{q\} \rangle$.
Whenever p holds, it must hold at least until q holds (p **unless** q), and there must exist a statement in the program that establishes the truth of q.

—$p \mapsto q$.
The assertion $p \mapsto q$ (p leads to q) is true iff it can be derived by a finite number of applications of the following inference rules:

$$\frac{p \textbf{ ensures } q}{p \mapsto q}$$

$$\frac{p \mapsto q,\ q \mapsto r}{p \mapsto r} \qquad \text{(transitivity)}$$

$$\frac{\langle \forall m : m \in W :: p(m) \mapsto q \rangle}{\langle \exists m : m \in W :: p(m) \rangle \mapsto q} \quad \text{for any set } W \text{ (disjunction)}$$

ACKNOWLEDGMENTS

The authors thank H. C. Cunningham, K. C. Cox, and J. Y. Plun for their reviews of the article.

REFERENCES

BACK, R. J. R. AND SERE, K. 1990. Stepwise refinement of parallel algorithms. *Sci. Comput. Program 13,* 2-3, 133–180.

CHANDY, K. M. AND MISRA, J. 1988. *Parallel Program Design: A Foundation*. Addison-Wesley, Reading, Mass.

CUNNINGHAM, H. C. AND ROMAN, G.-C. 1990. A UNITY-style programming logic for a shared dataspace language. *IEEE Trans. Parall. Distrib. Syst. 1,* 3, 365–376.

DALLY, W. J. AND SEITZ, C. L. 1987. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Trans. Comput. 36,* 5, 547–533.

DIJKSTRA, E. D. 1976. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, N.J.

DIJKSTRA, E. W. AND FIEJEN, W. H. J. 1988. *A Method of Programming*. Addison-Wesley, Reading, Mass.

EBERGEN, J. C. AND HOOGERWOORD, R. R. 1990. A derivation of a serial-parallel multiplier. *Sci. Comput. Program. 15,* 201–215.

GRIES, D. 1981. *The Science of Programming*. Springer-Verlag, New York.

GRIES, D. AND PRINS, J. 1985. A new notion of encapsulation. *ACM SIGPLAN Not. 20,* 6, 131–139.

HOOGERWOORD, R. R. 1990. A calculational derivation of the CASOP algorithm. *Inf. Process. Lett. 36,* 297–299.

JOSEPHS, M. B., MAK, R. H., UDDING, J. T., VERHOEFF, T., AND YANTCHEV, J. T. 1992. High-level design of an asynchronous packet-routing chip. In *Designing Correct Circuits*. IFIP Transactions A: Computer Science and Technology. North Holland, Amsterdam, 261–274.

KNAPP, E. 1990. An exercise in the formal derivation of parallel programs: Maximum flows in graphs. *ACM Trans. Program. Lang. Syst. 12,* 2, 203–223.

LENGAUER, C. 1982. A methodology for programming with concurrency: The formalism. *Sci. Comput. Program. 2,* 19–52.

LENGAUER, C. AND HEHNER, E. C. R. 1982. A methodology for programming with concurrency: An informal presentation. *Sci. Comput. Program. 2,* 1–18.

MORGAN, C. C. 1988. The specification statement. *ACM Trans. Program. Lang. Syst. 10,* 403–419.

MORRIS, J. M. 1989. Laws of data refinement. *Acta Informatica 26,* 287–308.

NI, L. M. AND MCKINLEY, P. K. 1993. A survey of wormhole routing techniques in direct networks. *IEEE Comput. 3,* 2, 662–676.

ROMAN, G.-C. AND CUNNINGHAM, H. C. 1992. Reasoning about synchronic groups. In *Research Directions in High-Level Parallel Programming Languages*, J. P. Banâtre and D. L. Métayer, Eds. Springer-Verlag, New York, 21–38.

ROMAN, G.-C. AND CUNNINGHAM, H. C. 1990. Mixed programming metaphors in a shared dataspace model of concurrency. *IEEE Trans. Softw. Eng. 16,* 12, 1361–1373.

ROMAN, G.-C. AND WILCOX, C. D. 1994. Architecture-directed refinement. *IEEE Trans. Softw. Eng. 20,* 4, 239–258.

ROMAN, G.-C., GAMBLE, R. F., AND BALL, W. E. 1993. Formal derivation of rule-based programs. *IEEE Trans. Softw. Eng. 19,* 3, 227–296.

STASKAUSKAS, M. 1993. Formal derivation of concurrent programs: An example from industry. *IEEE Trans. Softw. Eng. 19,* 5, 503–528.

STASKAUSKAS, M. 1988. A formal specification and design of a distributed electronic funds-transfer network. *IEEE Trans. Comput. 37,* 12, 1515–1528.

Received December 1992; revised June 1994; accepted October 1994