Modeling Deflection Networks: Design and Specification

Nobuyuki Nezu and Huizhu Lu
Computer Science Department
Oklahoma State University
Stillwater, OK 74078

Keywords: deflection, network, routing, specification

Abstract

This paper presents a formal specification for the deflection network, which has been proposed for high speed metropolitan area communication infrastructures in recent years. The network is modeled as a closed system with unbounded I/O queues. UNITY (Unbounded Nondeterministic Iterative Transformations) logic, which is a fragment of linear temporal logic, is used to formulate the properties of the network. In the UNITY model, a system is viewed as a mathematical object; the properties of the system are expressed by using a set of logical relations. A topology independent specification that clarifies the nodal structure and processing of the deflection network is developed.

1 Introduction

Formal methods are mathematical techniques for specifying and verifying complex systems. The use of formal methods eliminates inconsistencies, ambiguities, and incompletenesses that may remain undetected with informal specifications [6].

In this paper, we develop a formal specification for deflection networks [2, 3, 5, 18, 21], which have been proposed in recent years as alternatives to bus and linear topology networks commonly used for local area communications. Deflection networks eliminate buffering resources that may become congested. The absence of congestion makes the behavior of the networks similar to that of conventional local area networks. The simplified nodal structure enables nodes to follow the link speed as close as possible. Their mesh-connected topologies increase the throughput and can support communications in a large metropolitan area.

As in the case of many proposed parallel and distributed systems, formal specifications for deflection networks have not appeared in the literature. We model a deflection network as a closed system and develop a specification for the network using the logic of UNITY (Unbounded Nondeterministic Iterative Transformations) [4, 19, 20, 23].

SAC '99, San Antonio, Texas

UNITY logic is closely related to temporal logic [15, 16, 17]. It enables us to develop descriptive, non-operational specifications for programs. A program is viewed as a mathematical object and not in terms of its possible executions. The elimination of operational reasoning makes UNITY logic a formal method.

The computational model of UNITY separates the concept of termination, which is central in traditional transformational programs, from the problem solving process. The UNITY model captures parallel and distributed programs that are ongoing (*reactive*) and nonterminating.

UNITY logic is surprisingly simple and compact. Its design decisions avoided introducing notational artifacts.

In his forward to Chandy and Misra's work of UNITY [4], Hoare states that a complete theory of programming includes methods for:

- specifying programs,
- reasoning about specifications,
- developing correct programs, and
- transforming programs for executions on available machines.

Dijkstra's work [11] provides the methods for sequential programming. Chandy and Misra's UNITY does the same for parallel and distributed programming.

The UNITY methodology has been applied to a variety of design and specification problems [8, 9, 10, 14, 22, 25, 26]. In [8, 9], formal specifications for a static (fixed routes) wormhole message router for a multiprocessor interconnection network (a grid of $N \times M$ switches, where N is the number of input lines and M is the number of output lines) are studied. The router is modeled as a closed system in [8]. Whereas, [9] attempts to model it as an open system. Our closed-system assumption in modeling deflection networks is influenced by [8].

The goals of this paper are to develop a formal specification for deflection networks and to identify methodological elements that provide a common foundation for the design and specification of data networks.

The remainder of this paper is organized as follows. Section 2 gives an overview of the UNITY computational model and logic; we re-define the operators of [19, 20], which are derived from [4], using the notion of the *strongest invariant* [23]. Section 3 describes the characteristics of deflection networks. A formal specification for deflection networks is developed in Section 4. Section 5 concludes this paper.

2 Overview of UNITY

This section gives an overview of the UNITY computational model and logic.

2.1 UNITY Computational Model

The UNITY computational model (program model) is built upon a traditional imperative foundation and a state transition system.

A UNITY program consists of a declaration/initialization of variables and a set of atomic, terminating, deterministic, guarded, multiple-assignment statements. A UNITY program has no control constructs. In each step of execution, a statement is selected nondeterministically and executed. (Executing a statement whose guard is false does not change the values of the variables.) Nondeterministic selection is constrained by the *fairness* rule, i.e., every statement is selected infinitely often.

The execution of an assignment statement corresponds to the transition from one state to the next. An execution sequence will be either infinite or end in a state in which no statement leading to another state exists, (i.e., a *fixed point* of the program is reached).

Fairness is an important hypothesis in the study of parallel programs. It guarantees that the computations exhibit all behaviors manifested by the execution of programs. In a multiprocess program, different processes will be individually allowed to proceed [1, 12, 16, 19].

2.2 UNITY Logic

The verification of a sequential program involves placing predicates at specific points; the predicates hold when the control reached the points. This is not the case for a UNITY program since UNITY does not have the notion of program control. The properties that must be satisfied are associated with the entire program.

2.2.1 Notation

Quantification A quantified expression is written in the form

$$\langle Op\ x : R(x) : T(x) \rangle,$$

where Op is an associative and commutative operator (e.g., $\wedge$, $\vee$, $+$, etc.), x is a list of dummy variables whose scopes are delimited by the angle brackets, $R(x)$ is a predicate giving the ranges of dummy variables over which the quantification is to be done, and $T(x)$ is the term of the quantification. (When $T(x)$ is a predicate, we will write $\forall$ instead of $\wedge$ and $\exists$ instead of $\vee$. Note that $R(x)$ may be omitted if the ranges (domains) of dummy variables are understood.)

Hoare Triple The *Hoare triple* [13] has the form

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

where p and q are predicates and s is a program statement. Its meaning is that if s is executed in a state where p holds, then q will hold after the execution of s.

Inference Rule An inference rule is written as

$$\frac{P}{Q},$$

where P and Q are lists of properties. Its meaning is that if P holds, then we may infer that Q to hold.

Set A set consisting of all elements x that satisfy the property P is written in the form

$$\{x \mid P\}.$$

A finite set may be specified by enumerating its elements between curly brackets. The cardinality of a finite set A is denoted by $\#A$.

Operators The operators that we use are summarized below, ordered by increasing binding power.

$$\equiv$$

$$\Leftarrow, \Rightarrow$$

$$\textbf{initially}, \textbf{co}, \textbf{stable}, \textbf{constant}, \textbf{invariant}, \textbf{transient}, \mapsto$$

$$\wedge, \vee$$

$$\neg$$

$$=, \neq, <, \geq, >, \leq, \in, \notin$$

$$+, -, \min, \max$$

"." (function application)

The definitions of operators **initially**, **co**, **stable**, **constant**, **invariant**, **transient**, and $\mapsto$ are given later in this subsection. All other operators have their usual meanings.

2.2.2 UNITY Logic Operators

We adopt the notion of the *strongest invariant* [23] and redefine the operators of [19, 20]. Note that although the operators of [19, 20] are derived from the original work of UNITY [4], they are developed for generic (discrete) *action systems*, which consist of a number of actions that may change the state of the systems, and are not specific to UNITY.

A predicate p is stronger than predicate q if $p \Rightarrow q$. The strongest invariant, denoted by SI, is the strongest predicate X that satisfies the following condition:

$$(\text{initial condition} \Rightarrow X) \wedge \langle \forall s : s \in F : \{X\}s\{X\} \rangle,$$

where F is a program (a set of program statements) and s is a program statement. The strongest invariant SI characterizes the set of states that are reachable during some execution of the program F.

- **initially** p means that p holds for the initial state of every execution sequence.

$$\textbf{initially}\ p \equiv \text{initial condition} \Rightarrow p$$

- p **co** q (p constrains q) means that if p holds for some reachable state, then q holds for the next state.

$$p\ \textbf{co}\ q \equiv \langle \forall s : s \in F : \{SI \wedge p\}s\{q\} \rangle$$

- **stable** p means that if p holds for some reachable state, then p continues to hold for all succeeding states. (In the program model, once p is established, it is preserved by every statement.)

$$\textbf{stable}\ p \equiv p\ \textbf{co}\ p$$

- **constant** p means that p is true for all reachable states if p is initially true and false for all reachable states if it is initially false.

$$\textbf{constant } p \equiv (\textbf{stable } p) \wedge (\textbf{stable } \neg p)$$

- **invariant** p means that p holds initially and continues to hold for all succeeding states. (In the program model, p is preserved by every statement.)

$$\textbf{invariant } p \equiv (\textbf{initially } p) \wedge (\textbf{stable } p)$$

or simply

$$\textbf{invariant } p \equiv SI \Rightarrow p$$

- **transient** p means that if p holds for some reachable state, then $\neg p$ holds (p being falsified) for a later state. (In the program model, if p holds at a point, there is at least one statement whose execution falsifies p and that statement is going to be selected for execution due to the fairness rule of the model.)

$$\textbf{transient } p \equiv \langle \exists s : s \in F : \{SI \wedge p\} s \{\neg p\} \rangle$$

- $p \mapsto q$ (p leads to q) means that if p holds for some reachable state, then q holds for a later state (within a finite number of execution steps). Formally $p \mapsto q$ holds if and only if it can be derived by a finite number of applications of the following three inference rules:

  1. (Basis)

$$\frac{p \wedge \neg q \textbf{ co } p \vee q,\ \textbf{transient } p \wedge \neg q}{p \mapsto q},$$

  2. (Transitivity)

$$\frac{p \mapsto q,\ q \mapsto r}{p \mapsto r},$$

  3. (Disjunction) For any set S of predicates,

$$\frac{\langle \forall p : p \in S : p \mapsto q \rangle}{\langle \exists p : p \in S : p \rangle \mapsto q}.$$

The operators **co**, **stable**, **constant**, and **invariant** are used to specify *safety* properties, which claim that undesirable state transitions will not occur during the execution of the program. The operators **transient** and $\mapsto$ are used for *progress* properties, which claim that the program performs useful work.

3 Characteristics of Deflection Networks

Deflection networks employ an adaptive routing method that can be applied to the networks in which the in-degree and the out-degree (the number of incoming links and the number of outgoing links) of each node are equal. (Note that the deflection networks are not necessarily regular. Different nodes in a network may have different connectivities.) The networks operate in a slotted mode with fixed length packets. The switching and transmission processes take place on a time slotted basis. Generally, all links are one slot in length, i.e., every packet travels one link per slot. (Links may be multiple slots long in some networks.) A packet from an input source enters the network if an empty slot is available. When two or more simultaneously arriving packets contend for the same output link, a contention resolution mechanism is invoked to select a packet that gets the use of the link. The packets that lost a contention are deflected (misrouted). Because the in-degree and the out-degree of a node are equal, it is always possible to assign each one of simultaneously arriving packets to a distinct output link.

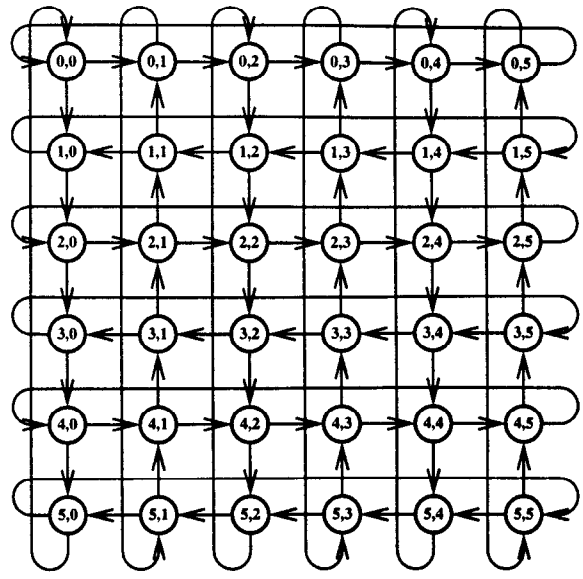

Figure 1: Manhattan Street Network.

For each time slot, a node of a deflection network

1. extracts a packet addressed to the node,
2. injects a source packet if the node obtains an empty slot (i.e., a slot has been received empty or a packet has been extracted), and
3. selects a switching configuration.

The switching and transmission of packets follow the above process.

The buffering resources required by the standard store-forward method are eliminated in deflection networks. Hence, there is no internal congestion. The behavior of the networks is stable. Packets are accepted as long as nodes recognize empty slots. The simplified nodal structure accelerates the nodal processing speed.

The most widely studied deflection network is the Manhattan Street Network (MSN) [18]. The MSN has the topology (Figure 1) that resembles the one-way streets and avenues in midtown Manhattan. The MSN has two incoming links and two outgoing links at each node and hence has the same degree of connectivity (number of links) as a bidirectional loop network. The alternation of row and column directions keeps inter-nodal distances small. The symmetric topology supports simple identical routing strategies at all nodes. The network provides multiple paths between a source and a destination and increases the throughput with the number of nodes by decreasing the fraction of the network capacity needed to communicate between nodes.

4 Formal Specification

We are interested in developing a specification that clarifies the structures of deflection networks and can aid the

construction of physical networks. We develop a topology independent, packet-level specification. The specification is based on a global observation. The network is modeled as a closed system. The development of the specification proceeds with the following principles: The specification for a system should be sufficiently strong to rule out any undesired behaviors of the system. At the same time, the specification should be sufficiently weak to provide implementers with the freedom to satisfy the specification in the most convenient and efficient way. In other words, it should avoid overspecifying the elements that are not essential to producing the desired system [26].

4.1 Basic Model and Topology

We start with a general graph model and make refinements on the model so that sufficient details of deflection routing can be specified.

A network is a directed graph $G = (V, E)$, where V is a set of nodes (vertexes) and E is a set of directed links (edges). The set E is a binary relation on the set V. (Note that the general graph model does not allow multiple links connecting two nodes in the same direction. That is the case for most general data networks.)

We identify each node in a network by a unique integer. Let n be the number of nodes in the network. Then the set V is defined as

$$V \equiv \{v \mid v \in \mathbf{N},\ 1 \leq v \leq n\},$$

where $\mathbf{N}$ denotes the set of natural numbers.

A deflection network has the following topological properties:

$$\langle \forall v, w : v, w \in V : (v, w) \in E^+ \rangle,$$

where E^+ denotes the transitive closure of E, and

$$\langle \forall v : v \in V : \#\{w \mid (v, w) \in E\} = \#\{w \mid (w, v) \in E\} \rangle.$$

The first property specifies that a deflection network is strongly connected. The second property states that each network node has equal in- and out-degrees, which are denoted as d_v for each node v, i.e.,

$$d_v \equiv \#\{w \mid (v, w) \in E\}$$

or equivalently

$$d_v \equiv \#\{w \mid (w, v) \in E\}.$$

Note that the transitive closure R^+ of a relation R can be constructed by the following rules:

$$\frac{(a, b) \in R}{(a, b) \in R^+},$$

$$\frac{(a, b) \in R,\ (b, c) \in R}{(a, c) \in R^+}.$$

The distance from node v to node w is given by the function Δ, which is defined as

$$\Delta : E^+ \rightarrow \mathbf{N}.$$

The distance between two nodes is usually defined as the smallest number of hops between the nodes. We leave the criteria for distance measurement to implementations.

The topological properties are static. We assume that there are no topological changes during the execution of the system, i.e., the following statements are assumed:

$$\langle \forall v :: \textbf{constant } v \in V \rangle,$$

$$\langle \forall v, w :: \textbf{constant } (v, w) \in E \rangle.$$

4.2 I/O Queues and Network Medium

In order to model the behavior of a deflection network, we must be able to distinguish the locations of packets in the network precisely according to the topology and the nodal structure of the network.

We abstract the inputs and outputs as unbounded queues of packets. Let S, D, and M be the set of locations in the input (source) queues, the set of locations in the output (destination) queues, and the set of locations in the network medium. We define the sets as follows:

$$Q \equiv \{(t, v, i, k) \mid t, i, k \in \mathbf{N},\ 1 \leq t \leq 2,\ v \in V,\ 1 \leq i \leq d_v,\ 1 \leq k\},$$

$$S \equiv \{(t, v, i, k) \mid (t, v, i, k) \in Q,\ t = 1\},$$

$$D \equiv \{(t, v, i, k) \mid (t, v, i, k) \in Q,\ t = 2\},$$

$$M \equiv \{(t, v, i) \mid t, i \in \mathbf{N},\ 1 \leq t \leq 3,\ v \in V,\ 1 \leq i \leq d_v\}.$$

The components v, i, and k are a node id, a link id, and a position number. The component t of an element in M indicates the location of a packet inside a node (Figure 2). The details appear later.

The sets S and D have countably infinite elements. An order of elements in the sets may be given by Cantor numbering. The order of locations is important only in the same queue. The component k determines the order of locations in a queue.

Instead of writing locations in the forms (t, v, i, k) or (t, v, i) in the specification, we often use the following more intuitive notations of the forms: $src^v{}_i k$ denoting $(1, v, i, k)$, $dst^v{}_i k$ denoting $(2, v, i, k)$, $in^v{}_i$ denoting $(1, v, i)$, $sw^v{}_i$ denoting $(2, v, i)$, and $out^v{}_i$ denoting $(3, v, i)$.

Figure 2 shows the locations in a node. (The node id is omitted in the figure.) We assign an input queue and an output queue to each link. This clarifies the nodal processing of the deflection network.

We use the symbol $\vdash$ to denote the connection of nodes (which outgoing link of a node connects to which incoming link of another node). The expression $out^v{}_i \vdash in^w{}_j$ is true if output link i of node v connects to input link j of node w. The operator has the same binding power as $=$, $\neq$, and so forth. The following statements hold:

$$(v, w) \in E \Leftrightarrow \langle \exists i, j :: out^v{}_i \vdash in^w{}_j \rangle,$$

$$out^v{}_i \vdash in^w{}_j \wedge out^v{}_{i'} \vdash in^w{}_{j'} \Rightarrow i = i' \wedge j = j'.$$

As mentioned in the previous subsection, topological properties are static. We assume the following property:

$$\langle \forall v, w, i, j :: \textbf{constant } out^v{}_i \vdash in^w{}_j \rangle.$$

4.3 Packet Representation

A packet has the format (s, r, i, k, h, z), where s, r, i, k, h, and z are, respectively, the source node, the destination node, the queue (link) id, the packet (position) number, routing information, and the data portion of the packet. In the specification, a packet is represented by a variable (α or β). The components of a packet are accessed through the following access functions: *source*, *destination*, *queue*, *number*, *rinfo*, and *data* (e.g., Given $\alpha = (s, r, i, k, h, z)$, $source.\alpha = s$). Note that the third and fourth components i and k are not necessary in implementations. We argument the packets with those components in order to uniquely identify each packet in the model. Note also that the fifth

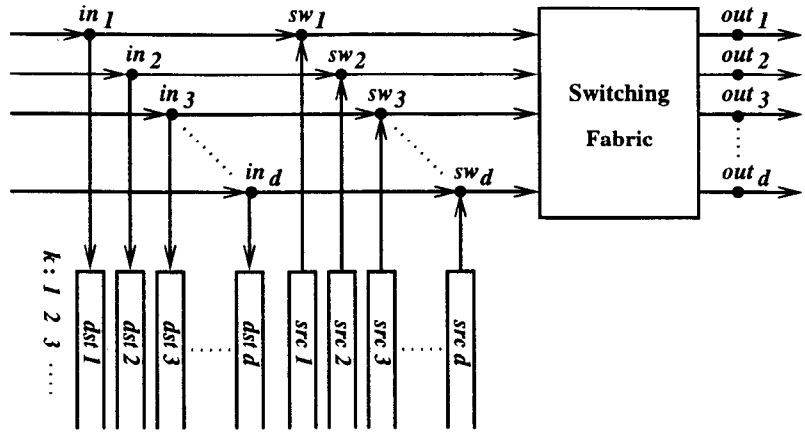

Figure 2: Locations in a node.

component h is optional and may not be needed in some implementation.

Let P^* be the set of all *logical* packets and P be the set of all *physical* packets that actually exist in the system. The set P^* is defined by the domains of the components of its elements as follows:

$$P^* \equiv \{(s,r,i,k,h,z) \mid s,r \in V,\ i,k \in \mathbf{N},\ 1 \le i \le d_s,\ 1 \le k,\ h,z \in \Lambda\},$$

where Λ is the set of all strings (data that can be represented by computers). The set P is a subset of P^*. Throughout the execution of the model, the set P is unchanged.

Packet Existence

$$\langle \forall \alpha : \alpha \in P^* : \textbf{constatnt}\ \alpha \in P \rangle \quad (1)$$

Property 1 states that no packets will be created or destroyed in the model. Only packets that initially exist continue to exist.

The location of a packet in the system is given by the function Θ, which is defined as

$$\Theta : P \to S \cup M \cup D.$$

We define a predicate *empty* as

$$empty.x \equiv \langle \forall \alpha : \alpha \in P : \Theta.\alpha \neq x \rangle.$$

The predicate *empty.x* is true if there is no packet at the location x and false otherwise.

For the rest of this paper, the domain of packets is P and may be omitted in an expression.

Packet Location

$$\textbf{invariant}\ \langle \forall \alpha, \beta : \alpha \neq \beta : \Theta.\alpha \neq \Theta.\beta \rangle \quad (2)$$

Property 2 means that no two packets have the same location, i.e., the function Θ is a one-to-one (or an injective) function. (This implies that a packet can not move to a location that is occupied by another packet.)

Network Initialization

$$\textbf{initially}\ \langle \forall \alpha :: \Theta.\alpha \in S \rangle \quad (3)$$

Property 3 means that initially all packets are in the input queues and there are no packets in the output queues and in the network medium.

Packet Validity

$$\textbf{initially}\ \langle \forall \alpha, v, i, k :: \Theta.\alpha = src^v{}_{i,k} \Leftrightarrow source.\alpha = v \wedge queue.\alpha = i \wedge number.\alpha = k \rangle \quad (4)$$

$$\textbf{initially}\ \langle \forall \alpha :: \langle \exists v :: destination.\alpha = v \rangle \rangle \quad (5)$$

$$\textbf{initially}\ \langle \forall \alpha :: \langle \exists z :: data.\alpha = z \rangle \rangle \quad (6)$$

$$\langle \forall \alpha, v, w, i, k, z :: \textbf{stable}\ source.\alpha = v \wedge destination.\alpha = w \wedge queue.\alpha = i \wedge number.\alpha = k \wedge data.\alpha = z \rangle \quad (7)$$

$$\textbf{invariant}\ \langle \forall \alpha :: \langle \exists h :: rinfo.\alpha = h \rangle \rangle \quad (8)$$

Properties 4–8 state that each packet must have a valid (s,r,i,k,h,z) value, which (except h) must not be changed during the execution of the system.

The position number of a packet is unique in its input queue. The number is assigned to each packet based on the initial location of the packet. The triple (s,i,k), where s, i, and k are a source node id, a queue (link) id, and a position number, uniquely identifies a packet in the model. This allows the existence of multiple packets that have the same source node id, the same destination node id, the same routing information, and the same data value in the model, i.e., in the set P.

The value of h, routing information in the packet header, may be used for contention resolutions in implementations. The domain of h is depend on implementations. Generally, h is a counter, i.e., a positive integer. Note that h does not have to have an atomic value; h may have a composite value. We leave the treatment of this field to implementations. As stated by Property 8, the value of h may change during the execution of the system.

4.4 Packet Move

Every packet in input queues must eventually move in some output queue, i.e.,

$$\langle \forall \alpha :: \Theta.\alpha \in S \mapsto \Theta.\alpha \in D \rangle.$$

We define the detailed properties of packet moves below.

Queue Move

$$\langle \forall \alpha, v, i, k : k > 1 : \Theta.\alpha = src^v{}_i k \text{ co } \Theta.\alpha = src^v{}_i k \vee \Theta.\alpha = src^v{}_i k - 1 \rangle \quad (9)$$

$$\langle \forall \alpha, v, i, k : k > 1 : \Theta.\alpha = src^v{}_i k \mapsto \Theta.\alpha = src^v{}_i k - 1 \rangle \quad (10)$$

$$\langle \forall \alpha, v, i, k : k > 0 : \Theta.\alpha = dst^v{}_i k \text{ co } \Theta.\alpha = dst^v{}_i k \vee \Theta.\alpha = dst^v{}_i k + 1 \rangle \quad (11)$$

$$\langle \forall \alpha, v, i, k : k > 0 : \Theta.\alpha = dst^v{}_i k \mapsto \Theta.\alpha = dst^v{}_i k + 1 \rangle \quad (12)$$

Properties 9–12 define the packet moves in input and output queues. Property 9 states that a packet at the position k in an input queue will either stay at the same position or move to the position $k-1$ and there are no other possible moves. Property 10 guarantees that the packet will move to the position $k-1$ in a finite number of execution steps. Properties 11 and 12 specify the analogous moves for packets in output queues.

Injection

$$\langle \forall \alpha, v, i :: \Theta.\alpha = src^v{}_i 1 \text{ co } \Theta.\alpha = src^v{}_i 1 \vee \Theta.\alpha = sw^v{}_i \rangle \quad (13)$$

$$\langle \forall \alpha, v, i :: \Theta.\alpha = src^v{}_i 1 \mapsto \Theta.\alpha = sw^v{}_i \rangle \quad (14)$$

Properties 13 and 14 specify that a packet at the head of an input queue must be injected into the network within a finite number of execution steps and there are no other moves. In order to implement Property 14, certain injection control mechanisms must be present for source lockout prevention. (Source lockout is a situation that a node is busy routing transit packets all the time and hence source packets can not be injected at the node.) Property 14 must be implemented in order for Property 10 to hold. Recall that no two packets can be at the same location (Property 2). Hence, the packet at the head of an input queue must be injected into the network so that the following packets in the queue can make their moves. (All packet moves must satisfy Property 2.)

Eventual Delivery

$$\langle \forall \alpha, v :: \Theta.\alpha \in M \wedge \mathit{destination}.\alpha = v \mapsto \langle \exists i, k :: \Theta.\alpha = dst^v{}_i k \rangle \rangle \quad (15)$$

Every packet injected into the network must eventually reach its destination. The routing algorithm in an implementation has to guarantee this property. The mechanisms to eliminate livelock (the indefinite circulation of packets without reaching destinations) must be present in deflection networks.

From Properties 10, 14, and 15, we can deduce (by applying the transitivity rule of $\mapsto$) the following property, which we gave earlier in this subsection:

$$\langle \forall \alpha :: \Theta.\alpha \in S \mapsto \Theta.\alpha \in D \rangle.$$

Note that for any packet in an input queue, there are only finitely many packets ahead of it in the queue. The packet will be out of the input queue within a finite number of execution steps. Hence, the property is technically correct.

The following properties define the packet moves in the network medium.

Node to Node Hop

$$\langle \forall \alpha, v, i :: \Theta.\alpha = out^v{}_i \text{ co } \Theta.\alpha = out^v{}_i \vee \langle \exists w, j :: out^v{}_i \vdash in^w{}_j \wedge \Theta.\alpha = in^w{}_j \rangle \rangle \quad (16)$$

$$\langle \forall \alpha, v, i :: \Theta.\alpha = out^v{}_i \mapsto \langle \exists w, j :: out^v{}_i \vdash in^w{}_j \wedge \Theta.\alpha = in^w{}_j \rangle \rangle \quad (17)$$

Delivery

$$\langle \forall \alpha, v, i :: \Theta.\alpha = in^v{}_i \wedge \mathit{destination}.\alpha = v \text{ co } \Theta.\alpha = in^v{}_i \vee \Theta.\alpha = dst^v{}_i 1 \rangle \quad (18)$$

$$\langle \forall \alpha, v, i :: \Theta.\alpha = in^v{}_i \wedge \mathit{destination}.\alpha = v \mapsto \Theta.\alpha = dst^v{}_i 1 \rangle \quad (19)$$

Transit

$$\langle \forall \alpha, v, i :: \Theta.\alpha = in^v{}_i \wedge \mathit{destination}.\alpha \neq v \text{ co } \Theta.\alpha = in^v{}_i \vee \Theta.\alpha = sw^v{}_i \rangle \quad (20)$$

$$\langle \forall \alpha, v, i :: \Theta.\alpha = in^v{}_i \wedge \mathit{destination}.\alpha \neq v \mapsto \Theta.\alpha = sw^v{}_i \rangle \quad (21)$$

Switching

$$\langle \forall \alpha, v, i :: \Theta.\alpha = sw^v{}_i \text{ co } \Theta.\alpha = sw^v{}_i \vee \langle \exists j :: \Theta.\alpha = out^v{}_j \rangle \rangle \quad (22)$$

$$\langle \forall \alpha, v, i :: \Theta.\alpha = sw^v{}_i \mapsto \langle \exists j :: \Theta.\alpha = out^v{}_j \rangle \rangle \quad (23)$$

Properties 22 and 23 specify that a packet will be switched at a node. In an implementation, the routing algorithm at each node determines the value of j in the properties. We will make refinements on Property 23 later for more specific implementations.

Up to this point, we specified only individual packet moves. Now, we must model and specify synchronized packet moves. First, we define an invariant on packet locations. Then, the synchronized packet moves are specified in terms of the number of packets at a node.

Relative Packet Location

$$\langle \forall t, v, i :: \textbf{invariant } \neg \mathit{empty}.(t, v, i) \Rightarrow \langle \forall s, j : s \neq t : \mathit{empty}.(s, v, j) \rangle \rangle \quad (24)$$

Property 24 specifies the relative locations of packets in a node. For example, if there is a packet at $in^v{}_i$ for some i, no other packets can be at $sw^v{}_j$ and $out^v{}_k$ for all j and k at the same time.

Synchronized Packet Move

$$\langle \forall t, v, k : k > 0 : \#\{i \mid \neg \mathit{empty}.(t, v, i)\} = k \text{ co } \#\{i \mid \neg \mathit{empty}.(t, v, i)\} = k \vee \#\{i \mid \neg \mathit{empty}.(t, v, i)\} = 0 \rangle \quad (25)$$

Property 25 implies that the switching and transmission of packets need to be synchronized. The number of packets that are at $in^v{}_i$, where i ranges from 1 to d_v, will remain the same or drop to zero. This implies that the packets at

$in^v{}_i$, $1 \le i \le d_v$ must move at once. The same property holds for packets at $sw^v{}_i$ and $out^v{}_i$, $1 \le i \le d_v$. Although the property is written in terms of the number of packets at a node, it globally synchronizes the node-to-node packet moves in the network. Packets coming into a node must be coming at the same time. Since all those packets are coming from distinct nodes, the transmissions of the packets at the nodes must be synchronized.

A packet at $src^v{}_i 1$ (the head of an input queue of a node) can be injected into the network if there are no other packets at $in^v{}_i$, $sw^v{}_i$, and $out^v{}_i$ or it may be injected at the same time that the packet at $in^v{}_i$ is moved to $dst^v{}_i 1$, i.e., an incoming packet on link i is extracted. There are no other cases that a packet at the head of an input queue can be injected into the network. Transit packets have higher priorities than source packets.

4.5 Selecting Switching Configuration

In this subsection, we make refinements on Property 23 for optimized switching.

The underlining routing scheme, which determines the path from the source to the destination for a packet when no contentions occur, is shortest path routing. The criteria for the shortest path is the criteria for the function Δ.

Let W_n denote the set of integers ranging from 1 to n, i.e.,

$$W_n \equiv \{i \mid i \in \mathbf{N},\ 1 \le i \le n\}.$$

We define the set C_v as

$$C_v \equiv \{\pi \mid \pi : W_{d_v} \to W_{d_v}, \\ \langle \forall i, j : i, j \in W_{d_v}, i \ne j : \pi.i \ne \pi.j \rangle\}. \quad (26)$$

The set C_v is the set of all permutations on W_{d_v}, representing all possible switching configurations at node v. There are $d_v!$ possible switching configurations (i.e., $\#C_v = d_v!$). (Note that a permutation is a one-to-one function from a finite set to the set itself.)

We also define the distance function δ_v as follows:

$$\delta_v : W_{d_v} \times V \to \mathbf{N},$$

$$\delta_v.(i, w) \equiv \Delta.(v, connect_v.i) + \Delta.(connect_v.i, w),$$

where the function *connect* is defined as follows:

$$connect_v : W_{d_v} \to V,$$

$$connect_v.i = w \equiv \langle \exists j :: out^v{}_i \vdash in^w{}_j \rangle.$$

The value of i indicates the output link from which the distance is measured. The function gives the distance from node v to node w through v's neighbor u to which output link i of v reaches.

Although we define the function δ as a local function of each node, the procedures implementing this function will be identical for all nodes in a typical symmetric topology network.

The function Φ is defined in terms of Θ as follows:

$$\Phi : \{x \mid \neg empty.x\} \to P,$$

$$\Phi.x = \alpha \equiv \Theta.\alpha = x.$$

The operator min is defined as

$$a = b \min c \equiv (a = b \vee a = c) \wedge a \le b \wedge a \le c.$$

Optimized Switching

$$\langle \forall \alpha, v, i, \sigma :: \Theta.\alpha = sw^v{}_i \wedge \sigma = \langle \min \pi : \pi \in C_v : \\ \langle +k : \neg empty.sw^v{}_k : \delta_v.(\pi.k, destination.\Phi.sw^v{}_k) \rangle \rangle \\ \mapsto \Theta.\alpha = out^v{}_{\sigma.i} \rangle \quad (27)$$

Property 27 specifies that the switching configuration that minimizes the total remaining distance of packets should be selected. The selected switching configuration may not be in favor of all the packets at the node. The switching configuration may result in some packets being deflected.

Note that there may be multiple configurations (permutations σ) that give the same minimum total distance. One of them may be selected randomly in a simple implementation. (Routing information in the packet header may be used to select one of them.) Implementations may further examine the consequences of those switching configurations. Some switching configuration may prevent the packets from being deflected at their succeeding nodes. In networks with a high degree of connectivity, the examination may be difficult and may cause nodal operations to slow down.

4.6 Switching in 2-Connected Networks

Property 27 specifies optimized switching in a general form. This subsection gives the properties for 2-connected networks ($d_v = 2$ for all v). The properties clarify the switching mechanism for 2-connected networks and suggest a simple case-analysis, which may be difficult for higher connectivity networks, in implementations. Note that analogous cases are omitted.

$$\langle \forall \alpha, \beta, v :: \Theta.\alpha = sw^v{}_1 \wedge \Theta.\beta = sw^v{}_2 \wedge \\ ((\Delta.(connect_v.1, destination.\alpha) < \\ \Delta.(connect_v.2, destination.\alpha) \wedge \\ \Delta.(connect_v.1, destination.\beta) \ge \\ \Delta.(connect_v.2, destination.\beta)) \vee \\ (\Delta.(connect_v.1, destination.\alpha) \le \\ \Delta.(connect_v.2, destination.\alpha) \wedge \\ \Delta.(connect_v.1, destination.\beta) > \\ \Delta.(connect_v.2, destination.\beta))) \\ \mapsto \Theta.\alpha = out^v{}_1 \wedge \Theta.\beta = out^v{}_2 \rangle \quad (28)$$

$$\langle \forall \alpha, v :: \Theta.\alpha = sw^v{}_1 \wedge empty.sw^v{}_2 \wedge \\ \Delta.(connect_v.1, destination.\alpha) < \\ \Delta.(connect_v.2, destination.\alpha) \mapsto \Theta.\alpha = out^v{}_1 \rangle \quad (29)$$

Property 28 may be simplified for faster processing in implementations.

5 Conclusion

We have developed a formal specification for deflection networks in UNITY logic. The specification is descriptive and non-operational. The network is viewed as a mathematical object. The advantage of this approach is that error prone operational reasoning is eliminated in the specification.

We represented the I/O queues as well as the locations in the network medium by sets of distinct locations rather than sequence variables, which may seem to be more natural for communication networks. The set model allows us to use

the standard mathematical tools in the specification. This approach may be applied to the specifications of routing schemes in other types of networks.

Although it is known that the use of a small amount of buffer space for each link (to hold a few packets that lost contentions) could reduce the number of deflections, we designed the nodes to be bufferless. At the level of current technology, buffering may be difficult in optical networks without electrooptic conversions, which prevent high speed nodal processing [2].

As we have seen in the specification, deflection routing may be difficult to implement for the networks with a high degree of connectivity. A large number of possible switching configurations in high connectivity networks makes cost efficient, high performance implementations challenging.

We modeled the network with its environment together as a closed system. This formulation avoids dealing with conditional properties, which add a certain degree of complexity to the development of the specification. An open system model imposes an examination of the compositionality of the defined logic.

We adopted the notion of the strongest invariant in defining the UNITY operators. This eliminates the non-equivalence between the axiomatic and informal operational semantics of the operators in [4]. However, the properties specified are weaker than those specified in the original logic of [4] in the sense that the properties hold only for reachable states.

In practice, the difference in formulation of the logic has little effect on the derivation of programs [24]. A recent study of the compositionality of properties and a discussion of the differences in logic formulations can be found in [7].

References

[1] K. Apt and E. Olderog. *Verification of Sequential and Concurrent Programs.* Springer-Verlag, New York, 1991.

[2] C. Baransel, W. Dobosiewicz, and P. Gburzynski. Routing in multihop packet switching networks: Gb/s challenge. *IEEE Network*, 9(3):38–61, 1995.

[3] F. Borgonovo and L. Fratta. Deflection networks: Architectures for metropolitan and wide area networks. *Computer Networks and ISDN Systems*, 24(2):171–183, 1992.

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

[5] T. Chung and D. Agrawal. Design and analysis of multidimensional Manhattan street networks. *IEEE Transactions on Communications*, 41(2):295–298, 1993.

[6] E. Clarke and J. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996.

[7] P. Collette and E. Knapp. A foundation for modular reasoning about safety and progress properties of state-based concurrent programs. *Theoretical Computer Science*, 183(2):253–279, 1997.

[8] C. Creveuil and G. Roman. Formal specification and design of a message router. *ACM Transactions on Software Engineering and Methodology*, 3(4):271–307, 1994.

[9] H. Cunningham and Y. Cai. Specification and refinement of a message router. In *Proceedings of the Seventh International Workshop on Software Specification and Design*, pages 20–29, Redondo Beach, California, December 6–7, 1993.

[10] H. Cunningham, V. Shah, and S. Shen. Devising a formal specification for an elevator controller. Technical Report UMCIS-1994-10, Software Methods Research Group, Department of Computer and Information Science, University of Mississippi, 1994.

[11] E. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

[12] N. Francez. *Fairness.* Springer-Verlag, New York, 1986.

[13] C. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580,583, 1969.

[14] E. Knapp. An exercise in the formal derivation of parallel programs: Maximum flows in graphs. *ACM Transactions on Programming Languages and Systems*, 12(2):203–223, 1990.

[15] F. Kröger. *Temporal Logic of Programs.* Springer-Verlag, Berlin, 1987.

[16] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer-Verlag, New York, 1992.

[17] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety.* Springer-Verlag, New York, 1995.

[18] N. Maxemchuk. Routing in the Manhattan street network. *IEEE Transactions on Communications*, COM-35(5):503–512, 1987.

[19] J. Misra. A logic for concurrent programming: Progress. *Journal of Computer and Software Engineering*, 3(2):273–300, 1995.

[20] J. Misra. A logic for concurrent programming: Safety. *Journal of Computer and Software Engineering*, 3(2):239–272, 1995.

[21] N. Nezu and H. Lu. Performance of toroidal deflection networks. In *Proceedings of the Second IASTED International Conference European Parallel and Distributed Systems (Euro-PDS '98)*, pages 103–110, Vienna, Austria, July 1–3, 1998.

[22] A. Pizzarello. An industrial experience in the use of UNITY. In J. Banâtre and D. Métayer, editors, *Research Directions in High-level Parallel Programming Languages: Mont Saint-Michel, France, June 17–19, 1991: Proceedings (Lecture Notes in Computer Science, 574)*, pages 39–49. Springer-Verlag, Berlin, 1992.

[23] B. Sanders. Eliminating the substitution axiom from UNITY logic. *Formal Aspects of Computing*, 3(2):189–205, 1991.

[24] B. Sanders. On the UNITY design decisions. In J. Banâtre and D. Métayer, editors, *Research Directions in High-level Parallel Programming Languages: Mont Saint-Michel, France, June 17–19, 1991: Proceedings (Lecture Notes in Computer Science, 574)*, pages 50–63. Springer-Verlag, Berlin, 1992.

[25] M. Staskaukas. Formal derivation of concurrent programs: An example from industry. *IEEE Transactions on Software Engineering*, 19(5):503–528, 1993.

[26] M. Staskauskas. The formal specification and design of a distributed electronic funds-transfer system. *IEEE Transactions on Computers*, 37(12):1515–1528, 1988.