Modeling and Simulating Computer Networks Using Formalized Data Flow Diagrams

Michael D. Haverdink
Iowa State University

Albert L. Baker
Iowa State University

Armin Mikler
Iowa State University

Follow this and additional works at: http://lib.dr.iastate.edu/cs_techreports

Part of the OS and Networks Commons, and the Systems Architecture Commons

Recommended Citation
http://lib.dr.iastate.edu/cs_techreports/80

This Article is brought to you for free and open access by the Computer Science at Digital Repository @ Iowa State University. It has been accepted for inclusion in Computer Science Technical Reports by an authorized administrator of Digital Repository @ Iowa State University. For more information, please contact digirep@iastate.edu.
Modeling and Simulating Computer Networks Using Formalized Data Flow Diagrams

Michael D. Haerdink
Albert L. Baker
Armin Mikler

TR #96-13
November 1996

Keywords: Formal methods; Computer networks; Specification languages; Simulation

Submitted for publication.

© Copyright 1996 by Michael D. Haerdink, Albert L. Baker, and Armin Mikler. All rights reserved.

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1040, USA
1 Introduction

A group of researchers at Iowa State University has targeted their research, in part, toward making formal software development methods more effective in real software development environments. These efforts have yielded results like:

- a simple, yet conveniently expressive, model-based specification language for C++ classes, SPECS-C++ [CB94, WBL94, Hav94].
- a technique for the direct execution of assertions in SPECS-C++, e.g., precondition and postcondition assertions for C++ class member functions [Wah95, WBL94].
- a formal, integrated CASE test tool supporting the convenient testing of specified classes and automated testing of corresponding implementations [Gur94].
- an executable formalization of data flow diagrams (FDFD's). [Col91, CB94, IWBBL96].

The authors have recently begun work on the application of FDFD's, and the corresponding techniques for direct execution of assertions developed by Wahls, et. al., to the modeling and simulation of computer networks.

FDFD's provide simple and natural abstractions for nodes and links in networks, as well as a formal language for specifying network behavior. We have augmented FDFD's by providing
• a notion of asynchronous timing of events,

• a convenient mechanism for describing the behavior of a general type of network node (of which there may be more than one in a given network), and

• features for specifying network architecture.

This paper introduces our approach to formal network modeling, which we have dubbed NET-SPECS. In the next section we provide a concise introduction to FDFD’s. Section 3 contains a summary of the extensions to FDFD’s required to facilitate the modeling of computer networks. Section 4 contains an example model expressed in NET-SPECS of a simple “shortest-path” routing algorithm with routing tables maintained at each node. In Section 5 we discuss how a network model expressed in NET-SPECS can be used to directly generate a simulation of the modeled network. We also summarize our plans for continuing our research in this concluding section.

2 Formalized Data Flow Diagrams

This section briefly introduces FDFD’s, which are based on traditional data flow diagrams (DFD’s) [DeM78, You89]. Informally, a DFD is a directed graph in which the arcs are called flows and the nodes are bubbles, stores, or terminators. Bubbles represent data transformers, stores typically represent persistent data, and terminators represent sources and sinks of information. Flows represent data paths between bubbles and bubbles, bubbles and stores, stores and bubbles, terminators and bubbles, and bubbles and terminators.

DFD’s typically serve as static roadmaps of information flow in systems. As is pointed out in almost every paper that includes a formalization of DFD’s, DFD’s have no rigorous semantic foundation, and thus can not be used to rigorously specify system functionality. For example, there is no clear view of whether bubbles represent processes, as in concurrent or distributed systems, or procedures and functions, as in code with a single execution thread. In addition, there is no notion of data movement on flows – flows represent roads, not cars, referring back to the “static roadmap” analogy.

Succinctly stated, FDFD’s describe the movement of cars. Of course, by “cars” we mean the data in the system. The flows may be one of two types. Consumable flows are viewed semantically as unbounded, first-in, first-out (FIFO) queues of tokens (values) of a particular type. Persistent flows behave much like shared variables, but we do not use persistent flows in this paper and omit any further discussion of them. We are basing our NET-SPECS work on the definitions of FDFD’s developed by Coleman, Wahls, Baker, and Leavens in [Col91, CB94, WBL93, IWBL96]. In this approach the types of flows are modeled abstractly using the discrete mathematical structures set, sequence, and tuple. Assertions are formed using a fixed set of operations over these intrinsic types, e.g., ∪ for sets and \ for sequences.

An informal view of these definitions is that each bubble in an FDFD has a unique identifier
and a set of firing rules. A firing rule has three parts: an \textit{enabling condition} that describes the conditions that must be true for a bubble to “start its work”, a \textit{precondition} that describes acceptable inflow data, and a \textit{postcondition} that defines the outflow data. The enabling condition is a conjunction of flow names preceded by + or − and assertions. A +\textit{flowname} expression is \textit{true} exactly when there is data on the flow named \textit{flowname}, and a −\textit{flowname} is \textit{true} when there is no data on the flow named \textit{flowname}. Enabling conditions define the synchronization primitives of the specified system, i.e., what values have to be “present” in order for a bubble to “do its work”. The enabling conditions and preconditions are written as first-order predicate calculus assertions over the data values on the inflows. The postconditions are analogous assertions, which define outflow values as functions of the inflow values. The post state values of outflows are denoted with a “prime” notation.

The syntax for a firing rule is:

\[
enabling \ condition : \ precondition \ \models \ \postcondition
\]

Since FDFD’s are used to model concurrent systems, the semantics must capture all possible execution sequences. Thus, bubbles fire in two steps. First a bubble reads its inflows, and then it writes to its outflows. A bubble is \textit{working} when it has read its inflows but not yet written to its outflows. A bubble is \textit{idle} at all other times. Initially, all bubbles are idle. The operational semantics of an FDFD is summarized below (a formal operational semantics is provided in [WBI93]):

1. Find the set of bubbles that may fire. This includes all bubbles in the \textit{working} state, and any bubble in the \textit{idle} state that has values on its inflows satisfying the enabling condition of at least one of its firing rules.

2. Nondeterministically choose one bubble from among the firing candidates.

3. Fire the bubble:
   - If the bubble is \textit{idle}:
     \begin{itemize}
     \item Choose one of the bubble’s rules whose enabling condition is satisfied by the inflow values. The precondition is assumed to be true, in that the precondition must be met by the bubble that produced the value.
     \item Read the values referenced in the enabling condition. Consume the values from these inflows. Note which firing rule was selected.
     \item Change the state of the bubble from \textit{idle} to \textit{working}.
     \end{itemize}
   - If the bubble is \textit{working}:
     \begin{itemize}
     \item Produce output on the outflows. This output is defined by the postcondition of the rule chosen when the bubble changed to the working state.
     \item Change the state of the bubble from working to idle.
     \end{itemize}

4. Repeat the above steps until the set of bubbles allowed to fire in step one is empty. (Which may be a long time, indeed.)
One might well argue that using FDFD’s to model network behavior may be advantageous even in the absence of extensive tool support. But the work of Wahls, Baker and Leavens on the direct execution of FDFD’s [WBL94] is, we believe, fundamental to the usefulness of NET-SPECS. It will allow the testing of the model as well as support the highly automated generation of a simulator for the modeled network. The cornerstone of the direct execution of FDFD’s is the ability to directly execute a large and useful subset of the class of all assertions that might appear in an FDFD specification. We will refer to this again in our discussion of direct generation of network simulations from NET-SPECS in the concluding Section 5.

3 FDFD Extensions in NET-SPECS

We view modeling (specifying) and simulating a computer network as two distinct steps. The development of a model in NET-SPECS of the computer network, to include a precise specification of network behavior, is the first step. The second step is the generation of a simulator from the NET-SPECS model. This section explains the modeling facilitators of NET-SPECS.

Over time in any computer network, packets enter and exit the network, links come up and go down, and nodes may behave erratically. Thus we have to extend FDFDs to include a feature for generating timing events. In the example presented in Section 4, the basic unit of time is a single state change in the execution of a NET-SPECS model of a network, i.e. the firing of a single bubble. However, we will generalize this notion of time in NET-SPECS along the lines taken in [SB96]. Regardless of the specification of time, we define a new “bubble-like” entity in NET-SPECS, the Timer.

Timers are like bubbles in that they are connected to other FDFD bubbles via outflows. Timers simply send tokens to bubbles based on a specified probability distribution over time. They can be used to simulate events like the generation of new message data packets into the network, the sharing of routing information, and any other time-based network activity. A simple timer can be specified to generate a token every $n$ time units. Another possibility would be to specify that tokens be generated with a Poisson distribution at a certain rate. Timers are depicted graphically in NET-SPECS with an oval, and the type of Timer outflows is always signal, a single-valued type.

By definition, every bubble in an FDFD must have its own collection of firing rules. If we use an FDFD to specify a large network, however, defining every node as a bubble is unnecessarily tedious, since there are undoubtedly groups of nodes that behave similarly. The only difference between nodes in such groups is that different nodes have different sets of neighboring nodes. To facilitate the convenient description of a network, we have added node classes to NET-SPECS.

Each node in a class behaves exactly the same (i.e. they have the same firing rules). Networks can be made up of nodes from one or more node classes. In general, a node will have a set of inflows and a set of outflows that connect it to its neighbors. One might use different node classes to describe machines and routers in a given network.
Finally, NET-SPECS needs an expressive suite of operations that can generate a particular network made up of nodes from one or more node classes. Operations must be provided to specify the number and classes of nodes in a network as well as the "degree of connectiveness" of the network. The specifier should also be able to quickly generate common network layouts like rings, grids, and hypercubes.

4 A NET-SPECS Example

This section contains the specification of a distributed shortest-path network routing algorithm to illustrate the suitability of NET-SPECS for modeling network behavior. An informal description of the shortest-path routing algorithm is given first, followed by the formal NET-SPECS model interspersed with comments. We do not include specification of the initiation of messages into the network, as this does not reveal any new features of NET-SPECS.

In this shortest-path algorithm, each node maintains a routing table with an entry for each other node in the network. Each routing table entry contains the destination node, the neighbor through which an outgoing packet will be routed, and a distance estimate to the destination (measured in hops for simplicity). Periodically each node sends its routing table to each of its neighbors. When a node receives a routing table from a neighbor, if, for a particular destination, a shorter path exists, the node updates the corresponding entry in its own routing table.

Specifically, suppose node $i$ receives a routing table from node $j$. Further assume that node $j$’s estimated distance to destination $d$ is $j_d$ (and similarly for node $i$). If $j_d + 1 < i_d$, then the entry for node $d$ in node $i$’s routing table is updated to route through neighbor $j$ with estimated distance $j_d + 1$. Assume that initially each node’s routing table indicates known paths of length 1 to every neighbor.

Figure 1 is a graphical representation of the only node class required in this simple example.

![Figure 1: Node $i$ of the FDFD](image)

The following type definitions define several abstract types, ultimately defining the types of the flows in Figure 1: *Packet* and *NodeData*. For simplicity, we assume nodes are identified by an integer.
So each node maintains a current set of neighbors and its own routing table. A packet consists of an \textit{originalSource}, an \textit{ultimateDestination}, and either an \textit{UpdateMessage} or a \textit{DataMessage}. Routing tables are modeled as a set of tuples containing a \textit{dest}, a \textit{routNeighbor}, and a corresponding \textit{cost}. The type of data on the inflow from \textit{Timer} is \textit{signal}, which is just a single-valued type. What follows is an English description of each firing rule for our distributed shortest-path algorithm, as well as the formal firing rule.

Some of the assertions contain references to \textit{abstract functions}. Abstract functions are simply parameterized assertions. All the abstract functions that appear in the firing rules will be explained and defined later in this section. Also note that “special symbols” in assertions are written with an ASCII representation to facilitate parsing and type checking. For example, the universal quantifier is written \texttt{\forall} and not \texttt{\forall}, and the “and” operator is written \texttt{\wedge}, and not \texttt{\land}.

The first firing rule listed sends a routing table to each neighbor when a token is present on the inflow from \textit{Timer}.

\texttt{+T \land +ND:}
\texttt{\mid = ND' = ND \land}
\texttt{\forallall (IDType n) [n \elem neighbors(ND) \Rightarrow}
\texttt{OutPacket'_n = (i,n,(i,rt(ND)))]}

The enabling condition requires that a signal be present on the inflow \textit{T} from \textit{Timer} and the
inflow \( ND \). In fact, every firing rule will contain \( +ND \) as part of the enabling condition, as there is always exactly one token on flow \( ND \). The precondition is blank and assumed to be true, indicating that there are no other restrictions on the inflow value, other than those provided by the abstract type of the flow. The first portion of the postcondition requires that \( ND' = ND \).

The other part of the postcondition is a universal quantification that defines values for the outflows that are connected to each node that is a neighbor of node \( i \). A packet \((i, n, (i, rt(ND)))\) is sent to each neighbor \( n \).

The next firing rule handles routing tables received from neighboring nodes. The enabling condition in this rule states that there must be a Packet present on an inflow from a neighbor, and that the message must be an UpdateMessage. The postcondition uses the updated routing table defined by the abstract function UpdatedTable.

\[
\text{+InPacket} \land -T \land +ND \land \text{message(InPacket)} \land \text{oftype UpdateMessage}: \\
\begin{align*}
| &= \text{neighbors}(ND') = \text{neighbors}(ND) \\
rt(ND') &= \text{UpdatedTable}(rt(ND)), \\
\text{handedFrom(message(InPacket))}, \\
rt(message(InPacket))
\end{align*}
\]

The following firing rule forwards DataMessage packets based on node \( i \)'s routing table, or consumes the packet if \( i \) is the ultimateDestination. The enabling condition is similar to that of the previous rule, except that the packet must contain a DataMessage, and another abstract function canForward must be true. The postcondition states that the data on flow \( ND \) is unchanged and defines the output value for the outflow that corresponds to the appropriate neighbor, as long as the packet is destined for a node other than \( i \).

\[
\text{+InPacket} \land -T \land +ND \land \text{message(InPacket)} \land \text{oftype DataMessage} \land \\
\text{canForward}(rt(ND), \text{ultimateDestination(InPacket))}: \\
\begin{align*}
| &= ND' = ND \\
&((!\text{ultimateDestination(InPacket)} = i) \Rightarrow \\
{(\exists \text{RoutingTableEntry e} [e \in \text{rt}(ND) \land \\
\text{dest}(e) = \text{ultimateDestination(InPacket)} \land \\
\text{OutPacket'}_\text{routNeighbor(e)} = \\
(\text{originalSource(InPacket), ultimateDestination(InPacket), \\
message(InPacket))})})
\end{align*}
\]

The last firing rule in this example handles the routing of a DataMessage packet if no path is known to its ultimateDestination. Its enabling condition is identical to the previous rule except that the canForward abstract function is false. In this case, the firing rule simply routs the message to a random neighbor using an existential quantification over all the neighbors of node \( i \).
When defining firing rules, it is important that one be conscious of how they interact. Recall from the description of an FDFD execution that the rule to be fired next is chosen nondeterministically from the set of all rules whose enabling condition is satisfied. The rules must be carefully constructed to specify the desired behavior and to prevent starvation or deadlock conditions.

For example, the last three rules presented above all contain \(-T\) as part of the enabling condition. This was done to give top priority to signals generated by \(Timer\). Alternatively, the \(-T\) could be omitted from some or all of the three firing rules to change the specified behavior of the network to nondeterministically choose between sending out routing tables, forwarding packets, and updating routing tables. In any case, the formalism forces the specifier to think about these issues early in the modeling and simulating process.

Abstract functions allow for the parameterization and modularization of assertions. They simply define values. Our example model references the four abstract functions defined below. As before, comments about the abstract functions are interspersed with their definitions.

The abstract function \(canForward\) defines a boolean value of \(true\) if there is an entry in the routing table \(myTable\) for the given \(destination\). Recall that a \(RoutingTable\) is modeled as a set of tuples containing routing information. Rather than maintaining a set containing entries for each destination (and representing unknown paths with infinite costs), our model will only insert entries into a routing table if the path is known. So \(canForward\) is simply an existential quantification over the entries in the routing table.

\[
\text{define CanForward(RoutingTable myTable, IDType destination) as boolean such that} \\
\text{CanForward(myTable, dest) =} \\
(\exists e (RoutingTableEntry e) [e \in myTable \wedge} \\
\text{dest(e) = destination])}
\]

The next abstract function \(IsAShorterPath\) also defines a boolean value. Its arguments are two routing tables, one from the node referencing the abstract function \((myTable)\) and one from the node's neighbor \((neighborTable)\). Its value is true if there is a shorter path to at least one common destination in the two routing tables.
define IsAShorterPath(RoutingTable myTable,
RoutingTable neighborTable)
as boolean such that
IsAShorterPath(myTable, neighborTable) =
(\exists (RoutingTableEntry e1) [e1 \in myTable \land
\exists (RoutingTableEntry e2) [e2 \in neighborTable \land
  dest(e1) = dest(e2) \land
  cost(e2) + 1 < cost(e1)]])

The abstract function IsANewPath is true if there is a destination in neighborTable that is not in myTable.

define IsANewPath(RoutingTable myTable,
RoutingTable neighborTable)
as boolean such that
IsANewPath(myTable, routingTable) =
(\exists (RoutingTableEntry e1) [e1 \in neighborTable \land
\forall (RoutingTableEntry e2) [e2 \in myTable =>
  dest(e2) != dest(e1)])]

The last abstract function, UpdatedTable defines an updated routing table based on a node’s existing table (myTable), a neighboring node’s ID (neighbor), and the neighbor’s table (neighborTable). It is a relatively long function, the assertion is just a conjunction of three implications that can be studied separately. The overall structure of the function is as follows,

(antecedent1 => consequent1)
\land (antecedent2 => consequent2)
\land (antecedent3 => consequent3)

where the three antecedents are written such that exactly one of them is true. So, by the definition of implication, the entire assertion is true if the consequent matching the satisfied antecedent is true.

The first implication handles the case in which there is no shorter path to any destination, and there is no new path to any destination. In this case, the defined routing table is simply myTable. The second implication handles the case in which there is a shorter path to at least one destination. If this is the case, existential quantification is used to identify one such destination. Then the new routing table is defined with a recursive call to UpdatedTable, with the new first argument defined as the old routing table updated with the shortest path for the quantified destination. The third implication is handled similarly for new paths.

define UpdatedTable(RoutingTable myTable, IDType neighbor,
RoutingTable neighborTable
as RoutingTable such that
(!IsAShorterPath(myTable, neighborTable) \/
!IsANewPath(myTable, neighborTable) =>
UpdatedTable(myTable, neighbor, neighborTable) = myTable)
\ (IsAShorterPath(myTable, neighborTable) =>
\ (RoutingTableEntry e1) [e1 \elem myTable \/
\ (RoutingTableEntry e2) [e2 \elem neighborTable \/
  dest(e1) = dest(e2) \/
  cost(e2) + 1 < cost(e1) \/
  UpdatedTable(myTable, neighbor, neighborTable) =
  UpdatedTable(myTable - {e1} \union
  {(dest(e1), neighbor,cost(e2)+1)}, neighbor,
  neighborTable)]])
\ (IsANewPath(myTable, neighborTable) \/
!IsAShorterPath(myTable, neighborTable)) =>
\ (RoutingTableEntry e2) [e2 \elem neighborTable \/
\ (RoutingTableEntry e1) [e1 \elem myTable =>
  dest(e1) != dest(e2)] \/
  UpdatedTable(myTable, neighbor, neighborTable) =
  UpdatedTable(myTable \union
  {(dest(e2), neighbor,cost(e2)+1)}, neighbor,
  neighborTable)])

Finally, initial values on at least one flow must be specified before an execution can be initiated. As mentioned previously, each node initially knows its neighbors. So, the initial values on flow ND are defined as follows:

neighbors(ND) = \{IDType n \mid "node n is a neighbor of node i"\}

rt(ND) = \{RoutingTableEntry e \mid
dest(e) \elem neighbors(ND) \/
routNeighbor(e) = dest(e) \/
cost = 1\}

5 Conclusions

Our primary intention with this first paper on NET-SPECS has been to introduce the features we have added to FDFD’s to facilitate the modeling of computer networks using NET-SPECS. These features include:

- timers, which allow for the modeling of asynchronous timing events,
- node classes for defining an entire class of nodes, of which there may be many instances in a given NET-SPECS model, and

- operators that allow the specifier to define the desired network architecture, e.g., the number of nodes in the network, the average number of nearest neighbors for a given node class, etc.

It is our hope that our precise specification of the shortest path routing algorithm in our example will not contradict the claim that NET-SPECS can serve as both an expressive and precise specification technique for large scale networks.

However, as mentioned earlier, the value of NET-SPECS for modeling large scale computer networks is significantly enhanced due to the ability to directly execute a NET-SPECS model (in order to test the model itself), and the potential for direct generation of a simulator of the network from a NET-SPECS model.

The convenient generation of a simulator from a NET-SPECS model will require operators for the user to specify simulation parameters. Such parameters might include the exact model of time to be used, the length of time the simulation should run, the number of iterations, the parameters for timer behaviors, i.e., the probability distributions over time of the Signal events from timers, and the data to be "collected" on each simulation run.

In addition to conveniently setting the simulation parameters, there may be features which we can provide which can be defined in terms of the NET-SPECS features already presented, but which may be generalized for convenience. An example of this would be an intrinsic node class that will model packet loss on a flow. Such a feature could be modeled as a bubble, but might more conveniently be expressed as a property of a particular flow type.

Thus our current research on NET-SPECS is focused on defining an appropriate and convenient set of operators and features for generating a network simulation from NET-SPECS models. We are also focusing on the appropriate simulation technique to employ in the executable simulation.

When combined in an integrated CASE tool supporting the development of a model of a computer network in NET-SPECS and the convenient generation of a corresponding simulation, we are confident that this CASE tool will find widespread application in the analysis of existing and planned computer networks.

References


