Using Data-Integrity Constraints in Model-Based Design

Sukhamay Kundu
Computer Science Dept, Louisiana State University
Baton Rouge, LA 70803, USA (email: kundu@bit.csc.lsu.edu)

Abstract

The success of model-based software design in terms of early detection of high level (design) faults depends heavily on the way the model abstracts the software. We present a new model of software which emphasizes the integral role of data-integrity constraints in the design activity and analysis to enhance our ability to detect faults in functional and data design. We describe an application of the new model in designing a complex networking simulation software.

Keywords: *Model-based design, data-integrity constraints, functional design.*

1. Introduction

A model of a software represents an abstraction which captures some of its high level features and gives a design overview. We can have several alternate models of a software from different (or partially overlapping) perspectives. A model helps us, above all, in early detection of high level design errors and models at different levels can help us detect faults of different types. A model also helps us in creating a high-level architecture for the software which can in turn help in determining a multi-phase development plan for the software and in planning high level test phases and test criteria. For most software development tasks, there is a critical need for more systematic design methods and better analyses techniques using lightweight formalisms than for more formal reasoning methods using exact and heavy weight formalisms [1]. Since the Unified Modeling Language (UML) [5] already contains a large number of modeling concepts, including constraints specification, our crying need of the moment is better methods for using those concepts in designing a software. Our use of the data-integrity constraints is a step forward in this direction.

A software model must include its two core components: the data items D and the operations (functions) F that process or produce them. The dataflow model of a software shows D, F, and the inputs and outputs of each $f \in F$ in terms of D. The formal specification of a software in Z [8] gives in addition the type information of each data item in D and the pre- and post-conditions for each f. A state-chart [7] or finite-state model (*FSM*), on the other hand, shows the interactions or uses of the functions in F in response to external events. The *FSM* is useful when different occurrences of an event may give rise to different responses based on some of its preceding events. If the response to an event is not dependent on the past events then the FSM becomes trivial and we can use the simple call-relationship (*CR*) among the functions as an alternative. The *FSM* can be used to check the correctness of a functional design in terms of their use in the required fashion. The data-integrity constraints (*DIC*) provides us an alternative light-weight formalism to support the analysis a functional design from a different perspective.

The functional design is at the heart of a design task for both object-oriented softwares and otherwise, Our model applies to the functional design level.

2. The New Model

Fig. 1 shows our new dynamic model of a software with four components: D, F, *FSM*, and *DIC*. The *DIC* provides information that are not properly captured by the pre- and post-conditions, particularly when they are expressed in incomplete fashion because of the difficulties in formulating them.

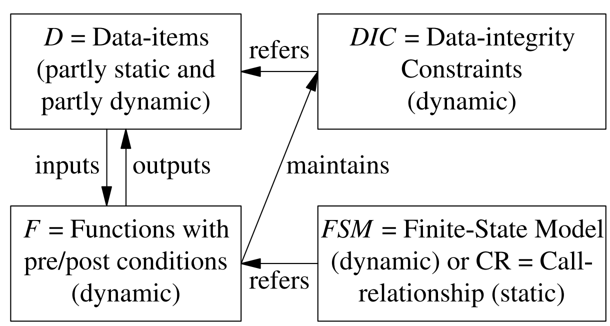

Figure 1. The four components in the new model.

For our purpose, it will be convenient to regard the data-items in D as global variables. We write $f_0 \in F$ for the initialization function, which sets the stage for the application of some other functions in F by making their pre-conditions true. In the extreme case, it may be a trivial function doing nothing. If the formal design verification (correctness proofs) is not the primary goal, one may either omit the pre- and post-conditions or state them in a crude form, e.g., by just listing the data-items involved in them. We include the outputs of each $f \in F$

in D irrespective of whether that data-item is immediately consumed by some other function in F or not. This is so because an output data-item may be part of an integrity-constraint. Note that each output of f takes part in the post-condition of f in a Z-specification, except for functions like a random number generator.

2.1. Static and dynamic properties

A property is static if it can be defined and verified without considering an actual (or symbolic) execution. The call-relationship between two functions is a static property as are the inputs inp(f) and the outputs out(f) of a function f. The inp(f) consists of the data-items used in f, including those used as actual parameter-values (if any) in different calls of f. Indeed, a particular call or execution of f may not use all items in inp(f); likewise, a particular execution may not modify (produce) all items in out(f). Note that if v is a local static variable in f in an implementation, then for our purpose v is a global data-item which is accessed only by f and $v \in$ inp(f)$\cap$out(f). On the other hand, if a function f has an integer parameter n and reads n data-items from a given file "input.dat" to compute the sum of those items, then inp(f) = $\{n$, input.dat$\}$ and not the actual data-items read from the file input.dat; out(f) = $\{$sum$\}$. In what follows, we assume for simplicity of this presentation that the functions $f \in F$ do not have any parameters; this is equivalent to saying that for each formal parameter in an f there is a distinct global variable which is used for passing the value for that parameter.

The pre- and post-conditions of a function f, denoted by preCond(f) and postCond(f), are dynamic properties since they relate to "before" and "after" of an execution. The preCond(f) characterizes the situations, i.e., the environments in which f is intended to be used, and its correctness can be verified only by executions of the software and not just of f. The correctness of postCond(f) can be verified by executions of f for which preCond(f) holds. Shown below are two static properties (P1)-(P2) for preCond(f) and postCond(f). The preCond(f) depends on the functions whose execution are followed by that of f, and postCond(f) depends on the functions whose execution follow that of f.

(P1) Each $v \in$ inp(f) appears in at least one of preCond(f) and postCond(f).

(P2) Each $v \in$ out(f) appears in postCond(f).

In contrast, some data-items in D may be static and the others dynamic. The static parts define the problem-instance being solved and are not modified by the functions in F after initialization by f_0. The dynamic parts of D are related to the computations of $f \in F$ and depend on the algorithm being used for solving the problem. The functions themselves are static.

We point out that in our model in Fig. 1 each execution of the software involves a sequential execution of one or more functions in F in some order starting with f_0, including repeated uses of some of them. That is, we actually have a "follows" relationship $f_i \rightarrow f_j$, meaning f_j is used after f_i, among F instead of a typical call-relationship where f_i may call f_j in the middle of its execution. This basically means that we model the functions f_i at a fairly high level (the calling of f_j in the middle of execution of f_i arises typically when we model the functions at a low-level). We shall use the term call-relationship to mean its more specialized form "follows-relationship".

2.2. Data-integrity constraints

For our purpose, a data-integrity constraint $C = C(v_1, v_2, \cdots, v_m) \in DIC$ describes a property (constraint) among the data-items $v(C) = \{v_1, v_2, \cdots, v_m\} \subseteq D$ that holds on termination of each $f \in F$, including the initialization function f_0. Thus, a constraint C is relevant to f if and only if $v(C)\cap$out(f) $\neq \emptyset$. Unlike preCond(f) and postCond(f), an integrity constraint C is not directly attached to a specific function f. Since all C hold before the start of each $f \in F - \{f_0\}$, we can say that each $C \in$ preCond(f)$\cap$postCond(f) for all $f \in F$, $f \neq f_0$. We recognize this special role of DIC in our model and use them accordingly in analyzing a design. We write $M(f) = \{C \in DIC\colon v(C)\cap\text{out}(f) \neq \emptyset\}$. The relationship "maintains" in Fig. 1 refers to this connection between $f \in F$ and $M(f) \subseteq DIC$. We consider DIC to be dynamic for the same reasons that pre- and post-conditions are considered dynamic. We have the following property (P3) of D, DIC, and the inputs/outputs of $f \in F$.

(P3) $D = \bigcup_{f \in F} [\text{inp}(f)\cup\text{out}(f)] \supseteq \bigcup_{C \in DIC} v(C)$.

We remark that an integrity constraint represents a less stringent requirement on F than an invariant.

3. Model Analyses

The explicit presence of DIC in our model provides us an opportunity for a more refined analysis in many ways than is possible otherwise. We describe some of these analyses below.

3.1. Proper choice of a modeling level

Given a set of $DIC = \{C_1, C_2, \cdots, C_n\}$, an appropriate level for choosing the functions F is determined by the property (P4) (see Section 2.2).

(P4) Each $C_i \in DIC$ holds after every function $f \in F$.

A function which is at a too low level is likely to violate (P4). It is easy to see that if (P4) is satisfied by F

and we replace a subset of functions $F' \subset F$ by a mega-function g, then g and hence $F - F' \cup \{g\}$ also satisfy (P4). In [2], we characterized the subsets $F' \subset F$ which can be replaced by a mega-function g in terms of the path-closed property in the dataflow diagram on F.

The level at which we choose D and F are interdependent in terms of the sets inp(f) and out(f) for $f \in F$. In contrast, *DIC* is determined only by D. In particular, if certain aspects of D are not relevant in *DIC*, then we abstract them out and simplify D. Thus we should first choose D and *DIC* and then choose F, which may force us to refine D and F. We then go back and reformulate *DIC* if necessary in view of the new D, and repeat the cycle till (P1)-(P4) hold for $\{D, F, DIC\}$. This gives us a different analyses of the functional design than is possible with a Z-specification. We shall see that sometimes the data-integrity constraints force us to use certain functions in F in a particular order, although the pre- and post-conditions of those functions themselves do not require this.

Finding meaningful names for a function f_j is another important outcome of this analysis, as this can help determination of potential violation of (P4).

3.2. Analysis of call-relationship

We can identify a missing but required dataflow from an analysis of the call-relationship based on the property (P5) below. The property (P5) may indicate the need to decompose a function f_i into smaller ones when inp(f_i) is too large.

(P5) For $f_j \neq f_0$, the current values of all data-items in inp(f_j) are available (computed or supplied externally) prior to each use of f_j.

3.3. Determining a layered structure of software

We can identify a layered architecture of the software by analyzing the *DIC* as follows. Recall that a layered structure can be very helpful in determining a stepwise-development plan and a high level test-plan for the software. Let v(preCond(f)) $\subseteq D$ denote the data-items which appear in preCond(f), and let v(postCond(f)) be defined similarly. Given a subset $D' \subset D$, we write $F' = F(D') = \{f \in F\colon \text{inp}(f) \subseteq D'\} \subseteq F$.

Def. 1. We say a subset $D' \subseteq D$ is *F-closed* if the following three conditions hold:

(c1) F' is non-empty and contains the initialization function $f_0 \in F$.

(c2) $D' = \bigcup\{\text{inp}(f)\colon f \in F'\}$.

(c3) v(preCond(f)) $\subseteq D'$ for each $f \in F'$.

The conditions (c1) and (c3) say that D' should be sufficiently large, and the condition (c2) says that D' should not contain unnecessary elements with respect to its associated F'. It is easy to see that if D_1 and D_2 are F-closed, then the union $D_1 \cup D_2$ is also F-closed, but $D_1 \cap D_2$ may not be F-closed. If D_1 is an F-closed subset, then the corresponding F_1 together with $D_1^+ = D_1 \cup \{\text{out}(f)\colon f \in F_1\}$ and $DIC_1 = \{C_i \in DIC\colon v(C_i) \subseteq D_1^+\}$ form a subsystem, and we can let $S_1 = \{D_1, F_1, DIC_1\}$ form the first layer of the software. Ideally, we want to choose D_1 such that it would allow us to isolate a corresponding finite-state submodel FSM_1 of *FSM* involving only the functions F_1 that completely describes the interactions among F_1. An absence of such an FSM_1 only means that we cannot do a full integration test of the functions F_1 before the remaining functions are developed, and we will have to be content with the unit level and a partial integration testing of F_1. In general, $D - D_1$ may not be F-closed and thus $\{D - D_1, F - F_1, DIC - DIC_1\}$ cannot be considered a subsystem, even with the original *FSM* (because $FSM - FSM_1$ would not be typically very meaningful). Nevertheless, we may be able to obtain an increasing sequence of closed subsets $D_1 \subset D_2 \subset \cdots \subset D$ and a corresponding sequence of subsets DIC_i, F_i, etc giving us a layered family of subsystems S_i. A function $f_i \in F_i$ may be responsible for maintaining an integrity constraint in DIC_j for $i < j$. Note that we might be able to replace the original initialization function f_0 in F by two or more simpler ones and distribute them among F_1, F_2, etc, without duplication.

If we have another such chain $D'_1 \subset D'_2 \subset \cdots \subset D'_k \subset D_{k+1} \subset D_{k+2} \subset \cdots \subset D$, which have all but the first $k \geq 1$ terms identical to the previous chain and $D'_i \cap D'_i = \varnothing$ for $i \leq k$ (and hence $F_i \cap F'_i = \varnothing$ for $i \leq k$), then we could reap the same benefits again as that for the first chain. This would, however, require that f_0 be decomposable so that parts of it can be used in D_1 and the other parts in D'_1. In that case, we would get an architecture of components which are organized in the form of a tree. Otherwise, we would get a structure of components organized as an acyclic digraph, where each component forms a complete subsystem when combined with all its subordinate components. A particularly simpler structure would arise if we can decompose, say, $D_2 = D_{21} \cup D_{22}$ such that $D_{21} \cap D_{22} = D_1$, with the corresponding F_{21} and F_{22} having the properties $F_2 = F_{21} \cup F_{22}$ and $F_{21} \cap F_{22} = F_1$, and similar properties for DIC_{21} and DIC_{22}, and likewise for other D_i.

3.4. Analyses of F and *DIC*

One can talk of actually two kinds of integrity constraints C_i: (1) $v(C_i)$ consists of only the input data-items which specify a problem instance that the software is to solve; here, C_i may contain other input events that the software may receive during its operations. These inputs are usually called the "external data" in the dataflow terminology. Such a C_i can be a part of one or more preCond(f), $f \in F$, but not of any postCond(f). (2) $v(C_i)$ contains at least one data-item not related to the problem instance specification, i.e., $v(C_i)$ contains at least one $v_j \in$ out(f), $f \neq f_0$. (We view the initialization function f_0 simply as representing the problem instance specifications, excluding the run-time inputs, into a form for manipulation by the other functions.) Our DIC is concerned primarily with the second kind only. Each constraint C_i of the second kind reflects certain characteristics of the underlying high level algorithm adopted in the software. For the analysis of the relationship between F and DIC, our focus is then the constraints of second kind, and we require that each $C_i \in DIC$ is enforced by the functions $f \in F$ whenever $C_i \in M(f)$. That is, C_i is made a part of postCond(f) for each such f, even if the latter is formulated informally and incompletely. (The difficulties in complete formal specification of pre- and post-conditions is the main reason why formal specifications are still not a common place in most software development projects).

4. Application to a Network-Simulator

We briefly describe now a network-simulator design problem, which we use to illustrate the application of data-integrity constraints. We have a network G with a distinguished source-node denoted by s, which has to deliver n data packets $\{p_i: 1 \leq i \leq n\}$ to all nodes which are at a distance d_0 from s using the policy PQFS (Pump or send packets Quickly and Fetch or request missing packets Slowly). Here, each link is considered to have length 1 and nodes x with distance $d(s, x) = d_0$ are called the destination-nodes. The specific application environment of PQFS has the constraints (a_1)-(a_6) for sending and receiving messages:

(a_1) The nodes x do not have distinct identifications, and thus a data packet p_i (more generally, any message) from x cannot be directed to a specific neighbor of x; the message is sent instead to all neighbors $N(x)$ of x.

(a_2) Because the nodes communicate with each other via radio-transmission, two nodes x and x' can send a message at the same time only if $d(x, x') \geq 2$, i.e., $N(x) \cap N(x') = \emptyset$; otherwise, the messages from x and x' interfere at each node $y \in N(x) \cap N(x')$, preventing y to receive the message from either of x and x'.

(a_3) Because of the noise in the environment, a message from x reaches a neighbor y successfully (without error) with a given probability $0 < p < 1$.

(a_4) The lack of a node's identity makes it impractical to try to determine a shortest sx-path from s to a node x and to force each packet p_i to reach x via such a fixed path; instead, different p_i is allowed to reach x through different sx-paths.

(a_5) As a data packet p_i is sent out by x to its neighbors, it is assigned a time-to-live TTL(p_k) which determines the number of steps the packet can still travel in the network. The TTL(p_i) is set to the current nodeTTL(x)−1, where nodeTTL(x) = max {TTL(p_j): p_j received at x}; thus, nodeTTL(s) = d_0 and nodeTTL(x) $\leq d_0 - d(s, x)$. We initialize nodeTTL(x) only when x receives the first packet with a non-negative TTL, and it can go up when x receives a message with a higher TTL. If x receives a packet p_j with TTL(p_j) < 0, then x ignores the packet.

(a_6) A node x is temporarily labeled as a destination-node if nodeTTL(x) = 0.

The main source of the complexity and most of the computations in PQFS-simulator (more generally, in any networking software) lies in updating and maintaining the integrity of message control information: which message to send, whom to send, when to send, and the message-queues and other data at each node.

4.1. The component D for PQFS-simulator

Let DC(x) be the data-cache of all packets p_j received by x so far and let TQ(x) be the queue of packets in DC(x) that are yet to be transmitted (send out) by x. If DC(x) $\neq \emptyset$ and DC(x) is not full (i.e., $|$DC(x)$| < n$), then x can send retransmission request message RRT(p_j) for a packet $p_j \notin$ DC(x). On receiving RRT(p_j) message by a node $y \in N(x)$, it first puts p_j in its retransmission queue RTQ(y) provided $p_j \in$ DC(y) and p_j is not in the transmission queue TQ(y) or $p_j \in$ RTQ(y) (due to RRT(p_j) from another node $x' \in N(y)$ or a previous RRT(p_j) from the same x that has not been served yet by y). The name PQFS comes from the selection priority for the transmission-message at each x: TQ(x) > RTQ(x) > RRTQ(x), where RRTQ(x) consists of $p_k \notin$ DC(x). To prevent x sending RRT-messages too frequently and to allow a sufficient time for arrival of the requested packets, we maintain nextRRTsendTime(x), which is incremented by RRTdelay(x) (a constant proportional to $|N(x)|$) every time x sends an RRT-message; in addition, we maintain nextRRTtoSend(x) to indicate the next packet in RRTQ(x) for which to send an RRT-message by x, by regarding RRTQ(x) as a circular list.

The middle section of Fig. 2 shows the data-items $v_i \in D$ in our simulator model, where each v_i is a set of items grouped together to simplify the diagram. Here, v_2 is a 0/1 matrix indicating the node-pairs $\{(x, y): d(x, y) \leq 2\}$. The item v_3 gives the state of each node x, which consists of all dynamic information about x. It includes nodeTTL(x), DC(x), TQ(x), RTQ(x), nextRRTsendTime(x), selectionStatus(x) = 0/1 of x for transmission at the current simulationTimePoint, transmMssg(x) = message to be transmitted by x (if x is selected), and the vector transmSuccess[y] to indicate success/failure in transmission of transmMssg(x) from x to $y \in N(x)$. Finally, v_4 consists of the counts of each message-type from each node x, including the counts of their successful transmissions to the neighbors $N(x)$; this is used for performance analysis of PSFQ.

4.2. The component F for PQFS-simulator

The major functions $F = \{f_1, f_2, \cdots, f_{10}\}$ in PSFQ-simulator are shown on the leftside in Fig. 2. There are also several print-functions, shown in shaded boxes on the rightside separated from the other functions because they are used only in the debugging phase and they can be "turned off" during a simulation run. We do not show the external file giving data items v_1 (and the number of runs to be made for a multi-run), which would be the input for f_1. We can think of $\{f_1, f_2\}$ together serving here as the initialization function f_0.

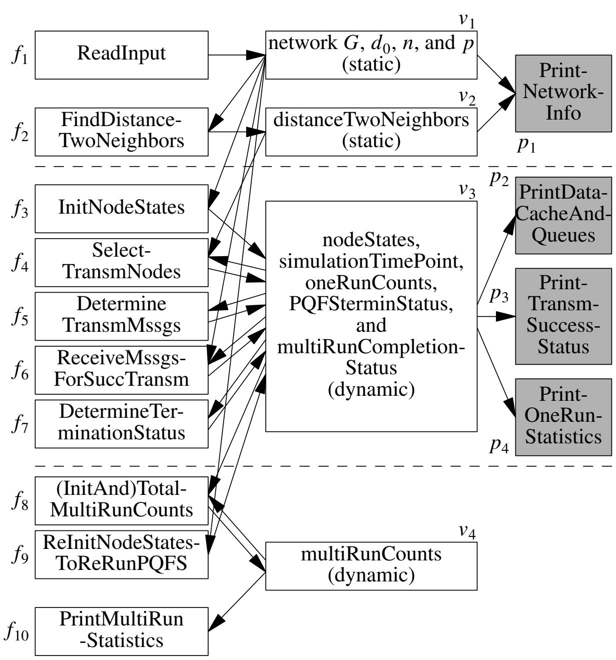

Figure 2. The data-items D, the functions F, and the input-output relationship for PQFS-simulator.

Since v_2 is used only by f_4, one may be tempted to eliminate v_2 to simplify the model by making v_2 a local-static variable in f_4. In that case, we need to eliminate f_2 and make its computations a part of f_4 with those computations to be executed only in the first call of f_4. In addition, the print-function p_1 would have to be broken into two parts as well, with the part that prints v_2 being absorbed within f_4 and to be executed only in the first call of f_4 along with f_2. A similar remark would also apply for v_4 and the function f_8 (which initializes the relevant counts in its first call) and f_{10}. These simplifications, however, hides some important details and are not worth at the design level; moreover, they prevent the formation of the layered-architecture (see Section 4.4) and its resulting advantage of stepwise development and testing (see Section 4.8).

There are several reasons for not combining f_5 and f_6 into a single function f_{56}, say. This would involve delete-operation on some queue at the selected nodes x, add-operation on some queue at nodes y, and many other updates, giving too many inputs and outputs for f_{56} and overloading f_{56} with too many integrity constraints to be responsible for. In is generally safer to separate message-sending action from message-receiving action in a network-software to prevent nodeStates being modified in unexpected ways in a synchronous message-operation as is the case for PQFS-simulator (the constraint C_5 in Table 1 actually prevents this problem in PQFS-simulator). By keeping f_5 and f_6 separated, we simplify every thing without loss of efficiency. For a similar reason, we keep f_7 separate although its input and output are the same as those of f_5 and f_6.

In contrast, it is advantageous to merge f_3 and f_9 into a single function f_{39} = InitAndReinitNodeStates because they have the same inputs and outputs, same relationship to the constraints (see Table 2), and the way they are used (see in Fig. 3). The only difference between them is that f_3 does the additional storage allocation for nodeStates.

4.3. The component DIC for PQFS-simulator

Tables 1 and 2 show the data-integrity constraints and their assignments to the functions in F. We focus on the constraints related to the primary message-delivery operation and thus do not show the constraints like "count(node x is a candidate for message-transmission) $\geq$ count(node x is selected for message-transmission)", which are maintained as a part of nodeState by f_4 and used only for debugging and performance analysis. Likewise, the relationships of total-counts of various items in nodeState before and after an application of f_7 is part of the postCond(f_7), and we do not show them as integrity-constraints. We state the constraints without using predicate formulas for the most part to avoid introducing additional symbolism; our main purpose here is to recognize the constraints in terms of the data-items involved and make sure that they are addressed.

Table 2 shows the non-empty $M(f_i)$ for various f_i. We also briefly indicate how f_i maintains the constraints $M(f_i)$. For some constraints C_i, several functions f_j bear the responsibility to maintain C_i, each on its own as

Table 1. Some key data-integrity constraints.

C_1:	TQ(x) and RTQ(x) are disjoint subsets of DC(x).
C_2:	RRTQ(x) = complement of DC(x).
C_3:	DC(s) = $\{p_1, p_2, \cdots, p_n\}$ and nodeTTL(s) = d_0.
C_4:	For $x \neq s$, nodeTTL(x) = −1 (undefined) unless x received at least one T/RT/RRT-message for a packet p_j with associated TTL ≥ 0. (Thus, it is possible to have nodeTTL(x) ≥ 0 and DC(x) = ∅.)
C_5:	The candidate nodes for transmission is S = {x: TTL(x) ≥ 0, TQ(x) ∪ RTQ(x) ≠ ∅} ∪ {x: nodeTTL(x) ≥ 0, RRTQ(x) ≠ ∅, nextRRTsendTime(x) ≤ simulationTimePoint}, and the selected nodes for transmission is a randomly selected maximal subset $S' \subseteq S$ such that distance(x, y) ≥ 2 for any two nodes x, $y \in S'$.
C_6:	Message priority: T > RT > RRT.
C_7:	If node x is selected for transmission and the transmission-message(x) = an RRT-message, then the new nextRRTsendTime(x) = nextRRTsendTime(x) + RRTdelay(x).
C_8:	Probability p (> 0) that a message from node x reaches node $y \in N(x)$.
C_9:	nodeTTL(x) = max{TTL(p_j): all T/RT/RRT messages received at x}.

it modifies one or more data-items in $v(C_i)$. The isolation of most deletion operations on various queues within f_5 and likewise for the addition operations within f_6 as indicated in Table 2 is a major design simplification in PQFS-simulator.

4.4. A layered architecture for PSFQ-simulator

The analysis in Section 3.1 gives us three layers for the model in Fig. 2 as indicated by the dashed lines. The first layer correspond to $F_1 = \{f_1, f_2, p_1\}$, $D_1 = \{v_1, v_2\}$, and $DIC_1 = \varnothing$. The second layer corresponds to $F_2 = F_1 \cup \{f_3, \cdots, f_7, p_2, p_3, p_4\}$, $D_2 = D_1 \cup \{v_3\}$, and $DIC_2 = DIC$. The third layer is the whole system.

4.5. Call-Relationship in PQFS

Fig. 3 shows the call-relationship for the functions in Fig. 2. It includes three additional functions shown here in boxes with solid lines; they serve as drivers of the other functions. A thin elliptical curve connecting the lines from a node to a subset of its children indicates that those child functions are called iteratively in a cyclic fashion. The child node g of f_6, shown in a dashed box, is not an actual function and is not in Fig. 2; the same remark holds for the children g_1 and g_2 of g. We shown them here only to indicate the complex computation involving multiple layers of iterations within f_6, where g is used iteratively for each node x of the network G and each application of g in turn involves iterations of g_1 or g_2 for all neighbors $y \in N(x)$.

Since there are no dynamic external events in the PSFQ-simulator as run-time inputs to which it needs to respond, the call-relationship in Fig. 3 suffices here to describe the interactions among the functions and we do not use a finite-state model for the latter.

Table 2. The integrity constraints $M(f_i)$ maintained by f_i.

f_3:	C_1, C_2, C_3, C_4. Initialize RTQ(s) = RRTQ(s) = ∅, DC(s) = TQ(s) = $\{p_1, p_2, \cdots, p_n\}$, and nodeTTL($s$) = d_0; for $x \neq s$, let RRTQ(x) = TQ(s), DC(x) = TQ(x) = RTQ(x) = ∅, and let nodeTTL(x) as in C_4 in Table 1.
f_4:	C_5.
f_5:	C_1, C_2, C_6, C_7. For a T-message, delete TQ(x).first; for an RT-message, delete RTQ(x).first if TQ(x) = ∅; finally, for an RRT-message, update nextRRTtoSend(x) and nextRRTsendTime(x) without deleting an item form RRTQ(x) if TQ(x) = RTQ(x) = ∅.
f_6:	C_1, C_2, C_8, C_9. Receive a T/RT/RRT-message at x with probability p if some $y \in N(x)$ is selected for transmission. If a packet p_j is received at x for the first time (via a T/RT-message), then add p_j to both DC(x) and TQ(x) and remove p_j from RRTQ(x). If x receives an RRT-message for p_j, then add p_j to RTQ(x) only if $p_j \in$ DC(x)−[TQ(x) ∪ RTQ(x)]. In each case, update nodeTTL(x) as in C_9.
f_9:	Same as f_3.

4.6. Analysis of Call-Relationship CR

The analysis of CR lead to, after several trials, to the names for f_5 and f_6 as shown in Fig. 2 and having them as two separate functions instead of a large single function. Initially, f_6 was named as AddToDCandUpdateTQandandRTQandRRTQ based on the operations it performed on various queues, but even this long and complex name is somewhat inappropriate as f_5 too modified TQ and RTQ and nextRRTtoSend associated with RRTQ (but not DC). The names of f_5 and f_6 had gone through several changes as the various update operations were being moved back and forth between them in enforcing different parts of the relevant integrity constraints in a piecemeal fashion. The present names are much superior to those earlier ones and they came up immediately once we allocated the integrity constraints to various functions as in Table 2.

4.7. Development plan based on CR and DIC

The usual bottom-up approach, i.e., developing the functions in the post-order according to the call-relationship tree in Fig. 3 is not suitable here. For example, we

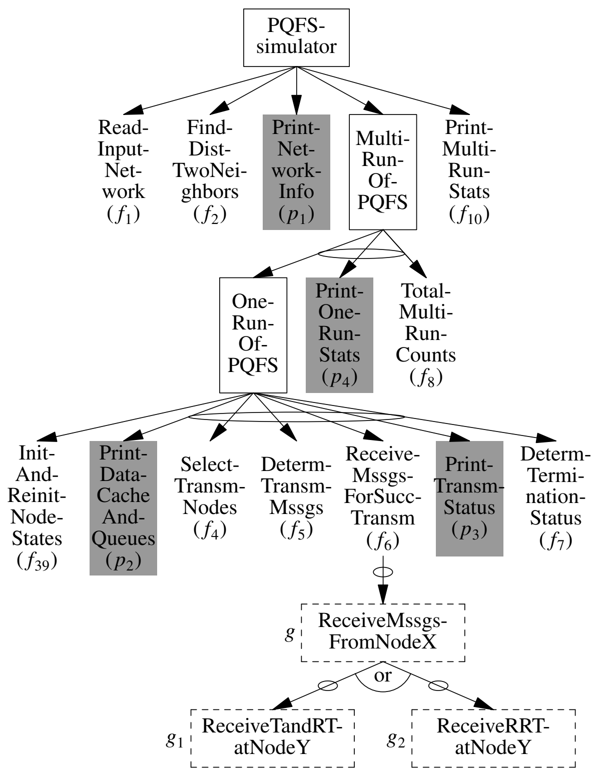

Figure 3. The functions at the child nodes of a node are called by the latter in left to right order.

cannot test f_4 (or f_5) without having tested f_2. In fact, we need f_1, f_2, and f_3 even for a partial testing of f_4; we can fully test f_4 only with all other functions in the loop involving f_4 are in place. The top-down approach (based on the pre-order traversal of the tree in Fig. 3), where one puts empty-body for the functions not yet completed, is also not suitable here. For example, we cannot fully test p_2 or f_5 without having all the other functions in the loop involving them.

Guided by the combined use of the call-relationship and data-integrity constraints and their allocation to the functions, we developed the PQFS-simulator in several phases using a combination of horizontal and vertical extensions. We first created a subset of the functions partially but coherently (so that the implemented features of those functions can still work together) and tested them. Then, we completed those functions more fully by adding some of their remaining features (horizontal extensions), and also added other functions to the list of (partially) implemented functions (vertical extensions), and tested the new group of functions, and so on.

One major phase in our development process were based on implementing the relevant parts of various functions for processing of only T-messages, tacking into account the relevant parts of the various constraints. In then second phase, we added processing of RT- and RRT-messages (which are inter-related) to bring every thing to completion. In each phase, we included some of the output functions to show the detailed time-sequence of events and node-states to help debugging of the program. In this way, we completely avoided any undoing and redoing parts of the previous work and all functions are developed in-situ and often in multiple stages by addition of new features. The structure of the constraints and allocation of their responsibilities to the functions as described in Tables 1 and 2 allowed us to decompose the development process in this way.

The function f_4 is algorithmically the most complex function here; it is also used once in every simulation time cycle and it is also the most computationally expensive part in a time cycle. This means its computational efficiency dictates the overall efficiency of the whole PQFS-simulator. We needed to use some special data-structures to optimize (linear in the number of nodes in the network G) the performance of f_4. For this reason, we developed and tested f_4 for the most part in isolation, using f_2 along with an input network G and the additional input of a subset of nodes S (for the candidate nodes); the output of f_4 is a randomly selected maximal subset of $S' \subseteq S$ such that distance$(x, y) \geq 2$ for any two nodes x, $y \in S'$. In integrating f_4 with the rest of PQFS-simulator, we simply had to replace the input S by the computation of it from the information of various queues TQ(x), RTQ(x), RRTQ(x), etc for nodes x in G.

Shown below are the list of functions developed in the successive phases of the PQFS-simulator:

Phase-1: $\langle f_1, f_2, p_1 \rangle$

Phase-2: A partial development of each of $\langle f_3, f_4, \cdots, f_7, p_2, p_3, p_4 \rangle$, by restricting each function to T-messages except that f_4 is fully developed and tested separately as remarked earlier. The partially developed print-functions p_2-p_4 showed only the values of DC(x), TQ(x), nodeTTL(x), etc. for various nodes;

Phase-3: Completion of f_3 and f_5-f_7 for handling RT- and RRT-messages and completion of p_2-p_4 for RTQ(x) RRTQ(x), etc. for various nodes.

Phase-5: f_8, f_9, and f_{10}.

Fig. 5 shows a skeleton code for testing a Phase-2 version of p_2 = PrintDataCacheAndQueues, which has a complex output format to show the multiple queues (TQ, RTQ, and RRTQ), dataCache DC, and some other information about each node's state.

4.8. Summary of Experience in Using The Model

The proper way of handling the data-integrity constraints in the new model was the key to the successful

```
//uses global variable numPQFSrun (read from
//datafile specifying the network, etc) ;
void PQFSimulator()
{ int i, numSuccRuns;
  ReadInputNetwork();
  for (numSuccRuns=i=0; i<numPQFSruns; i++)
      if (OneRunOfPQFS() > 0) {
         numSuccRuns++;
         //*** InitAndTotalMultiRunCounts()
         //*** ReInitNodeStatesToReRunPQFS();
      }
  //*** PrintMultiRunStats()
}

int OneRunOfPQFS()
{ int i, time;
  InitNodeStates();
  for (time=0; terminationStatus; time++) {
      PrintDataCacheAndQueues();
      //*** SelectTransmNodes();
      //*** DetermineTransmMssgs();
      //*** ReceiveMssgsForSuccTransm();
      //*** PrintTransmSuccessStatus();
      //*** DetermineTerminationStatus();
      terminationStatus = 0;
  }
  //*** PrintOneRunStats();
  if (terminationStatus > 0) return(time);
  else return(0);
}
```

Figure 5. A skeleton code to test print-function p_2.

design and development of PQSF-simulator without requiring any rework. An earlier attempt without an explicit focus on the *DIC*s and trying to fix errors based on observed test-runs proved futile; the manual examination of test-outputs became very tedious and time consuming due to the large number of message-queues and the probabilistic events affecting those queues. A small change in the code or in the probability for a successful message transmission often caused many changes in the outputs that needed close examinations. As much as 10-15% of the effort in this earlier attempt went into redesigning the output formats for the print-functions p_2 and p_3 in Fig. 2 in order to make the outputs compact and sufficiently informative to simplify their manual analysis and error detection. It finally became clear that we needed a different approach to redesign the simulator properly from the start, and the success came in a short few days with the new design using *DIC* after weeks of waisted effort in the first attempt.

5. Conclusion

The key advantage of using the data-integrity constraints *DIC* explicitly in our model is that it forces us to choose both the data-items D and the functions F at an appropriate level as we relate them tightly via the outputs of the functions f_i and the many-to-many mapping $M(f_i)$ between f_i and the integrity constraints C_j. The input-outputs of the functions and the call-relationship (or finite-state model) provide an additional layer of checks for the appropriateness of a design. In the process, we may be required to perform some decomposition or aggregation of our initial choices for the data-items and the functions. Since the integrity constraints can interfere with each other in terms of how we group and define the data-items, we address them jointly in our model. This results in a better and simpler design. The layered structure resulting from the combined analysis of D, F, and *DIC* also helps to determine a clear development and test-plan for the software.

Our future work will expand the use of data-integrity constraints to the object-oriented design.

6. References

1. M. Joseph, Adding formalism to methods or where and when will industry use formal reasoning, *5th Intern.. Conf. on Formal Methods and Software Engineering,* (#LNCS 2885, Springer-Verlag), Singapore, Nov. 2003, pp. 26-33.
2. S. Kundu, The concept of path-closed sets and its application in software functional design, *Proc. of 7th Asia-Pacific Software Engg. Conf., APSEC-2000,* Singapore, Dec 5-8, 2000, pp. 112-119.
3. S. Kundu and G. Nigel, A formal approach to designing a class-subclass structure using a partial order on the functions, *The 29th Annual Intern. Computer Software & Applications Conf., COMPSAC-05,* Edinburgh, Scotland, Jul 26-28, 2005, pp. 213-220.
4. L. Murray, D. Carrington, I. MacColl, J. McDonald, and P.Strooper, Formal derivation of finite-state machines for class testing, *The Z Formal Specification Notation: Zum-98,* (#1493 LNCS, Springer-Verlag), 1998, pp. 42-59.
5. G. Booch, J. Rumbaugh, and I. Jacobson, The unified modeling language user guide, Addison-Wesley, 1999.
6. P. Bourque, R. Dupuis, A. Abran, J.W. Moor, L. Tripp, and S. Wolf, Fundamental principles of software engineering - a journey, *The Journal of Systems and Software,* 62(2002), pp. 59-70.
7. D. Harel, Statecharts: a visual formalism for complex systems, *Science of Computer Program.,* 8(1987), pp. 231-274.
8. J. Jackey, *The way of Z*, Cambrdge University Press, 1997.