Analyzing the Logical Structure of Data Flow Diagrams in Software Documents

G. Butler, P. Grogono, R. Shinghal, I. Tjandra
Department of Computer Science, Concordia University
1455 deMaisonneuve Blvd. West
Montréal, Québec, Canada, H3G 1M8

Abstract

Understanding software documents requires the logical structure analysis of diagrammatic notations, such as data flow diagrams. A data flow diagram (DFD) represents the functional dependencies within a system: it shows how output values in a computation are derived from input values. We show how the logical structure of a DFD can be described using the formalism of Calculus of Communicating Systems (CCS). We present the process of generating the logical structure, and show how the formalism is used for deep understanding of DFDs. Using the Edinburgh Concurrency Workbench (CWB), we can use the formal description of a DFD to reason about the equivalence of two DFDs, and to simulate the behavior of a DFD.

Keywords: document understanding, logical structure analysis, data flow diagrams, software documents

1 Introduction

A data flow diagram (DFD) (Figure 1) represents the functional dependencies within a system: it shows how output values in a computation are derived from input values [10]. DFDs are frequently used in paper software document for so-called legacy systems. Here, we illustrate the benefits of a formal representation of the logical structure of a DFD, and describe the steps of document understanding necessary to generate the logical structure. The formalism is based on Calculus of Communicating Systems [3, 4].

Understanding a diagram requires a number of steps [6, 7]. The system involves two phases: Recognition and Understanding. The sequence of steps included in the recognition phase are as follows:

- Scanning. The printed document yields a digital representation of the DFD.
- Feature extraction yields the components of the DFD, such as lines, arrows, boxes, circles, and text. Feature extraction also includes the determination of relationships between components of the DFD. For example, arrows should link nodes and each label should be associated with a node or an arrow.
- Syntactic Analysis checks that the relationships between features satisfy the rules of a grammar. The analysis detects errors such as a node with no label, or an arrow that goes nowhere.

The output of the recognition phase is the layout structure of the DFD. The understanding phase consists of the semantic processing that yields the meaning of the DFD in a suitable formalism. The output of the understanding phase is the logical structure of the DFD that enables us to check equivalence of two DFDs, simulation of a DFD, and to do top-down design of a DFD.

Techniques for the recognition phase are fairly well understood [2] although this is still an active area of research. In [8, 5] recognition of flow charts is described. A survey [9] presents methods for describing document layout structures. Our contribution is primarily to the understanding phase. We assume that scanning, feature extraction, and syntactic analysis have yielded the layout structure of the DFD in a format that we define below. Our concern is to show how semantic processing enables us to understand the DFD at a deep level.

In section 2, we shall present a semantics for DFDs informally. The notions introduced in section 2 will be used in section 3 to describe the steps used for generating the logical structure of a DFD. Section 4 concludes the paper with a discussion of the advantages of our approach.
2 Semantics

We introduce a semantics for DFDs in which each node of a DFD is associated with an agent and each arrow in a DFD is associated with communication between agents. We describe four functions that are used to construct the logical structure of a DFD from its layout structure: the sequence, or, composition, and restriction functions of the Calculus of Communicating Systems (CCS). Below, we use the DFD of Figure 1 to explain the use of these functions.

There are two actions inherently possessed by P1: “receive a and b as inputs” and “send c and d as outputs”. The sequential ordering of actions is expressed using the sequence function “;”, so the expression “receive a and b ; send c and d” indicates that the action “receive a and b” can only be performed after the action “send c and d”.

The action “send c” is the complementary action of “receive c” performed by the agent T3. For synchronization purposes, we shall write “receive x”, which is read as “receive x bar”, to express the action “send x”.

Furthermore, we need an additional function for expressing alternative courses of actions, which may be determined by the abilities of other agents to interact. The agent P1 may send a to T3, if T3 is able to receive a, or P1 may send d to T4 as well. An interaction between P1 and T4 may happen. Such alternative courses are represented by using the function or which is written “+”. Consequently, we can represent P1 by using recursion and the sequence and the or functions as follows: P1 = (receive a ; receive b ; receive c ; P1) + (receive a ; receive b ; receive d ; P1). Furthermore, we introduce two actions, input and output, that are associated with terminators. These actions are used to represent the input from a source and the output to a sink.

The composition function “I” is used to express the interaction between agents. Two agents, which interact with each other, can be composed into one agent by using this function. For example, P1 °T3 expresses an agent in which P1 and T3 may proceed independently but may also interact through the complementary actions “send c” and “receive c”.

A transition of the form E -ap E' indicates that agent E can perform the action a and becomes E'. Consider the composition (x ° E)(x ° E'). If the agent (x ° E) performs the action x and becomes E and, simultaneously, the agent (x ° E') performs x and becomes E', the composition will become E'F'.

Consequently the DFD in Figure 1 can be described as in Figure 3.

The layout structure of a DFD may correspond to several logical structures that are semantically equivalent, that is, they possess the same behavior. In Figure 3, we use the notations for agents and state spaces of the Edinburgh Concurrency Workbench (CWB) [1] to show a description of the logical structure of process P1 of Figure 1.

3 Translator

In this section, we describe how to generate a logical structure of a DFD based on its tuple representation (Figure 2). The tuple representation of the layout structure is produced in the recognition process. Each
tuple represents a node of the DFD and contains information about a node. The set of tuples is subdivided according to the hierarchical levels of the DFD. For example, in Figure 2, each of the labels: 0, 1, and 2: is followed by the tuples representing the nodes at the corresponding level. Each tuple contains information of a node, such as:

- the kind of the node: SOURCE-TERMINATOR, SINK-TERMINATOR, PROCESS, DATA-STORE, or AUXILIARY-NODE;
- the identifier; for example, T1, T2, T3, P1, P11, P12, A, B, C, or D;
- the internal process number representing the position of the process in the hierarchical structure of the DFD; for example the internal number of the process P11 at level 1 is 0.1 indicating that P11 is a process in the refinement of the process with number 0. The process with number 0 is P0 at level 0;
- the external input or output in a refinement of a process, if available; for example the external input of the node P11 at level 1 is a and b; and
- the successors of the node and the data flowing to the successors; for example, the successors and the data going out from P1 at level 0 are (T3, c) and (T4, d).

The parser checks both syntactic and structural correctness of the tuple representation, and if no errors are found, the parser generates a DFD tree, that represents the layout structure. A grammar is used to check the syntactic correctness. Structural correctness relies on certain constraints being satisfied. Such constraints are that there is no connection between a terminator and a data store, and that all processes have at least one incoming and one outgoing arrow.

The task of the translator is to generate the logical structure of a DFD from its layout structure described as a DFD tree. The translation process is based on the semantics that have been described in Section 2. For these purposes translation rules are introduced. To illustrate how the logical structure of a DFD is generated, we use our running example of Figure 1.

The logical structure of the DFD is generated by traversing the DFD tree. The translator generates CWB code for each node traversed. The DFD tree is traversed level by level, in the sense that the nodes of level n of the DFD are traversed fully before the nodes of level (n + 1).

For each node, N, being traversed the translator generates code based on the kind of node as follows:

1. If the node is a process then actions corresponding to receiving and sending data must initially be captured. We adopt the convention that one or more inputs of a process suffice to compute the outputs. When a process requires all of its inputs to compute the outputs then we shall use an auxiliary node to express this situation. The treatment of an auxiliary node will be described below. The agent expression representing a process is defined recursively and uses the or and sequence functions. Returning to our example, for the process B at level 2, where p is the only input and q is the only output, the translator produces the expression: B = p . q . B.

2. If the node being traversed is a data store, then actions representing inputs can be performed independently of the actions representing outputs, since a data store does not perform calculations to derive outputs from inputs.

3. In case of a source terminator, the only task of this node is sending data to other processes. If there are more than one input, the sending of each input depends only on the ability of the corresponding process to receive it. For synchronization purposes, we introduce the special action input which precedes the action of sending the data. In our example, the source terminators T1 and T2 are translated to: T1 = input . a . T1, and T2 = input . b . T2.

4. If the node is a sink terminator, then it behaves similarly to a source terminator. Hence, we need to introduce the special action output for synchronization purposes. In our example, the two sink terminators are translated to: T3 = c . output . T3, and T4 = d . output . T4.

5. If the node is an auxiliary node then we consider all permutations of the inputs. There are no auxiliary nodes in our example.

6. Finally, we have to establish connections between the agents representing the nodes in a DFD. The connections of communications between the agents are represented by agent composition. For each refinement of a process P, and for the DFD at level 0, the translator creates the following expression: Let N1, ..., Nk be the nodes of the refinement of P, and Internal be the set of internal data flows through the refinement, then
the translator generates the expression: \( P = (N_1 \cdots |N_x) \setminus \text{Internal} \). In our example, the communication of the nodes at level 0, 1, and 2 are represented as:

\[
DFD_1 = (T_1 | T_2 | T_3 | T_4 | P_1) \setminus \{a, b, c, d\}
\]

\[
P_1 = (P_{11} | P_{12}) \setminus \{a, b, c, d\}
\]

\[
P_{11} = (A | B) \setminus \{q\}
\]

\[
P_{12} = (C | D | E) \setminus \{r, s, v, u\}
\]

The complete logical structure of the DFD, as produced by the translator, is illustrated in Figure 3.

**Figure 3: Logical structure**

### 4 Conclusion

The steps in analyzing the logical structure of a data flow diagram as implemented in our translator have been presented. A formal representation of the logical structure based on CCS is generated by the translator. This allows deep understanding of DFDs. For example, CWB can determine the behavioral equivalence of two DFDs with quite different layout structure; and it further allows simulation of the behavior of a DFD.

Future work on understanding software documents might consider further annotations to DFDs, such as constraints on temporal behavior, and may also consider other diagrammatic notations.

**Acknowledgements** The research described in this paper was funded by the Natural Sciences and Engineering Research Council of Canada, *Fonds pour la Formation de Chercheurs et l’Aide a la Recherche*, and the Institute of Robotics and Intelligent Systems.

**References**


