A DFD Based Approach to Model Data Processing Solutions

Falko Schmalenberg*, Ralf Vandenhouten

Abstract

This paper introduces procedures, which facilitate the development of as well as the linking between modular data processing processes into different and versatile system solutions. Based on the concept of data flow diagrams (Yourdon 1989), processes and data flows are used. With the ability of a process to harness any amount of in- and outputs with different sources and destinations, it is possible to model any conceivable data processing problem. By aggregating a diagram as a new process, it becomes possible to outsource partial process chains to different abstraction levels. As a result, even very complex solutions can be modeled and explained in a clear way. The introduced uniform termination flow while traversing the graph creates a good readability, which also makes it possible to quickly detect weaknesses. Moreover, this solution is comprehensive and advances the cooperation in teams and development.

1. Introduction

Nowadays, software is being developed in almost every area. It is intensively used in our private, our social and our business life. Since the early stages, architects, programmers and other engineers have invented rules, methods and tools to improve the development process with regard to different aspects like:

- speeding up the creation time
- reducing the effort
- reusing program elements
- maintaining order and structure, even in gigantic systems

Design patterns can be seen as the major rulebook for software development. This very popular collection of templates has become indispensable. It was first introduced by the, so-called, gang of four in their book: Design Patterns. Elements of Reusable Object-Oriented Software (Gamma et al. 1994). Apart from providing uniform solutions for certain contexts, the major benefit, besides reusability, is the improvement of readability. Developers, who have not yet been involved, can better understand existing software and start to contribute. This improves teamwork and allows for the creation of powerful software systems.

Model-driven software development (MDSD) is another section of mechanisms to improve the development process. It provides techniques to automatically generate software from previously created models (Stahl & Völter 2006). There are many MDD tools and frameworks available. The eclipse-based solutions, for instance, are as powerful as they are popular (Eclipse Foundation 2015).

Unified Modeling Language (UML) is a graphic based modeling language for software development (International Organization for Standardization 2012). It is an ISO Standard, famous and often used to specify, construct and document software parts. It covers many, not to say almost every, possible programming facet. It became essential for software engineering.

Besides the development, maintaining the software is the most expensive and time-consuming factor. In general, more than 70 % of the total costs represent maintenance (Polo et al. 2003). This underlines once more how important it is to keep and improve code readability. Therefore, methods of structured analysis can be helpful.

Zusammenfassung


*corresponding author
One of them is the Data Flow Diagram (DFD). A DFD visualizes the flow of data through an information system and its functional processes. It clarifies, where data is created and consumed. There are different terms like: Bubble Chart, Process Model or Functional Model, which can be used synonymously, as well as different notations. Figure 1 illustrates a simple DFD example. The DFD is an important and powerful tool used in the field of systems analysis. The modeling technique presented in this paper is based on the DFD model. The following DFD elements and notations are used:

1.1 The Process

Figure 2 shows an example of a process. The notation used in this paper is a circle containing the name of the process. A process is a part of the system that consumes input data and creates output data.

1.2 The Flow

Figure 3 shows an example of a flow. A flow describes the movement of data from one part of the system to another. It represents data in motion. For representing data at rest, the store (see 1.3) is used.

1.3 The Store

Figure 4 shows an example of a store. An oval shape is the notation used in this paper. The store is used to model (a collection of) data at rest.

1.4 Problems

Data flow diagrams do not address all aspects of data processing. In particular, procedural features are missing.

Referring to figure 1, it is not determined (by help of a symbol or sign notation), which process will act first, and which process or even processes will act subsequently, and so on. There is no implicit or explicit order. Moreover, according to process RECEIVE ORDER, it is uncertain if the flow orders of the CUSTOMERS take place before the flow invalid orders. Data flow diagrams require intelligent reading, and are, therefore, suitable for humans, but not for machines. Thanks to our knowledge, we are able to integrate additional information from the meaning of the words (of processes, stores and flows), and extract possible procedures and preconditions. It has been one important motivation of this work to enable computers to execute previously modeled graphs (without the need of further explanation). In order to achieve this, we enhanced the DFD and added new elements like a distinct sequence policy (section 2.3).

Another limitation of the current model is the fact that flow arrows only indicate the direction of data motion. It is not apparent, however, if data is pushed from a source or requested by a sink. Section 2.2 shows our solution to remove this deficit. This is not the first attempt to extend Yourdons design methodology. Decomposing DFD and state transition logic diagrams (STLs) was a different approach to design and implement a software system (Molla 1989).

2. Solution

This solution of a data processing model (we will use the acronym DPM in this paper) addresses issues in regard to modeling software data processing by taking the DFD-concept and adding new elements. To make the notation more compact, some DFD-elements are summarized. For instance, a process, which creates and stores data, can be represented in DFD-notations as follows:

Figure 5 shows an example of a data producer in DFD.
This will be compacted as follows:

2.1 Nodes and Slots

The place in which data is read, modified, and/or created, is called node (e.g. Camera). A result is placed in a store, which will be called slot. A node is able to handle any number and any kind of data. Thereby, each input is read from a corresponding input slot, and each output is written into a corresponding output slot. Figure 7 shows how a node with several in- and output slots is derived from a DFD. The DPM representation, in contrast to the DFD notation, combines stores and their using process into one unit. As a consequence, the representation of constellations with many processes and data becomes more obvious. This is possible, as in DPM there never is a direct flow from one process to another. The data transfer always takes place over stored data (slots).

Figure 8 pictures a process named Stereo Processor consuming the data from two webcam processes and creating two kinds of data output, two images used for creating a depth image and a point cloud. However, the DFD notation does not tell us how, when and under which circumstances the Stereo Processor is obtaining the input data and creating its outputs. Due to the determination of input and output slots the direction of a flow is now able to illustrate whether inputs are being pulled or pushed from outputs of other previously acting processes. Using DPM the system pictured in DFD-Fig. 8 can be transformed in several different but more explicit models (see Fig. 9).

2.2 Push and pull connections

The data flow from an input slot to an output slot is called pull connection. Vice versa, the data flow from an output slot to an input slot is called push connection. An input slot can have a zero or one-pull connection. An output slot can have zero up to any number of push connections. A node contains an algorithm, which can be executed. Dependent on connections via slots, other nodes (predecessors and/or successors) will be executed as well. This solution introduces a consistent execution rule. Figure 10 explains the process for a node, which is about to be executed.
2.3 Execution process

First, it is checked whether the current node was executed during the current execution process chain. If it was not, the node will be marked and prepared for execution. Therefore, the availability of input data has to be ensured. This is done by following all pull connections, from the input slots of the current node to predecessor nodes. These predecessor nodes will be executed according to the same rule. This means that in terms of graph theory a depth-first search takes place to procure the inputs. After that, the program or algorithm of the current node will be executed.

Side note: It is not sure whether the execution will be successful because not all inputs might be available by pull-connections. Some may be pushed by other nodes, which have not been executed yet.

In case of execution results, output slots of the current node are containing new data. This data will be written into input slots of successor nodes according to the push connections. Each successor will then be executed with the pushed data by the same schema. This means a depth-first search is also used for committing the outputs by following the push edges to successor nodes. In conclusion, it might be possible that by executing one node all other nodes in the graph will be executed as well, depending on the constellation of push-and-pull connections.

2.4 Executors

Finally, in order to enable the modeling of procedural aspects, a new notation is added. A cycle attached to a node indicates the availability of an executor. Such an executor declares where the execution process (introduced in section 2.3) is initiated. It always belongs to a concrete node, and can be triggered from the outside to start or stop its execution run. When executed, it traverses the nodes and executes its containing programs, as explained before. During one execution run, a node can only be executed once. Upon completion, it depends on the executor type if, for instance, the executor terminates or sleeps for a certain period of time, and afterwards runs again. Figure 11 complements figure 9 by adding the minimal amount of executors for making each of the four constellations work (although it is possible and occasionally useful to use more executors).

The example mentions two types of executors. ‘L25’ means that after being started, this executor loops until stopped from the outside. It sleeps a computed time between each period to ensure that it executes 25 times within a minute. ‘S’ stands for single execution and means that it immediately runs for one time after being started.

Figure 11b only uses one executor. After starting, it first executes both Camera-nodes because both inputs of a Stereo Processor have pull connections. The executor in figure 11c starts at the right Camera and pushes the color image to the Stereo Processor. In order to then execute the Stereo Processor, the other input is needed. As this input is connected via pull connection to the other Camera, the second Camera is executed, and afterwards, finally, the Stereo Processor. Figure 11d works similar to figure 11c by starting at the left instead of the right Camera. All three figures 11b-d have in common that...
their constellation ensures that all nodes are only executed in order to create the results of Stereo Processor. In contrast, figure 11a shows executors creating results that are not only necessary to procure the results of Stereo Processor. The left and the right camera are executed 25 times each within one second. However, Stereo Processor is executed only on demand. This constellation makes sense if there is, for instance, a live view that demands the results of the webcams in order to achieve a fluent display. But if the L25-executors do not run, the Stereo Processor does not create results when executed because its input slots do not contain data (because of the lack of pull connections).

2.5 From node description to node instances

Reusable modular program elements (algorithms) are kept in node descriptions. A node description describes to the outside, which inputs it uses and what outputs it creates. For each input and each output, there is a data description defining:

- the type of data,
- the presence of a default value,
- a range (in case of numeric and if appropriate),
- or an enumeration of valid values (if appropriate).

Each input and each output have a unique naming within such a node description. During the execution, the algorithm only works with these synonyms. The standardized execution environment ensures the linking between each synonym and an actual data value at runtime. The algorithm does not have to perform other tasks like locating and obtaining input data from distributed sources. As a consequence, each algorithm also is decoupled and independent from other algorithms or acquisition tasks.

Each node description is the basic model for related node instances. For instance, the two Camera nodes of figure 11 are related to the same node description, holding the algorithm for acquiring data from an USB video device. The solution provides a clean separation between the declaration and the usage of modular program components. Developers of algorithms are able to create solutions independent of the progress of others by declaring self-explaining input synonyms. Users without development skills are also able to create (complex) processing solutions by reusing and linking together existing sub solutions.

2.6 Metadata

All data handled (read, processed, created or manipulated) in DPM are encapsulated in a metadata container. The metadata holds additional information about the encapsulated data. There are two kinds of information:

a) Information, which is related to the individual data type, for instance in case of an image: color format, height, width etc.

b) The other group consists of common information, available in each metadata independent of the kind of encapsulated data:

- a timeframe explaining the time interval, when the data is valid (when it occurred in reality)
- a timeframe explaining the time interval, when the data was handled by a node
- the name of the node
- the (IP) address of the node
- the output slot name in the node

Data and metadata are detached from slots and nodes and can be distributed independently. The metadata allows for merging information in time as well as in content at any location and from different sources. This makes distributed processing and load balancing possible.

2.7 Aggregation and compositum feature

Another feature of DPM is the possibility to aggregate an existing graph with a certain number and constellation of

Fig. 9) Push-and-Pull Connections in DPM

Fig. 12) Aggregation example
nodes and connections to a compositum node. Figure 12 shows an example. By default, all slots involved in push or pull connections disappear. Only the slots without connections are available for use outside the new compositum node. This characterizes a graph as a new node, which can be part of another graph as well. This makes it possible to outsource partial components and work on a higher abstraction level. Thus, even very complex systems can be modeled without a loss of structure or overview.

Similarly, the idea and inner structure of a solution can be hidden as a black box while keeping its functionality.

2.8 Export feature

Referring to section 2.7, it is also possible to easily export solutions as a compositum node. As not only the processing flows, but also the access to originated outputs is standardized, it is easy to integrate DPM solutions into external programs.

3. Conclusion

The concept presented forces developers to create standardized solutions. As each node, its slots and the data-flow can be visualized in a self-explaining overview, it encourages the user to split a big and complex processing solution with many sub-steps into several modules. As a result, it is more likely that existing solutions are being reused and the degree of redundancy is minimized. Because of the uniform program flow and the abstraction feature (see section 2.7), the visualization also supports the understanding and speeds up the development process. Therefore, new developers can be integrated faster.

Compared to visualization tools like DFD or UML, DPM provides a bijective relation between the visualization and the realization of the program. In UML class or object diagrams, for instance, it is possible to generate POJOs from the visualization model. It is, however, not possible to derive procedures and sequences. On the contrary, and again because of the uniform processing model (see section 2.3), DPM can do this. At any time, there is a distinct visual representation of the current data processing system. As a consequence, there are advantages and savings in costs and time – not only in terms of development, but in maintenance as well.

As the nodes are decoupled and all handled data come with additional metadata (see section 2.6), the solution can be used for the computation of hard problems. The approach is also suitable for distributed systems and load balancing. In order to implement this, DPM has to be enhanced further. The concept and realization of push- and pull connections over the network will be one of the future tasks.

REFERENCES


AUTHORS

Falko Schmalenberg, M. Eng.
Prof. Dr. rer. nat. Ralf Vandenhouten
Technische Hochschule Wildau
Forschungsgruppe Telematik
E-mail for correspondence:
falko.schmalenberg@th-wildau.de