

Use of Data Flow Diagrams for Building Process with Message Passing: A Parallel Design Proposal

Mario Rossainz López, Mireya Tovar Vidal, Nallely Morales Lozada,
Jesús Alberto Islas Fuentes

Benemérita Universidad Autónoma de Puebla,
Facultad de Ciencias de la Computación,
Mexico

{rossainz, mtovar}@cs.buap.mx,
{nalle.ml29, albertisfu}@gmail.com

Abstract. The present work shows a method to design parallel programs with message passing using Data Flow Diagrams (DFDs), which are graphs that are used within the classic structured design of Software Engineering. It shows the modification and semantic adaptation of the graphic elements of the DFDs to the semantics of the elements that are used in the passing of messages of the concurrent / parallel programming to map the processes, communication channels, geometric parallelism, parallel composition, generalization, specialization and nesting of processes to the graphic elements of DFDs so that the design made with them turns out to be a transparent design in the coding of a parallel system that uses processes, communication channels, composition and process nesting. The complete parallel design assisted with DFDs of an application called Parallel Generator of Natural Numbers is shown to demonstrate the usefulness of the proposal.

Keywords. Data flow diagram, parallel programming, message passing, distributed memory, parallel design, structured design.

1 Introduction

There is interest by a large part of the community dedicated to computational discipline, to develop increasingly powerful systems taking advantage of the benefits of current computer architectures. The most important approach in the development of such systems is the use of parallel programming at various levels

Parallel systems are found today in practically every type of device, not only in computers, the same are present in embedded devices in household appliances (screens, audio components, refrigerators, video players, etc.), than in supercomputers of institutes, universities, the army or government agencies [1].

But the way in which this parallel software is developed in order to make efficient use of these components takes great relevance with respect to the parallel algorithms designed and implemented, adapting them to the different existing platforms.

Within this area, the knowledge that is had on the concepts related to the design of programs and parallel algorithms takes great importance to implement efficient parallel programs in its logic, in its execution and in its performance. The parallel program design model adopted in this work is the so-called channel-process model [1]. This model consists of a parallel message passing architecture whose components are two: the processes that have tasks to be performed and the means of communication between said processes to share information through the concept of the channel [2]. A process consists of a sequential program to which a task is associated, a locality of non-shared local memory and a collection of ports of inputs and / or outputs that are the communication channels [3].

The channel used is a zero capacity storage structure (that is, only one piece of data travels through the channel, sent by one process to another that are connected to each other by the same channel; where a sending process sends a message through the shared channel and the receiving process receives it, so that while the message or data is within the channel, the sender will not be able to send any other data until he knows that the receiver has already received the message and the channel will be empty and ready to store another message of the same type as the previous one). A process that is connected to another through a communication channel and that is a receiver will be blocked in its execution if it tries to receive a message that has not yet been sent. In the same way the sending process will be blocked in its execution if when sending a message through its communication channel it has not been received by the receiving process. Therefore, the communication between the sending process and the receiving process through the communication channel is synchronous, thus generating the concept of rendezvous, [3].

The classic design methodology for this type of parallel programming as indicated by [4] is the one proposed by Ian Foster: to divide the computation (tasks) and the data into pieces, determine the communication patterns between the tasks that they will be the processes, generate the composition of tasks / processes (nesting tasks) and assign each process to a processor or thread of execution. The idea is to divide data as independent as possible and then determine how many processes should be created and how to associate those processes with the data. A functional decomposition is generated that consists of dividing the total calculation into several processes and associating the data with them. There will be processes that can be divided into simpler processes generating more decompositions and there will be processes that can be joined and nested in a composition of more processes. The objective is to determine primitive processes that can no longer be divided and that indicate the starting point of the more general processes that make up the system, giving rise to a degree of fine or coarse parallelism; that depends on the problem to be solved and the input data [4]. Ideally, determine as many primitive processes as possible in order to maximize the degree of parallelism.

Finally, the communication patterns between primitive and non-primitive processes must be determined through the creation of communication channels.

The processes can carry out both their communications and the execution of their associated algorithms in parallel. If we analyze and relate the design methodology of Ian Foster with the method of use and creation of a data flow diagram or DFD we can see a one-to-one mapping of the components of the Foster methodology with the graphic components of the DFDs, so that this type of graphs can help the novel programmer in parallel to propose a good design of a parallel application.

This is the proposal of this research work that is organized as follows: section 2 shows the characteristics of DFDs, their components, the types of connections and how to build them, section 3 talks about parallel programming with message passing, section 4 shows how to design a parallel system using DFDs and applies the methodology proposed in a case study, and finally in section 5 the conclusions and future work are shown.

2 State of the Art

According to [5] the design of a software system describes the organization of the system, expressed in terms of its components, the relationships between them, the relationships with its environment, and the fundamental principles that govern the design and evolution of the system.

The design of a system focuses on what is known as "viewpoint", which is a form of abstraction that focuses on some specific aspects of the system, abstracting from the rest. As there are different viewpoints based on the generality and particularities that each software engineer uses to define a design, it is necessary to use unified graphic models that correctly represent the logical, process, development and physical point of view of scenarios etc., of the system to implement. In the case at hand, the viewpoint of the design of parallel applications using message passing can be unified with different graphic models, from contextual diagrams to represent the components that make up the system, generating a software architecture, to the use of diagrams of transition of states in UML, but without a doubt an ad-hoc graphic model for such a design is the use of Data Flow Diagrams (DFD's) that is used in the classic structured design of Software Engineering.

Currently there are several works in the literature that make use of DFDs to show the design of the parallel systems that they propose. In [6] shows the Design of Applications in Distributed Systems with the two possible environments, conventional Operating Systems and the Internet, using DFDs to model the design of the concept of specialization of a service in the client-server architecture. In [7] the design and implementation of a distributed video on demand application based on the client-server architecture is shown, where it uses the DFDs to design the exchange of RTP and RTCP packets in the request of the video service.

3 Data Flow Diagram (DFD)

A DFD is a network-shaped diagram that represents the flow of data and the transformations that are applied to them when moving from the entrance to the exit of

a software system [8]. DFDs are commonly used to model the functions of a software system and the data flow between them at different levels of abstraction in a concept of structured design within software engineering. The software system is modeled through a set of level DFDs in which the upper levels define the system functions in general and the lower levels define these functions in detail [8].

3.1 Components

- **Procedures:** They represent the functions of a software system. A process represents a function, procedure or operation system, that transforms the input data streams into one or more output streams [9]. Its graphic representation is a circle and inside it includes a number and the name that represents the function which must be unique within the DFD (see Fig. 1).
- **Storage:** Represent the data stored in a specific structure such as a database or a file. A data storage represents system information stored temporarily or permanently [9]. The storage is a logical repository in the DFD that can physically represent a file, a database, a file cabinet drawer, etc. In a DFD there may be more than one different data storage. Its graphic representation is two parallel bars (see Fig. 1).
- **External Entities:** They are the sources or destinations of the system information. It represents a source (generator) or destination (consumer) of information for the system but that does not belong to it [9]. It can represent a subsystem, a person, a department, an organization, etc., that provides data to the system or that receives it from it. They are represented in the DFD by a square with a representative name of the external entity inside (see Fig. 1).
- **Data Flow:** they represent the information flows that flow between the functions of the system. It is a path through which data travels from one part of the system to another. They represent the moving data within the system. Data flows are the means of connection of the DFD components [9]. They are represented with directed arcs, where the arrow indicates the direction of the data, (see Fig. 1).

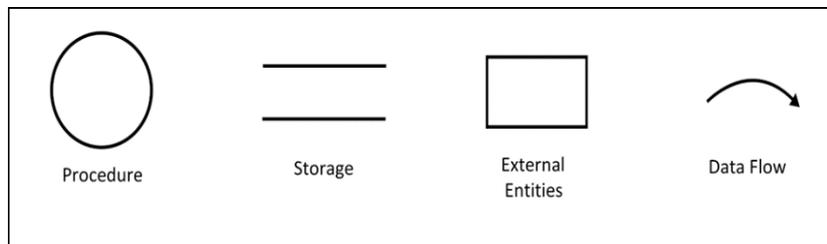


Fig. 1. Graphical representation of the components of a DFD.

Table 1 shows the connections allowed in DFDs [10].

Table 1. Allowed connections between the components of a DFD.

Destination/ Source	Procedure	Storage	External Entity
Procedure	TRUE	TRUE	TRUE
Storage	TRUE	FALSE	FALSE
External Entity	TRUE	FALSE	FALSE

3.2 Connections

The Procedure-Procedure connection: The direct connection between two procedures through a data flow is possible if the information is synchronous, that is, that the target procedure begins at the moment when the source procedure ends its function. If this is not the case, it is necessary that there is a temporary storage that saves the data of the origin procedure [10, 11]. The target procedure will then capture this data when needed (see Fig. 2). **The Procedure-Storage connection:** There are different types of connections that can be made between procedures and storages in a DFD. The Fig. 3 illustrates the following cases:

- **Query Flow:** shows the use of the storage information by the procedure for one of the following actions: use the values of one or more attributes of a storage occurrence or check if the values of the selected attributes meet certain criteria.
- **Update Flow:** Indicates that the procedure will alter the information that is in the storage to: create a new occurrence of an existing entity or interrelation in the storage, delete one or more occurrences of an entity or interrelation and modify the value of an attribute.
- **Dialogue Flow:** Between a procedure and a storage it represents at least one query flow and one update flow that have no direct relationship. Dialog flows are also used to simplify the interface between two components of a DFD.

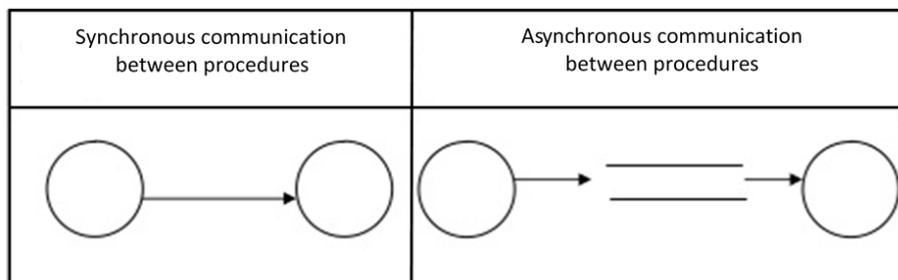


Fig 2. Procedure-Procedure Connection in a DFD.

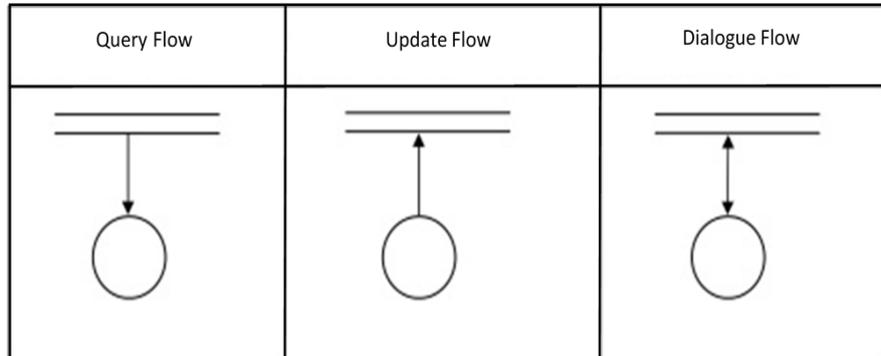


Fig. 3. Procedure-Store connection in a DFD.

3.3 Construction

The construction of a DFD is based on the principle of decomposition by levels of detail. The idea is to generate a model of a system represented by DFDs through layers.

The decomposition by levels allows the system to be analyzed from the general scope to the detail through successive intermediate levels (top-down approach), according to. This form of design of a system provides us with a number of advantages:

- it helps to build the specification from top to bottom,
- the different levels can be addressed to different people,
- independent functions of the system can be modeled at the same time,
- it facilitates system documentation since each diagram can be explained separately.

Thus, the decomposition of a process in a DFD produces another DFD (see Fig. 4):

- **Context Diagram:** It is the highest level of the hierarchy in the design of a system. In this diagram there is only one procedure that represents the complete system.
- **System Diagram [LEVEL 0]:** It is the decomposition of the DFD of the Context diagram into another DFD in which the main functions of the system or subsystems are represented.
- **Middle Level Diagrams [LEVEL 1, 2, 3, ...]:** These are the DFDs that result from the decomposition of each of the DFD procedures of the System Diagram into new diagrams that represent simpler functions.
- **Primitive Function Diagrams [Level n]:** These are DFDs that represent functions that are detailed enough that the creation of new DFDs is not necessary.

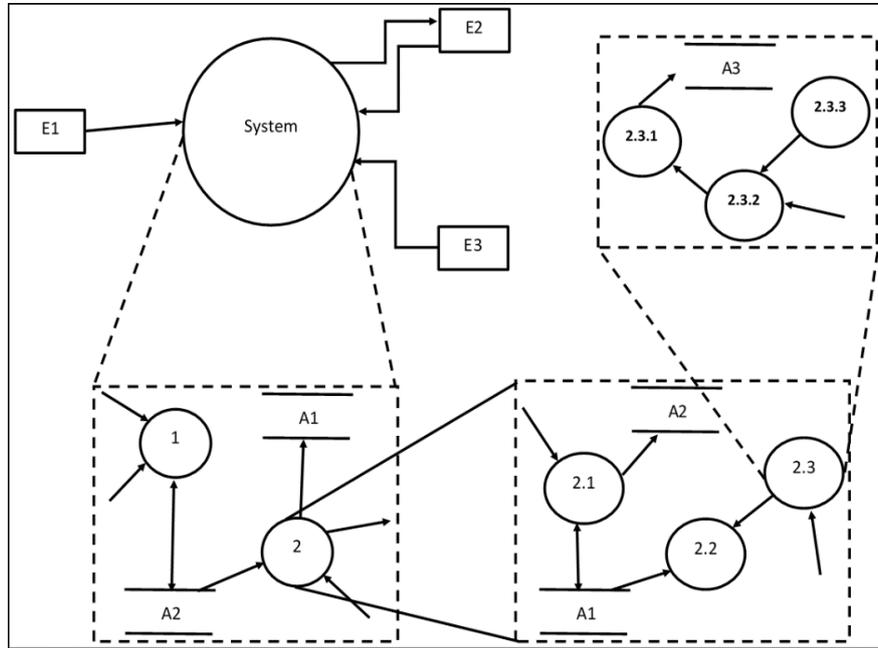


Fig 4. Decomposition by levels of a DFD.

4 Parallel Programming with Message Passing

The natural way to communicate and synchronize processes in this type of systems is using message passing: The processes exchange messages with each other through explicit send and receive operations that constitute the basic primitives of any system of communication of this type [12]. The fundamental elements involved in the communication in systems with message passing are: a sending process (transmitter), a receiving process (receiver), a communication channel (channel), the message to be sent/received (message) and operations of sending (send ()) and reception (receive ()), see Fig. 5.

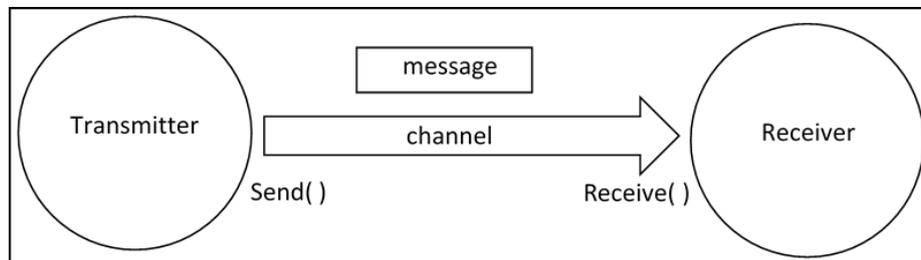


Fig 5. Message passing.

4.1 Types of Communication Between Processes

It is the way in which the sender indicates to whom the message is addressed and vice versa, that is, the way in which the receiver indicates from whom he expects a message. We talk about direct or indirect communication [12].

- **Direct-Symmetric Communication:** It is characterized in that the sender explicitly identifies the recipient of the message in the sending operation. The receiver, in turn, identifies the sender of the message and establishes a communication link between them.
- **Direct-Asymmetric Communication:** The sender continues to identify the receiver, but the receiver does not identify a specific sender.
- **Indirect Communication:** The sender and receiver processes are not explicitly identified. Communication is done by depositing messages in an intermediate storage that is supposed to be known for processes interested in communication. That intermediate store is called the mailbox.
- **Indirect Communication with Channels:** The sending and receiving operations are carried out through the specification of a channel (communication link), which generally has an associated type and on which only data of the same type can be sent. In addition, a channel cannot be used by multiple transmitters and receivers and is unidirectional. The concept of channel arises in languages like Occam and Pacal-FC.
- **Asynchronous Communication:** The sender can carry out the sending operation without it being necessary to coincide in time with the receiving operation by the receiver. It is necessary to store the messages in buffers until the receiver removes them.
- **Synchronous Communication:** The sending and receiving operations by senders and receivers must coincide in time, that is, the sender will be blocked in the send operation until the receiver is ready to receive the message, or vice versa, the receiver will be blocked in the receive operation until the sender sends the message. The processes will be blocked until the coincidence of both occurs in time (rendezvous).

4.2 Channels and Messages

- **Data flow.** Once a communication channel between sender and receiver is established, this, according to the flow of data that passes through it, can be of two types: unidirectional or bidirectional. For the first, information always flows in one direction between the two interlocutors, while for the second, information flows in both directions.
- **Channel capacity.** It is the possibility that the channel has to store the messages sent by the sender when they are not received immediately by the receiver. The channels can be of zero capacity (where there is no buffer where messages are stored), channels of finite capacity (where the existing buffer has a fixed size) and channels of infinite capacity (where the buffer associated with the link of communication is assumed infinite in its capacity).

- **Message size.** The messages that travel through the channel can be of fixed length or of variable length.
- **Channels with type or without type.** Some communication schemes require defining the type of data that will flow through the channel, imposing the restriction of sending data only of the type for which the channel was declared.
- **Message passing by copy or by reference.** Communication through message passing requires sending information between the processes involved in the communication. This can be done in two ways: make an exact copy of the data (message) that the sender wants to send from his address space to the address space of the receiving process (passing by copy or value) or simply send the recipient the address in the address space of the sender where the message is located (passing by reference). The latter requires that processes share memory.

5 Designing a Parallel System with Data Flow Diagrams

A parallel program consists of several processes that run at the same time [13]. A process is a sequential program in execution. The parallel program has control over its processes. Within the human cognitive aspect, the composition of processes through more processes, that is, through simple processes build more complex processes; It becomes natural when parallel programs based on distributed memory are developed [13]. The processes are highly independent because they are limited to the task they have to solve and provide the fundamentals of the program structure. In the field of object orientation, processes can be considered as active objects because they have the capacity to execute themselves. During the analysis phase of the development of a parallel software, the tasks to be performed by the program are determined, among other things [14].

However, it is in the design phase that these tasks are usually represented with graphic models that represent both the structure and semantics of the processes associated with the system and for this, we use activity diagrams, or state transition diagrams in the OO design. However, the structured design gives us through the data flow diagrams an important graphic model that we can adopt in the design of a parallel program with message passing in a completely transparent way. A data flow diagram, as defined in section 2, is a directed graph that can be adapted in the design of a parallel system to define processes, as well as the degree of parallelism that we intend to program by generating Intermediate level DFDs and program structure as a whole using the context diagram DFD.

Each circle of a DFD model represents in the design of a parallel program with message passing a process or a control flow, while the arrows will now indicate the communication channels that the processes use to communicate with each other and thus define the flow of data or its express communication. The rectangles that represent external entities in the DFD will now represent processes of input or output of information that reach other processes modeled with the circles and data storage of the

DFDs will now represent mailboxes where the processes can leave or collect information stored through communication channels, if required and according to the type of communication used in the processes (see section 3.1). The graphical representation of a data flow diagram is therefore a powerful model in the design phase, which represents processes at different levels of nesting, that is, specialization and generalization.

5.1 Case Study: Parallel Generator of Natural Numbers

The design of a parallel system with message passing that generates natural numbers in sequence is shown. This case study has been taken from [15, 16]. The design is done using the DFDs. First the context diagram (more general level in the DFD hierarchy), then the level 0 DFD diagram which is the decomposition of the context diagram in the main processes of the system, along with the communication between them through channels and Medium-level DFDs that represent the decomposition of the processes identified in the system DFD and finally the design of the DFDs of the primitive processes that are those that can no longer be broken down into more process.

The context DFD of this case study is shown in fig. 6. The process represented by the circle in the figure generates consecutive natural numbers starting from the number zero, although the user can indicate the starting number. The numbers generated are sent through the communication channel represented by the arrow or flow to another process called “Display” in charge of receiving them in sequence and printing them on the screen. This process is responsible for defining the limit of reception of natural numbers. Both the Parallel Generator of Natural Numbers process and the Display process are created in a “Parallel Composition of Processes” in concurrent execution [17]. The communication between these processes occurs through the communication channel in accordance with the principle of rendezvous, see Synchronous Communication of section 3.1 for details.

The Display process of fig. 6, is a primitive process and cannot be broken down into more sub-processes. On the contrary, the Parallel Generator of Natural Numbers process can be broken down into more sub-processes which are designed through the Level 0 DFD of fig. 7. The DFD of level 0, consists of three processes (Prefix (0), Delta and Successor or Suc) which in accordance with the logic of operation of the system are connected by their respective channels and whose communication between them is carried out through of the rendezvous concept. Each process is an active object to which a control thread is associated, while a channel is a passive object without its own life.

Of the three DFD processes of Level 0, the Suc process is a primitive process. Its operation consists in increasing by one the value that enters through its Channel b and then sending it to the Prefix process through Channel c. The other two processes, Prefix (0) and Delta, are processes composed of more processes so we can design their corresponding DFDs of Middle Levels (see Fig. 8).

The level 1 DFD of the Prefix process has two channels, one input (Channel c) and one output (Channel a) that connect to the Id process that composes it. The first time the Prefix process is executed, it sends the N value provided by the user through its output channel. N is the initial natural number. In its next executions the process

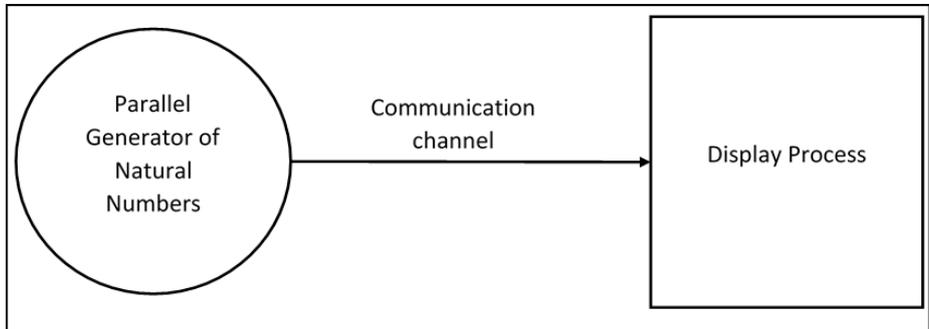


Fig 6. Context DFD.

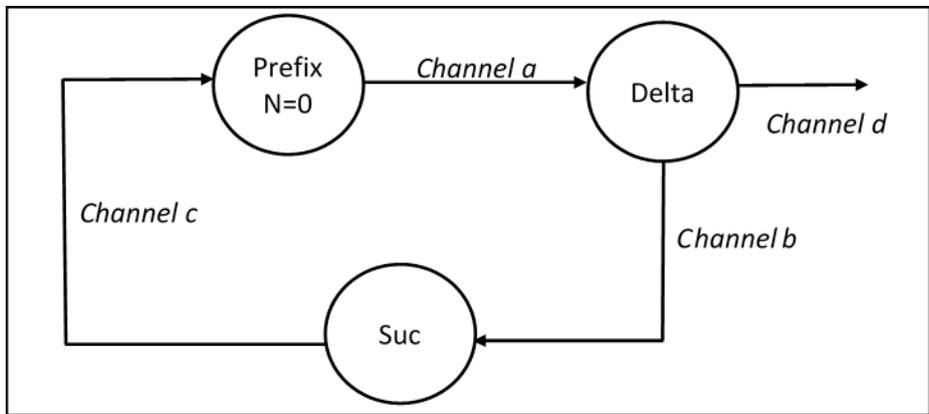


Fig 7. Level 0 DFD.

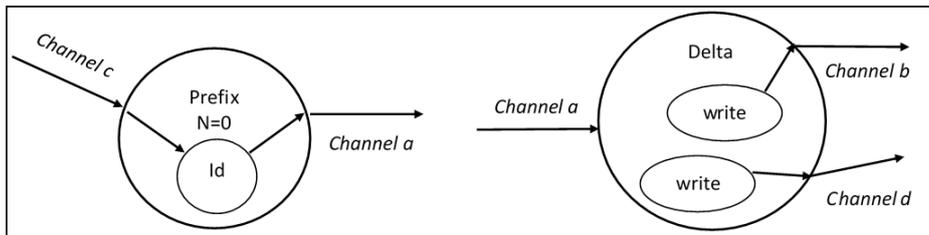
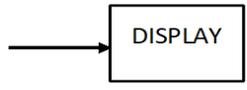
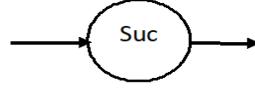
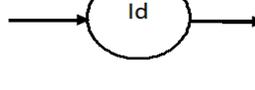
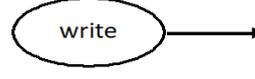


Fig 8. Level 1 DFD (Middle Levels).

executes the behavior of its internal process Id (it is said to behave like him). The internal process Id is a primitive process. In the level 1, DFD (fig. 8) the Delta process has three communication channels, one input channel and two output channels.

Delta's behavior is as follows: in an infinite execution, the number received by its input channel (channel a) is sent by the internal processes (processes Write), which are executed in parallel, so that they send through their respective channels output (Channel b and Channel d) the data received without any modification. Write processes are primitive processes. Finally, the design is completed with the graphic representation of the level 2 DFDs of the primitive processes (see table 2).

Table 2. Level 2 DFDs (Primitive Processes).

TYPE OF EXECUTION	LEVEL 2 DFD (PRIMITIVE PROCESS)	DESCRIPTION
Infinite 		Process that receives from its input channel a data (natural number) and then prints it on the screen.
Infinite 		Process that receives a natural number through its input channel and then generates the successor of that number and the new value is sent through its output channel.
Infinite 		Process that receives a natural number through its input channel and is immediately sent by its output channel without modification.
Finite 		Process that receives a natural number as a parameter and is then sent through its output channel without any modification.

6 Conclusions

A way to design parallel programs using Data Flow Diagrams of the structured design of classical software engineering has been proposed. The same graphic symbols of the DFD were used by changing their semantics of the structured design by the semantics of parallel programming with message passing, so that their uses in the design of a parallel system result in a transparent mapping of processes, communication channels, geometric parallelism, logical partition of processes, generalization and specialization in different levels of nesting that correspond to the different levels of generalization and particularization of DFDs (context, system, levels 1, 2, 3, etc ... until reaching processes primitives).

To demonstrate its usefulness, this proposal was used in the parallel design of a Natural Number Generator. This application has been programmed in JAVA using a particular class library called JPMI based on the process algebra of Hoare or CSP.

References

1. Moyano, J.: Programación Paralela - Conceptos y Diseño de Sistemas Distribuidos (2016)
2. Fujimoto: Parallel and Distributed Simulation Systems. USA, Wiley-Interscience (2000)
3. Palma-Méndez, J.T., Garrido-Cabrera, M.C., Sánchez, F., Quesada-Arencibia, A.: Programación Concurrente. Paraninfo (2003)

4. Wilkinson, B., Allen, M.: Parallel programming techniques and applications using networked workstations and parallel computers. Prentice Hall (2000)
5. Salguero, E.: Diseño de Aplicaciones Distribuidas. GitHub Gist (2018)
6. Martínez, G.E.: Diseño de Sistemas Distribuidos. LWP, Comunidad de Programadores (2019)
7. Montalvo, O.P., Byron, P.V.: Diseño e Implementación de una aplicación distribuida de video bajo demanda basada en la arquitectura cliente-servidor. En: XXV Jornadas en Ingeniería Eléctrica y Electrónica, 25, pp. 344–355 (2014)
8. Sumano, M.A.: Análisis estructurado moderno (2012)
9. Cillero, M.: Diagrama de flujo de datos (2018)
10. García, S., Morales, E.: Análisis y Diseño Detallado de Aplicaciones Informáticas de Gestión. Madrid, Thomson Paraninfo (2003)
11. De Amescua-Seco, A.: Análisis y Diseño Estructurado y Orientado a Objetos de Sistemas Informáticos. Madrid, McGraw Hill (2003)
12. Capel, M., Rodríguez, V.S.: Sistemas Concurrentes y Distribuidos. Granada, Copycentro Editorial (2012)
13. Hoare, C.A.R.: Communicating Sequential Processes. London, Prentice Hall (2003)
14. Kendall & Kendall: Análisis y Diseño de Sistemas, México, Pearson (2011)
15. Hilderink, G., Broenink, J., Vervoort, W., Bakkers, A.: Communicating Java Threads. IOS Press (1999)
16. Hilderink, G., Broenink, J., Bakkers, A., Schaller, N.C.: Communicating Threads for Java, Architectures, Languages and Techniques. IOS Press (2000)
17. Friborg R.M., Brian V.: PyCSP – Controlled Concurrency. International Journal of Information Processing and Management, 1(2), pp. 40–49 (2010)