FUNCTION-BASED METHODOLOGY TO MODEL COMMUNICATION PROTOCOLS FOR PERFORMANCE EVALUATION

 ATIKA COHEN and RADOUANE MRABET
 Université Libre de Bruxelles
 Bd du Triomphe, CP 230, Brussels, Belgium
 cohen or mrabet @helios.iihe.rtt.be
 C=be; ADMD=rtt; PRMD=iihe; O=helios; S=cohen or mrabet

ABSTRACT: The aim of this paper is to describe a methodology based on the function concept, as defined in the OSI Reference Model, to model protocol entities in order to assess their performance while being used in a communication network. This concept provides a high degree of flexibility to model protocol entities, because it allows the modeler to combine some functions needed to build a given protocol which will provide a particular level of performance. The functions of a protocol entity are structured inside a Library of functions, and are mainly related by means of the precedence relation.

I. INTRODUCTION

Software reusability is the key-word of the OSISIM Project (Open Systems Integrated Simulator). This four-year project which began in 1992, has two main complementary objectives. The first one is the design and development of an atelier called AMS\(^1\) to model and simulate open communication systems \([1, 2, 3]\). The second objective is to define a methodology that allows to model, in a generic way, mechanisms, services and protocols related to the communication systems. Software reusability can be seen at three levels. The first level is the use of different packages allowing to carry out the AMS atelier. Currently, a prototype of the atelier is based on the following packages: QNAP\(^2\) (Queueing Network Analysis Package 2), GSS4 (Graphical System Support 4) and MODLINE\(^3\) \([4, 5, 6]\). The second level of reusability is the library of basic models; actually, each basic model, denoted DBM\(^4\), can be used several times to model different network architectures. Further details on how software reusability is carried out at these two levels can be found in \([1, 2]\). The goal of this paper is to focus on the third level of reusability which will be based on the function concept as defined in the OSI Reference Model \([7]\).

The internal structure of a DBM has been defined in an abstract manner in \([2]\). Besides, in \([8]\), the QNAP2 structure of a DBM has been described, and the most important guidelines have been given. This structure is mainly composed of three blocks, the Behavior Engine Block (BEB), the Interfaces Block (IB) and the Measurements Block (MeB). The description of the BEB of a DBM, which is its main part, has been left to the responsibility of the modeler, and neither methodology nor guidelines have been defined to help him.

In this paper, we will partly fill the gap between specifying and modeling by means of a methodology based on the function concept. This methodology assumes that the model of a protocol entity follows the structure of a DBM. The function concept will be used as a fine grain to model protocol entities. As defined in the OSI RM, a function is a part of the activity of a protocol entity, and most protocol entities can be expressed in term of functions. This approach will allow a modeler to construct rapidly and efficiently some models of protocol entities.

The function concept is already being used to design the new generation of high speed transport protocols for high speed communication systems \([9, 10, 11, 12, 13, 14]\). This concept provides a high degree of flexibility to model protocol entities, because it allows the modeler to combine some functions needed to build a given protocol which will provide a particular level of performance.

The second section of this paper presents the general framework of the methodology. The third section deals with the function concept, and the relations between functions. The fourth section outlines how a pattern of a protocol entity is made up. An example is given in the fifth section. Finally, a conclusion is drawn.

II. GENERAL FRAMEWORK

In this section, the general framework for the methodology is described through its three main components, the library of functions, the event classes and the protocol entities.

The main idea of the methodology is to model each function separately, and to store it in a specific library, called LoF for Library of Functions. Hence, a protocol will be regarded as a set of available functions selected from this library. The selection is driven by the needs of the application and by the internal and/or external constraints. The library can be seen as a protocol with menu such as defined in \([10]\).

A. Library of Functions (LoF)

The Library of functions (LoF) is a collection of functions. LoF is referred to as a set of available functions: $\text{LoF} = \{f_i\}$,
f_2, ..., f_n) with n \geq 2. It is interesting to have a large number of functions into LoF, which means that the library is rich in functionalities, and a large number of services can be provided.

A simple function is defined as a function performing one task, provided this task is atomic, i.e. does not need the cooperation of other functions to be achieved. For example, the sequencing function is a simple function, while the error recovery function is not a simple function because it is a set of simple functions such as sequencing and retransmission functions. A cooperation between functions is possible when the functions all together or parts of them are asked to provide a given service. Henceforward, only the simple functions will be considered and only the term ‘function’ will be used instead of ‘simple function’.

Function f_i can perform its task in different ways, which implies that it has to be associated with different algorithms. A_{ij} denotes the jth algorithm associated with the ith function. We write f_i/A_{ij} to denote that function f_i is associated with algorithm A_{ij}.

The available functions into LoF are related to each other by means of some relations which will be defined in the next sections.

### B. Event Classes

A protocol entity can handle a certain number of events, which can be classified into several classes, mainly three^4. The first class of events is defined by the arrival of messages from the application layer, the layer which asks for a service. The second class of events is defined by the arrival of segments from the sub-network which is beneath the protocol entity. The third class of events is defined by the expiration of timers. Additional classes can be defined, for instance, if the interactions of the protocol entity with the operating system are modeled, and/or the control plan is considered.

Each event is associated with a data structure containing information that will be treated by the functions which have to handle the event. At this level, a data structure can be seen simply as a collection of fields containing numeric or symbolic data.

The functions of the library can be partitioned into non-empty sub-sets. The partitioning is based on the class of events the entity will process (let m be the number of event classes). Partitioning does not exclude the existence of relations between the functions belonging to different parts.

We assume that LoF is partitioned into m non-empty parts (m \leq n) : F_1, ..., F_m. This partition permits us to distinguish between the functions in terms of the events handled. In addition, it will allow us to define a partial order between the functions belonging to the same part.

### C. Protocol Entity

A protocol entity has to be modeled as a DBM to be added to the AMS Library. Therefore, the internal structure of a DBM has to be taken into account.

A protocol entity is defined by a set of functions selected from LoF. The selection is driven by the service required by the application and/or by the type of the available sub-network.

Each protocol entity E is associated with a data structure D_E. This structure contains, on one side configuration data related to the selected functions, and on the other side information to be used by these functions to achieve their tasks.

## III. FUNCTION CONCEPT

### A. Definitions

Each function f_i/A_{ij} is associated with a set of inputs and a set of outputs.

**Input Set** : Sin(f_i/A_{ij}) is the set of the objects’ attributes whose values can be used by function f_i/A_{ij} during its execution. These objects’ attributes either belong to D_E or to the handled event. The size of this set is variable, it depends on the handled events and on the context where f_i is requested. This set is never empty.

**Output Set** : Sout(f_i/A_{ij}) is the set of the objects’ attributes whose values are updated by f_i during the execution of the algorithm A_{ij}. These objects’ attributes either belong to D_E or to the handled event. As Sin(f_i/A_{ij}), the size of Sout(f_i/A_{ij}) is variable. It is also never empty.

**Remark** : Sin(f_i/A_{ij}) and Sout(f_i/A_{ij}) will partly determine the dependencies between f_i/A_{ij} and other functions.

To define a function exactly, its associated algorithms and its input/output sets, the following syntax is used. Each function has a unique name (denoted Fct_name). The definition of a function contains a CHOICE block. The block contains one or several parts depending on the number of algorithms associated with the function.

Each part begins with the name of an algorithm (denoted Algo_name) used as a label in the CHOICE block. For each algorithm, the sets of inputs and outputs are given. For each set, two lists can be given, one for the attributes which have to be found in the data structure of the handled event (IN(v) or OUT(v)), and the other one for the attributes which have to be found in the data structure of the protocol entity (IN(e) or OUT(e)). Finally, the part contains the name of the algorithmic block (denoted statement_block_name) implementing the algorithm.

---

^4 This methodology does not depend on the number of event classes. The number may be larger or smaller than three.
Fct_name (Algorithm);
CHOICE
{ +^{Aalgo_name} : BEGIN
{ IN(v) = +{var_name,} var_name ;
{ IN(e) = +{var_name,} var_name ;
{ OUT(v) = +{var_name,} var_name ;
{ OUT(e) = +{var_name,} var_name ;
statement_block_name;
END;
}
END;

Remarks
• Not all the attributes listed in IN(v) or IN(e) have to be used. This is why it was said earlier that the set of inputs is variable. The same remark is true for OUT(v) and OUT(e).
• IN(v) or IN(e) can be omitted but not both. Namely, at least an input list has to be present inside the definition of a function for a given algorithm. The same remark is valid for OUT(v) and OUT(e).

Mandatory functions
A function is declared mandatory if and only if it is always present in a protocol entity, whatever the functions selected by the user, and whatever the service required may be.

With the mandatory functions only, a protocol entity is able to execute its code normally without the cooperation of any non-mandatory function.

The set of mandatory functions can be seen as a minimal service that a protocol entity is able to provide. It can be a service which merely transfers data to the sub-network and in this case the only mandatory function is "transfer DT".

B. Precedence Graph
A partial order between the functions belonging to the same part of LoF is defined as the acceptable order in which operations can be performed. The reason for dependencies could be viewed as some sort of shared information, manipulated by a function and required by another one. So, the necessary condition for function f_{i/Aip} to precede function f_{j/Ajq} is :
\[ \text{Sout}(f_{i/Aip}) \cap \text{Sin}(f_{j/Ajq}) \neq \emptyset. \]

Two main types of precedence are defined : strong precedence and weak precedence.

Strong precedence
A function f_{i/Aip} preceded by a function f_{j/Ajq} (i \neq j) by means of a strong arc has to start its execution if and only if the execution of function f_{i/Aip} has been completed; this is true when the two functions (f_{i/Aip} , f_{j/Ajq}) are selected. But, if only f_{i/Aip} is selected, two cases can take place :
1) f_{i/Aip} is not preceded by another f_{r/Aip} with r \neq p. In this case, f_{i/Aip} is selected automatically and mandatorily, then f_{i/Aip} has to be executed first, and only when this has been done, can f_{j/Ajq} start to be executed.
2) f_{i/Aip} is preceded by another f_{r/Aip} with r \neq p. In this case, a choice has to be made between these algorithms. If A_{ic} is the choice then f_{i/Aip} has to be executed first, and only when this has been done, can f_{j/Ajq} start to be executed.

The necessary condition for linking two functions by a strong-arc is defined hereafter :
\[ f_{i/Aip} \rightarrow f_{j/Ajq} (i \neq j) \text{ if at least a part of "} \text{Sout}(f_{i/Aip}) \cap \text{Sin}(f_{j/Ajq})\text{" is used by } f_{j/Ajq} \text{ during its execution.} \]

This condition is not sufficient. Actually, additional knowledge about dependencies between the functions is needed because no specific dependency is described through the input and output sets.

Weak precedence
If function f_{i/Aip} precedes function f_{j/Ajq} (i \neq j) by means of a weak arc and both are selected, then f_{i/Aip} has to be executed after the execution of function f_{i/Aip} has been completed. If only f_{i/Aip} is selected, then f_{i/Aip} will be executed all the same. The necessary condition for relating two functions by a weak-arc is defined hereafter :
\[ f_{i/Aip} \rightarrow f_{j/Ajq} (i \neq j) \text{ if } \text{Sin}(f_{j/Ajq}) - \text{Sout}(f_{i/Aip}) \neq \emptyset \]

This condition means that when f_{i/Aip} is selected and f_{i/Aip} is not, then the input set of f_{j/Ajq} has to contain other inputs than those contained inside Sout(f_{i/Aip}); because, as mentioned before, a function cannot be executed without input information.

This condition is not sufficient. As explained before, additional knowledge about dependencies between the functions is required.

Precedence Graph
A precedence graph G_{i}, associated with a sub-set F_{i}, is a digraph, where each vertex represents a function belonging to F_{i}, and each arc represents the precedence between the two vertices extremities of the arc. It is either a "strong-arc" or a "weak-arc". Each arc, connecting function f_{i} to function f_{j}, is labeled and the label is denoted L_{ij}^{pq}, where p represents the pth algorithm associated with f_{i}, and q represents the qth algorithm associated with f_{j}.

A precedence graph has two special vertices, the "Begin" vertex and the "End" vertex. The former is not preceded by any other vertex, the later does not precede any vertex; clearly :
\[ +{\{x\}} : \text{this notation means that } x \text{ is present at least once.} \]
\[ \{x\} : \text{this notation means that } x \text{ may be present zero, one or several times.} \]
\* in degree(Begin) = 0 and out degree(Begin) \neq 0
\* in degree(End) \neq 0 and out degree(End) = 0

On the other hand, except the vertices "Begin" and "End", vertex \( f_i \) with \( \text{in degree}(f_i) = 0 \), will be preceded by "Begin" and vertex \( f_j \) with \( \text{out degree}(f_j) = 0 \) will precede "End". Consequently, each vertex different from "Begin" and "End" is preceded by at least one vertex, and precedes at least another one.

C. \( \varepsilon \)-Graph

In order to build a protocol entity completely and coherently, the precedence graphs are not sufficient to identify all the relations between the functions of LoF. Another type of graph, called \( \varepsilon \)-Graph, is defined.

An \( \varepsilon \)-graph is a non-directed graph. Its vertices are the functions of LoF. An edge linking two functions \( f_i \) and \( f_j \) is labeled either \( \Gamma_{ij}^{pq} \) or \( \Lambda_{ij}^{pq} \). The two extremities of an edge belong to two different parts of LoF.

An edge labeled by \( \Gamma_{ij}^{pq} \) (resp. \( \Lambda_{ij}^{pq} \)) means that if the function \( f_i/A_{ip} \) is selected for a protocol entity then the function \( f_j/A_{jq} \) will be selected automatically for this protocol entity (resp. for the peer protocol entity); the opposite is true.

D. Parameters and Measurements

The measurement block is a part of a DBM and it is as important as the Behavior Engine Block itself, since our final objective is to assess the performance of the DBM. In the case of the protocol entities, a measurement is related to the function concept. A measurement will be seen as a hook on a function or a sub-set of functions. A function can be associated with zero, one or more than one measurement(s).

When one measurement is associated with one function, this measurement will be used to assess the behavior of this function. If a measurement is associated with a group of functions, then it will be computed only if all the functions of the group are selected.

Example: For the segmenting function, the measurements can be the number of segments created, and the average number of segments created by message.

IV. A PATTERN OF PROTOCOL ENTITY

Building up a pattern of a protocol entity is driven by the required service, and/or by a set of functions selected by the user. A given service can be seen as a set of functions which have to be implemented by a protocol entity in order to provide the service asked for.

The algorithm to be able to construct a pattern of a protocol entity from LoF uses the precedence graphs and the \( \varepsilon \)-graph defined for the available functions into LoF. In fact, the algorithm constructs at the same time a pattern of a protocol entity and a pattern of the peer-protocol entity.

In these two patterns, the mandatory functions have to be present, plus the selected functions, plus may be other functions which have to be present in order to give coherence to the assembled set of functions, and to make the transfer of segments between them feasible. The algorithm to make up a pattern of a protocol entity is explained in detail in [15].

The algorithm takes into account the external constraints which generally come from the sub-network. These constraints either require to select additional functions or on the contrary, to cancel the pre-selected function(s). For instance, the segmentation function has to be selected because the sub-network can handle only segments of a maximal length; on the other hand, each segment may have a fixed length so that the padding function has to be selected as well.

V. EXAMPLE

Let LoF = \{segmenting, reassembling, concatenation, separation, numbering, retention, acknowledgment, retransmission on time-out, resequencing, transfer DT, transfer ACK\}.

In order to simplify the example, we assume that each function is associated with only one algorithm; in this case all the labels of the precedence graphs are useless. Furthermore, we assume that the patterns of a protocol entity and its peer-protocol entity are identical.

Basing ourselves on the three main event classes, defined previously, LoF can be partitioned as follows:

\[ F_1 = \{\text{segmenting, concatenation, numbering, retention, transfer DT}\}, \]
\[ F_2 = \{\text{reassembling, separation, acknowledgment, resequencing, transfer ACK}\}, \] and
\[ F_3 = \{\text{retransmission on time-out}\}. \]

The precedence graphs of these three parts are represented in figure 1; a strong-arc is represented by a solid line while a weak-arc is represented by a dashed line. \( \varepsilon \)-graph is represented in figure 2, while figure 3 depicts a pattern of a protocol entity.

\[ ^6 \text{DT for Data and ACK for acknowledgment.} \]
VI. CONCLUSION

The methodology described in this paper will help to model efficiently and rapidly new protocols (i.e. XTP [16]) designed for the new generation of networks where several services can be provided. These new protocols can be dynamically tailored to the user’s requirements. The methodology will help to assess earlier the performance of the protocols when used in different contexts. The methodology is mainly based on the function concept introduced in the OSI RM.

VII. ACKNOWLEDGMENT

We wish to express our gratitude to SAIT Electronics with whom we are collaborating on the OSISIM project. We also gratefully acknowledge the financial support granted by IRSIA.

VIII. REFERENCES