INTERFACE, A DISPERSED ARCHITECTURE

Chris A. Vissers
Twente University of Technology
Enschede, The Netherlands

0. Abstract

Past and current specification techniques use timing diagrams and written text to describe the phenomenology of an interface.

This paper treats an interface as the architecture of a number of processes, which are dispersed over the related system parts and the message path. This approach yields a precise definition of an interface. With this definition as starting point, the inherent structure of an interface is developed. A horizontal and vertical partitioning strategy, based on one functional entity per partition and described by a state description, is used to specify the structure. This method allows unambiguous specification, interpretation, and implementation, and allows a much easier judgement of the quality of an interface. The method has been applied to a number of widely used interfaces.

1. Introduction

Many committees, charged with standardizing an interface struggle many years (8 to 10 years makes no exception) to get the job done. What are their problems ? We can find at least three. First, an interface is always much more complex than a first estimate suggests. Qualification and quantification of the needs of the users is a difficult task, application dependent, and subject to different opinions. The definition of the functional contents of an interface that satisfies these needs introduces an extra choice, and consequently makes agreement one level more difficult. Second, the disclosure of an interface allows the linkage of products of different companies into one system, which requires the political will to make this happen. The third problem is the available methodology and language for the specification of an interface and its preliminary versions. Conventional methodology uses timing diagrams and written text, often illustrated with tables and drawings. This methodology has a number of serious disadvantages. The most important of these are discussed in this paper. Bad specification methodology makes an interface difficult to master and document, and enhances the risk of errors, incompleteness, inefficiency and vagueness. It also opens the door to obstruction of progress through vague reasoning. An interface with such characteristics contributes to non-uniform and unintended interpretations. And faithful to Murphy's law this has led to system malfunctioning, even though the interface was scrupulously interpreted.

Therefore, the availability of an efficient tool that allows unambiguous and clear specification and interpretation of an interface would be of great profit to both its designers and users. For the designers it facilitates a clear discussion and expedites a correct, complete, efficient and clear specification. For the users it will help to avoid system malfunctioning, caused by misinterpretation or unauthorized extension of a given interface.

This paper presents a specification method that tries to incorporate the desired characteristics. It has been applied to a number of existing and proposed standard interfaces [1,2,3] with satisfactory results. The state description technique, which is closely related with the method, is reflected in an interface which is now becoming an international standard. It proved to be of extreme value both in the development and use of this interface [4,5]. The method is based on the definition of an interface, which will be discussed first. Next, technique and language for the specification of an interface are discussed. This is followed by the development of a structuring discipline for an interface, and a discussion of the character of a standard interface. Finally some conclusions are drawn.

2. Definition

Few attempts seem to have been undertaken to state a manageable definition for the concept of interface. It may be that the term interface itself, and its translations to various languages(cutting place, tangent plane), pretends to be clear enough. But the term interface is currently used with many divergent interpretations. (The term 'connection' is used in [6] to indicate the same kind of relation definition as discussed in this paper). Therefore, if we want to discuss a specification methodology for it, an adequate definition is demanded.

What we clearly want, is to be able to bring system parts that can be considered and designed as separate functional entities, into relationship to form a system with a higher level of functional performance. The possibility that some system parts can be brought into relationship makes us say that these system parts can interface. Therefore an interface can intuitively, but still informally, be called a 'relation definition'. In order to interface, the system parts must be given certain properties which are attuned to each other.(e.g. two system parts know the same variable, one as its producer, the other as its consumer). All properties of a system part are defined by the complete specification of its functional behaviour, its architecture. Therefore we are able to define an interface by specifying the architectures of the related system parts. Through the definition of the architecture of each part, the interfaces of these parts are concurrently established. However, in some cases we may desire, or be forced, to define an interface first, and the complete architectures later (e.g. for the definition of a standard Channel-to-I/O interface). In these cases it is undesirable, unfeasible, and unnecessary to define the complete architecture of the related system parts in order to be able to define their interface.

System parts have a relation if they can affect each other's functional behaviour. Without mutual effect they ignore each other and the system parts are independent and unrelated. The mutual effect is through variables (messages) with a defined behaviour and exchanged via a message path. Suppose we want to define the effect of system part A on system part B through variable V. The behaviour of V can be defined through the definition of the generative mechanism of V, which belongs to A and forms a portion of A's architecture (the influence of A). The effect on B through V can be defined through the definition of that <u>portion</u> of B's functional behaviour expressing that effect (the effect on B). Conversely we can define the effect of B on A through W. This yields, another portion of A's and B's architecture. The two portions of A's architecture can be specified either as separate portions if there is no correlation between them, or integrated if there is. A similar statement can be made for B.

An <u>interface</u> of two or more systems parts defines for each system part that portion of its architecture that allows a relation between those system parts to form a system providing a desired function.

The previous reasoning assumed a functional passive message path for the exchange of the variables V and W. Though this is often the case, an interface may contain a message path that performs logic operations on the variables it exchanges as explained in section 4.3: Central message path. Consequently the definition of an interface has to be extended to contain the architecture of the message path, as shown in figure 1. As with the definition of an architecture, the definition of an interface is a specification problem. This specification problem concerns not one architecture, but a portion of each of the related architectures and the message path in between. In this sense the concepts of architecture and interface are equivalent. The term relational function is given to that portion of an architecture that is part of the considered interface. The remaining portion of the architecture is called local function. It forms the complement of the relational function in the architecture's total relation with its environment.

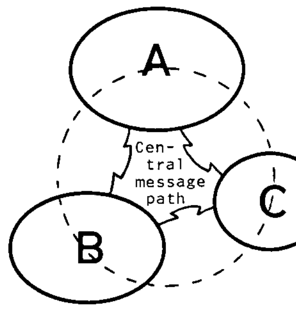


Figure 1a: Three related architectures A, B, and C.

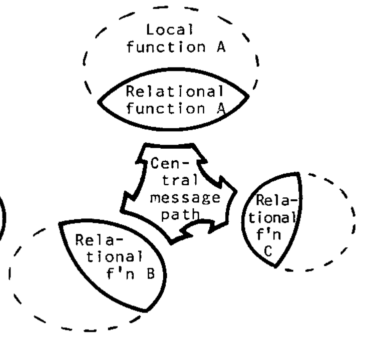


Figure 1b: The A-B-C interface.

This definition provides the basis for a sound specification method: the interface is specified by the separate specification of each relational function and of the message path. Since these items are portions of an architecture, their description may use the same techniques and languages as applied to architectures in general.

3. Description techniques and languages

Three basically different description techniques are conceivable and used to specify an architecture: the phenomenological, the assertive, and the generative description.

3.1 Phenomenological description.

The phenomenological description gives an observation of the behaviour of the input and output variables. As known from automata theory, that bases its definition of an automaton on it, this is a valid specification method. But in order to be complete, the observation must include all input and output variables, and all possible sequences of their values. Though this requirement is not important for the development of (automata)theory, it is impractical for any architecture of some complexity, because of the sheer monotony and inordinate length of the sequences. It is not surprising that this method is not used in practice for the specification of architectures. Therefore it is a real surprise to observe that the conventional specification method for interfaces is still based on the phenomenological description, since the timing diagrams are literally an observation of the signal lines that exchange the messages among the system parts. Those diagrams are furthermore by definition incomplete, as long as they do not contain all input and output variables of the relation functions. This incompleteness however, is normal in conventional specification methodology, since the variables exchanged across the local function/relational function boarder, are normally missing in the timing diagrams. (In 4.1 Sources and sinks we come back to this point). To make things even worse, most specifications only contain the most significant sequences. Although this cuts down on the monotony and length of the sequences, it makes the specification even more incomplete. Hence, the written text, which usually goes together with the diagrams, becomes essential to fill up the gaps in the specification with timing diagrams. The text, however introduces several new problems. The use of another language will inevitably tempt the writer of the specification to 'explain' the diagrams. And so the reader must carefully distinguish between text that contains additional specification and text that contains redundant written specification of the diagrams. Furthermore it is in most cases not clear whether the text is meant to be assertive, generative, or phenomenological. The observation of the message path as basis for the description, results in an intermixed description of the contributing relational functions and the message path itself. This burdens the implementer of an architecture to untangle the relational function, that is part of this architecture, from the total specification. The phenomenological description is maximally unstructured, opposite to the nature of human thinking. Therefore the implementer has to bridge the 'maximum distance' from the phenomenological specification to his product, a realizable generative specification.

The previous observations suggest a specification method for an interface in which each relational function and the message path is specified individually. Each individual specification uses one description technique, preferably not the phenomenological, and one description language.

3.2 Assertive description.

The assertive description method, originally introduced to prove program correctness [7,8], specifies an architecture by specifying assertions on the behaviour of the input and output variables. In so doing, the assertions form in fact a shorthand notation for the phenomenological description of the input and output variables, and allow the latter's drawbacks to be avoided . The assertive specification can not be simulated, since this requires a model of the generative mechanism between inputs and outputs. Simulation can be highly desirable if we want to check the assertions against samples of the phenomenological description. The specification of the assertions themselves is the biggest problem in using this technique, in particular when the architecture is complex. This often requires that the architecture is specified as a collection of related partitions, and each partition is specified assertively. Through this partitioned specification, internal variables are defined, and the assertive approach comes close to the generative approach which is followed in this paper.

3.3 Generative description.

Associated with the phenomenological specification of a finite automaton, the type of system to which we are restricted when we start an implementation, is a (minimum) state machine. This state machine can be considered as the generative mechanism that maps the input onto the output. A description of it can replace the phenomenological description. For complex systems, as frequently encountered for interfaces, it will generally be difficult to establish this minimum state description. Since the generative description is used to replace a desired input/output behaviour (phenomenology) the latter is not available to deduce a minimum state description from it. Associated with the minimum state machine are many equivalent machines with the same phenomenology which do not contain the minimum set of internal states. Therefore, although the minimum state description often appears to be the most attractive one [2], we often have to be satisfied with a reasonably good equivalent description. Most of our experience is based on the use of this type of description for interfaces, though an assertive description might equally well have been chosen.

3.4 Language.

The most primitive language in which the generative mechanism for a finite automaton can be expressed is the state transition diagram or table, as used in sequential circuit theory. Though this language has succesful been used in a number of applications [1,3,4], and remains quite suitable for specific ('logic') functions, it appears to work inefficiently for more complex interfaces. In these cases an algorithmic language, that contains primitives for many (numerical and logical) operations is much more powerful [2]. Some examples of this are shown in figures8 and 9. It remains essential though, that the algorithmic description is interpreted as a state machine description. The representation of the states is free, since they are internal to the automaton. The optimum choice is a representation that provides maximum clarity of specification. This can often be achieved by adapting state representation and formulation of transition conditions to each other [2]. The algorithmic language allows also many different representations for the state transition diagrams or tables. So these descriptions can be mixed with algorithmic statements, and simulated on a computer [9]. The possibility of simulating the interface is an important advantage of the generative description.

4. Structure.

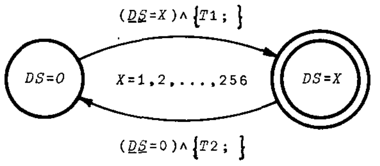
Human nature does not favour the specification of a complex system by one single homogeneous function, such as a large diagram, table or algorithmic expression. We no longer have confidence that it represents what we want. Instead we start with smaller individual parts of specification. For each part we have confidence that its specification is, or can made to be, what we want. And we link up (interface) those parts, often by extension of their specification, into a larger part of specification. Such a partioned specification method raises problems of its own: where to start, and how to link up the parts. A general structuring discipline, providing a structure in which partitions of specification can be embedded, can greatly help in reducing these problems. Such a structuring discipline for the specification of an interface is discussed in the following sections.

4.1 Sources and sinks.

The first class of partitions is called the source and sink functions layer. To operate as one functional entity -e.g. a Channel- information is exchanged between the local and relational function of the channel. Conventional methodology often hides the variables carrying this information in vague statements, such as
' if the Channel is able to communicate with a device, it places an address on the bus. . .'
Such a statement gives rise to numerous questions:
- Where and how is the ability to communicate generated ?
- When the channel is able, will it actually communicate ?
- What happens when the channel is able, but other activities require the channel's attention ?
- When, where and how are addresses generated, how and where are they coded ? Etc.

It is therefore necessary to make a clear distinction between the variables generated by the local function and by the relational function, and to decide which of these cross the boundary between local and relational. The interface is only interested in those locally generated variables that are inputs to the relational function. Since their generation is a part of the local function, which is per definition unknown, we cannot define their behaviour. But we can define their required behaviour via a finite automaton, called a source function:

A source function defines the existence, set of values, and required behaviour of a local variable, which is made available as input to the relational function [3].



(a)

```
       ∇ DECSOURCE
[1]    W1:→(DS=0)/W1
[2]     DS←DS
[3]     DELAY T2
[4]    W2:→(DS≠0)/W2
[5]     DS←DS
[6]     DELAY T1
[7]     →W1
       ∇
```

(b)

Figure 2: Decoded source.

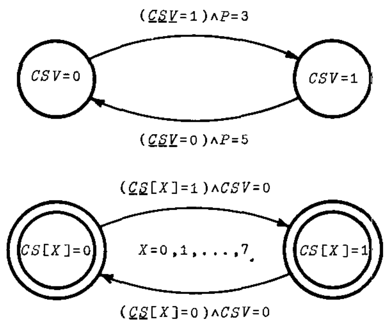
Graphic (a) and algorithmic (b) description of a decoded source. The local variable DS, whose behaviour is free, is used to determine the behaviour of the relational variable DS. The behaviour of DS is as follows: DS=0 remains at least T1 seconds valid, DS≠0 remains at least T2 seconds valid. Each value of DS≠0 is enclosed by the 'separation message' DS=0. The variable DS is used in the relational function. If DS behaves as DS (i.e. as required), the implementation of the source is trivial: a short-circuiting from DS to DS.

The local function represents the complement of the relational function in the architecture's total relation with its environment. This leads to the introduction of the sink function, the counterpart of the source function:

A sink function defines the existence, set of values, and behaviour, of a relational variable, which is made available as input to the local function [3].

A sink function has the same appearance as a source, therefore no examples are shown. Practical applications often require the combination of a source and sink function into one function. Such a function is called a conversational source or sink function, dependent upon its main task. The conversational character is required when the validity time (the time that the value of the variable remains unchanged) of a relational variable is defined in a logical way (figure 3) instead of by the use of time (figure 2).

(a)



(b)

```
       ∇ CODEDSOURCE
[1]    WAIT:→(CSV←(((CSV=1)∧P=3)∨CSV)∧~(CSV=0)∧P=5)/WAIT
[2]     CS←CS
[3]     →WAIT
       ∇
```

Figure 3: Coded converational source.

Graphic (a) and algorithmic (b) description of a coded conversational source. The local variables CSV and CS are used, together with the relational variable P, to determine the behaviour of the relational variables CSV and CS. As long as CSV=0, the code of CS may change. Analogous to figure 2, CSV=0 may be interpreted as the separation message. CSV is used both in the relational function and in the local function. The code of CS remains valid, as long as CSV=1.

When all sources and sinks for each relational function are identified, they form a layer that shields the local functions from the remaining part of the interface. This remaining part is called in figure 4a basic interface function. The source and sink layer defines the behaviour of all inputs and outputs of the interface and is therefore a suitable place to start the definition It shows that an interface can be considered as an architecture that is dispersed over several other architectures and the message path.

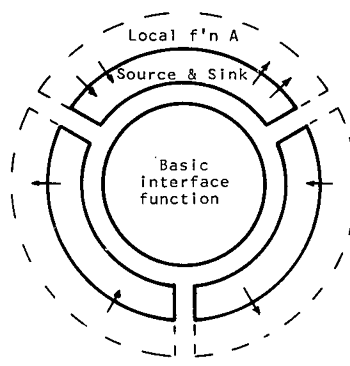


Figure 4a: Source and Sink layer and Basic interface function.

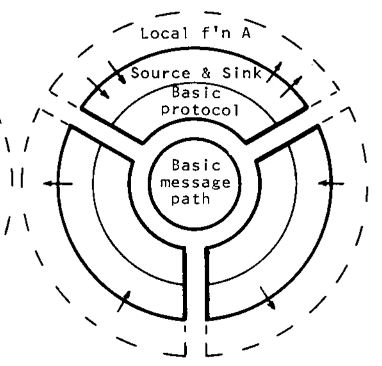


Figure 4b: Partitioning of the Basic interface function into the Basic protocol functions and the Basic message path.

When an interface is considered as a dispersed architecture, one can view a system either as a collection of related architectures (figure 5a), or as a collection of interfaces (figure 5b). The latter viewpoint is used when a system makes use of one or more predefined interfaces.

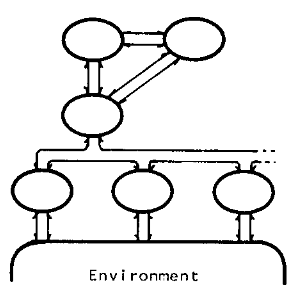


Figure 5a: System, specified by a collection of architectures.

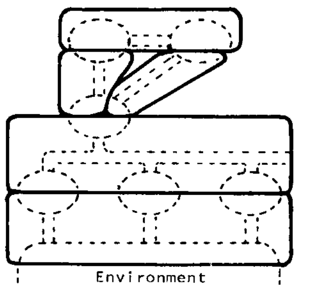


Figure 5b: System, specified by a collection of interfaces.

4.2 Basic protocol.

When the source and sink layer is defined, the remaining part of the interface represents its basic function. This basic interface function contains the flow of the data from sources to sinks and the operations performed on the data. If the basic interface function is defined as one automaton and subsequently partitioned into the basic protocol functions (the parts that are accomodated in the related architectures), and the basic message path (figure 4b), it will usually result in a voluminous, inflexible and costly basic message path. Most interfaces require a reduction of this cost through a reduction in the space and time allowed for the exchanged variables. At the same time increased flexibility is desired and obtained through a general purpose message path. Consequently the basic protocol functions have to be adapted to this reduced and generalized message path to form part of a definite specification. Starting with the basic protocol, however, is very useful in formulating the interface's basic task and in deciding how the elements of this task are allocated to the related architectures.

4.3 Central message path.

The next step is the definition of the central message path. The basic message path indicates what variables are to be exchanged. As a first step it can be decided how much space and time will be assigned to these variables. Three most important and competing parameters influence this choice. The first one is the possibility of physical separation of the architectures, in particular their maximum physical distance. The second is the desired time performance of the interface, and the third is the desired reliability of the interface. Another important parameter is the level of independence of the message path from the related architectures. The weight of those parameters is highly dependent on the application of the interface (e.g. industrial plant control versus laboratory experiments), which makes a universal central message path for all applications unlikely.

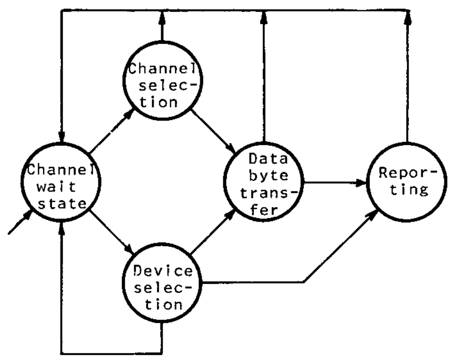


Figure 6: Basic protocol function.

A simplified basic protocol function of a complex Channel-to-i/o interface [1]. The function is located in the Channel. The diagram shows how a data transfer sequence can be build up. The elementary steps in the sequence are represented by the states in the diagram. The transitions in the diagram indicate how these steps may be sequenced. It follows that data transfers from different devices may be mixed. This allows the multiplexing of the message path.

As a second step, the function of the central message path is defined. In some message path configurations, such as in string or loop configurations, this function is trivial: just one or more connections. A less trivial function is represented by the so called bus or party line configuration that can be found in most Channel-to-I/O interfaces. Such a bus structure gives the 'or' of the coded variables presented to it by the relational functions. This 'or' is a simple, but essential function that allows multiplexing of data from various destinations. The implementation of the 'or' function is generally distributed over the relational functions.

```
     ∇ CMP
[1]   L:SRL←PSL
[2]    SRLV←PSLV
[3]    PRL←∨/SSL
[4]    PRLV←∨/SSLV
[5]    →L
     ∇
```

Figure 7: Central message path function of SDLC [1].

The central message path function connects one primary station to multiple secondary stations. The S(econdary) R(eceiving) L(ine) is connected to the P(rimary) S(ending) L(ine). Idem for the S(econdary) R(eceiving) L(ine) V(alid), which carries the clock. The value of the P(rimary) R(eceiving) L(ine) is the 'or' function of the sending inputs to the line, represented by the vector S(econdary) S(ending) L(ine). Idem for P(rimary) R(eceiving) L(ine) V(alid). PSL and PSLV are generated in the function FRAMETRANSMITTER of figure 8. PRL and PRLV are used in the function FRAMERECEIVER of figure 8.

More sophisticated message path functioning can be found in many CPU-Channel interfaces. Here the message path contains store operations, priority assigments to regulate concurrent acces to the same storage locations, and the like. These functions are performed by main storage.

The CPU-Channel interface is a prominent example of the use of memory in the message path. Exchange of variables via memory (indirect transfer) allows either parallel or sequential operation of the relational functions. A freedom of choice which is left to the implementer. When no memory is used, the transfer is direct, which requires parallel operation of the relational functions. The determination of the space, configuration and function of the message path of the interface plays a definite role in the total performance and applicability of the interface. The message path is therefore central to the relational functions as illustrated in figure 10. It is not surprising that some names of interfaces are based on it (e.g. the unibus). But this does not justify the identification of the message path, or its momentary condition (e.g. the state of main storage as the interface between the program modules), as the entire interface. The message path should be derived from the basic protocol functions and not vice versa.

4.4 Transfer.

The introduction of the central message path requires an adaption of the basic protocol to the space, time, and function of the central message path. This requires a number of functions to adapt the format (multiplex, serialize) of the variables supplied by the basic protocol to the message path and vice versa. The sequencing of different variables over the same transmission path requires a mechanism to indicate the type and (in)validity of these variables. The introduction of the coded representation of the variables on the message path, discussed in the next paragraph, also makes these mechanisms necessary. Dependent on the choices made for the message path, such a mechanism can make use of handshaking, strobing, or enveloping techniques. The chance of message mutilation caused by the message path, often requires the introduction of message protection mechanisms These mechanisms can range from a simple parity check to complex methods such as cyclic redundancy check, buffering, numbering and retransmission.

The functions charged with these types of tasks form a layer, shielding the basic protocol from the message path. This layer is called Transfer in figure 10.

```
     ∇ FRAMETRANSMITTER;SPTR;SFLAG
[1]   W1:→(~TRANS)/W1
[2]    SPTR←FLAGPTR,(CHECKBITS SFRAME),SFRAME,FLAGPTR
[3]   W2:→PSLV/W2
[4]    SFLAG←((ρSPTR)>24+ρSRFAME)∨(ρSPTR)≤8
[5]    PSL←(0,¯1↑SPTR)[SFLAG∨SCNT≠5]
[6]    SPTR←(-SFLAG∨SCNT≠5)↓SPTR
[7]    FRSEND←0=ρSPTR
[8]    SCNT←PSL×SCNT+1
[9]   W3:→(FRSEND,PSLV,~PSLV)/W1,W2,W3
     ∇
```

```
     ∇ X←CHECKBITS Y;N
[1]    N←ρY
[2]    X←16ρ1
[3]   L:N←N-1
[4]    X←1↓(X,0)≠POL∧X[0]≠Y[N]
[5]    →(N≠0)/L
[6]    X←ϕ~X
     ∇
```

```
     ∇ FRAMERECEIVER;FINB;SUPR
[1]   W1:→(~PRLV)/W1
[2]    FINB←RFLAG
[3]    SUPR←(~PRL)∧RCNT=5
[4]    RFLAG←(~PRL)∧RCNT=6
[5]    RCNT←PRL×RCNT+1
[6]    RFRAME←(8×RFLAG)↓SUPR↓PRL,(~FINB)/RFRAME
[7]    FRDY←RFLAG∧ρRFRAME≥32
[8]   W2:→(FRDY,PRLV,~PRLV)/L,W2,W1
[9]   L:RFOKE←∧/(16↑RFRAME)=CHECKBITS 16↓RFRAME
[10]  W3:→(PRLV,~PRLV)/W3,W1
     ∇
```

Figure 8: Transfer functions of SDLC [2].

The FRAMETRANSMITTER generates PSL and PSLV (see figure 7) from the variable S(ending)FRAME, that it receives from the ENCODER function, shown in figure 9. It indicates when the frame is transmitted (by FRSEND) etc. The FRAMERECEIVER assembles a variable R(eceived)FRAME from PRL and PRLV (see figure 7), and presents this to the DECODER function of figure 9. The subfunction CHECKBITS generates the cyclic redundancy checkbits, and is part of both the transmitter and receiver function. It is not an individual automaton.

4.5 Coding and decoding.

As mentioned under 3.4 Language, the representation of the state of the automaton is free as long as we are in the architectural phase, and can be chosen to provide maximum clarity of specification. When the automaton is implemented, the implementer is free to represent the state of the variable by any suitable set of code elements according to his criteria. Source and sink variables are internal to the individual architecture's as are most of the variables contained in the basic protocol and transfer functions. Their final representation is the implementer's decision. The situation is different however, for the variables that are exchanged via the message path. Usually an interface is designed to allow the implementation of each architecture by an independent group without requiring them to communicate with all other groups. When this is the case, the representation of the variables crossing the message path is public and must be settled by the interface designer. The variables are principally provided or accepted by or via the basic protocol, and are subjected to the transfer operations. Hence the functions performing the coding and decoding form a layer in between the (basic) protocol and the transfer, as shown in figure 10. The message path provides the code elements for the representation of the exchanged variables.

```
     ∇ ENCODER;SAF;SCF;SIF
[1]   W1:→(~TRANS)/W1
[2]    SAF←(8ρ2)⊤IADD
[3]    SCF←(8ρ2)⊤(16×FB)+(SNSI,SRQI,SROL,SNSA,SCMDR,SRR,SRNR,SREJ,SI)/
       3 7 15 99 135 ,(1 5 9 ,2×SNS)+32×SNR
[4]    SIF←(SCMDR/HELPFIELD),(SNSI/OUTNSI),SI/OUTI
[5]    SFRAME←SIF,SCF,SAF
[6]   W2:→((ENCODERDY←TRANS),~TRANS)/W2,W1
     ∇
```

```
     ∇ DECODER;RAF;RCF
[1]   W1:→(~FRDY∧RFOKE)/W1
[2]    RAF←¯8↑RFRAME
[3]    RCF←¯8↓¯16↑RFRAME
[4]    RIF←¯16↓RFRAME
[5]    MA←(2⊥RAF)∊(IADD,CADD)
[6]    PB←RCF[3]
[7]    INR←2⊥RCF[0 1 2]
[8]    INS←2⊥RCF[4 5 6]
[9]    RVIF←~RCF[7]
[10]   RNSI←3=2⊥RCF[0 1 2|4 5 6 7]
[11]   RSIM←7=2⊥RCF[0 1 2|4 5 6 7]
[12]   RORP←19=2⊥RCF[0 1 2|4 5 6 7]
[13]   RDISC←35=2⊥RCF[0 1 2 4 5 6 7]
[14]   RSNRM←67=2⊥RCF[0 1 2 4 5 6 7]
[15]   RRR←1=2⊥RCF[4 5 6 7]
[16]   RRNR←5=2⊥RCF[4 5 6 7]
[17]   RREJ←9=2⊥RCF[4 5 6 7]
[18]  W2:→((DECODERDY←FRDY),~FRDY)/W2,W1
     ∇
```

Figure 9: Encoding and Decoding functions of SDLC [2].

The ENCODER function generates the variable SFRAME from the variables provided by the protocol functions of SDLC. The DECODER function performs the opposite operation. Both functions are linked to the transfer functions of figure 8.

4.6 Protocol.

The introduction of the central message path, the transfer layer, and the coding layer may affect the basic protocol functions. If so, these functions must be adapted to the communication facilities provided by this central part of the interface. This adaption predominantly involves the introduction of sequencing functions, due to the time limitations of the message path. The adapted basic protocol functions thus form a layer, in figure 10 called Protocol, in between the source and sink layer and the coding and decoding layer. The protocol layer which is defined through this procedure contains the highest level of functions representing the substance of the relation of the architectures. In a standard Channel-to-I/O interface, the interface is primarily concerned with the exchange of different classes of messages, such as commands, data, and status. The protocol layer in this type of interface controls such things as the setting up, maintenance and closing down of message transfers, as well as the interleaving of message transfers from different origins and their priorities in case of concurrency. In general few arithmetic and other data manipulation functions, are found in this type of interface. Other interfaces such as the Channel-Main Storage interface, or CPU-Main Storage interface, may contain in high degree of data manipulation and buffering functions. The CPU-Main Storage interface can be considered as including the definition of almost the entire CPU instruction set.

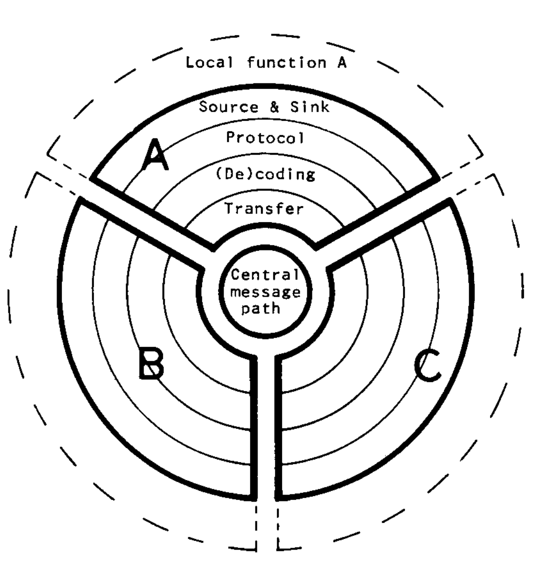


Figure 10: Layered structure for an interface

4.7 Further development.

The previous discussion of the structure of an interface suggests a sequence in the development of the layers, according to the sequence of the sections 4.1 through 4.6. This development is based on a strategy of successive definition. First the architecture of the total interface is determined, and its partitioning and dispersion over the related architectures. Next the architecture of the central message path is determined, and finally the architectures of the individual relational functions. Though this procedure is a useful guideline, a practical application often requires a substantial number of iterations through this sequence, due to the high dependency among the layers.

A further substructuring per layer may result in either the development of sublayers per layer (extended horizontal partitioning) or a partitioning of a layer into functions which are not or only slightly related (vertical partioning). The previous discussions have already used the vertical partitioning by interpreting each layer as a class of functions, and showing examples of such functions. Much is dependent on the possibility of defining a function first as an independent entity, and next of establishing the linkages to and from other functions. As is true for the vertical partitioning, the extended horizontal partitioning may also provide more clarity in the specification of the interface. The protocol function of figure 6 shows what type of operations may be sequenced. The way these operations are organized in detail can be specified in a lower protocol layer. Complex data transmission interfaces may build up their transfer layer as a stack of sublayers. Such a sublayer, and all that it encloses, may be interpreted as the central message path of the transfer layer that is just one level higher. An opposite development also occurs frequently: variables pass a layer unchanged.

The structure so far developed for the interface is shown in figure 11 from the perspective of an individual architecture. Each box in the figure represents a function, that exists in parallel with the other functions and is related with them via the exchange of variables. This horizontal and vertical structuring is different from the structuring in which functions on a lower layer are used to implement an abstract machine on a higher layer [10].

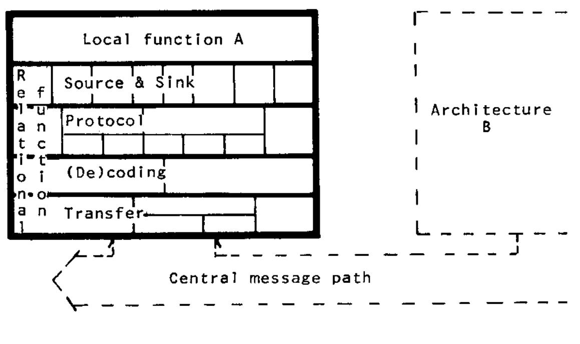


Figure 11: The interface from the perspective of architecture A.

5. What is a standard interface ?

As stated, a system can be understood as a collection of interfaces (figure 5b) as well as a collection of architectures (figure 5a). This viewpoint is significant when an interface is defined first, and the associated architectures later. This happens with a so called standard interface. A standard interface, such as a Channel-to-I/O interface, is always defined to meet many different architectures, e.g. printers, tape units, disc units, display devices, architectures that still have to be invented, etc. in different quantities and configurations. At the time of the definition of the standard, the current application area is known, and there is a rough estimate of the characteristics of future applications. Definition of a standard to include all current and future applications is not only impossible, it is also highly inappropriate since it loads any particular application with the overhead of a multiplicity of unused application functions. Instead the standard is defined to suit all requirements of current and future applications without containing the specific functions of individual applications. The standard is by definition incomplete. Consequently, when the standard is used in a particular application, each relational function has to be extended with application dependent functions. Those application dependent functions form yet another layer around the source and sink layer of the standard interface, and are designated 'Application' in figure 12a.

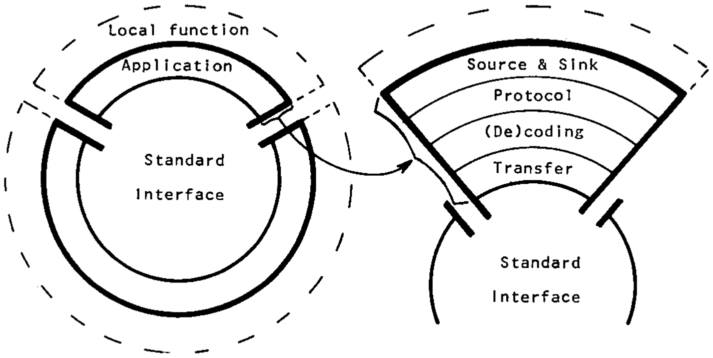


Figure 12a: Application of a Standard Interface

Figure 12b: Partitioning of the Application layer.

The consequence of this structure is that the variables exchanged among the application functions are unknown, i.e. transparent to the standard interface, and yet pass all layers and the message path. Since we want the function of the standard te remain invariant with each application, it implies that the standard has to provide for the space and time for the exchange of those variables. If on the level of the central message path the available space is to be defined in terms of available code elements, the definition of the available space at the level of the source and sink functions has to be in terms of the same number of code elements, since the coding of the variables is transparent with respect to the standard. The coded source in figure 3 is an example. Therefore, in using a standard interface in a particular application, the application dependent interface can be defined according to the procedure explained above. The standard interface is now embedded as a central message path with a high level of complexity. (See figure 12b).

6. Application

The structuring and description discipline has been succesful applied to a number of existing and proposed standard interfaces. Among these are a complex data transmission interface [2], two I/O interfaces, one complex Channel-to-I/O interface [1], and an instrumentation interface [4]. The relational function of the secondary station of SDLC [2] was for example described by 25 functions, each of an average complexity as shown in the figures 8 and 9. It contains 4 sources, 2 sinks, 8 protocol, 2 decoding, 3 encoding, and 6 transfer functions. A formal specification was developed as far as the intentions of the interface architects were stated unambiguously. This specification was generally a fraction of the length of the original document. As part of the description process ambiguities and omissions in the original documents were systematically uncovered.

The state description technique was introduced in an IEC (TC 66/WG 3) standardization activity in june 1973 [4], and eventually accepted as the method to define the considered interface. In the opinion of the committee it has contributed much to the fact that the definition work was practically completed within 9 months, that is May 1974 [5]. A structured and complete description of this interface can be found in [3].

7. Conclusion.

The proposed design discipline facilities fast, correct, efficient and clear specification, interpretation, and judgement of an interface through the definition and its evaluation into a structured specification methodology. As such it can be profit for both interface designers and users:

- The definition provides a better understanding, of what an interface substantially is: a specification of a portion of each of the related architectures (relational functions) and the architecture of the message path, defined to provide cooperation of the related architectures. It is not the story of the reporter, who is sitting on a grandstand, viewing the communication between the related architectures, observing, interpreting and logging what happens. It is the rules of the game according to which the teams play.
- The architecture of each relational function and the message path is specified individually. For all these architectures one specification methodology and language should be used. Poor interface specification mixes relational functions and message path, as well as specification methodologies and languages.
- The horizontal and vertical partitioning strategy for the specification of the relational functions facilitates the recognition of the nature of functions of a particular application and their embedding in such a structure. It facilitates easier specification and recognition of quality and correctnes of the individual and compound functions.
- A standard interface is by definition incomplete. It can be interpreted as a complex central message path, that can be extended to a complete interface in a particular application.
- The method has proven to be applicable to a number of widely used interfaces.

8. Acknowledgement.

The project Interface was suggested by G.A. Blaauw. It is one of the projects of the digital techniques group at Twente University of Technology that focusses on the control of the design process of digital systems. The author would like to thank G.A.Blaauw, B.v.d.Dolder, and R.Davey for their constructive criticism during the preparation of this paper.

9. References.

1. M.J. Heg - Formal description and evaluation of a proposal for an international standard for an input/output interface for electronic data processing systems (ISO TC97/SC13) - M. Sc. Thesis (Dutch/English). June 1975 - Twente Univ. of Techn.
2. B.v.d. Dolder - Algorithmic description of a data transmission interface - M. Sc. Thesis (Dutch) - June 1975 - Twente Univ. of Techn.
3. C.A. Vissers - Digital Techniques IV: Interface-Lecture notes - Spring 1975 - Twente Univ. of Techn.
4. Byte-serial bit-parallel standard interface for programmable measuring apparatus - Drafts of July 1973 and June 1974 - IEC TC66/WG3.
5. D.E. Knoblock, D.C. Loughry, C.A. Vissers - Insight into Interfacing - IEEE Spectrum - May 1975 - pp. 50-57.
6. D.L. Parnas - Information distribution aspects of design methodology - Proc. IFIP, 1971 - North-Holland Publishing Company (1972).
7. R.W. Floyd - Assigning meanings to programs - Proceedings of symposia in Applied mathematics, Vol. 19, Mathematical aspects of computer science, pp. 19-32, American Mathematical Society, 1967.
8. C.A.R. Hoare - An axiomatic basis for computer programming - Comm. ACM. 12, 576-580, 583 (1969).
9. F. Wijnstra - A conversational system for representation and verification in APL of interfaces - M. Sc. Thesis (Dutch) - sept. 1975 - Twente Univ. of Techn.
10. E.W. Dijkstra - The structure of the THE multiprogramming system - CACM, Vol. 11, No.5, May 1968. pp. 341-346.