ETSI GS NFV-INF 007 V1.1.1 (2014-10)

Network Functions Virtualisation (NFV); Infrastructure; Methodology to describe Interfaces and Abstractions

Disclaimer

Reference
DGS/NFV-INF007

Keywords
interface, NFV

ETSI

650 Route des Lucioles
F-06921 Sophia Antipolis Cedex - FRANCE

Tel.: +33 4 92 94 42 00 Fax: +33 4 93 65 47 16

Siret N° 348 623 562 00017 - NAF 742 C
Association à but non lucratif enregistrée à la
Sous-Préfecture de Grasse (06) N° 7803/88

Contents

Intellectual Property Rights

IPRs essential or potentially essential to the present document may have been declared to ETSI. The information pertaining to these essential IPRs, if any, is publicly available for **ETSI members and non-members**, and can be found in ETSI SR 000 314: *"Intellectual Property Rights (IPRs); Essential, or potentially Essential, IPRs notified to ETSI in respect of ETSI standards"*, which is available from the ETSI Secretariat. Latest updates are available on the ETSI Web server (http://ipr.etsi.org).

Pursuant to the ETSI IPR Policy, no investigation, including IPR searches, has been carried out by ETSI. No guarantee can be given as to the existence of other IPRs not referenced in ETSI SR 000 314 (or the updates on the ETSI Web server) which are, or may be, or may become, essential to the present document.

Foreword

This Group Specification (GS) has been produced by ETSI Industry Specification Group (ISG) Network Functions Virtualisation (NFV).

Modal verbs terminology

In the present document "**shall**", "**shall not**", "**should**", "**should not**", "**may**", "**may not**", "**need**", "**need not**", "**will**", "**will not**", "**can**" and "**cannot**" are to be interpreted as described in clause 3.2 of the ETSI Drafting Rules (Verbal forms for the expression of provisions).

"**must**" and "**must not**" are **NOT** allowed in ETSI deliverables except when used in direct citation.

1 Scope

The present document describes how Network Functions Virtualisation (NFV) related interfaces and abstractions are to be derived and specified. It describes the concepts associated with these interfaces and abstractions. It covers the specification process / methodology in general. It presents a cross-cutting framework which covers compute, hypervisor and infrastructure network domains, also data, control and management planes.

The present document does not specify all the interfaces and abstractions as these are covered by other documents, e.g. the NFV INF domain specific documents. Examples of interfaces and abstractions are nevertheless supplied to illustrate the methodology.

The present document does not provide any detailed specification but makes reference to specifications developed by other bodies and to potential specifications, which, in the opinion of the NFV ISG could be usefully developed by an appropriate standards development organization (SDO). Furthermore the NFV INF domain specific documents will not provide detailed specifications either.

2 References

References are either specific (identified by date of publication and/or edition number or version number) or non-specific. For specific references, only the cited version applies. For non-specific references, the latest version of the referenced document (including any amendments) applies.

Referenced documents which are not found to be publicly available in the expected location might be found at http://docbox.etsi.org/Reference.

NOTE: While any hyperlinks included in this clause were valid at the time of publication, ETSI cannot guarantee their long term validity.

2.1 Normative references

The following referenced documents are necessary for the application of the present document.

Not applicable.

2.2 Informative references

The following referenced documents are not necessary for the application of the present document but they assist the user with regard to a particular subject area.

Not applicable.

3 Definitions and abbreviations

3.1 Definitions

For the purposes of the present document, the following terms and definitions apply:

unicode: unique, unified and universal encoding

zeroconf: zero configuration networking

3.2 Abbreviations

For the purposes of the present document, the following abbreviations apply:

3GPP	3rd Generation Partnership Project
ACL	Access Control List
API	Application Programming Interface
CLR	Common Language Runtime
CPU	Central Processing Unit
EJB	Enterprise JavaBeans
ETSI	European Telecommunications Standards Institute
GS	Group Specification
HFB	Host Functional Block
IETF	Internet Engineering Task Force
ISG	Industry Specification Group
IT	Information Technology
ITU-T	International Telecommunication Union Telecommunication Standardization Sector
JIT	Just In Time
JSON	JavaScript Object Notation
MANO	Management and Orchestration
NFV	Network Functions Virtualisation
NIC	Network Interface Card
NPU	Network Processing Unit
OS	Operating System
OVS	Open Virtual Switch
OVSDB	Open Virtual Switch Database
SATA	Serial Advanced Technology Attachment
SDN	Software Defined Networking
SDO	Standards Development Organization
SR-IOV	Single Root I/O Virtualisation
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
UML	Unified Modelling Language
UTF	Unicode Transformation Format
VF	Virtual Function
VFB	Virtualised Functional Block
VM	Virtual Machine
VNF	Virtual Network Function
VNIC	Virtual Network Interface Card
VT	Virtualisation Technology
XML	eXtensible Markup Language

4 Objectives

The three key features of the NFV approach are:

1) Separation of the software defining the network function from generic high volume hardware servers, storage devices and network switches.

2) Independent modularity of the software and hardware components.

3) Automated orchestration which will automate remote installation and management of the virtual functions on the generic hardware.

4.1 Requirements

The overall vision of Network Functions Virtualisation gives rise to the following overall requirements w.r.t. interfaces and abstractions:

- Each functional block or component shall be described as an abstraction, documenting the purpose and behaviour of the functional block or component, and as a set of interfaces to the abstraction.
- The interfaces and abstractions shall be documented at a sufficiently detailed level to permit standardizing organizations to create specifications for these interfaces and abstractions.
- The NFV ISG is expected to liaise with standardizing organizations in order to ensure that such specifications are sufficiently detailed to enable interoperability between platforms and devices hosting Virtualised Network Functions that are offered by different vendors.
- The NFV ISG is expected to liaise with standardizing organizations in order to ensure that such specifications are decoupled from vendor-specific design and implementation choices within platforms and devices hosting Virtualised Network Functions.

4.2 Standardizing Organizations

The standardizing organizations that are responsible for creating detailed specifications of each interface and abstraction are listed in the overview document and in domain specific documents.

Other organizations that define methodologies relevant to interfaces and abstractions include the Object Management Group (specifically for UML) and the International Council on Systems Engineering (specifically for Systems Engineering).

5 Architectural Principles w.r.t. Interfaces and Abstractions

Many network systems, including those specified by 3GPP, IETF, and ITU-T, are specified using the principles of systems engineering. Each component of the overall system is specified as a functional block and the interactions between the functional blocks are specified as interfaces.

This clause details the basic functional block based system composition methodology and extends it to cover the process of virtualisation.

The representation of functional blocks is part of the working methods of many industries as well as different disciplines and perspectives within those industries. As a result, there is not a clear common representation of functional blocks which is unambiguous across different industries, disciplines, and perspectives.

As tools that describe functional blocks are most often used by engineers for the design, development and construction of functional blocks, quite naturally, many tools give considerable emphasis to these phases of the functional block life cycle. For example, in the construction phase, the reuse of common design features is especially important as reuse increases efficiency. Many tools therefore give considerable emphasis to the reuse of such features. In this case classification of functional blocks according to common design features is of considerable value and the natural starting point for describing functional blocks is the class. It is natural to start by representing a class of functional blocks which can be built using the same design. The class diagram can also contain hierarchy, for example an inheritance hierarchy, which can show increasing scope of design reuse at higher levels of the class hierarchy.

However, when describing the operation of functional blocks, the individual instances of functional blocks and the way individual functional blocks interact with each other are important. In this case the natural starting point is not the class but the individual instances. Classification and hierarchy of classification is much less relevant at the operations stage. More important is the way individual functional block instances are interconnected and interact.

There are of course many other aspects to functional blocks which may be important to represent in different way. For example the nature of what is passed between functional blocks may be important to differentiate. In all information and network systems, it is information that is passed between the functional blocks. However, even in this narrow category, in may be important to distinguish a continuous flow of information from discrete events. More general systems may pass fluid (pressure and flow), electricity (voltage and current), rotation (revs and torque), money, etc.

The present document is concerned with the basic characteristics of information functional blocks. We can assume that all the parameters passed between functional blocks are information of one form or another.

This clause considers some of the basic properties of information functional blocks. It focuses on the case where a functional block such as a server or a network acts as a host functional block, hosting virtual functional blocks such as virtual machines and virtual networks.

In this case, it is important to highlight the properties of functional blocks in operation and so all the discussion and diagrams in the present document show functional block instances (and not classes of functional blocks) unless otherwise stated.

5.1 System Composition using Functional Blocks

5.1.1 Functional Blocks as the Primary Specification Method

The great majority of specification of telecommunications systems specifies functional blocks using the methodology of systems engineering. A functional block is the basic unit of a system and its specification can and should be precise.

A functional block, that is a single functional block instance, consists of:

- A set of input interfaces.
- State.
- A transfer function.
- A set of output interfaces.

When describing practical functional blocks, the interfaces may be described such that an input interface is paired with an output interface. This is normally convenient when describing the interconnection between functional blocks.

When considering the functional block at its most fundamental level, the proper operation of a functional block is causal and the flow of causality from input to output is central to the methodology. In this case is normally more convenient to consider all input as separate from all outputs. A fundamental view of a functional block illustrated in Figure 1.

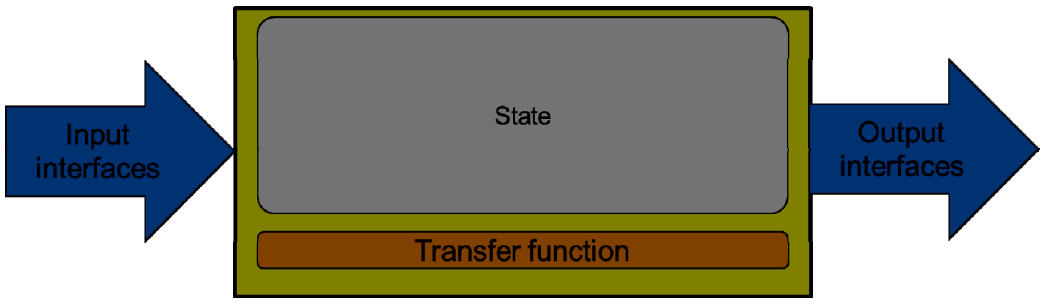

Figure 1: The fundamentals of a functional block

There a number of fundamental properties of functional blocks:

- The transfer function is fixed and defining of the functional block.
- The set of all possible values of state is fixed and is defining of the functional block.
- The set of all possible input values is fixed and is defining of the functional block.

- The set of all possible output values is fixed and is defining of the functional block.
- The transfer function is the set of mapping from a specific value of tuples of input and state to a specific value of tuple of output and state.
- The state is the ability of the evolution of the functional block to be dependent on historic inputs and not just on the current input.
- The process of acquiring the current input value, the current state value, calculating the transfer function mapping, and setting the next state value and the next output value takes a finite amount of time.
- The exact amount of time may vary with each specific mapping.

A central property of functional blocks is the complete and formal separation of the static from the dynamic. Using a more IT oriented terminology, the input, output, and internal (i.e. state) data structures and all the methods (i.e. the transfer function) and static. They shall not change. Only the values of data within the data structures can change; these values are the only things which are dynamic.

For standardized functional blocks, in order to ensure proper interoperability, the goal would normally be to fully define all the static parameters of the functional block in the standard.

5.1.2 Interconnection of Functional Blocks

Having defined what a functional block is from the inside, the next fundamental property of a functional block the ability to interconnect functional blocks. This is achieved by connecting an output interface of one functional block with the input interface of another functional block. For this to work the following shall be true.

- The data structure of the output interface of the functional block on side of the interconnection shall be compatible with the data structure input interface of the functional block on the other side of the interconnection. The output set of values shall a subset of the input set of values.

This interconnection of interfaces is called an interface binding. This arrangement is illustrated in Figure 2.

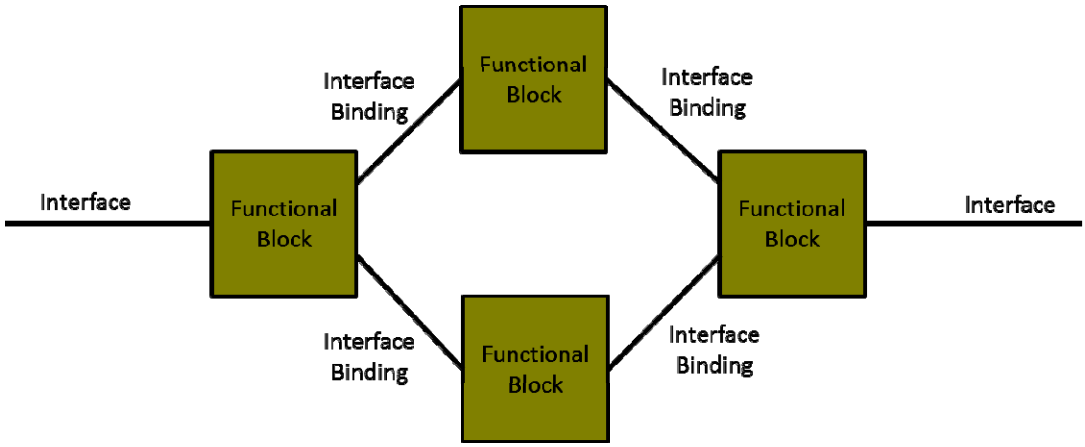

Figure 2: Interconnection of functional block with interface bindings

As illustrated in Figure 2, when a number of functional blocks are interconnected all together, the interface bindings form a topology between the functional blocks.

5.1.3 Recursive Structure of Functional Blocks

When a number of functional block are interconnected in a topology, some input interfaces and some output interfaces remain. A fundamental property of the functional block methodology is that the entity as viewed through these remaining input and output interfaces is also a functional block and meets all the properties of a single functional block.

This is illustrated in Figure 3.

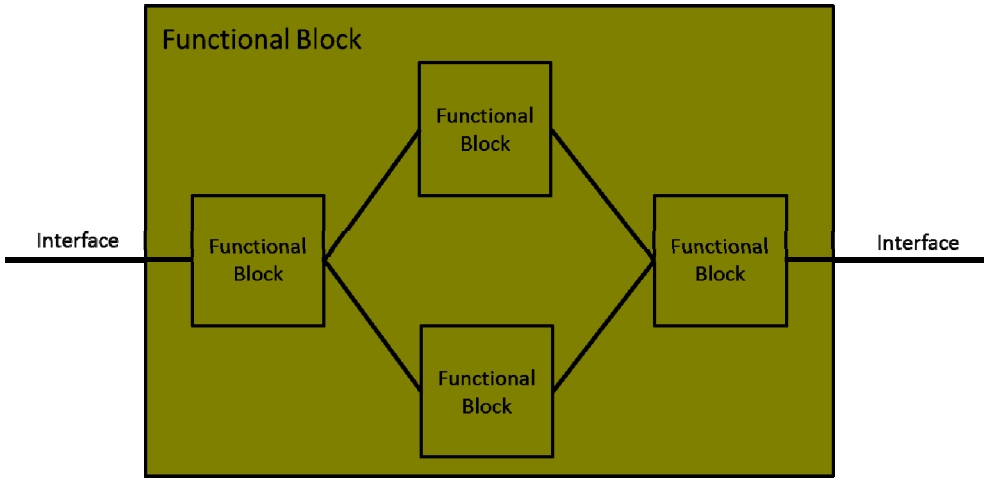

Figure 3: Recursive property of functional blocks

This means that functional blocks have a fundamental recursive property. A larger functional block can be created by aggregating a number of a smaller functional block and interconnecting them with a specific topology.

Another fundamental property of functional blocks which is immediately apparent from this recursive property is that functional blocks are inherently parallel, concurrent, and asynchronous. Any sequential and synchronous properties will arise only as a special case, normally by imposing explicit design constrains on the static properties of all the constituent functional blocks.

5.1.4 General UML Diagram for Basic Functional Block Methodology

It is possible to summarize the basic entities of the functional block methodology using a UML class diagram. UML class diagrams show classes, that is, sets of things which all have the same property. Most often classes in UML class diagrams are there to define the construction of members, often called instances, of the class. In this way, members of the class have the same properties because the class definition from the class diagram is used directly to create the instances. However, the class can also be used to categorize things even if they were not created directly by the specification. They can be classified in retrospect, rather by design.

The ability to classify in retrospect is important for many aspects of practical systems. Generally component functional blocks are designed, built, and made operational at different times and often it is not possible to change existing functional blocks when introducing new components. The UML diagrams used here show general classification, and instances of functional block classes may be classified as instances of the classes in the diagram after they have been designed, built, and made operational.

UML class diagrams show classes and relationships between classes. A relationship is shown between two classes. Two completely different type of relationship are as follows.

- Generalization (also called inheritance). This relationship shows that two classifications are different viewpoints of the same thing. The instances of the classes are the same instances. The relationship is often referred to as an 'is a' relationship. In UML, generalization is shown by an open triangular arrowhead
- Associations. These are relationship between two separate instances which interact with each other. UML allows three different strengths of association:
  - Association. General interaction between instances.
  - Aggregation. An association where the instance on one class is a component part of an instance of the other class.
  - Composition. An aggregation where the existence of the component instance depends on the existence of the aggregate instance.

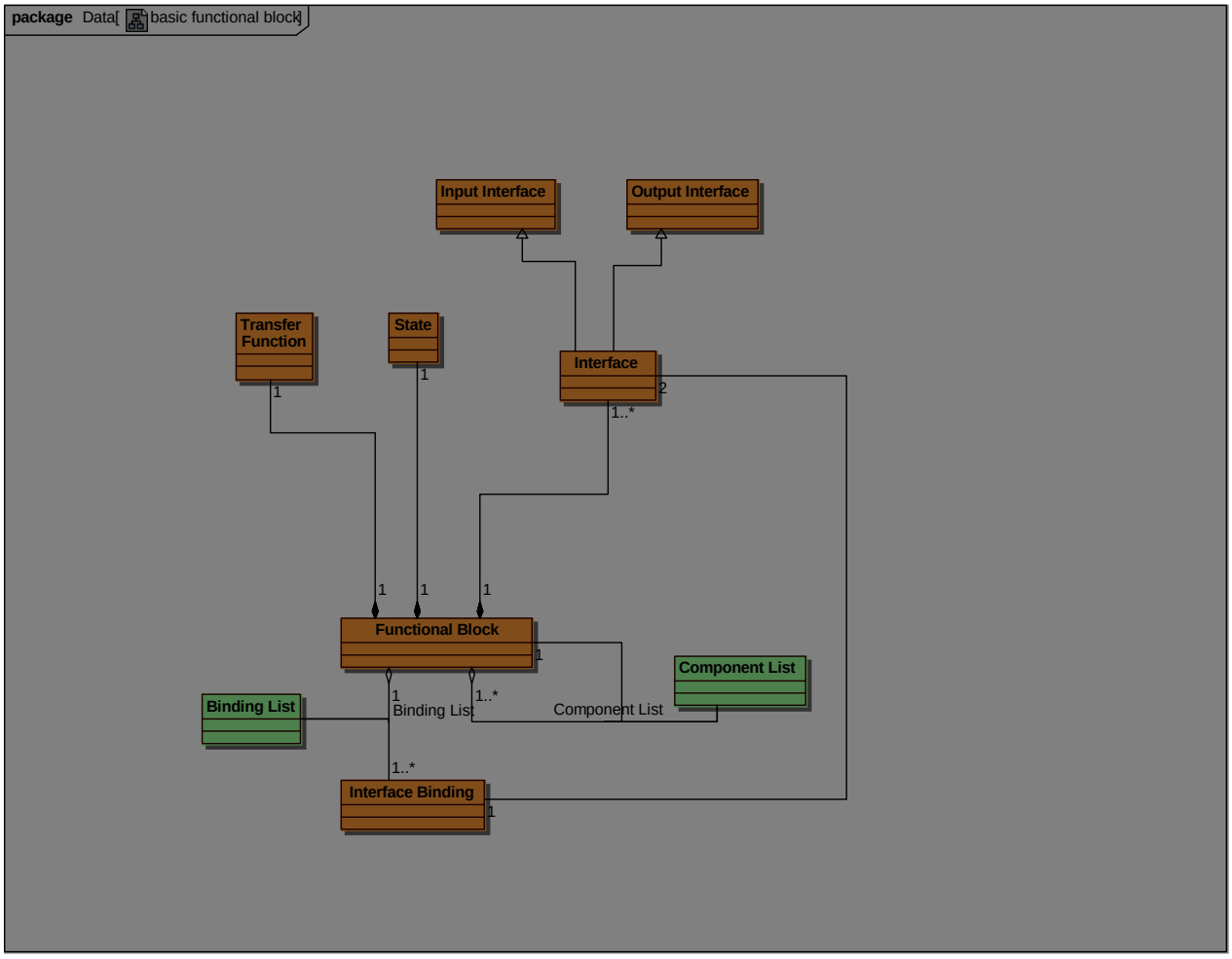

Figure 4: UML class diagram of functional blocks

Figure 4 shows a UML class diagram of functional blocks (note the functional block class is called virtual function for consistency with the following clause). The diagram specifies the three basic parts of a functional block (transfer function, state, interfaces), the ability to bind interfaces, as well as the property of recursive composition. Note that this use of composition is different to the UML definition.

5.2 Extension of Functional Block Methodology to Virtualisation

Functional block methodology does not anticipate or direct support virtualisation. However, the foundations of the methodology are very general and mathematically robust. It is still possible therefore to understand virtualisation in terms of functional blocks. This clause develops the extension of the methodology in terms which are still fully based on the same general, mathematical principles and so still retains the formal robustness of the methodology.

In summary, the essence of virtualisation is to revisit the boundary between the static and dynamic parts of the functional block specification. We will see that virtualisation uses a process with the following steps:

- A host function has some dynamic state which can be set to a value (configured) and held constant for a prescribed period of time (which will be the lifetime of the virtualised function).
- This configuration allows the host to appear to operate according to the specification of the virtualised function - this configuration of the host implements the virtualised function.

In fact, it is the case that all implementation is exactly this process. It is indeed, this mechanism of virtualisation that allows any implementer freedom to choose and optimize their own implementation of the virtual function. Moreover, all implementation independent specification is a specification of a virtual function.

This means that the requirement to extent the method of functional blocks to NFV has also provided a complete and robust solution to implementation independent specification.

5.2.1 Virtualisation: Virtual Interfaces and Container Interfaces

Figure 5 shows an illustrative set of interconnected functional blocks. For the purposes of illustration, we will now consider what happens when the second two are implemented as virtualised functions on a host function.

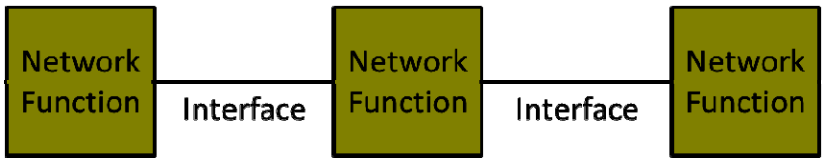

Figure 5: Illustrative set of interconnected functional blocks

Figure 6 shows the results of this process.

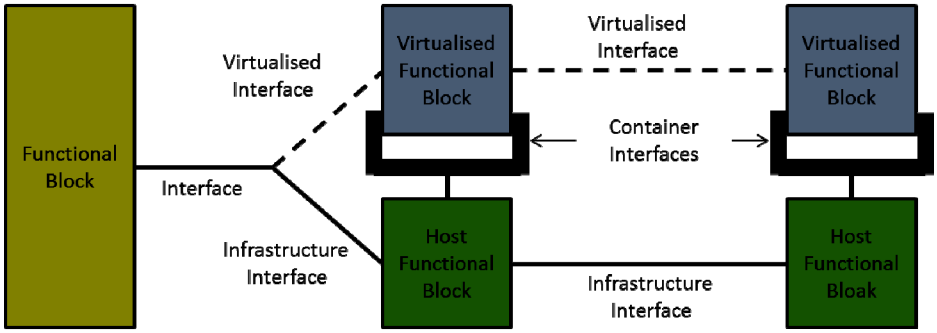

Figure 6: Implementation of two functional blocks as virtualised functions on host functions

This process has resulted in the following:

- The division of a functional block between a host functional block (HFB) and a virtualised functional block (VFB).
- The creation of a new container interface between the HFB and the VFB it is hosting.

NOTE: The use of 'container' in this context is general and is consistent with its use over many years for the hosting of remotely deployable executable modules such as Enterprise Java Beans (EJB). It's use should not be confused with the use of the term for a specific Linux technology of protected execution domains which are not full virtual machines.

- The division of the interface between the two network functions which are now virtualised between an infrastructure interface and a virtualised interface.
- The interface to the non-virtualised functional appears to be a homogeneous interface at its end with the non-virtualised function, however, at its virtualised end, it appears to be divided between an infrastructure interface and a virtualised interface.

However, in carrying out this process, there are two very important distinctions from the standard functional block description:

- The VFB is ***not*** a functional block independent of its HFB.
- The container interface is ***not*** an interface between functional blocks equivalent to other interfaces.

This arises as the VFB cannot exist autonomously in the way a functional block can. The VFB depends on the host function for its existence, and if the host function were to be interrupted, or even disappear, then the VFB is also be interrupted or disappear. Likewise, the container interface reflects this existence dependency between a VFB and its host function.

The relationship between the VFB and its HFB is:

- the VFB is a configuration of the host function;
- the VFB is an abstract view of the HFB when the HFB is configured to be the VFB.

Therefore a HFB, when configured as a VFB, has the ***external appearance*** as being a functional block implementing the VFB specification. It is the HFB that is the functional block but it appears from the outside to be the VFB. Equivalently, the VFB is an abstract view of the HFB.

5.2.2 Virtual Functions and Host Functions

The interesting property of virtualisation is the things that are predefined. As set out above, the data structures defining the value ranges of the inputs, outputs, and state are all predefined, as is the transfer function. These are static to the functional block and defining of the functional block. Defining these static parameters fully specifies the functional block. The specific values of the inputs, outputs, and state are dynamic to the functional block and are not part of its specification.

With this background to the formal definition of a functional block, this can be developed to formally define virtualisation. This is set out in the four steps shown in Figure 7.

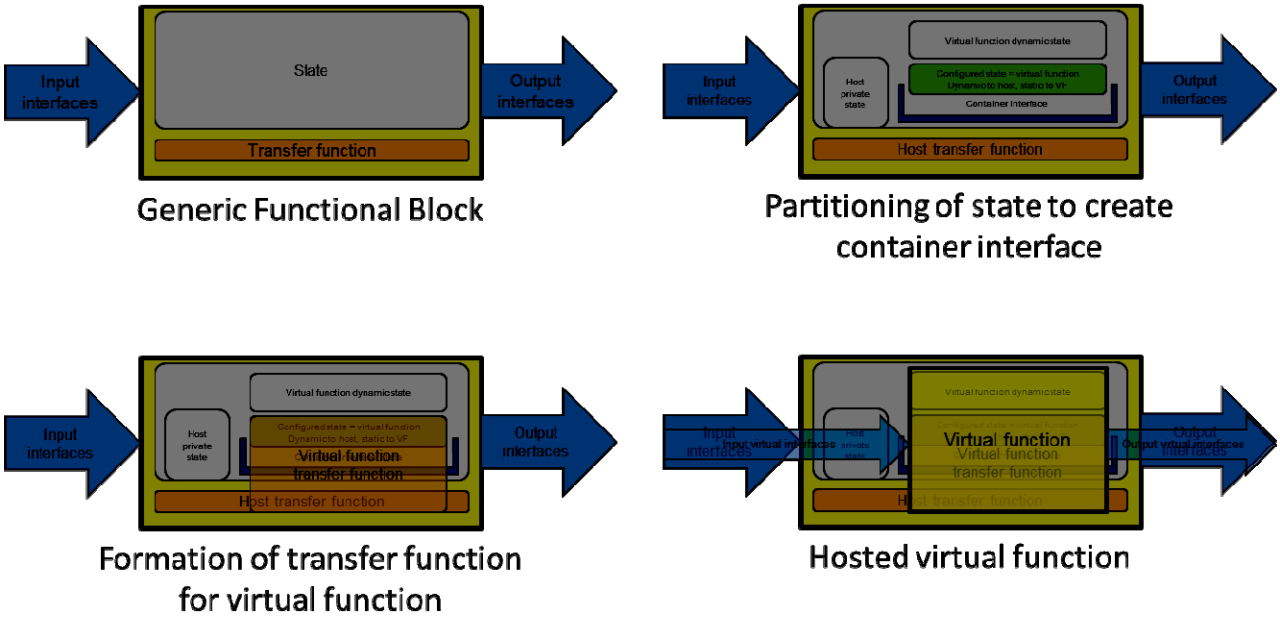

Figure 7: Functional Block Virtualisation Process

The first step shows a generic functional block with inputs, outputs, state, and a transfer function.

The second step shows the state of the functional block partitioned into three distinct types of state:

- state which holds the dynamic state which is private to the host function;
- state which will be configured and then held constant and invariant during the life of the virtualised function and which defines the virtualised function;
- state will hold the dynamic state of the virtualised function;

This second step shows the essence of virtualisation: this fixing of some of the host's state so that it is held constant for the life of the virtualised function. The state which is available for fixing is the aspect of the container interface that allows the virtual function to be defined.

The third step shows how this fixed state combines with this host transfer function in order to define the transfer function of the virtualised function. This combination of the fixed state and the host transfer function allows the virtual function to execute.

The fourth step shows how the range of values defined for the host input and output interfaces are partitioned so that only those values which control the execution evolution of the virtual function are identified as belonging to the virtual interfaces of the virtual function. The virtual input interface is therefore defined as a subset of the range of values of the host function's input interface. Similarly, the virtual output interface is defined as a subset of the range of values of the host function's output interface. The behaviour of the virtual function as perceived through the virtual interfaces, is now exactly that of the virtual function.

With this formal definition of virtualisation, it demonstrates:

- a VFB is a configuration of its HFB;
- a VFB is an abstract view of the HFB when appropriately configured.

5.2.3 Recursive Virtualisation

Having defined the process of virtualisation, if can now be seen that the process is fully recursive. An operating virtual function block can itself be a host functional block. This is illustrated in Figure 8.

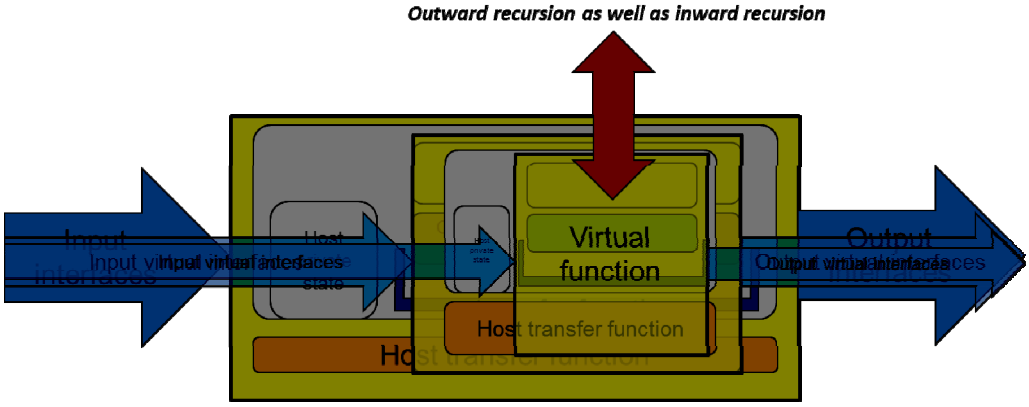

Figure 8: Fully recursive nature of virtualisation of functional blocks

A point of which emerges from this view is when the recursion is viewed 'downwards' to look at the nature of a HFB. We see that a HFB is itself a VFB. This is actual a statement of the observation that any functional block is itself an implementation.

The following are consequences of the recursive nature of virtualisation:

- Implementation is a process of hosting.
- The means of achieving an implementation independent specification is to specify a virtual function.
- There is no 'bottom' to the recursion meaning that there is no 'atomic' level of implementation.
- There is normally a useful level of implementation visibility, however, this is selected based on practical consideration.

NOTE: While it would not normally be of any practical relevance to NFV, the concept of host configuration does include manual configuration of the spatial configuration of elements. In other words, the sitting and installation of physical equipment together with the plugging in of physical interfaces is still a logical configuration.

5.2.4 Configuration Lifespan, Operational Interfaces and Configuration Interfaces

Having precisely defined the process of virtualisation, it can be seen that the existence of a VFB depends precisely on the specific configuration of the HFB which is hosting the VFB. Given that the definition of a functional block requires that the transfer function is static and is defining of the functional block, this means that the configuration which creates the VFB shall not change during the lifespan of the VFB. Indeed, the lifespan of the VFB is defined by the period during which the configuration which defines it remains unchanged.

This means that there is a profound and fundamental difference between input interfaces of the HFB which determine the configuration state from the input interfaces which determine the dynamic state of the VFB. This distinction can define a formal partitioning of the HFB interfaces:

- HFB configuration interfaces which allow the creation/destruction of VFBs. The data structure of these interfaces is, by definition, a programming language.
- VFB operational interfaces which are the virtual interfaces to the VFBs.
- Host private interfaces which control aspects of the HFB which are neither part of the definition VFB or the operational interfaces of the VFB, control aspects private to the HFB.

This partitioning of the HFB interface is illustrated in Figure 9.

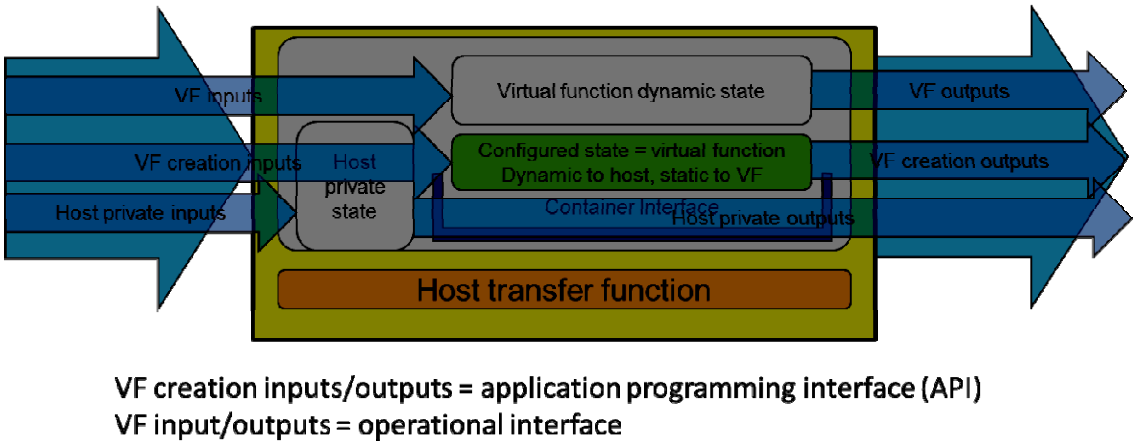

Figure 9: Partitioning the interfaces of a HFB

5.2.5 Mapping Between VFBs and HFBs

The functional block methodology is fundamentally parallel and concurrent. This means that VFBs and HFBs may be distributed with high degrees of concurrency.

This means that, when decomposed, the mapping of VFBs to HFBs may be a complex many to many relationship. In particular:

- one HFB may host more than one VFB;
- A single VFB may be hosted across many HFBs.

This complex mapping may be viewed in two ways:

- The mapping arises from the configuration of HFBs which created the VFBs and there the mapping is simply an observation of has is in actual operation.
- HFBs has certain properties based on their level of concurrency which will place constraints on the time required for some specific states to evolve and this will place significant constraints on the mapping if certain performance for the evolution of state is required.

The second of these is especially important where state is required across state which will be common across geographically distributed HFBs. Managing this constraint on the mapping between VFBs and geographically distributed HFBs is very often a primary design consideration for the performance and/or stability of an end to end VFB.

This mapping is illustrated in Figure 10.

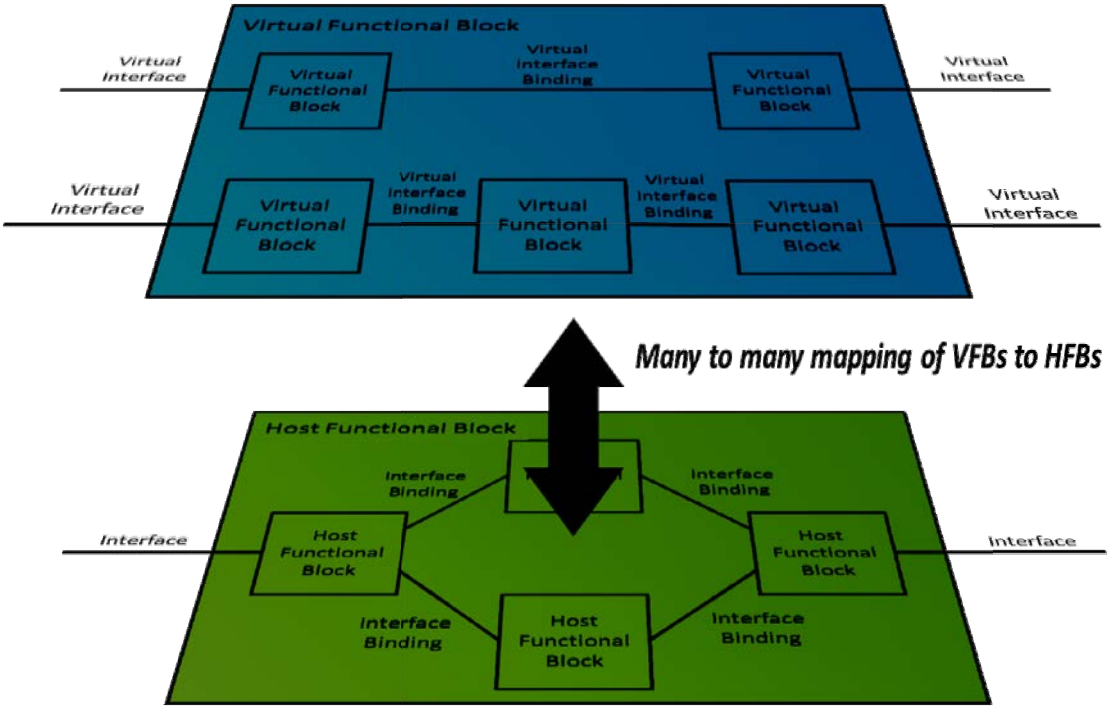

Figure 10: Mapping of VFBs to HFBs

5.2.6 General UML Diagram for Extended Functional Block Methodology

It is possible to extend the UML class diagram to include virtualisation. This is shown in Figure 11. The primary extension is the inclusion of the generalization that a virtual functional block 'is a' host functional block. The hosting aspects now have the configuration interfaces and the private interfaces. The mapping of VFBs to their hosting HFBs is shown by an association.

Figure 12 further extends this to also depict the relationship data structures.

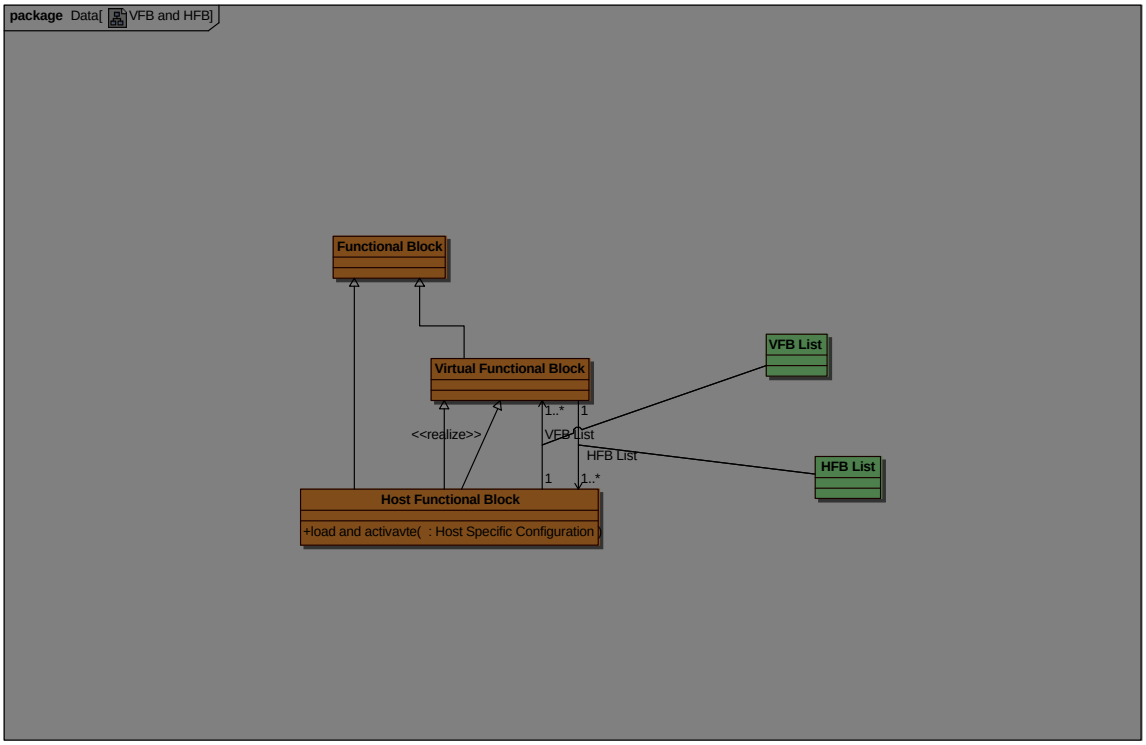

Figure 11: UML class diagram of functional blocks including virtualisation

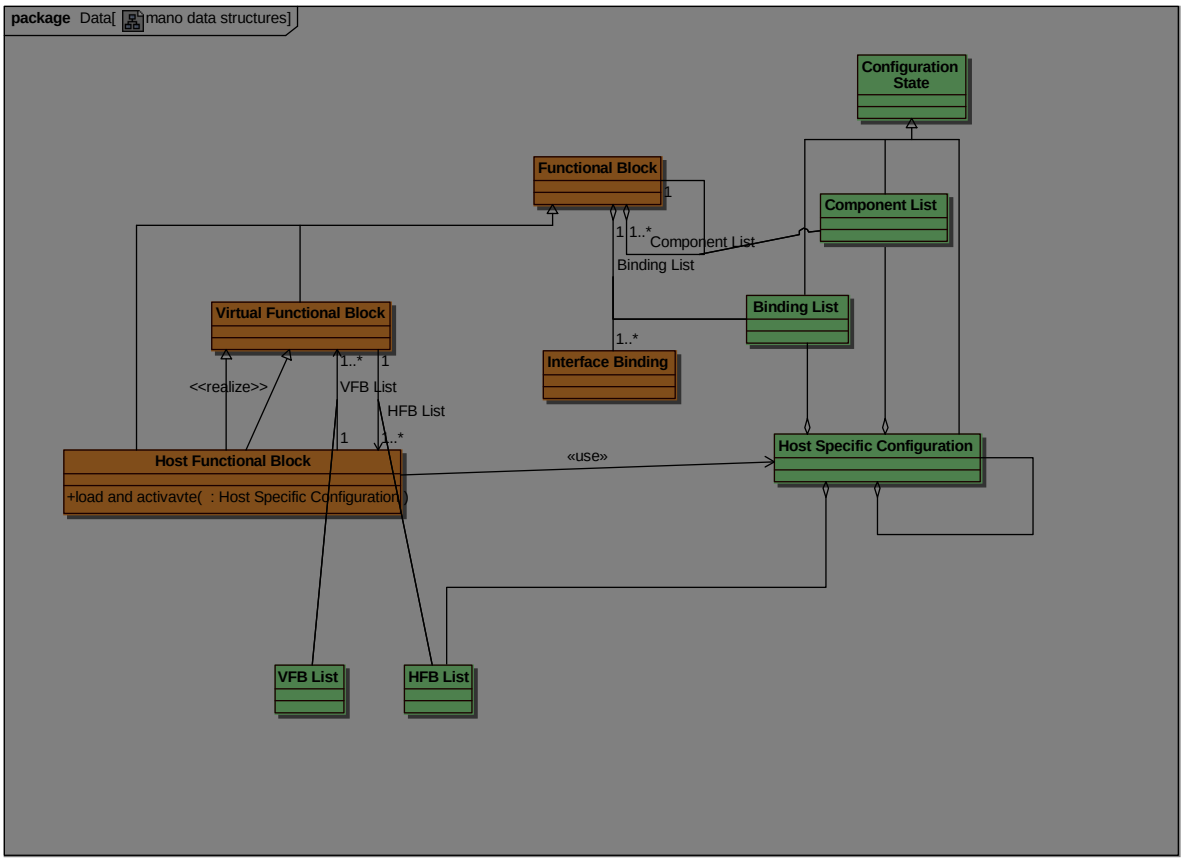

Figure 12: Functional blocks with relationship data structures

5.3 Describing and Specifying Interfaces and Abstractions

5.3.1 Functional Blocks, Components, Abstractions and Interfaces

As detailed in the previous clause, a significant, fundamental principle of the systems engineering approach is that the complete system is specified by the specification of the component functional blocks and their interconnection, and all components are always functional blocks.

The term abstraction is used to refer to the high level description of the required behaviour of a functional block or component. For a more specific definition of the term abstraction, refer to clause 7.

5.3.2 Specifying Organizations and Level of Detail

Note that the specification of standards related to NFV is delegated to individual standardizing organizations. The descriptions of interfaces and abstractions produced by the NFV ISG therefore need to be sufficiently detailed to clearly document requirements for such specifications, but not detailed enough to constitute actual specifications.

The present document recommends minimum standards w.r.t. describing interfaces. (These only apply to the interfaces that are described / specified.) NFV working groups need to at least discuss each aspect when describing interfaces in order to make informed and deliberate decisions when deviating from the recommended minimum standards.

Note that not all interfaces and abstractions need to be specified. The focus needs to be on specifying those that affect critical aspects of the overall system's behaviour, e.g. interoperability. Additional guidance is supplied in clause 7.

Perspectives w.r.t. Abstractions:

1) Build time
2) Run time
3) Configuration time (e.g. as viewed by MANO)

5.4 Types of Interfaces

Hardware interfaces permit access to e.g. the network (Ethernet), storage (SATA) etc. As these are manipulated and accessed via a software interface (a "driver" implemented as a user mode library or a kernel mode entity), these are represented as the software interface, and therefore not discussed separately.

Callable interfaces ("APIs") can be invoked directly by software, provided appropriate language bindings are supplied. They may be implemented by local procedures, by stubs that invoke remote procedure calls, or by software which invokes kernel or hypervisor facilities to implement the required behaviour.

Protocol based interfaces are specified by documenting the protocol details. The required protocol is typically specified with reference to an underlying transport (e.g. TCP or SSL), or by referring to a standard marshalling format.

Protocol stacks can be used to supply callable interfaces for protocol based interfaces. Conversely a callable interface can be turned into a protocol based interface by specifying a remote procedure call protocol.

As the NFV ISG merely supplies requirements that are to be conveyed to standards development organizations, the details w.r.t. callable or protocol based interfaces should not be specified by the NFV ISG. Whether interfaces should be callable, protocol based, or hardware based, and the required behaviour would typically be specified. In cases where interfaces are not specified to be protocol based or callable, the semantics implied by the interface being standardized as being callable or protocol based need to be considered.

The behaviour of interfaces and the behaviour of the associated abstractions need to be specified in sufficient detail. Either callable or protocol based interfaces can be defined as synchronous or asynchronous interfaces. Which of these applies would have significant implications w.r.t. the behaviour. Many additional attributes of the behaviour need to similarly be specified.

5.5 Interface Adaptation Mechanisms

This clause details various mechanisms that can be used to adapt interfaces, resulting in the presentation of different interfaces.

The term "shim" refers to an entity which attaches to an interface on one side of the shim while presenting the same or another interface on the other side of the shim.

The shim therefore supplies the following capabilities:

- Conversion / translation between interfaces.
- Shim specific behaviour other than conversion / translation, i.e. shim specific processing, as described by an abstraction.

The following practically useful types of shims can therefore be identified:

- Same interface on both sides: in this case, the function of the shim is not interface conversion/translation, but some other type of processing performed by the shim.
- Different interfaces on both sides: conversion/translation of interfaces may or may not be combined with other processing.

The following shim architectures can be identified:

- Custom proxy / conversion layers, which are purpose designed for each case where a shim is required, e.g. where a shim is retrofitted on an existing interface.
- Generic service usage / service provision plug in mechanisms, where the architecture of the interface itself makes provision for shims or for that matter other layers to be inserted on the interface.

In general, shims can be inserted at various times, e.g. compile time, link time, boot time, run time etc. Which can apply / will apply to a given shim needs to be specified.

Shims can be used to implement the following operations:

- Interface adaptation / translation: the "new" interface could be at a higher, a lower, or the same abstraction level as the "original" interface.

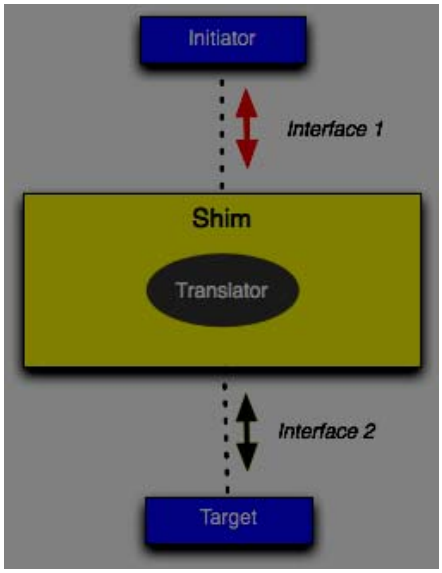

Figure 13: Interface Translation

- Filtering, whereby only a subset of the operations invoked via the interface, or data communicated across the interface, is permitted to cross the interface, e.g. for security purposes certain operations may be prohibited by a so called Access Control List, or only notifications relevant to an application might be retained.

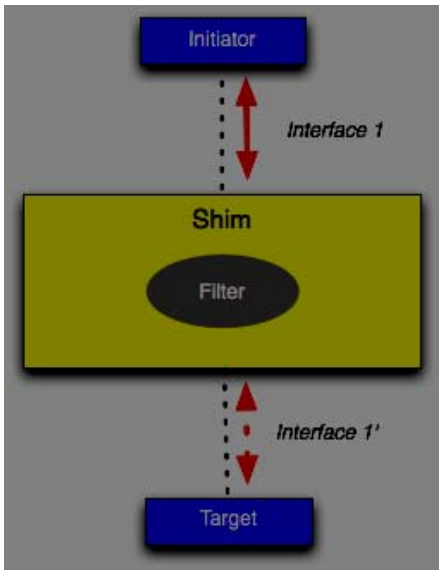

Figure 14: Interface Filtering

- Virtualisation / Partitioning: in this case one underlying / original entity is represented as m entities, or an underlying / original entities are represented as m entities (with n < m).

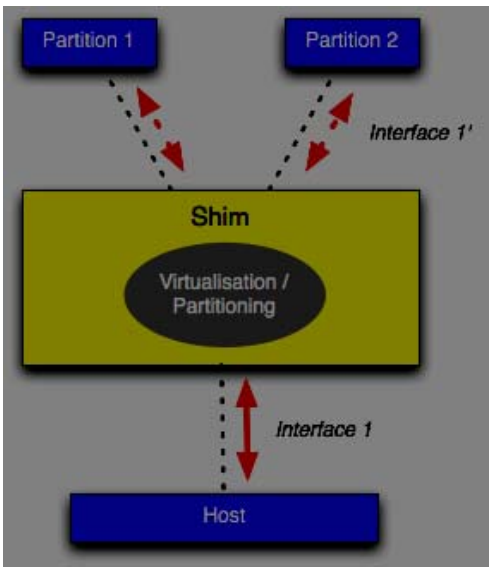

Figure 15: Virtualisation / Partitioning

  - This case is described in detail in clause above.
  - Note that the interface presented to the partitions may differ from the interface to the underlying (host) entity.

- Bonding / Aggregation: here n underlying interfaces are represented as one interface or as or m (m < n) interfaces.

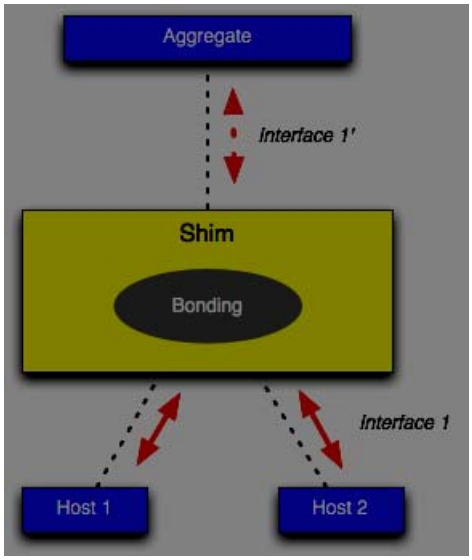

Figure 16: Bonding / Aggregation

- In this case, the bonded interface cannot exist unless at least one underlying interface exists. Some types of bonding require all the originally configured interfaces to exist - if any of them disappear, the bonded interface will also disappear.
- Note that the interface presented to the aggregate may differ from the interface to the underlying (host) entities.
- This is a corollary of virtualisation / partitioning.

- Replication for high availability: here information is replicated to a local or remote entity in order to for example ensure state is synchronized in order to implement high availability systems. For this case, loss of replicated information is typically not acceptable.

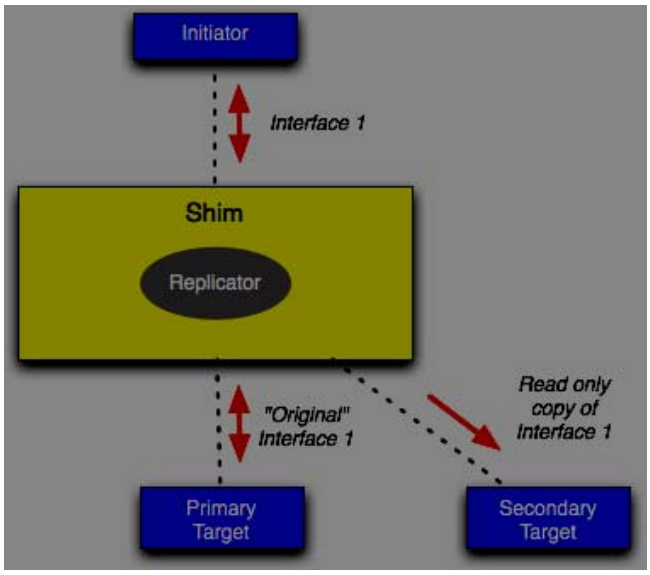

Figure 17: Replication / Monitoring

- Monitoring: here information is replicated for e.g. debugging, auditing, billing purposes, with depending on the mechanism / requirements, the replication potentially being lossy / statistically sampled, etc.

5.6 Naming and Versioning

Interfaces need to be identified, i.e. named. Ideally unified or at least aligned naming conventions should be followed when naming related interfaces.

In order to promote the evolution of standards, versioning information needs to be included to ensure that suppliers and users of interfaces can evolve while maintaining the capability to interoperate. The versioning information would be used to distinguish between stages of evolution of a given interface.

Where possible / applicable, different versions of a given interface should have the ability to co-exist as this would facilitate upgrading individual subsystems / components independently.

When interfaces are requested from SDOs, these requirements will be communicated.

5.7 Discovery of Initiators / Targets and Bootstrapping

The interacting entities may need to discover each other. An entity initiating communication could for example solicit targets by sending broadcast or multicast traffic over a network. This would for example permit a node which is to be centrally managed to discover the available controllers, whereupon it would connect to one or more of these controllers. The most suitable target could be selected, with suitability being dependent on factors like availability, proximity, load, interface version and supported capabilities, etc. Where possible, standardized mechanisms for such discovery should be employed, for example the Zeroconf suite for network (i.e. protocol) based interfaces.

Where discovery is impossible or impractical (e.g. too time consuming or deemed not reliable enough), the applicable targets may need to be statically configured at an initiator.

Dynamic discovery or static configuration can both be employed as bootstrapping mechanisms. Which is employed would depend on whether the mechanism meets the requirements of the system (e.g. the dynamic discovery mechanism may not return a result quickly enough) and whether the prerequisites of the mechanism are available during the bootstrapping phase (e.g. dynamic mechanisms may require the availability of network communication).

When standards are requested from SDOs, discovery and bootstrapping related considerations (if applicable) should be communicated.

5.8 Security

As the overall security of a system is almost always dependent on the security of underlying abstractions and interfaces, requirements need to be specified to cover security of interfaces and abstractions. The following aspects need to be considered:

- Authentication, i.e. how initiators and targets are identified, and how these identities are verified.
- Authorization, i.e. how access rights are determined and managed.
- Privacy, i.e. how access to data in flight and at rest is limited to authorized parties.
- Auditing, i.e. how valid and denied accesses are logged and how these records are made available to those entitled to access them.

Mechanisms that are employed to address security concerns often vary between individual SDOs. The NFV ISG therefore needs to make recommendations w.r.t. these aspects and, once individual standards have been created, examine the resulting standards, ensuring that any remaining gaps / mismatches are addressed. Where new standards are requested, unambiguous requirements need to be communicated.

5.9 Performance and Availability

Performance and/or availability may be the responsibility of a specific subsystem or virtual network function. The achievable performance and/or availability may however be dependent on the performance / availability supplied by a specific abstraction as well as the applicable interfaces. The assumptions in this regard need to therefore be documented for interfaces and abstractions. The following aspects need to be considered:

- Configuration of requirements w.r.t. interfaces - e.g. detailed requirements (throughput / latency) or approximate requirements (relative requirement or class), as well as whether guarantees are needed vs. whether best effort attempts are acceptable.
- Monitoring of performance and availability achieved over interfaces, for management and/or billing purposes.
- Mechanisms to achieve availability, e.g. redundancy.
    - Clause above describes how this can be achieved using interface shims.

5.10 Error and Anomaly Handling

Error, alarm and other anomaly indication mechanisms can be described as synchronous, where success or failure indications are returned immediately after operations have been performed via an interface, or asynchronous, where the success / failure indication is not specifically coincident with the operation, e.g. indications could be delayed, grouped (to reduce overheads), or even distributed to a different party or multiple parties.

Interface specifications need to clearly state which error handling aspects are the responsibility of the initiator (caller) and which will be handled by the target (caller). Error indications may for example need to be communicated to a management system. Furthermore dealing with error conditions (e.g. restarting subsystems) may be the responsibility of an entity itself, or may be handled by an underlying entity (e.g. an operating system).

In some cases, the state of health is coupled between entities, e.g. error conditions in a server / host would affect the clients attached to the server / virtual machines running on the host respectively.

When standards are requested from SDOs, these aspects need to be considered and where appropriate communicated. Where multiple standards are combined, explicit verification that all required aspects are addressed is advisable as the standards may have incompatible approaches (e.g. assignment of responsibility).

5.11 Platform Independence and Portability

To ensure portability and platform independence, interfaces need to be able to accommodate various platform specific aspects, e.g. differing word sizes, endianness, packing, alignment, padding, character sets, etc. Interfaces could specify in detail which values for each aspect are supported, or employ generic mechanisms that cope with various options, e.g. one technique involves the initiator encoding its endianness in each message, with the target being required to perform byte swapping if its endianness differs. Another technique involves translation into marshalling formats where platform dependent concerns do not arise, e.g. employing text based formats like XML / JSON with specified character sets like UTF-8 avoids word length and endianness related incompatibilities.

Employing platform independent formats can impose a performance penalty. Software compatible with various underlying instruction sets could for example be encoded as byte codes instead of being distributed as native code. By translating the platform independent format into a platform dependent format (i.e. instantiating a platform dependent equivalent instance, or binding it to the platform), the performance penalty can be ameliorated. An example would be JIT compilation of byte codes. Platform independence can be maintained by treating the instantiation / binding as a cache which is invalidated when the underlying entity is changed. Whether this technique will need to be applied to improve performance should be considered when selecting the platform independent format as not all platform independent formats are amenable to this approach.

As defining interfaces in detail is the remit of SDOs, the NFV ISG would merely state which interfaces need to be inherently fully platform independent and accessible from any platform at any time vs. which interfaces only operate within a given platform and therefore need to merely have bindings for each supported platform. Performance related concerns would also be elucidated.

5.12 Level of Abstraction and Granularity of Interfaces

The overall goal of the identification and definition of abstractions and interfaces is to specify enough detail to permit different implementations to interoperate while omitting the level of detail which would require or imply specific implementation approaches. This implies that abstractions need to unambiguously define the material elements of the required behaviour of a specific component, with interfaces defining the interaction with the component.

Note that as the NFV ISG is not intending to create and specify standards, the identification and definition of abstractions and interfaces would need to be formulated as requests to other standards developing organizations to create and specify standards which cover requested behaviours and/or have requested interfaces. In order to not be overly prescriptive to the standards developing organizations, the requests need to be kept at a high level. This might be at odds with the principle of specifying enough detail to ensure interoperable implementations. The solution will be to request that the standards developing organizations add enough detail to ensure interoperability.

6 Illustrative Examples

This clause illustrates a variety of interface types and "operations" on interfaces (e.g. translation) in the context of a specific example consisting of an SDN-controlled system hosting VMs which is equipped with an accelerated network interface card. For each type of interface observable in the diagram, a brief description of the interface type or "operation" is supplied. Refer to the sub-clauses of clause 5 for additional information w.r.t. each concept.

Note that the diagram and text in this clause is NOT intended to be a depiction / description of a specific actual architecture, interface, or abstraction. Although it depicts a few subsystems relevant to the NFV problem domain in order to make the example more relevant to the audience, the intent is merely to depict a small number of interfaces and abstractions (not an exhaustive list of all relevant interfaces and abstractions) to illustrate selected concepts associated with interfaces and abstractions. This is consistent with the main aim of the present document, which is to describe the methodology of interface and abstraction creation and description, not the interfaces and abstractions themselves (refer to clause 4on page 4 for details). Consult documents like the Overview and the Domain specific documents for actual architecture diagrams that depict actual interfaces and abstractions.

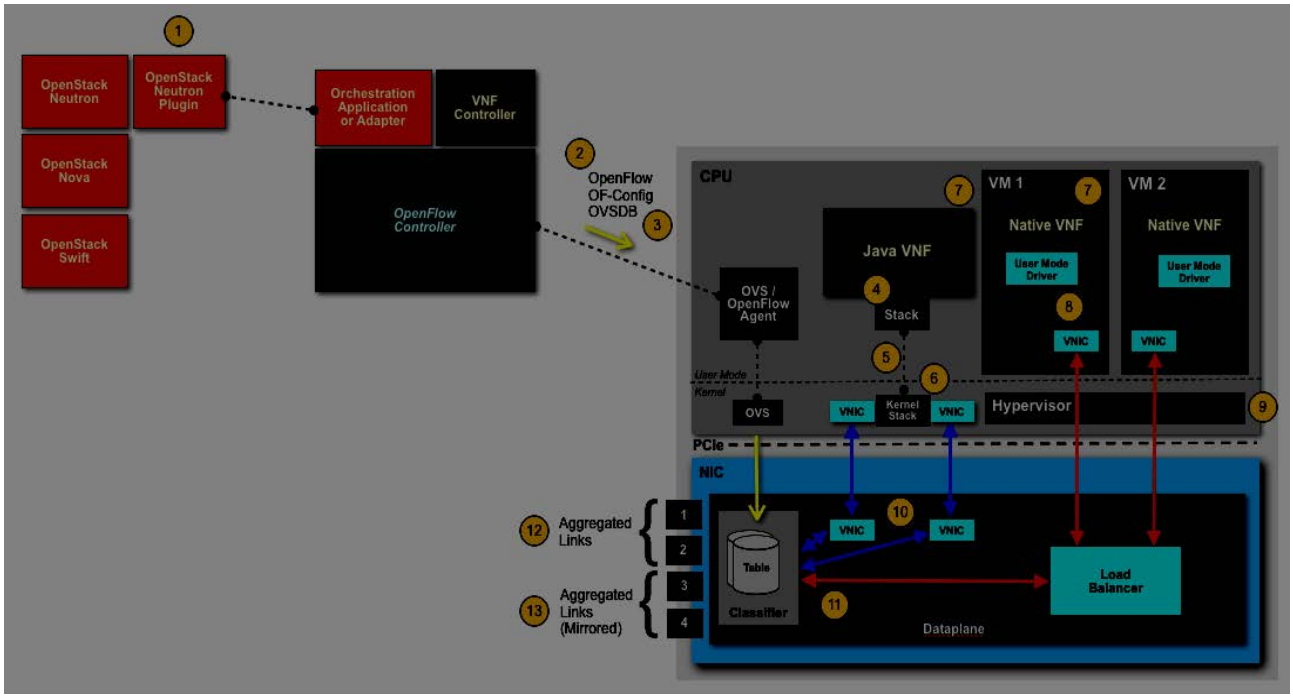

Figure 18: Selected Interfaces and Abstractions Illustrating Concepts and Methodology

The following concepts are illustrated in the example. The numbers in parentheses refer to the numbered items on the diagram.

- (2) Another SDO defining an interface / abstraction: OpenFlow™
  - Members of the ISG who are also Open Networking Foundation members would in this case influence the standard by contributing requirements via for example use cases or by directly contributing to the evolution of the standard. The official liaison channels can also be employed.
- (3) A vendor / an open source project defining an interface / abstraction: OVSDB
  - In this scenario, influencing the interface / abstraction would entail communication with the vendor or making contributions to the open source project.
- (9) Partitioning in hypervisor domain: VMs on host
  - Here the hypervisor, host OS and CPU together act as the Host Functional Block (HFB), while each virtual machine (VM) is a Virtualised Functional Block (VFB). The container interface is located between the hypervisor and the virtual machine. For more information refer to clauses 11 and 19.
- Partitioning in infrastructure network domain:
  - In this case the Network Interface Card (NIC) is partitioned by the NIC dataplane hardware / software into separate Virtual Network Interface Cards (VNICs). The Host Functional Block comprises the NIC hardware and the accompanying dataplane hardware / software hosted on the NIC, and (depending on the implementation) potentially drivers running on the CPU. The Virtualised Functional Block would comprise hardware / software on the NIC implementing the VNICs (10) or VNIC entities (partially) implemented by driver software on the host (8) depending on whether this is purely implemented on the NIC or whether this is also partially implemented using host CPU software. For more information refer to clauses 11 and 19.
  - (10) Fixed port assignment to VMs / host
    - In this case, each external port is assigned to a dedicated VNIC, which is in turn assigned to a kernel in the host operating system or a kernel in a guest operating system on a 1:1 basis. This simplified form of partitioning can be accommodated by non-intelligent hardware (e.g. most multi-port NICs).

- (11) Classification based assignment of traffic to VM / host destinations
  - Here traffic is directed to destinations within VMs or the host based on complex criteria, for example by rule based classification and/or load balancing (stateless or dynamic, e.g. considering the CPU utilization at the destination). This more dynamic / intelligent form of partitioning requires correspondingly intelligent hardware.

- (12) Bonding in infrastructure network domain: link aggregation
  - In the depicted example, links are aggregated into groups. Bonding could also be performed in other domains, for example multiple CPU cores and/or sockets could be grouped and managed as a unit, with the unit being assigned to a VM, or shared by a set of VMs. Refer to clause 19 for more information.

- (2) Protocol based interface: OpenFlow™
  - Here the interface is specified as a protocol. It is up to product vendors to develop or acquire a stack which implements the protocol. Refer to clause 18 for details.

- (4) Callable interface (API): protocol stack
  - In this case, the interface to an entity (a protocol stack) is specified as a directly callable interface. Refer to clause 18 for details.

- (1) Interface translation: SDN - Neutron adapter
  - This clause of the diagram depicts an adapter which translates between an SDN related interface and OpenStack®'s Neutron component. Depending on whether the protocol on the network is Neutron specific or more generic, the adapter can be deemed to consist of either just the Neutron plugin or the combination of the plugin and the module running on the SDN controller. This type of interface translation involves both callable interfaces (e.g. Neutron to plugin) and protocol based interfaces. Refer to clause 19 for more information.

- (5) Interface translation: user mode vs. kernel mode stack
  - Here callable interfaces are translated. The user mode stack offers a different interface to the kernel mode stack, but both interfaces can be used to interact with the network. Refer to clause 19 for more information.

- (13) Replication: traffic mirroring across links
  - In this scenario, traffic is replicated across an interface. The bearer interface is comprised of the result of bonding of other underlying interfaces. This scenario therefore also illustrates the composition of interface "shims". Refer to clause 19 for more information.

- (8) Performance of interfaces: data delivery to host / VM VNIC vs. to host / VM user mode driver
  - This scenario illustrates that different interfaces that serve roughly the same purpose may offer different performance levels. When packets are delivered to a VNIC, they need to transit the kernel stack and be forwarded to user mode, whereas should they be delivered directly to a user mode driver, higher performance levels can be achieved. Refer to clause 23 for details.

- (6) Abstraction level: network interface vs. stack (sockets) level
  - Here different abstraction levels are illustrated. The network interface (e.g. VNIC) level is able to deliver individual packets, whereas the interface offered by a stack is also able to supply reassembled data (e.g. the stream contained in a TCP connection), issuing and processing acknowledgements and handling other aspects required by the implemented protocol. Refer to clause 24 for more information.

- (7) Portability: Java vs. native code based VNFs
  - This part illustrates that interfaces can leverage cross-platform / portable technologies (e.g. Java based interfaces) or be platform specific (e.g. native code based interfaces). Refer to clause 23 for more information.

Annex A (informative): Additional Potential Illustrative Examples

In future revisions from the present document, additional examples may be selected from the following list to illustrate the interface and abstraction specification / description methodology. Note that these examples would merely be present to illustrate the methodology. Actual interfaces are specified in domain specific documents.

General

- Systems:
  - Embedded system / appliance: specific function implemented on underlying compute/storage/networking etc. platform
  - Traditional server hosting VMs running functions
  - Network switch
  - Hybrid of above
- Interactions between systems / arrangement of systems:
  - High Availability: active-passive fail over vs. active-active load sharing
  - Integrated management of virtualised compute/storage/networking
  - Central vs. distributed management and control (e.g. SDN, clouds)

Networking

- Connections and Tunnels
- Network Virtualisation and Network Overlays:
  - On-ramp / off-ramp: gateway with tunnel conversion
  - Partitioning of network adapters and switches
  - Management and provisioning of above: shims vs. built in capabilities
- SDN: Logically Centralized Control of Networking:
  - Interfaces: northbound, southbound (OpenFlow™), east-west between controllers (intra-domain vs. inter-domain)
  - Conversion between these interfaces, shims/filters, high availability etc.
- Hybrid Software Defined Networking - Traditional Networking:
  - Port based ships in night - gateway between SDN and traditional
  - Layer oriented hybrid: SDN overlay, traditional underlay
- Abstract Network Views in Software Defined and Traditional Networking:
  - Presentation of abstract views
  - Manipulation of abstract views

Storage

- Mandatory vs. discretionary access control - built in vs. as shim
- Logical Volume Management
- Disk images in files
- Distributed file systems
- Distributed databases

Compute

- Quotas: provisioning, monitoring and management
- Application level, OS level and VM level virtualisation
- VM migration - also move network traffic + ACLs
- Portability of code: interpreters vs. JIT compilation (e.g. CLR / Java)
- Portability libraries / frameworks

I/O

- Hardware based (VT / SR-IOV) vs. paravirtualised vs. VirtIO virtualisation

Security

- Network security:
  - Filtering: firewalls and intrusion detection / prevention systems
  - Usage in virtual environments: VM to protect other co-located VMs
- Host security:
  - File system filters: antivirus, access control, encryption, monitoring
  - Usage in virtual environments: VM to protect other co-located VMs
- General concerns:
  - Aggregation and abstraction: security information / event management

Annex B (informative): Authors & contributors

The following people have contributed to the present document:

Rapporteur:

Johann Tönsing - Netronome Systems, Inc.

Other contributors:

Andy Reid - British Telecommunications plc.

Calum Loudon - Metaswitch Networks Ltd.

Steven Wright - AT&T Global Network Services.

History

Document history		
V1.1.1	October 2014	Publication