- *independence with respect to target hardware configuration*, a requirement which is more an more expressed on large projects.
- *portability* for several targets, also a growing demand on large projects where identical software pieces are running on different targets and sites.
- *reusability* on frozen parts of a given application domain (reuse of high level architecture and of parts of the designs),
- *maintenability*, which is most improved when the design is easily understandable.

- Refinement of initial models. HOOD refinement simply adds more objects and details to existing ones. Refinement trading-off is made with non functional constraint implementation solutions such as existing of-the-shelf software, target constraints, as well as bottom-up, reusing components.
- Logical to physical refinement: this is rather a “restructuring” of the logical HOOD objects so as to fit into physical target constraints. For example, a grouping of the logical objects is needed when the target system is distributed over two processors. In order to avoid network communication bottlenecks, HOOD objects will be allocated to one or the other processor according to the functions they support, and at the same time trying to avoid heavy remote induced dataflows. This grouping can be formally stated through the definition of a physical model in term of HOOD Virtual Nodes, matching the target processors, and then allocating objects from the logical model to these VNs. Hence the designer gains advantages of automated code generation, that have as principle not to modify the logical code of allocated objects.

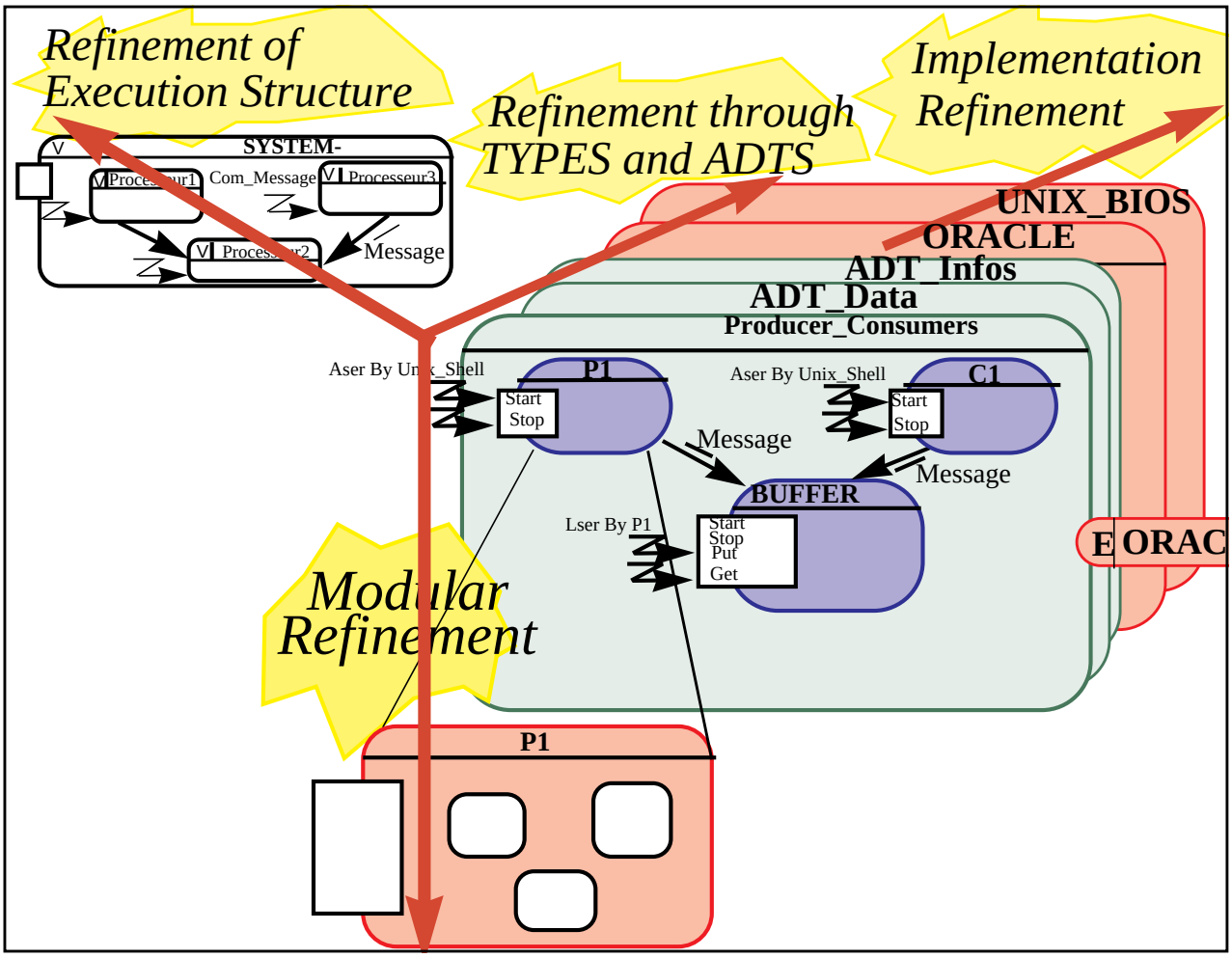

Figure 91 - The HOOD Development Model

This development approach leads to solutions supporting the following constraints:

18.5.4 Generic Architecture for Information Systems

Applying the above principles to the definition of an information system architecture, one comes to a model as the one represented in *Figure 90 - HOOD Architectural Model of a complex information system*. Four objects are associated to four development lines and partition the system into technology component objects that exchange data. The latter are again described through HOOD4 classes represented as uncle objects and where:

- the access interface to a datum (with all services provided to its clients) is formalised by the set of operations that manipulated the associated type(creation, update of a sub field, read, delete, checks, etc....) in a HOOD class,
- further refinement of classes producing classes of less complexity upto terminal specification directly rewritable in a target OO language, is based on the graphical extension and refinement rules outlined above.

18.5.5 Development Approach for Complex Systems

HOOD offers to a designer a client-server model of an application at different levels of abstraction and refinement, through parent-child decomposition, keeping consistency with initial representations of the design. Such properties are well suited in the framework of complex system developments, where successive refinements of an initial model down to the operational one can be produced with good traceability and high reuse potential.The fundamental principle behind this approach is «*it is always easier to develop a simplified system (and possibly redevelop it possibly later again) than to start a system integrating all constraints from scratch*».

Developing such an initial model with HOOD is highly efficient in that:

- it provides at hand a prototype of a solution that highlights the logical properties of the system. It is then possible to reason about the constraint implementation, instrument, and prototype them. Thus design decisions can be justified, and the testing process is more efficient.
- it provides a logical model, possibly defining a generic architecture that can be reused on similar applications

The development is thus a phased one and should enforce the following steps:

- Elaboration of an initial model or logical solution. This model is an abstraction of a functional solution structured into HOOD object hierarchies, where non functional constraints are ignored. Such a solution should be fully independent from target and non functional characteristics (languages, targets, efficiency, distribution,...)

18.5.3 Technology Component Architecture Definition

HOOD modular decomposition principles may simultaneously be used to find a modular structure in terms of components exchanging data. These components should be first identified according to the technology with which they are developed. For each level of parent/child decomposition, one applies decomposition criteria based either on allocation of functions to HOOD top-level objects or on component developed using a specific technology. Thus loosely coupled modules, with minimised provided interfaces may be defined and interfaces between the different development technologies will be highlighted. *Figure 90* - gives an example of a technology component decomposition defined for a large information system.

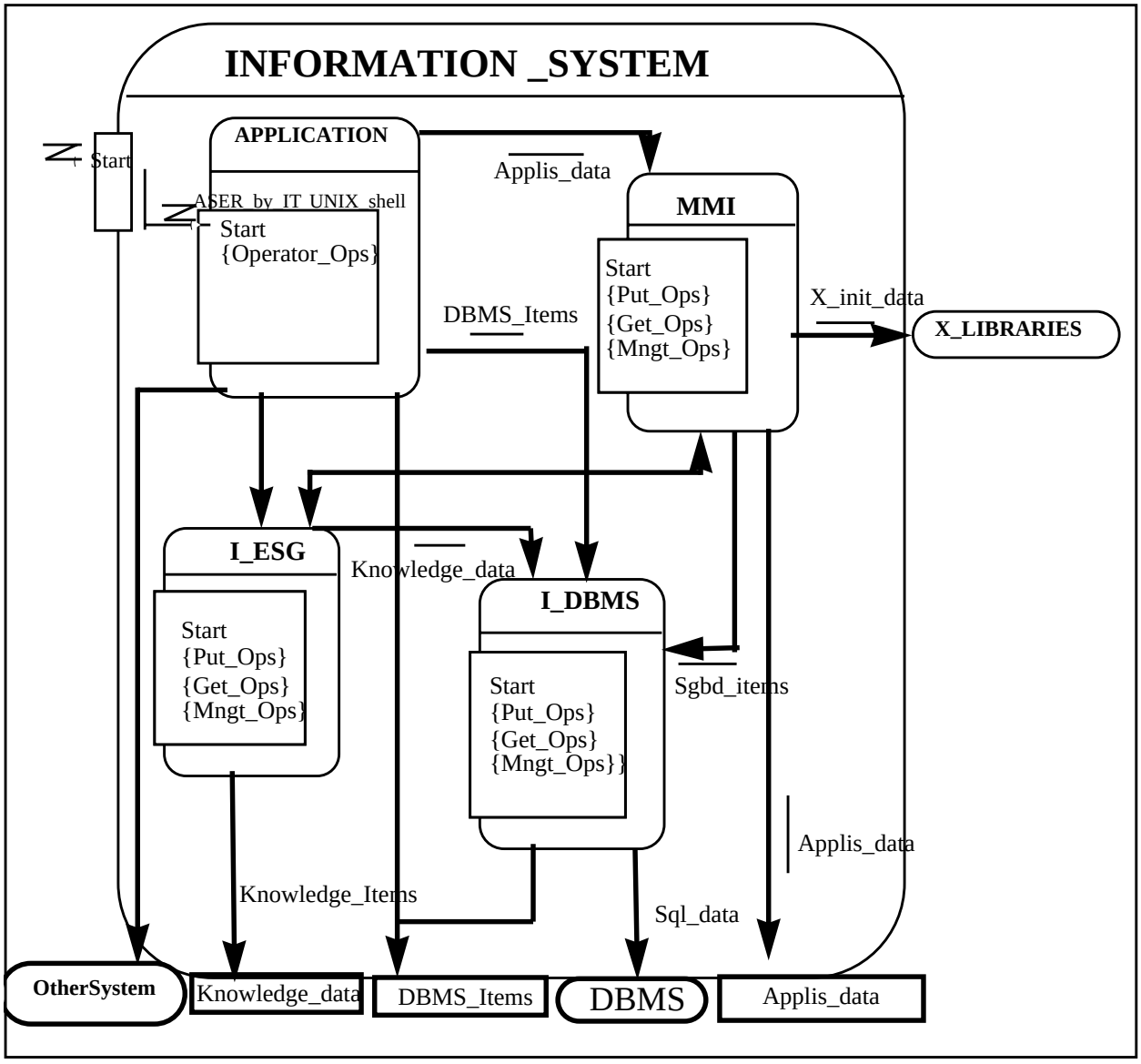

Figure 90 - HOOD Architectural Model of a complex information system

18.5.2 Implementation Refinement

This kind of refinement is as the modular refinement; we stress it here only to recall that the designer may elaborate a logical solution that relies on software layers that isolate the system-to-design from the specificities of the target infrastructure and OS. This kind of refinement is illustrated in *Figure 89* -, is one way to take into account the non-functional constraints such as:

- reuse of software architecture or components elaborated project by project in a given application domain.
- development with unclear requirement, or where full target requirement are not yet fixed at the time of the design.

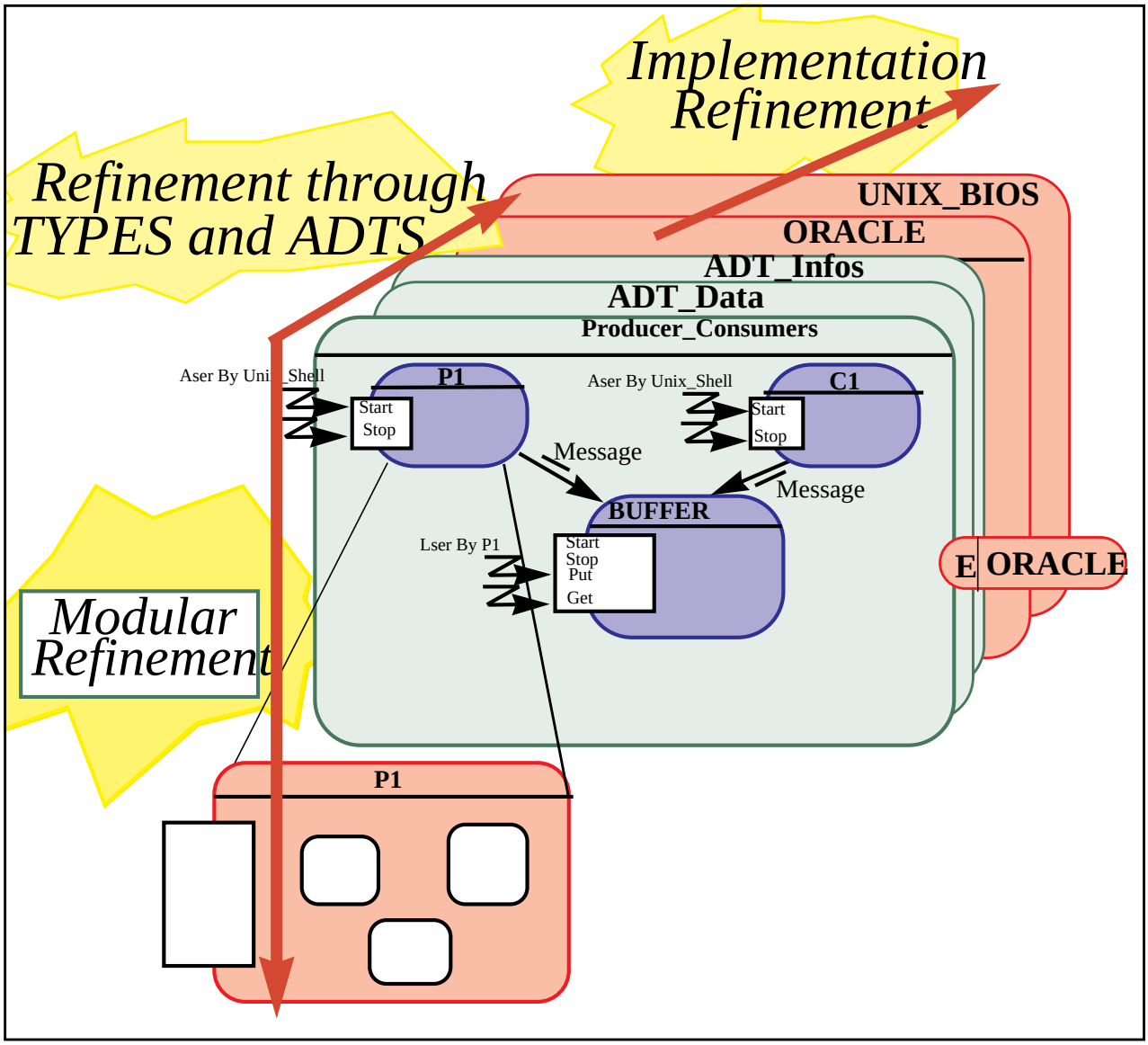

Figure 89 - Refinement Techniques of a HOOD model

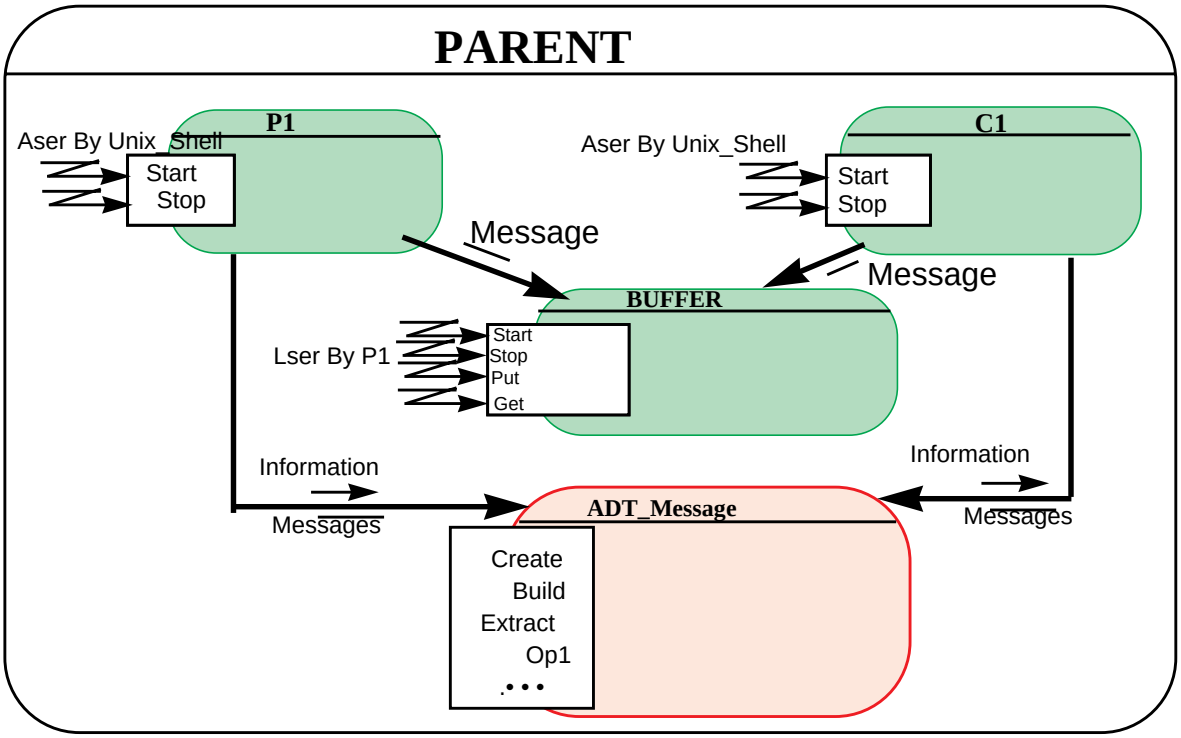

Figure 87 - ADT_MESSAGE implemented as an object of the current HDT

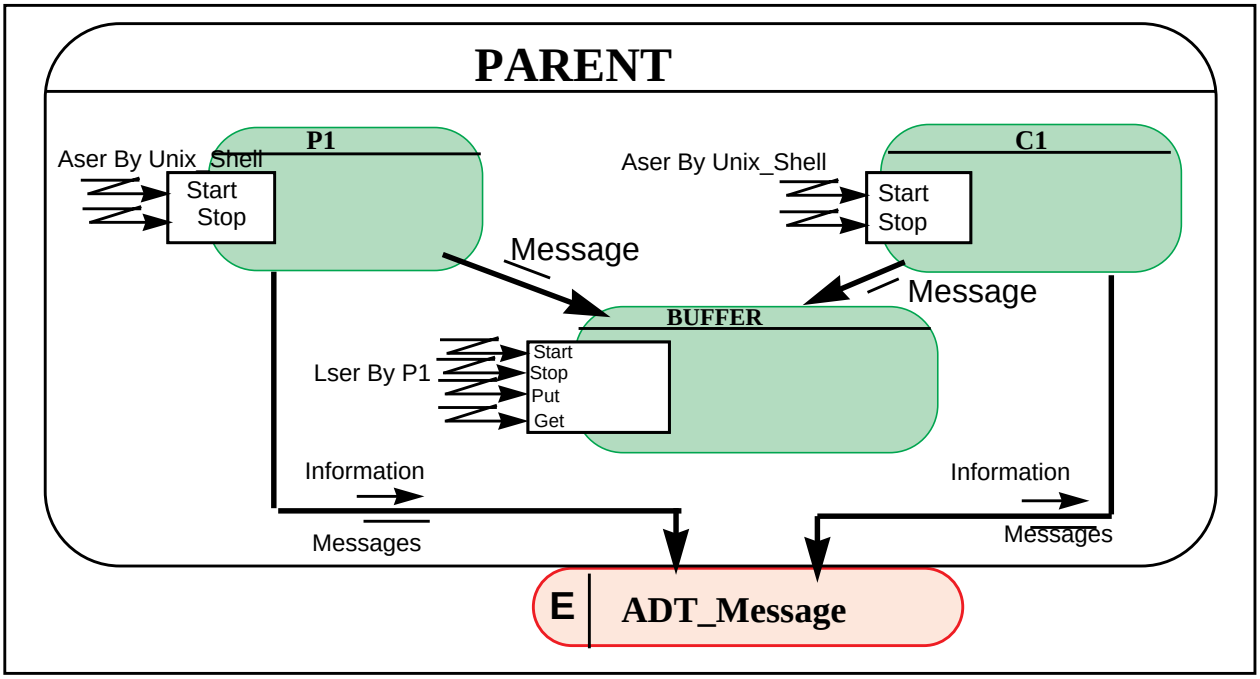

Figure 88 - ADT_MESSAGE implemented as an Environment of current HDT

When numerous classes are identified, they should be grouped and structured in HOOD object defining “class library” of logically related classes. *Figure 85* - illustrates attribute refinement for class TStack;

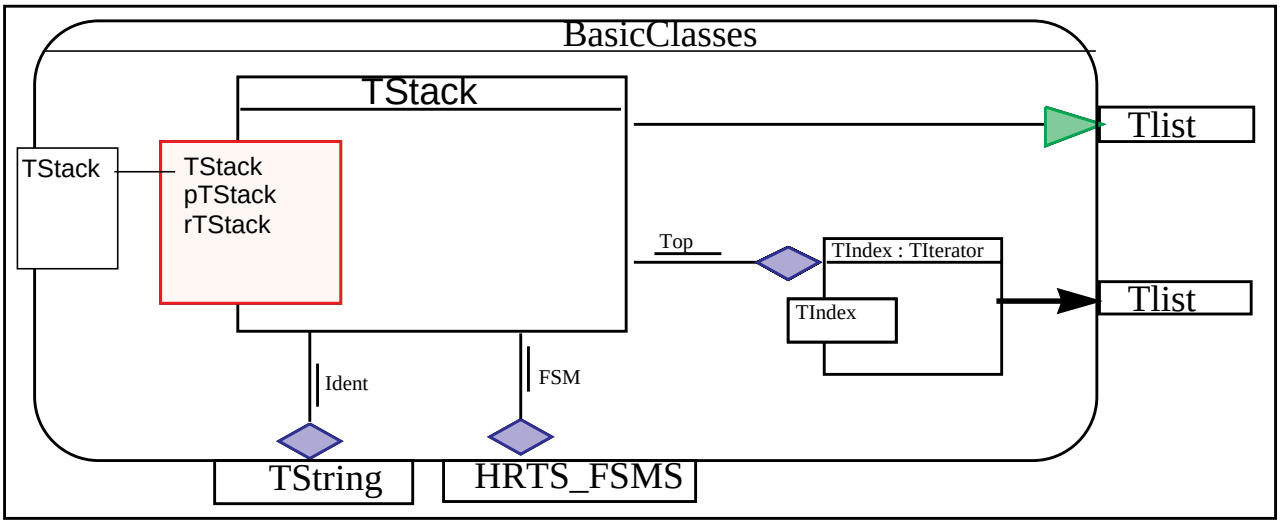

Figure 85 - Refinement through Attribution of class TSTACK

18.5.1.4 Combining ADTs with USE and INCLUDE

Figure 86 -, *Figure 87* - and *Figure 88* -illustrate how a complex data may be defined as an ADT object. The associated type is either hidden in the PARENT object or defined as new root as environment, possibly reusable from other HDTs.

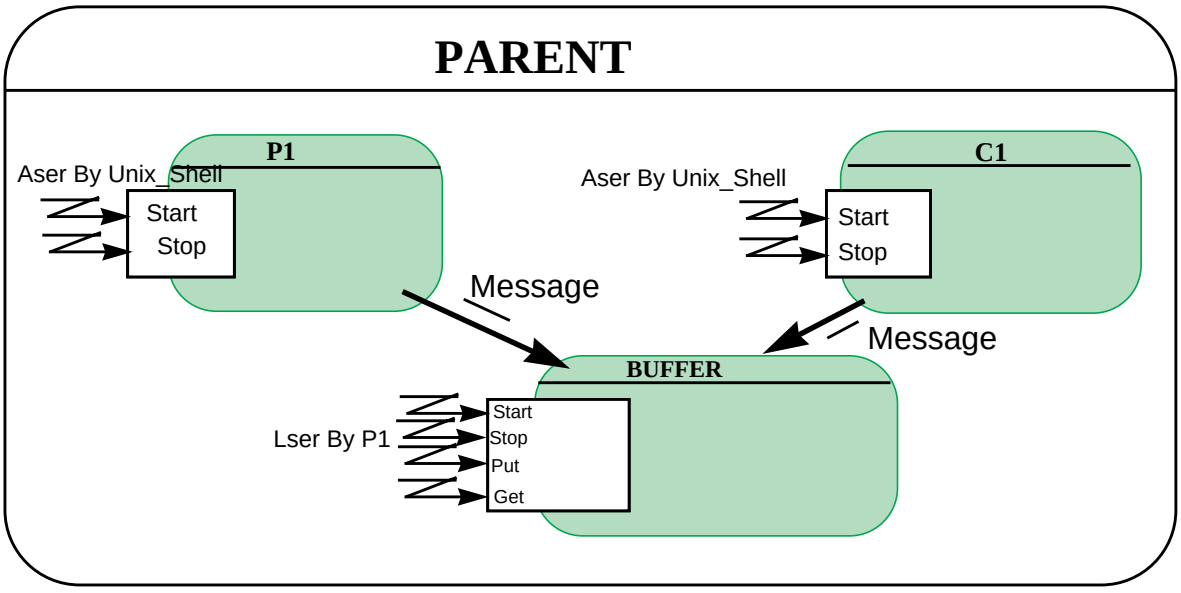

Figure 86 - Objects exchanging a complex data Message

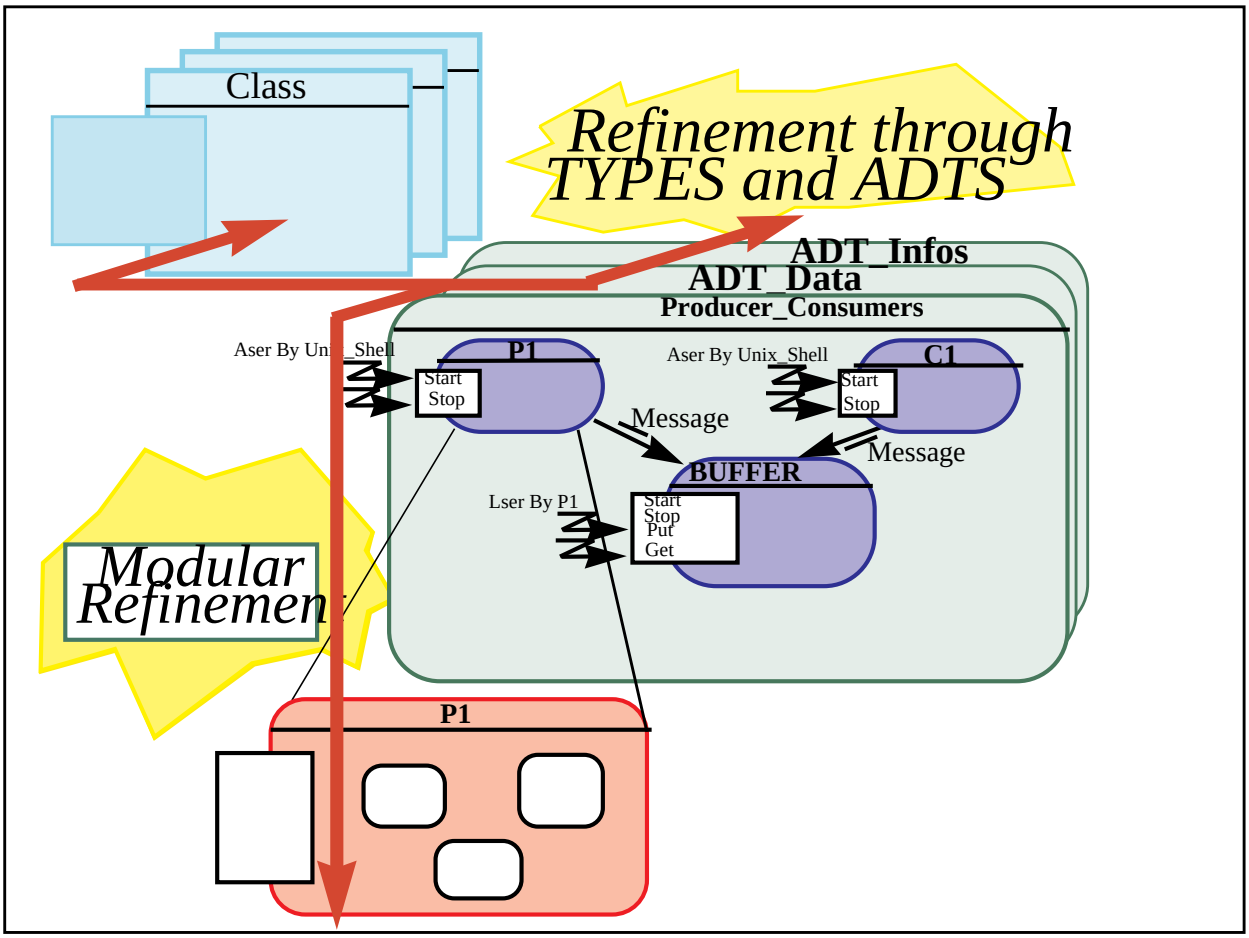

Figure 84 - Combining Modular with ADT Refinement

18.5.1.3 Class Refinement through Attribution and Inheritance

In the detailed design of HOOD object implementing an ADT three cases may show up:

- either the ADT is a complex type, which operations are groupings of operations on «sub» types, and which can thus be broken down in as many ADT objects.
- either a subset of these operations lead directly to a class identification
- either all provided operations of the ADT lead to a class definition

When a class is so identified, it is a HOOD terminal object, generally child of parent object ADT modelling a complex type. The class is, at that time, only defined by its provided operations, and when the detail design is performed the designer may:

- use attribution for defining properties and data structures common for all instances of the class
- use inheritance in order to factor attribute and operations declaration while sharing the associated code with the inherited class.

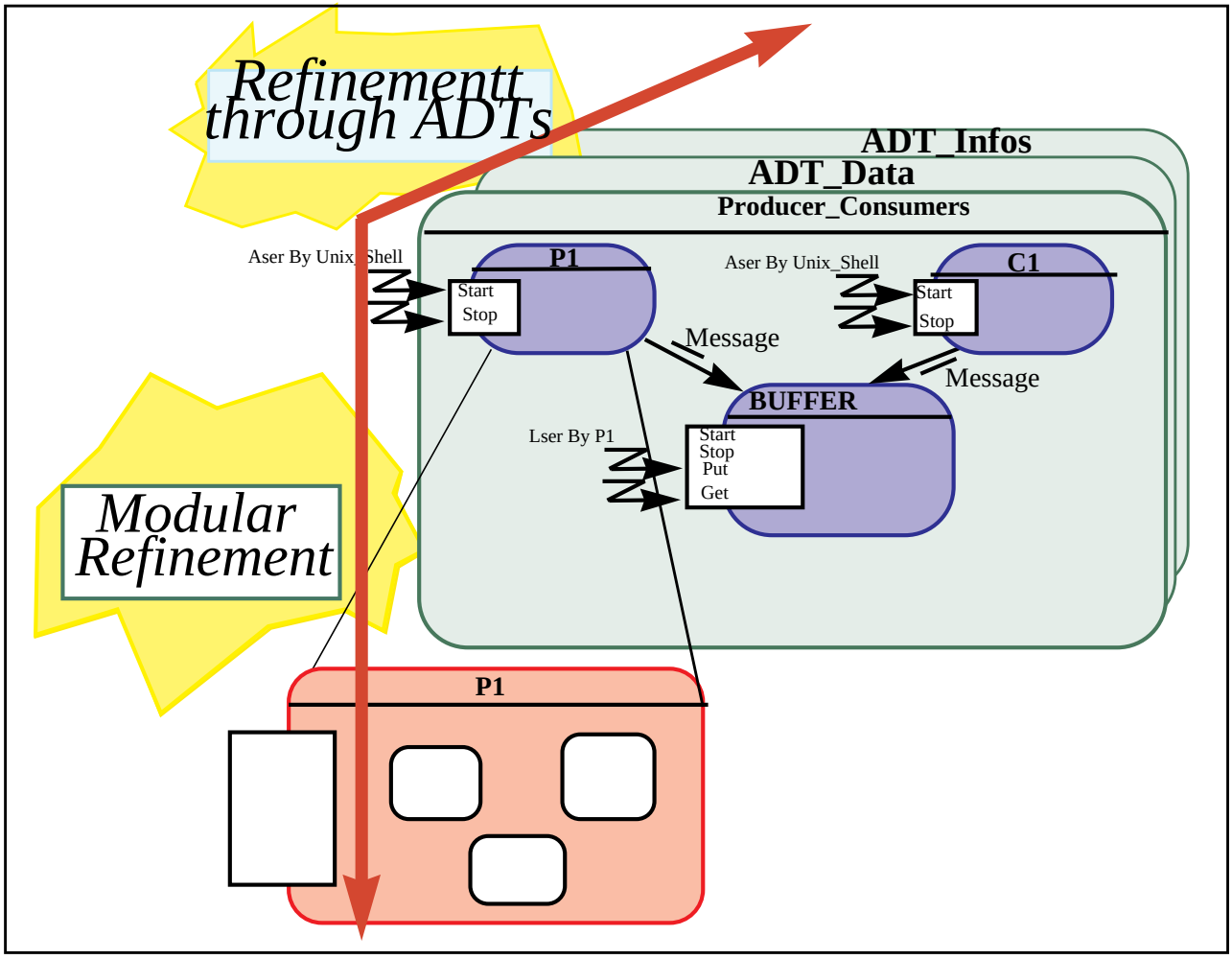

Figure 83 - Modular and ADT Refinement

An ADT implementation is defined as an encapsulation in a HOOD object of operation working on data of that type[30]. For best identification of the type on which the operation works, the receiver of the operation is indicated by the reserver parameter *me*, allowing to distinguish the main type from other parameters.

The difference between an ADT implementation and HOOD class are the following:

- an ADT implementation is not necessarily terminal and may be easily broken down into child object, possibly defining as many sub ADTs
- a class may inherit from another, or may be inherited, what is not the case of ADT implementations.
- a HOOD class is a terminal object. However a class may be refined by defining/adding attributes, or by extending existing properties through inheritance. A class allows thus, complex data structures to be defined step-wise, leaving possibly provided interface frozen with respect to client-server view

[30]. we mean the ADT

18.5.1.2 ADTs Identification and Interface Refinement

Once components and their data exchanges have been defined at one level, their interfaces can be formally expressed through ADTs (see *Figure 82* -):

- operations of OBJECTS have parameters that implement dataflows
- dataflows may be seen as “class instances”. Hence associated classes define the dataflows.

.

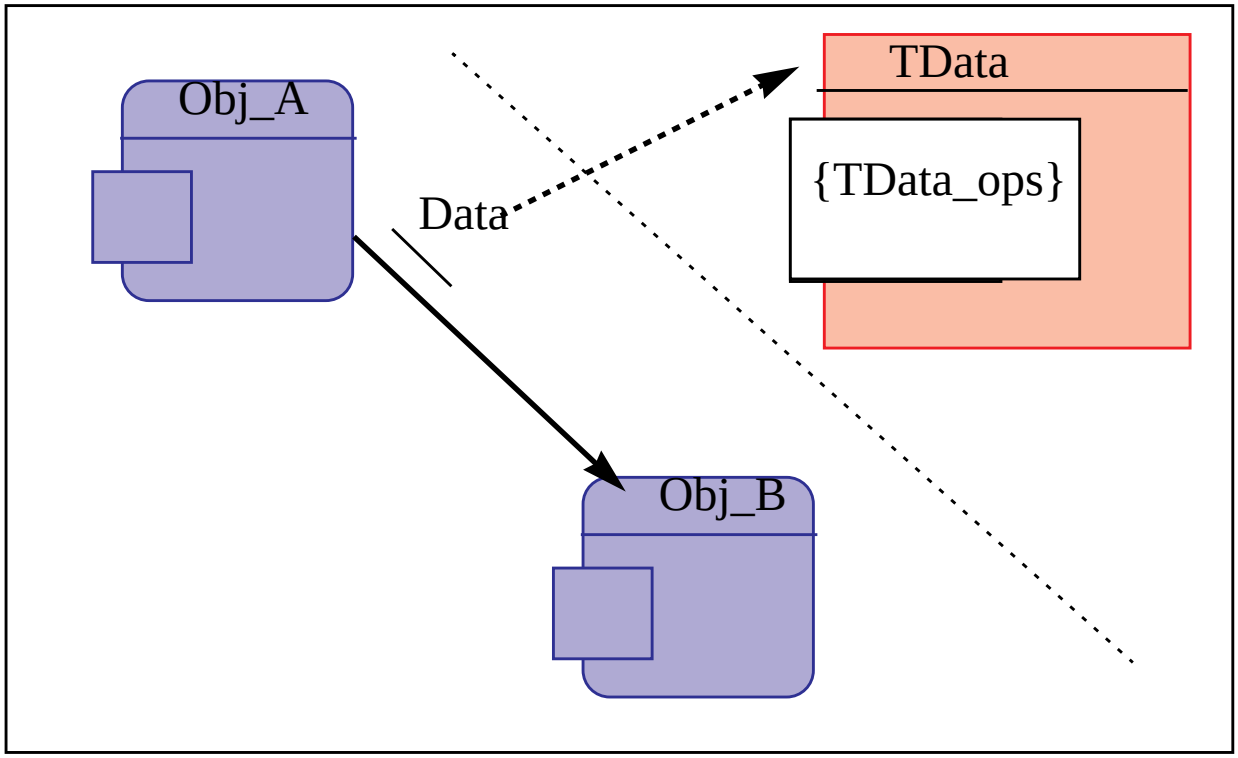

Figure 82 - Principle of specifying Interfaces through ADTs

Each dataflow identified in the decomposition may thus itself be implemented:

- either as an instance of a primitive type directly supported in the target language, or
- as an instance of an ADT (or a basic type of the target language). The operations on the data are identified as the client objects are further refined, in parallel with the modular refinement. When an ADT provided interface is fully defined, it can, in turn, be refined following a modular decomposition and/or HOOD class refinement rules. *Figure 83 - Modular and ADT Refinement* below summarizes these principles.

The interfaces between the different subsystems or software layers associated to the development activity lines can moreover be formalised as **A**bstract **D**ata **T**ypes(ADTs) that factor the exchanged data and provide for a common representation whatever the activity and technology context. The above concepts have been indeed after feedback and trials and numerous projects, built into principles and refinement techniques for integrating multiple technology developments. These principles must be applied all together within a development and list as:

- Modular and ADT Refinement Principles
- Technology Component Architecture Principles
- General Refinement Approach for Complex Systems

18.5.1 Modular and ADT Refinement Principles

18.5.1.1 Modular Refinement and BreakDown / Include Refinement

- Break down rules associated to the HOOD include relationship have as principle to hold the properties of the parent object with the ones of the child objects. *Figure 81* - illustrate this modular refinement technique of a given parent object into child objects that exchange data.

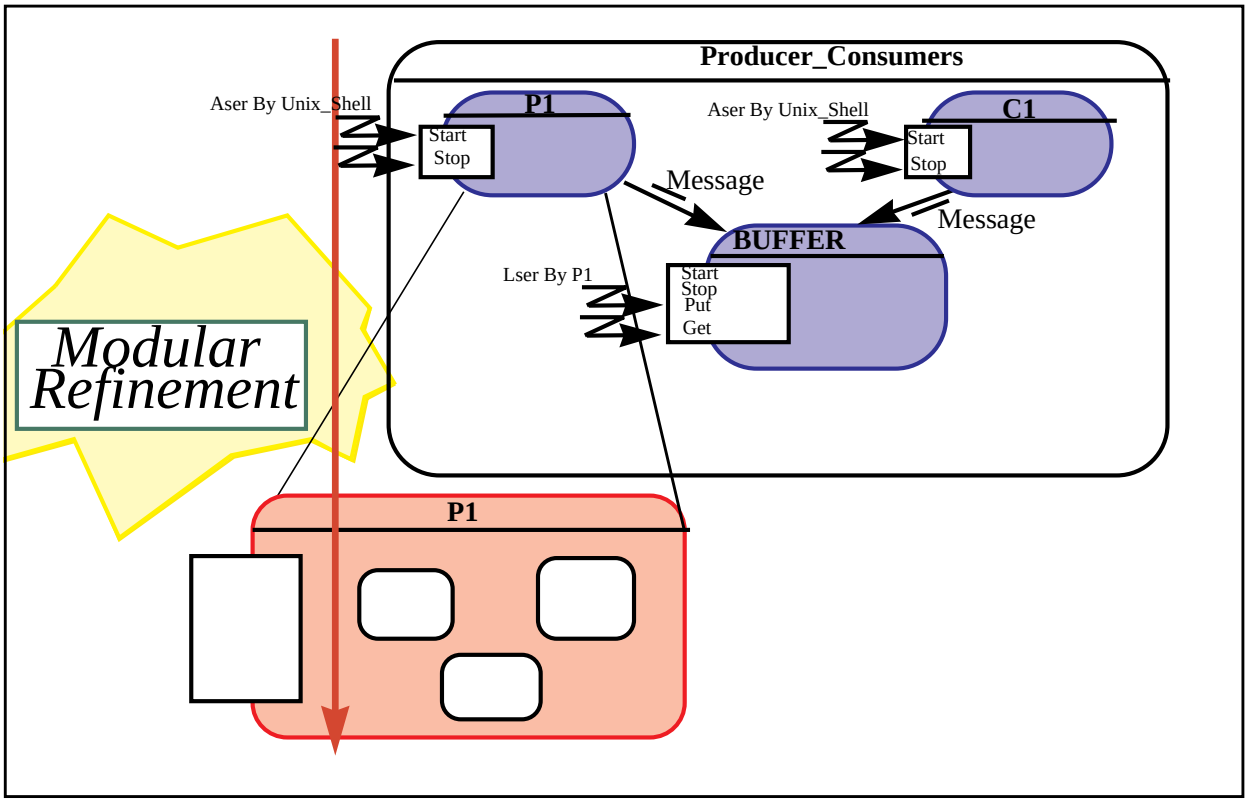

Figure 81 - HOOD Modular Refinement

18.5 DESIGN INTEGRATION APPROACH FOR MULTI-TECHNOLOGY DEVELOPMENTS

The development of complex systems involves modelling activities using intensively different supporting methods and must allow a smooth transition from:

- the conceptual models produced by the analysts,
- to the implementation models and associated constraints.

Such developments may include several lines of development activities, each of which is relying on a different methods and technical approaches, but well worked out and adapted to the specific application domain. As an example, the MMIs (Man Machine Interface) are more and more developed using UIMSs (User Interface Management System), whereas data management systems are more and more developed with the help of application generators (or 4th generation tools).

A development of such systems results thus in a set of more or less parallel lines of «technology specific» activities. The latter must be scheduled, synchronised, and interface mastered to support the associated management activities over a full life cycle framework.

The use of **HOOD, now extended with OO class support**, jumps in naturally **as the common representation formalism for the different development technologies** and brings several benefits:

- standardised external representation (defining formally the interfaces of a component) through class concepts,
- use (for the internals of the component object or class) of the best suited formalism according to the technology supported by each activity line, whereas the relationships towards the other external technologies are always captured by object and class interface descriptions,
- definition of transformation rules from the common HOOD representation formalism towards an implementation formalism allowing final code merging. (for example ODS towards an Object Oriented Language generated by an UIMS, or ODS toward Ada, or ODS towards C or assembler).

The four activities of that global process are shown in *Figure 80* - below:

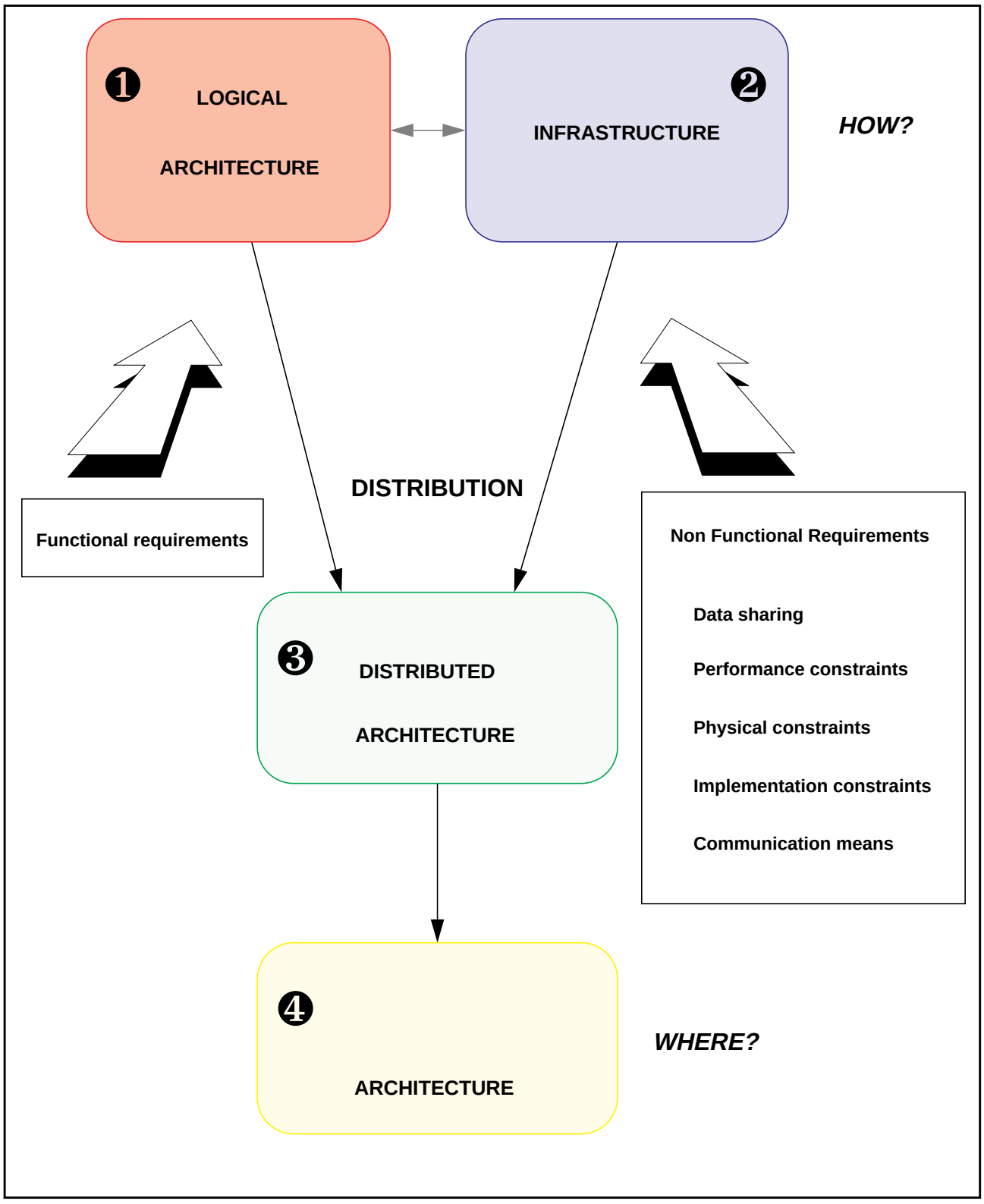

Figure 80 - Full Design Activities Overview

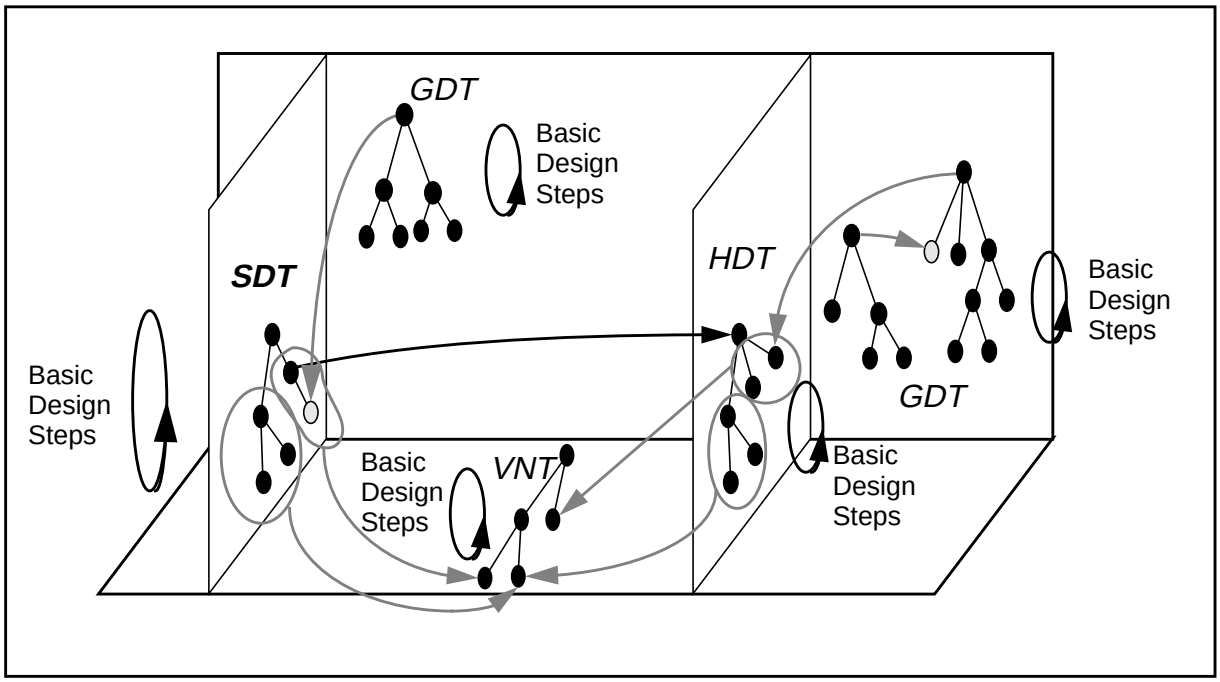

Figure 79 - Application of Basic Design Steps to the system configuration

This process allows to define the system configuration and associated elements. Associated activities may be further grouped into phases, according to the technical area of concern:

- **phase 1: Logical architecture**
  Logical is here meant as “non physical”; thus a HOOD design shall first be produced, ignoring all physical and implementation details and constraints. The principle of the approach is to produce first a solution ignoring all non-functional-constraints (*supposing we have an ideal with unlimited power reliable target*) and then rework it to add more features dedicated for the implementation of non functional constraints such as performance, reliability, distribution....
- **phase 2: Infrastructure**
  This is the basic support software (possibly also *middleware*) available for the project and which may be associated to the logical architecture e.g communication services, operating system, archiving system,...
- **phase 3: Distribution**
  These activities define the partition of the software associated to previous architectures and HDTs into hierarchies of VNs as suggested in *section 12.3*.
- **phase 4: Physical architecture**
  These activities define the partition of the VNTS def9ined in phase # onto a network of Physical Nodes (PNs) as described in *section 12.3*. PNs are interconnect through communication channels.

18.4 THE OVERALL HOOD DESIGN PROCESS

The previous basic HOOD design process allows to build a unique HDT. In order to take into account project management constraints such as subcontracting, parallel development, reuse, system design, etc... the designer has to include new principles allowing to build the whole system configuration, with the different spaces. The overall HOOD design process thus consists in (see also *Figure 80* - below):

- defining the system-to-design as an “interface” with respect to its environment

  ☞ **define a root “STD” object and environment objects**

  If one plans to **reuse** objects or classes from a previous project, they will have to be included in the system configuration

  ☞ **define the system configuration**

- performing the first basic design step (decompose the root STD into child objects)

  ☞ **update the system configuration**

  If child objects are similar or reusable,
  If child objects have to be sub-contracted, they will have to be included in the system configuration

  ☞ **update the system configuration**

- iterating basic design steps up to a level of detail enough for direct implementation and coding.

The basic design step is applied in the same way whatever the level of design. It structures the activities to be performed by the designer in a systematic approach, Thus the management procedures may be improved, allowing the distribution of design and development as well as the definition of milestones providing unique visibility over work progress.

- Preliminary Description of Child Objects Skeletons: Preliminary filling of the ODS. (some fields in the ODS are already defined by the drawing)
- Complete definition of the Parent Object.

At the end of this phase, some child ODSs are preliminary filled. The ODS of the parent object is completely and formally described. From now on, it will remain the only reliable piece of documentation for detailed design and code generation.

18.3.5 Analysis of the solution

18.3.5.1 Activity: review and justify all design decisions

This phase is the key for quality insurance. After each breakdown, one has to verify the correctness of the solution. Different activities are foreseen:

- Design Justification
- Consistency and Completeness Validation
- Identification of Re-usable Objects
- Identification of Potentially Generic Objects
- Analysis of the Dynamic Behaviour may include state transition modelling...)
- Post-Analysis Design Update possible updates in design step M and M-1
- Traceability entries: this is the right time to define which requirement the current design is fulfilling. The designer can thus define entries in a traceability matrix or directly within the ODS. Two attributes may be attached to requirements during this operation: partial implementation or full implementation
- Risk analysis in order to identify critical issues in the solution in terms of technical and management risks. For technical risks concerning failure management, detection means and recovery actions have to be studied and the solution have eventually to be updated.

18.3.5.2 Activity Outputs

- **Section H3.5: Analysis of the solution**
- This section may be a “stand-alone” document managed by the quality assurance organisation, or may be definitively attached to the ODS of the parent object under decomposition within the DESCRIPTION field of the ODS (using the ODS annotation pragma HCS - see *section F*). In that case this valuable information is immediately available in case of later reuse of this object/solution.

- **Section H3.1: Identification of objects and classes**. The designer expresses, either from the strategy text or from operation description text, how each child objects works with the others and what they do, what functions they embed. The result of this phase is a textual description of significant objects in the solution; the child objects.
- **Section H3.2: Identification of operations.** The designer identifies all operations, the objects on which or from whose they can be operated, and gives and gives for each one a textual description. He may point out all attributes relative to parallelism, synchronism, periodic execution.The result of this phase is for each child object, a textual description of the operations it provides to its users.
- **Section H3.4: Graphical description**. Using the HOOD graphical formalism, the designer produces a graphical description completed with most relevant data and exception flows. It is relevant to stress that the graphical description is only an abstraction of the textual descriptions: as a result not everything should be shown in the diagram, but only the most relevant information easing the understanding of the architecture. On the other hand, the consistency between these two kind of representations shall always be ensured.

18.3.4 Formalisation of the solution

18.3.4.1 Activity: Formalise the reviewed solution in the ODSs

The goal of this phase is to achieve a description of the solution (whose strategy could be refined and approved as the result of above activities) with the detailed characteristics of the objects being formalised in the ODS fields. The capture of this formal description consists in filling each field of the Object Description Skeleton (see *section 14* above).

18.3.4.2 Activity Outputs: ODSs

The way the ODS are completed will depend of the functionalities of the used toolset. In the worst case only a textual ODS editor is available and the designer has to complete the fields of an ODS skeleton text. However a suitable HOOD toolset provides facilities that help and automate the filling of the ODS fields that were already created as the graphical description was elaborated. Some toolsets even generate automatically some fields (REQUIRED INTERFACE) as they analyse relevant fields of the ODS.

Two kinds of formalisation have to be done:

- analysis of non functional constraints[28]
- user manual outline

18.3.2 Elaboration of an Informal Solution Strategy

18.3.2.1 Activity: Refine and work out a solution

This phase has as goals the expression of a solution that the designer should have started to imagine from the early phases of the H1 activities. The designer shall identify the main abstractions and actions and give a scenario of solution accordingly. Such a strategy shall be expressed in natural language in order to explicit clearly how the solution will work, without requiring forward references (e.g how a child object will do its work). This strategy is a starting solution and may be refined as the solution is refined, in parallel, as H3 activities are performed[29].

18.3.2.2 Activity Outputs

- **Section H2:** A clear and concise text explicating the solution according to the graphical description elaborated during H3 activities.

18.3.3 Formalisation of the strategy

18.3.3.1 Activity: Refine and work out the selected solution outline

This phase has as goals the extraction of the major concepts of a solution strategy in order to come smoothly to a formalised description of the solution. The designer refines the strategy by producing textual descriptions of all relevant objects and operations. The idea is that a solution which can be expressed clearly in natural language is an already mastered solution.

18.3.3.2 Activity Outputs

A set of textual descriptions as well as a HOOD graphical description with justification of the design decisions.

[28]. these are all constraints related to the particular target and context of the project (performances, reliability, distribution, maintenability, etc.)

[29]. hence if H2 activity starts always before H3 ones, it may finish after H3 ones.

18.3.1 Problem definition

18.3.1.1 Activity: Understand the problem to solve before jumping on a solution

The goal of this phase is to integrate all facets of the problem, before devising a solution. Recommendations here are to make the designer state himself the problem, and to analyse and restructure the requirements with respect to his own designers perception.

- **Statement of the problem**The designer states the problem in one correct sentence giving a clear and precise definition of the problem as well as the context of the system to design.
- **Analysis and restructure of requirement data**The designer gathers, analyses and organises all the information relevant to his problem, *clarifying all points which are not yet clear.* He possibly defines the software environment of the system-to-design. He classifies the requirements into *functional, behavioural and non-functional* ones, and performs *design sensitive analysis* upon them. He possibly produces a user manual outline of the system-to-design.

NB: It is here that the transition between requirement analysis (description of the WHAT), and the design (description of the HOW) is made.

18.3.1.2 Activity Outputs

To support the above activities, the following text sections (with suggested name Hij[24]) shall be produced:

- **Section H1.1: Statement of the problem** comprising a description of the problem and its context in a few sentences
- **Section H1.2: Analysis and restructure of requirement data** comprising, if needed, the following subsections:
  - analysis and definition of the object environment
  - analysis of functional constraints[25]
  - analysis of behavioural constraints[26]
  - analysis of data model constraints[27]

[24]. for HOOD Activity i, subsection j. The way the various sections of each design step are set together within the design process for a project is out of the scope of this manual. However these text sections provide the basic material submitted to verification in the context of a basic design step, and could be captured in the DESCRIPTION field of the ODS of the parent object under decomposition, using the pragma HCS (see *section F*).

[25]. Functional constraints are constraints not related to any implementation; they are just a result of the pure functional analysis

[26]. behavioural constraints are mostly analysed, expressed using state transition models

[27]. data model constraints define all relationships between data identified at analysis level.

18.3 THE BASIC HOOD DESIGN PROCESS

The basic HOOD design process consists in building a HOOD Design Tree. It is globally top-down. The system to design is first defined as a high-level object called *root object*, and then is broken down into several lower level objects or classes up to they can be directly implemented by target language units and environment services.

As seen above, the system to design can be represented by a *HOOD design tree* where branches represent parent objects broken in children and leaves represent terminal objects, which are no more refined into child objects.

The process of breaking down one object into child objects is called the **basic design step**. Thus, it is necessary to perform a succession of basic design steps to build a full HDT.

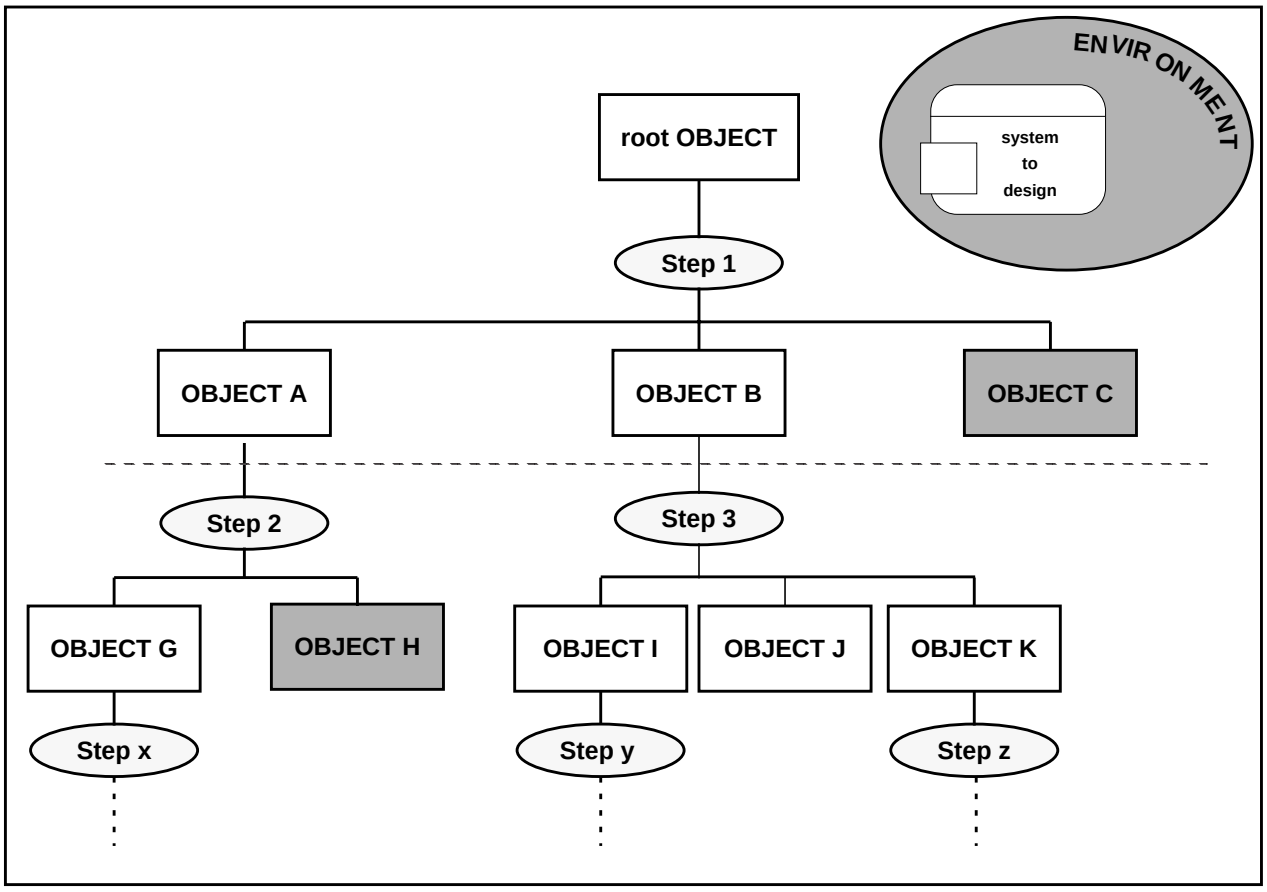

Figure 78 - HOOD Design Tree as broken down from the ROOT object

The **HOOD Design Process** with a detailed description of each activity is included in the HOOD User Manual *[HUM96]*. In the following we recall these activities and associated documentation for a basic design step.

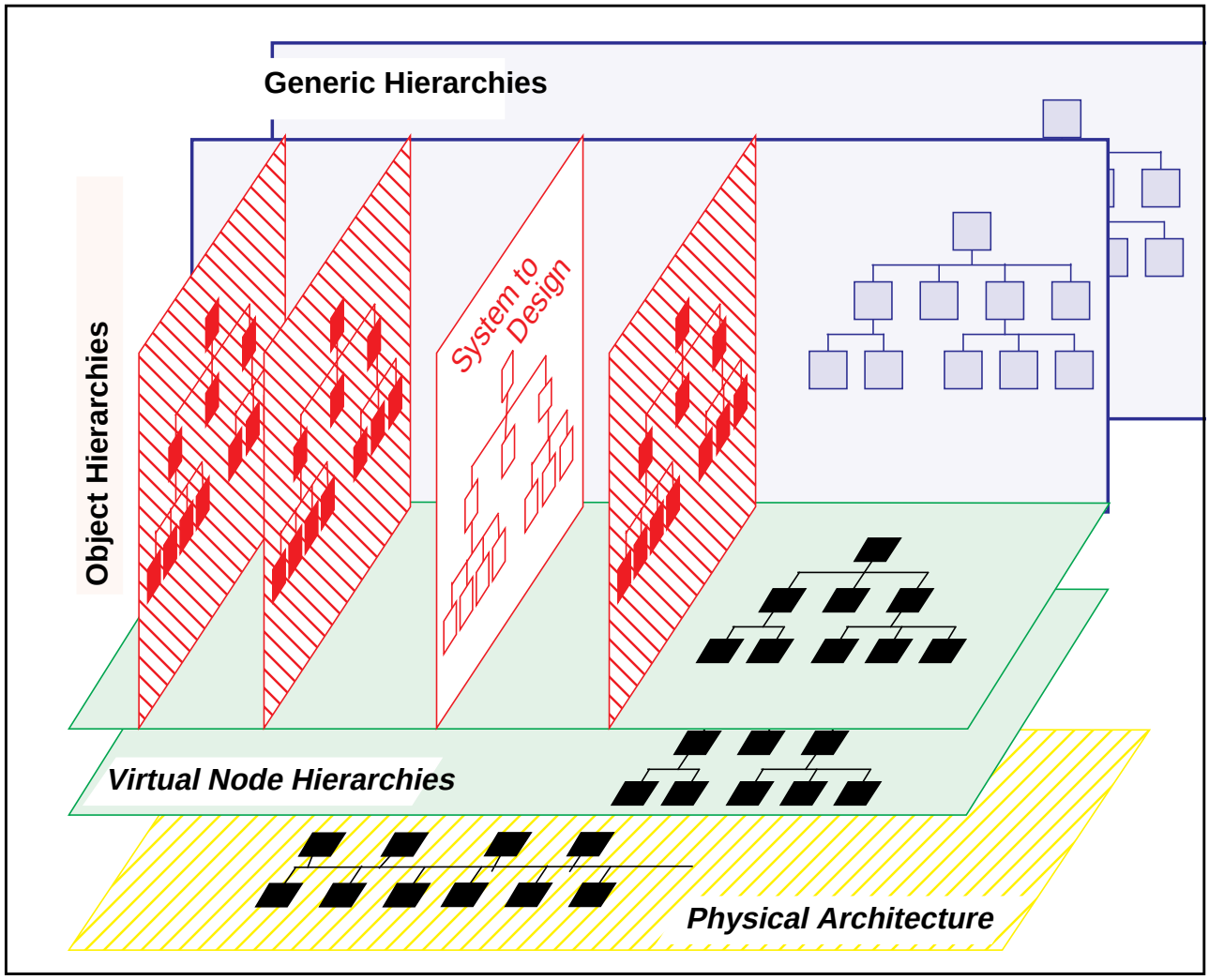

Figure 77 - The HOOD Design Model as a set of spaces and hierarchies.

The HOOD method proposes a process to guide the designer to create a design what ever the size and the complexity of the project. Two different processes are imbricated:

- **a general process driving the development approach, called "Overall HOOD Design Process"** in the following**.** This process describes the approach and activities to perform along the architectural design phase in order to organise the system according to the development constraints of the project (sub-contracting for example). It allows to define several models of the system down to the system configuration. This approach is more detailed in *section 18.4* below,
- **a basic HOOD Design Process** which defines the activities to break down a given object into children, from a top object to all terminal objects, and to get thus a a Design Tree (either HDT or GDT). This process is applicable for all objects and generics and is detailed in *section 18.3* below.

A whole project organisation may thus be defined with such a HOOD design model (see *Figure 77 - The HOOD Design Model as a set of spaces and hierarchies.*) as:

- A system to design hierarchy (HDT associated to the STD), which is an emphasised part of the object space and which defines orthogonal client-server and composition relationships between objects. Instantiation links to generic objects may appear in this HDT as some objects may be defined as parameterized instances of templates.
- A set of HDTs appearing as environment to the current STD, and which are part of the object space and allow to describe (formally) in an homogeneous framework and notation, external used entities from the STD.
- A set of generic hierarchies or GDTs, which are part of the generic space and which define parameterized reusable structures and templates, organised and structured by client-server and include relationships as defined above. The generic space includes all generics which are instantiated within the HDTs (STD or environment) of the object space.
- A set of VN hierarchies or VNTs, which are part of the Virtual Node space and which define the potential granularity for partitioning onto a physical architecture of processing nodes (either heavyweight processes or CPUs)
- The Physical architecture on which the VNs are allocated.

The scope of a HOOD design model is controlled through the concept of *System Configuration* which defines all root of hierarchies describing the system (Object hierarchies, Generic hierarchies and Virtual Nodes hierarchies).

A system to design is defined in a HOOD model as a set of root objects (some of which include HOOD classes), and generics. This is formalised by the **system configuration** illustrated in *Figure 76* -which lists all root Objects, Generics and VNs **in the scope** of a given system to design defined as a root of a HDT.

SYSTEM_CONFIGURATION
ROOT OBJECTS
List of root objects
ROOT GENERICS
List of generic root object or generic root lasses
ROOT VNs
List of VNs
END

Figure 76 - SYSTEM_CONFIGURATION Definition

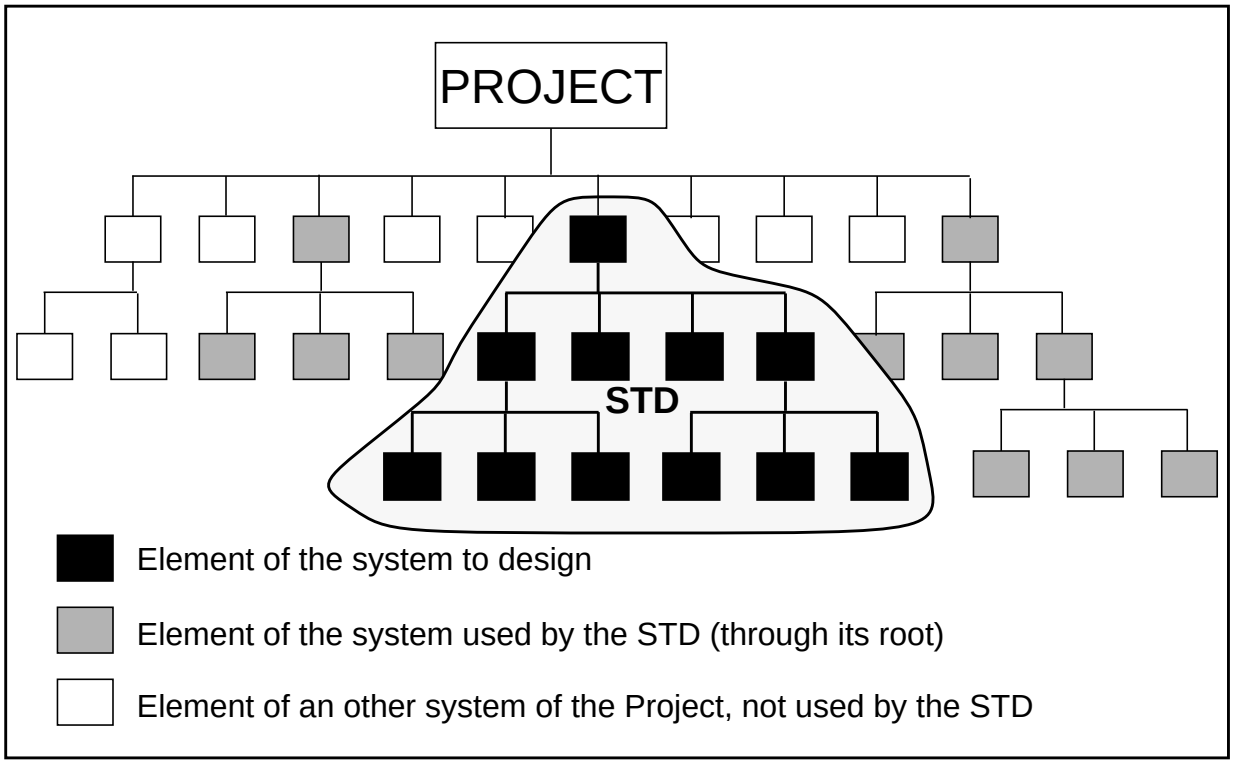

Figure 75 - A System To Design within a whole project[23]

18.2 SYSTEM CONFIGURATION

In order to manage HDT through interface descriptions still consistent with their environment, a HDT is defined within an *object space* comprising the current system to design and associated *environment hierarchies* of objects.

Generics define templates of objects. Generics are not objects that can be directly used. Thus, the generic hierarchies Generic Design Tree or GDT) may be defined separately from the object space. The GDT are described in the *generic object space*.

For similar reasons, the Virtual Node hierarchies (VNT) are defined in a separated *Virtual Node space* allowing to define distributable entities, memory partitions *or* heavyweight processes as defined in *section 12*

Finally, virtual nodes have to be allocated to physical nodes of the physical architecture. This architecture is defined in a *Physical space*.

[23]. Client/server use relationships are not shown

18 THE HOOD DESIGN PROCESS

The HOOD method has worked out a framework for the project design and organisation. Principles of that organisation are explained in detail in *section 18.1* and *section 18.2*below, whereas the HOOD design process is described in details in *section 18.3* and *section 18.4*.

18.1 SYSTEM TO DESIGN

The System To Design (STD) is the System (in the broad sense) which have to be designed from a specification. It may designate a part of, or a whole project.

A HOOD **System To Design** (STD) may be first seen as an object defined by its interface with its environment. The environment is considered as an other part of the project, not designed for the purpose of the current system, and for which the root is seen as an environment object from the STD. This root object is then broken down into child objects, which can themselves be further broken down (until the specifications of child objects can be directly implemented in the target language). A STD can thus be represented as a root object of a HOOD *design tree* (HDT) where branches represent parent objects broken down into child objects, and where leaves are terminal objects which are no more broken down.

Figure 75 - hereafter represents a STD as a root of a hierarchy of objects within a whole project including other hierarchies (other HDTs).

Thus, a project is defined by a set of root objects, each of them being considered in turn as the STD **by a development team**, all others appearing as potential environment objects for this particular system.

CppG.22 A formal type parameter is mapped to a class parameter of the template

CppG.23 A formal exception parameter is mapped to a data parameter of the template matching the type HRST_PE.TExceptions

CppG.24 A formal constant parameter of a generic is mapped to a data parameter of the template.

CppG.25 A formal operation parameter of a generic is mapped to a data parameter of the template.

CppG.26 An Instance of a Generics is mapped into a class instance of the template class that implements the generic.

Optional rules shall specify the mechanism for assuring the visibility of the class from the instantiation context. By default, visibility is ensured by include of the module header that implement the generic

CppG.27 An Instance Range of a generics is mapped into several instances of the generic, with instance_name defined by the concatenation of the instance object name and the string associated to each value of the range.

17.7 OTHER TARGETS

In other languages, one looks for preserving the object oriented architecture down to the target system. HOOD code generation principles (as defined in *section 17.1*) shall be applied, with possibly adaptations of the HRTS modules. Modularity and visibility mechanisms may have to be simulated by inclusion of files such as in C, or in primitive assemblers.

illustrated in Appendix *I.2.2"OSTM code illustration in C++"* and visible from the OPCS_BODY code

- the ObcsServer field[21] shall be mapped as code the body of the <NAME>_RB module.
- the ObcsClient field[22] shall be mapped as code in the body of <NAME> module.

CppX EXCEPTIONS

CppX.1 An exception declaration, is mapped in an instance of the class HRTS_PE.TExceptions. The constructor of this class allocates a unique exception id.

CppX.2 An exception provided by a parent object, is mapped into:
- a C++ comment if it is implemented_by a exception of same name.
- the same C++ object as the child exception otherwise.

CppX.3 The RAISED_BY clause may be translated into a comment.
Optional rules shall specify the conditions for no generation for such comments.

CppIh INHERITANCE and ATTRIBUTION

CppIh.1 An Instance of a Class is mapped into a variable or constant of the type defining the class, in the module body associated with the object where the instance is defined.

CppGr GENERICS

CppG.19 A Generic is mapped into a module of same name. A generic HOOD class or object is mapped into a «template Class» named as the Generic.
The formal parameters of a Generic are mapped into the parameters of the associated template Class. An additional parameter named «instance_name» and of *type HRTS_PE.TString* is generated to contain the name of the instance of the generic.

CppG.20 A class or an object included in a generic and requiring formal parameters is mapped to a template class, and its formal parameters are mapped to the parameters of the template class. An additional parameter named «instance_name» and of *type HRTS_PE.TString* is generated to contain the name of the instance of the generic.

CppG.21 classes and objects included in generics and not requiring formal parameters are generated as standard classes or objects.

[21]. when not empty
[22]. when not empty

OPCS_FOOTER part.

CppO.5 A function body associated to a state constrained and/or concurency constrained operation shall comprise at least an OPCS_HEADER part and an OPCS_BODY PART.

Optional rules shall specify the conditions for suppressing calls to mutual exclusion semaphores protecting object's state access.

CppO.6 An internal operation is mapped into the declaration of a C++ function in the module body associated with the object.

CppO.7 A terminal object or class of name <NAME> with protocol constraints operations shall be mapped into three C++ modules, named <NAME>, <NAME>_RB and <NAME>_SERVER
-C++ functions associated to protocol constrained operations comprising only OPCS_ER code will be generated in the module of name <NAME>.
-function bodies comprising only OPCS_SER code will be generated in the module of name <NAME>_RB.
- C++ functions associated to protocol constrained operations comprising non_empty OPCS_HEADER part, OPCS_BODY PART and possibly a OPCS_FOOTER part code will be generated in the module of name <NAME>_SERVER

Optional rules shall specify the mechanism used for giving visibility (include of header files or nesting within the module) to the different modules according to the used operations.By default visibility is ensured by use of inclusion of required module header files.

CppO.8 The OPCS of an operation of a terminal object is mapped as follows :
- all sections but code sections may be mapped into comments.
- the code section is mapped into a function body (as OPCS_BODY part).

CppO.9 The OBCS of a non terminal object is mapped as follows :
- the visible obcs section may be mapped into a comment in the package specification associated with the object;
- the internal obcs section may be mapped into a comment in the module header associated with the object;

CppO.10 The OBCS of a terminal object is mapped as follows :
- the visible obcs section may be mapped into a comment in the package specification associated with the object;
- the internal obcs section is mapped into:
  - all sections (except code sections) may be mapped into a comment in the module body associated with object.
  - the OSTM shall be mapped into a class of name <NAME>_OSTM> as

CppC CONSTANTS

CppC.1 A constant provided by a terminal object, is mapped into a public C++ constant object in the module header associated to the terminal object. (static const data member). If the constant is fully described in the INTERNALS, the target dependent description of the constant is inserted in the module header. If the value of the constant is given, the constant is also initialized with the target dependent value.

CppC.2 An internal constant is mapped into a public C++ constant object in the module body associated to the terminal object. If the constant is fully described, the target dependent description is inserted in the module body,; if only the value is given, the constant is generated and initialized with the target dependent value.

CppC.3 A constant provided by a parent object, is mapped into the declaration of a C++ constant object (static const data member) in the module header of the parent object and in the initialization of this data in the module body with a value equal to the name of the constant that implements it in a child object.

CppD DATA

CppD.1 A HOOD data is mapped into a C++ object declared as private members in the module header associated to the HOOD object.

Optional rules shall specify the conditions for declaring such a data as C++ static object.

CppO OPERATIONS

CppO1 An operation provided by a parent object, is mapped into:
- a C++ comment if it is implemented_by an operation of same name.
- an inline C++ function calling child operation otherwise.

CppO.2 An unconstrained provided operation of a terminal object is mapped into the declaration of a C++ function in the module header associated to the terminal object and in a function body in the associated module body. It is mapped into:
- a C++ Class operation it has a receiver parameter *me*
- a C++ static Class operation it has a receiver parameter *myClass*
- a C++ function if it has no parameter me.

CppO.3 A function body associated to an unconstrained operation shall only comprise an OPCS_BODY parts the target dependent field of the CODE section of the ODS.

CppO.4 A function body associated to a constrained operation shall comprise non_empty OPCS_HEADER part, OPCS_BODY PART and possibly a

Optional rules shall specify the conditions for no generation for such comments.

CppG.7 Required types, constants, operations and exceptions are mapped into comments.
Optional rules shall specify the conditions for no generation for such comments.

CppG.8 The internals part of a terminal object is mapped into comments and C++ declarations in the module body that implements the object. Internal Types, Constants and Data shall be mapped as C++ a declarations in the module body associated to the object.

CppV VISIBILITY

CppV.1 A C++ module implementing an object shall have visibility on the modules that implement the required objects. Such a visibility shall only be given to the module header if references to a required object appear in the provided interface.

Optional rules shall specify the mechanism used for giving such a visibility (include of header files or nesting within the module). By default visibility is ensured by use of inclusion of required module header files.

CppV.2 A required Environment Object is mapped into an include directive of the module header that implements the required object.

CppV.3 The modules implementing the objects required by an OP_CONTROL object shall be visible from its module body.

CppT TYPES

CppT.1 A type provided by a terminal object, is mapped into a public type member declaration in the C++ class related to the terminal within the module header associated to the object.

CppT.2 A internal type of a terminal object is mapped into a private type member declaration in the C++ class related to the terminal within the module header associated to the object

CppT.3 A type provided by a parent object, is mapped into a public type member declaration in the C++ class related to the parent object within the module header associated to the parent object. This member declaration renames the implementing child's type.

The extraction rules are partitioned into two classes : a kernel set which defines standard rules and an optional set which may be triggered by associated pragmas.

17.6.2 Kernel Extraction Rules

In the following, kernel extraction rules are listed. Additional rules are introduced by a sentence starting with *"Optional rules shall specify..."*.

CppG.0 General Rules

CppG.1 An object is mapped into a module named as the object. The object provided and required interface is mapped into C++ declarations in the module header file. The object internals are mapped into comments, preprocessor directives and declarations in the module body file.

Optional rules shall specify the conditions for generating an object as a C++ class or merely C++ declarations. By default an object mapped into a C++ class[20] named as the object.

Optional rules shall specify the conditions for the generation of nested modules. By default child objects are mapped into non nested modules.

Optional rules shall allow to modify the extension of header and body file associated to a C++ module in order to match C++ development environments implementations.

CppG.2 An OP_CONTROL object is mapped into a function named as the object, and nested in the module body file associated to the parent object.

CppG.3 A HOOD Class is mapped into a C++ class, exporting the provided interface of the HOOD class.

Optional rules shall specify the conditions for generating the class in a module header named as the class, or for nesting it in the module body. By default classes are mapped into module files of same name as the class.

CppG.4 Dataflow and exception flows are mapped into comments.
Optional rules shall specify the conditions for no generation for such comments.

CppG.5 A provided operation set are mapped into a comment.
Optional rules shall specify the conditions for no generation for such comments.

CppG.6 The MEMBER_OF clause may be mapped into a comment.

[20] such a class is a pseudo class that is never instanciated. May be replaced by a «name space» as implementation will be available.

VNCS.

AG-32. An object allocated to a virtual node and not remotely shall mapped in a standard package nested in the package associated to the VN.An object named <NAME> allocated to a virtual node and remotely called shall be mapped into:

- one package of name <NAME>[19] in the body of the package associated to the VN, and that shall contain code as defined in *rule-codeAG-5*. -for package <NAME>_SERVER, for all its provided operations, andcode associated to the internals of that object
- one package of name <NAME>RB.in the body of the package associated to the VN, and that shall contain code as defined in *rule-codeAG-5*. -for all its provided operations, and all code associated to the internals of that object
- as many package of <NAME> in the body of the package associated to clients VN, and that shall contain code as defined in *rule-codeAG-5*. - for package <NAME>, for all its provided operations, and all code associated to the internals of that object

17.6 C++ AND C TARGETS

17.6.1 Introduction

The translation to C or C++ consists in 2 steps :

- refinement of OPCSs and OBCSs parts from the pseudocode into associated code sections by the designer/programmer.
- code generation from the Object Description Skeleton including internal TYPES, CONSTANTS, DATA, OPCS and OBCS code fields; this generation can easily be automated.

In general, a HOOD object is mapped into a C or C++ module that is defined through two files:

- *module header file* with the extension of C or C++ include file (generally «.h»)
- *module body file* with the extension of C++ source file (generally «.c»)n.

[19]. there cannot be confusion between a package <NAME> and the one containing the code of the originally named <NAME>_SERVER, since the units are nested in different package bodies corresponding to client and server VNs

AG-25. The OBCS of a terminal object <NAME> is mapped as follows :

- the visible obcs section may be mapped into a comment in the package specification associated with the object;
- the internal obcs section is mapped into:
  - all sections (except code sections) may be mapped into a comment in the package associated with object.
  - the OSTM shall be mapped in a package named <NAME>_OSTM.
  - ClientOBCS and ServerOBCS parts are mapped into the package <NAME>_RB. ClientOBCS part shall be mapped into *renames* instructions mapping constrained procedure entries onto ServerOBCS entries
  - the ServerOBCS parts shall be mapped into a ServerOBCS unit (task or package)whose declaration is in the specification of the package named <NAME>_RB and whose body is nested in the body of this package.

    Optional rules shall specify the mechanism used for giving different visibility to the different modules according to the used operations. By default the <NAME>_OSTM is a library unit.

    Optional rules shall specify the type of the IPC communication and thus the type of CLIENTOBCS and SERVEROBCS units. By default the ServerOBCS is a task with one entry named ExecuteRequest.(see section 14.8)

AG-26. The OPCS of an operation of a terminal object is mapped as follows :

- all sections but code sections may be mapped into comments.
- code section is mapped into the subprogram body.

AG-27. The internal section for a terminal object is mapped into comments and Ada declarations in the package body that implements the object.

AG-28. An Instance of a Generic is mapped into a library unit as an instantiation of the generic package that implements the generic.

Optional rules shall specify the mechanism for assuring the visibility of the package that implements the class from the instantiation context. By default, visibility is ensured by a with clause onto the associated generic package.

AG-29. An Instance Range of a generics is mapped into a set of instances of the generic, whose names are the aggregation of the generic name with the different range values.

AG-30. For terminal objects, Internal Types, Constants and Data (including instances of classes) shall be mapped as Ada declarations in the body associated to the object.

AG-31. A virtual node shall be mapped into a package of same name, that includes one package, instance of the HRTS_VNCS generic package and of name

AG-14. A provided operation set may be mapped into a comment.

AG-15. An exception provided by an object X(terminal or not), which is also provided by its parent, is mapped into the declaration of an exception in the package specification of object X, renaming the parent exception.

AG-16. An exception provided by an object X(terminal or not) but which is not provided by its parent is mapped into the declaration of an exception in the package specification, of the object X

AG-17. The RAISED_BY clause may be translated into a comment.

AG-18. A package implementing an object shall have visibility on the packages that implement the required objects. Such a visibility shall only be given to the package specification if references to a required object appear in the provided interface.

Optional rules shall specify the mechanism used for giving such a visibility (with clauses or nesting within the same parent unit). By default visibility is ensured by use of with clauses.

AG-19. A required Environment Object is mapped into a with clause of the package that implements the required object.

Optional rules shall give precise guidelines for placing the with clause. By default, the with clause is placed at the lowest level of unit needed to satisfy the Ada visibility rules.

AG-20. An OP_CONTROL object is mapped into a private child subprogram named as the object.

AG-21. The packages implementing the objects required by an OP_CONTROL object shall be visible from within the private child unit that implements the OP_CONTROL object.

Optional rules shall specify the mechanism for ensuring such a visibility. By default, visibility is ensured by use of with clauses.

AG-22. Required types, constants, operations and exceptions may be mapped into comments.

AG-23. Dataflow and exception flows may be mapped into comments.

AG-24. The OBCS of a non terminal object is mapped as follows :

- the visible obcs section may be mapped into a comment in the package specification associated with the object;
- the internal obcs section may be mapped into a comment in the package specification associated with the object;

<NAME>_SERVER

- non constrained operations shall be mapped into subprogram bodies comprising only OPCS_BODY PART.in the package <NAME>_SERVER

Optional rules shall specify the mechanism for generating non constrained operation as standard procedures in the package <NAME>.
Optional rules shall specify the mechanism used for giving visibility to the different modules according to the used operations. By default these packages are library units.

AG-6. A Class is mapped into a package named as the class and providing a (tagged) type with the name “Instance”.

Optional rules shall specify the conditions for generating the main type with the same name as the encapsulating package and a classwide subtype with user predefined identifier.[18]

Optional rules shall specify the conditions for generating library units, nested units, subunits, (private) child units or protected types. By default classes are mapped into library units .

AG-7. A Generic is mapped into a generic package named as the Generic.

Optional rules shall specify the conditions for generating library units, nested units, subunits, (private) child units or protected types. By default generic units are library units with children requiring formal parameters mapped into private child units.

AG-8. The formal parameters of a Generic are mapped into the formal parameters of the generic package.

AG-9. A type provided by an object X(terminal or not), which is also provided by its parent is mapped into a subtype of the parent type in the package specification of the object X

AG-10. A type provided by an object X (terminal or not) but which is not provided by its parent is mapped either into a type declaration in the package specification of object. X

AG-11. A constant provided by an object X(terminal or not), which is also provided by its parent, is mapped into the declaration of a constant in the package specification of object X, renaming the parent constant.

AG-12. A constant provided by an object X(terminal or not) but which is not provided by its parent is mapped into the declaration of a constant in the package specification, of the object X

AG-13. The MEMBER_OF clause may be mapped into a comment.

[18]. this gives provision for implementing a naming scheme when using generics upon the base class inheritance tree.

The term module is used to designed object, class and generic.

AG-1. A module is mapped into a package named as the object.

Optional rules shall specify the conditions for the generation of library units, nested units, subunits, (private) child units or protected units. By default child objects are mapped into library units and operations into associated bodies.

AG-2. A provided operation of a terminal module is mapped into the declaration of a subprogram specification in the package specification and in a subprogram body in the package body associated with the object.

Optional rules may specify the condition for the generation of subunits[17]. By default they are nested in the package body associated to the object.

AG-3. An internal operation of a terminal module is mapped into the declaration of a subprogram specification body in the package body associated with the object.

AG-4. A terminal module without protocol and TimeOut constraints operations shall be mapped into a package named as the object.

- constrained operations shall be mapped into subprogram bodies comprising non_empty OPCS_HEADER part, OPCS_BODY PART and possibly an OPCS_FOOTER part code as defined in *section 17.2.3*
- non constrained operations shall be mapped into subprogram bodies comprising only OPCS_BODY PART.

Optional rules shall specify the mechanism used for giving visibility to the different modules according to the used operations. By default these packages are library units and visibility is provided through use of with clauses.

Optional rules shall specify how a concurency constrained operation can be mapped into the declaration of a procedure, function or entry of a protected record.

AG-5. A terminal module of name <NAME> with protocol and Time out constraints operations shall be mapped into three packages, named <NAME>, <NAME>_RB and <NAME>_SERVER.

- all operations shall be mapped into procedure bodies comprising only OPCS_ER code as defined in *section 17.2.3* in the package <NAME>.
- all operations shall be mapped into procedure bodies comprising only OPCS_SER code as defined in *section 17.2.3* in the package <NAME>_RB.
- constrained operations shall be mapped into subprogram bodies comprising non_empty OPCS_HEADER part, OPCS_BODY PART and possibly an OPCS_FOOTER part code as defined in *section 17.2.3* in the package

[17]. When operation are numerous generation of separate units leads to an inflation of files.

17.4 HOOD PRAGMAS

Pragmas are directives to be put in the ODS fields in order to give the designer a better control over documentation and code generation. Their syntax is the following :

PRAGMA <Pragma Identifier> <Optional_Parameters>.

The syntax of the different pragmas is described in the ODS BNF as well as the place where they can be inserted.

Most of those pragmas are project/tool dependent. Nevertheless, HOOD defines several pragmas independent of the target language and described in Appendix *F"HOOD Pragmas"*.

17.5 ADA TARGETS

17.5.1 Introduction

The translation to Ada consists in 2 steps :

- refinement of OPCSs and OBCSs from the pseudocode into associated code sections by the designer/programmer.
- code generation from the Object Description Skeleton including internal TYPES, CONSTANTS, DATA, OPCS and OBCS code fields; this generation can easily be automated.

In general, an object is mapped into a package and an operation is mapped into a procedure or function. Classes are mapped into packages containing a type with the same name. The Ada extraction rules are partitioned into two classes : a kernel set which defines standard rules and an optional set which may be triggered by associated pragmas.

17.5.2 Kernel Extraction Rules

In the following, kernel extraction rules are listed. Additional rules are introduced by a sentence starting with “*Optional rules shall specify...*”. Such rules may be triggered by code generation pragmas, specific to a given HOOD toolset and code generator.

When generating code for objects allocated to VNs, the target code shall be structured into :

- on client VNs, <ObjectName> target units for allocated objects requiring remote VNs and that is generated with OPCS_ER code.
- on server VNs, <ObjectName>_RB and <ObjectName>_SERVER target units for remotely called objects. Note that on constrained operation of objects remotely called, are generated in client VN as HSER constrained operations.

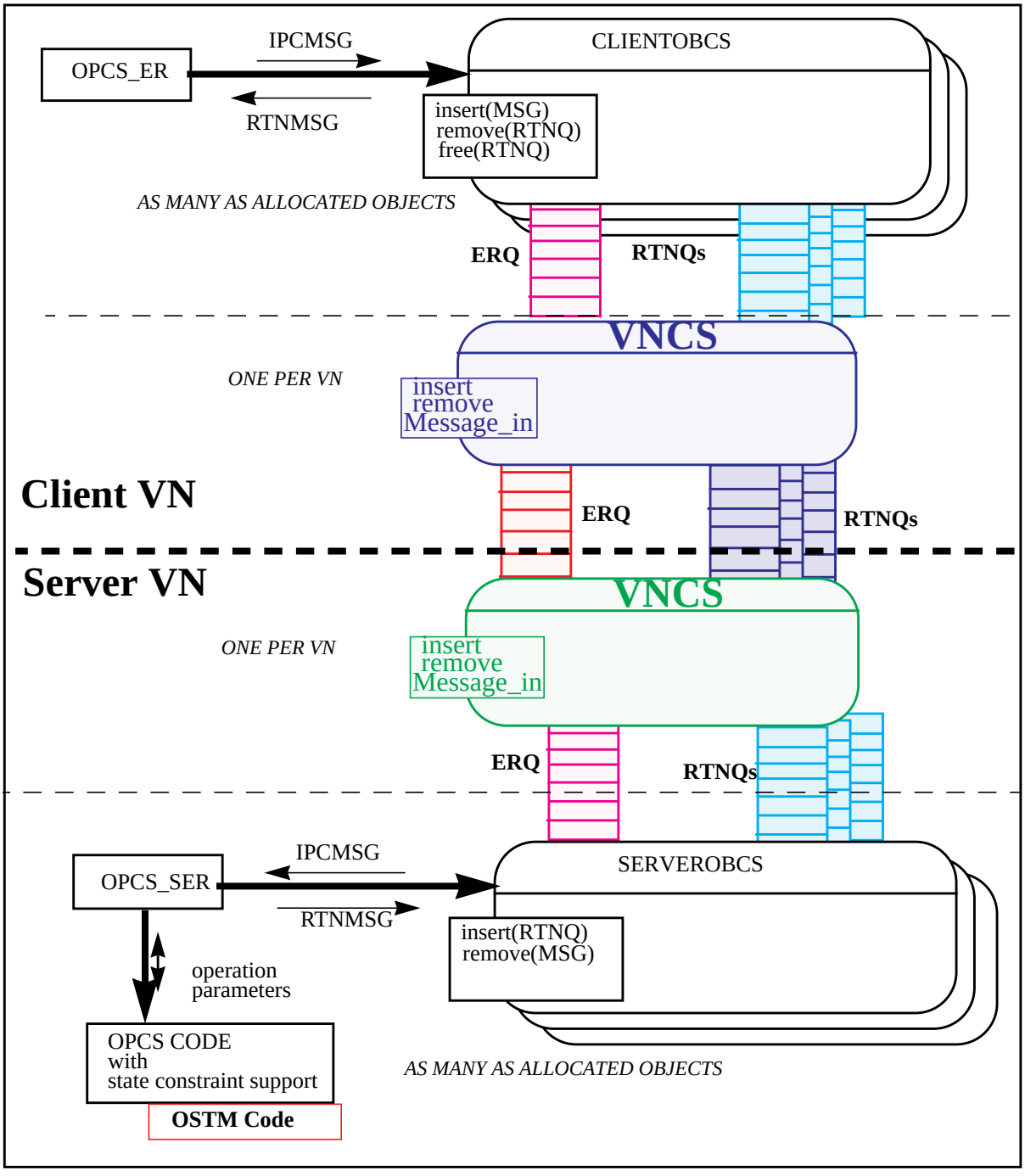

Figure 74 - VN and OBCS communication principle.

The communication software between the VNs in *Figure 72* - is moreover encapsulated within a VNCS object which shall be unique for each VN. This allows to have OPCS_ER or OPCS_SER code unchanged with respect to the specifications given for protocol constraints. However the OBCS objects shall[15] use the VNCS object as «User Defined Communication Mechanism».

Since a VN may be both a server and a Client the VN may be partitioned in at least two sub objects «ClientVNCS» and «ServerVNCS» as illustrated in *Figure 63 - Architecture Principle of the OBCS software*

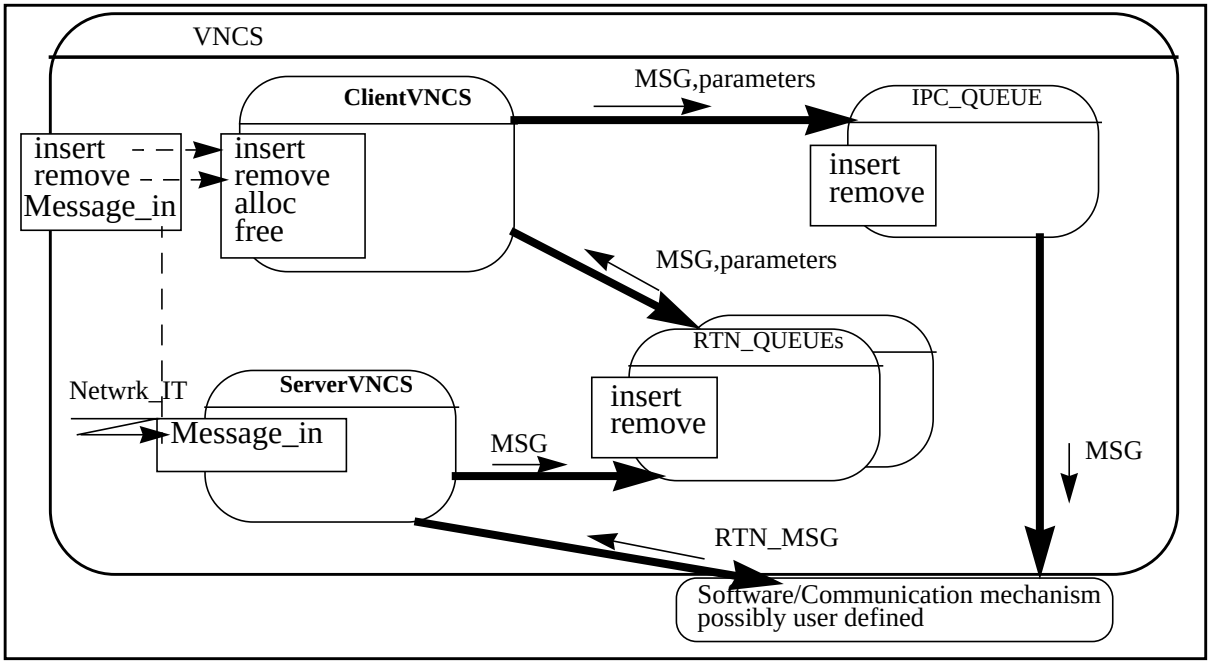

Figure 73 - Architecture Principle of the VNCS software

The operation VNCS.Insert builds up the inter VN communication message (IVN_MSG) Structure (especially it does the *marshalling* converting the parameters structure into streams, allocates a RTNQUEUE and puts its identification in the MSG, and is blocked on the return MSG by calling RTNQUEUE.remove operation. Messages incomming through the communication software (possibly user defined and provided) shall trigger[16] a Message_in operation of:

- of the remote VNCS which shall process the MSG and dispatch it to the appropriate OPCS_SER operation.
- of the local VNCS, which shall process the RTN_MSG and dispatch it (by calling RTNQ.insert) to the appropriate RTNqueue, thus releasing the blocked client process

[15.] This can be achieved by defining the OBCS as a generic with as formal parameter the type of the communication queues

[16.] Either the implementation is polling MSG events on I/O channel, either this procedure is «called_back» directly by the communication software as a MSG was received from the network.

17.3 VIRTUAL NODE IMPLEMENTATION

Figure 72 - below gives the principle of code generation for VN, *reusing and leaving unchanged the functional code generated of the logical model*.

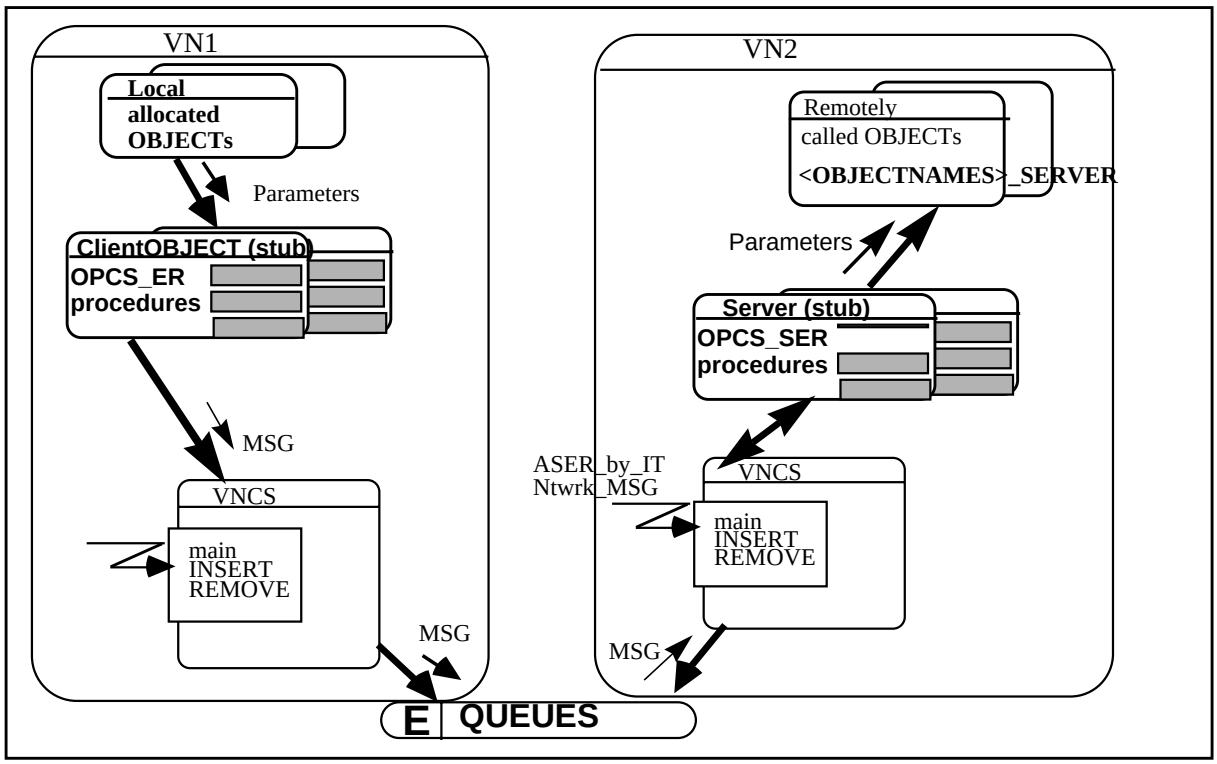

Figure 72 - Execution Model for protocol constrained operations for VN

Hence the implementation of VN target code can map directly the target code structure for protocol constrained operations with communication between client and server processes. In case of a VN, an additional communication mechanism is needed for data exchanges with remote VNs. Two kind of remote FIFO queues are thus used to standardize the implementation of the VN communications:

- an ERQ for queuing calls to remote VNs
- one or several[14] of RTNQ for processing return parameters and status

Queues allow to handle the problem of global time, network and process contention, synchronisation, etc.. in the most modular and flexible way. Furthermore it allows to have a layered implementation decoupling fully the HOOD application of any implementation infrastructure and communication software.

[14]. this shall depends on the number of communication protocols supported between VNs, and shall be tunable for each target

17.2.4 Active Class Implementation Support

The basic implementation principle is to enforce a CLIENT-SERVER architecture, with **one server per class in the standard scheme.** The code generation rules are similar to the ones defined for objects, with the difference that we have instances on the client side and one class and server instance on the server side.Thus for a< *ClassName*> active class the target code is structured in:

- a target module <*ClassName*> implementing the OPCS_ER code as defined above. In order to handle consistency with multiple instances this class shall have an OBCS ATTRIBUTE.
- a target module <*ClassName*>*_SERVER* implementing the effective code of the class. In order to support state constraints, this class shall have an FSM ATTRIBUTE in order to support consistent states.
- a class named <*ClassName*>*_RB* implementing the SER code for all instances of the class. This class has an OBCS ATTRIBUTE allowing to communicate with <ClassName> instance through IPC MSG queues. This class has also an <ClassName_>SERVER Instance ATTRIBUTE to access the effective server code.

Figure below illustrates the above principles for an active class of name **TBuffer**

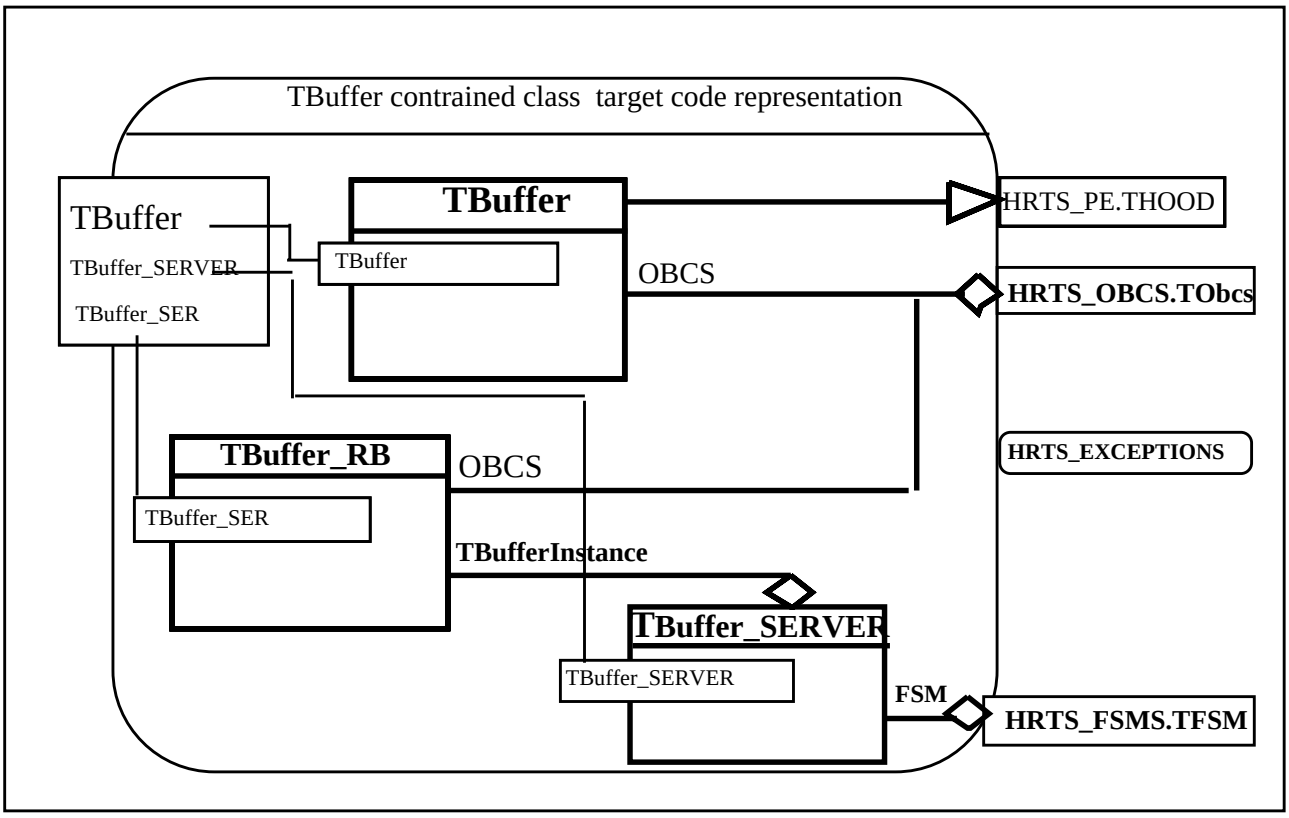

Figure 71 - Class target code representation for TBufferClass

```
package body <ObjectName>_RB is
   procedure <OperationName1> is --OPCS SER code
   • • • • •
   procedure <OperationName2> is --OPCS SER code
   • • • • •
   procedure SerDispatcher is --SerDispatcher procedure
   • • • • •
begin
   loop -- for ever
      OBCS.remove(MSG);
      if MSG.SENDER == <ObjectName> then
         SerDispatcher;
      else
         HRTS_EXCEPTIONS.LOG ("<ObjectName>_RB>", "INCONSISTENT OBJECT NAME"
         MSG.X=COMMUNICATION_ERROR;
         [Build RTNMSG]
         OBCS.insert(RTNMSG); --release client process immediately
      end ;
      exit on OBCS.ShutDown; -- all OBCS activities have to be stooped if set to true.
   end loop;
end <ObjectName>_RB
```

Figure 69 - Example of an ADA implementation for triggering the SerDispatcher procedure

```
procedure SerDispatcher is -- MSG data structure is visible as HOOD DATA
begin
   case MSG.OPERATION is
      when <OperationName1>=>
         <OperationName1>; -- call procedure of current package <OBJECTNAME>_RB;
      when <OperationNamei>=>
         <OperationNamei>; -- call procedure of current package <OBJECTNAME>_RB;
      when • • •
      when others =>
         HRTS_EXCEPTIONS.LOG ("<ObjectName>.<SerDispatcher>", "INCONSISTENT OPERATION
   NAME"
         MSG.X=COMMUNICATION_ERROR;
         [Build RTNMSG]
         OBCS.insert(RTNMSG); --release client process immediately
      end case;
end SerDispatcher>;
```

Figure 70 - Code structure for SerDispatcher procedure

17.2.3.3.2 OPCS_SER for Classes.

The SER code for operations is similar except that a <LocalInstance> of the class must be declared in the <object>_RB target module> The SER code of the class operations thus look like:

```
procedure <OperationName>is -- MSG data structure is visible as HOOD DAta
begin
   [Extract MSG]                          -- extract parameters
   if (MSG.CSTRNT==LSER|TOER_LSER )then-- or LSER we return acknowledge at once
      [build MSG]                         -- build MSG structure
      OBCS.Insert(MSG.RTNQ, MSG);     -- insert RTNMSG in return queue
   else
      begin
      <LocalInstance>.<OperationName>(Params)--effective call on local server code
      exception
      when X_BAD_EXECUTION_REQUEST =>
          HRTS_EXCEPTIONS.LOG("<CLASSName><OperationName>_SER",
                                "X_BAD_EXECUTION_REQUEST");
          MSG.X= HRTS_PE.T_X_VALUE'X_BAD_EXECUTION_REQUEST';
      when Others =>
          HRTS_EXCEPTIONS.LOG("STACK_SER.PUSH", "Others");
          MSG.X= HRTS_PE.T_X_VALUE'OTHERS;
      end;
      if MSG.CSTRNT /= ASER then
          [Build RTNMSG]
          OBCS.insert(RTNMSG); --release client process immediately
      end if;
   end if;
end <OperationName>;
```

Figure 68 - Code structure for Class OPCS_SER

17.2.3.3.3 <OBJECT>_RB Target code outline

Figure 69 - gives a sample of an <OBJECT>_RB structure. Note that active polling of the ER queue is implemented through a loop at elaboration of the package. However the polling could be replace by asynchronous interrupts on ER queues item arrivals. In order to isolate the OPCS_SER code from queuing processing policies, a SERDIS-PATCHER procedure was defined as illustrated in *Figure 70* -.

17.2.3.3 OPCS _SER Specifications

In order to distinguish the effective operation code from the surrogateclient one, and to allow fully automated code generation of OPCS_SER code:

- the server code shall be encapsulated in a target module of name <*ObjectName*>[11]*_RB.*
- the <ObjectName>_RB shall include a procedure called SerDispatcher that is triggered to remove IPCMSG from the ERQ queue and that dispatches execution to trigger the <OperationName> of the <*ObjectName*>*_RB* object. The way the procedure <SerDispatcher> is triggered is implementation defined[12].
- the effective[13] server code shall be encapsulated in a target module of name <*ObjectName*>*_SERVER.*

17.2.3.3.1 OPCS_SER for OBJECTS.

```
procedure <OperationName>is -- MSG data structure is visible as HOOD DAta
begin
   [Extract MSG]                          -- extract parameters
   if (MSG.CSTRNT==LSER|TOER_LSER )then-- or LSER we return acknowledge at once
      [build MSG]                         -- build MSG structure
      OBCS.Insert(MSG.RTNQ, MSG);         -- insert RTNMSG in return queue
   else
      begin
      <ObjectName>_SERVER. <OperationName>(Params)--effective call on local server code
      exception
      when X_BAD_EXECUTION_REQUEST =>
          HRTS_EXCEPTIONS.LOG("<OBJECTName><OperationName>_SER",
                                "X_BAD_EXECUTION_REQUEST");
          MSG.X= HRTS_PE.T_X_VALUE'X_BAD_EXECUTION_REQUEST';
      when Others =>
          HRTS_EXCEPTIONS.LOG("STACK_SER.PUSH", "Others");
          MSG.X= HRTS_PE.T_X_VALUE'OTHERS;
      end;
      if MSG.CSTRNT /= ASER then
          [Build RTNMSG]
          OBCS.insert(RTNMSG); --release client process immediately
      end if;
   end if;
end <OperationName>;
```

Figure 67 - Code structure for OPCS_SER

[11] in case the code is associated to a HOOD class, <*ObjectName*> is replaced by <*ClassName*>. RB stands for Server Execution Request Broker code.

[12] either it is a loop that is triggered on<*ObjectName*>*_RB* object elaboration, or it is triggered by call back from an event handler triggered by the host IPC communication software.

[13] in case the code is associated to a HOOD class, <*ObjectName*> is replaced by ***<ClassName>.***

17.2.3.2.4 OPCS_ER for LSER_TOER constraint

The client process is blocked until begin of OPCS execution or timeout event. In order to abstract from any timer implementation, TIMER operations are defined in the HRTS_TIMER module.

```
procedure <ER_LSER_TOER> (Time_out_Value : in T_Time_out_Value; Time_out in out boolean := false) is
begin
   HRTS_TIMER.Set(Time_out_Value); -- set TIMER to Time_out value
   [build MSG]                     -- build MSG structure
   OBCS.insrem(MSG);               -- insert execution request, and remove return message
   [Extract RTNMSG]                --extract return parameters
   if (TIMER.Timeout) then
      Timeout=true; --test and set timeout boolean
   end if;
end <ER_LSER_TOER>;
```

Figure 66 - Code structure for OPCS_ER with an LSER_TOER constraint

17.2.3.2.5 OPCS_ER for HSER_TOER constraint :

The client process is suspended until end of OPCS execution or timeout event hence OPCS_ER is identical to ER_LSER_TOER

17.2.3.2.6 OPCS_ER for ASER_BY_IT constraint :

The client process is the HW; OPCS_ER has no associated code in the client space (there exists only code in the server space).

- the expression [build MSG] should read as :;

```
MSG.SENDER=<Current ObjectName>;
MSG.OPERATION=<OperationName>;
MSG.CNSTRNT=<HSER|LSER|ASER|RASER|RLSER>;
MSG.PARAMS=<Operation_In_Parameters>;--- marshalling parameters
```

- the expression [extract RTNMSG] should read as :;

```
if not (MSG.X=OK)then
    HRTS_EXCEPTIONS.raise(MSG.X); --raise exception according to Xvalue MSG.X;
else
    HRTS_EXCEPTIONS.LOG ("<ObjectName><OperationName>", "OK.RTNMSG")
end if;
<Operation_out_Parameters>=MSG.PARAMS;--- unmarshalling parameters
if not (MSG.CSTRNT=ASER|RASER|RLSER) then
    OBCS.FREE[0](MSG);                    -- deallocate MSG structure and RTN queue
end if;
```

[0]·For ASER constraints no MSG structure was allocated, for RASER or RLSER constraints, the MSG structure is freed when the client code return from «OBCS.GtRprt>> operation.

17.2.3.2.1 OPCS_ER for HSER constraint :

The client process is suspended up to operation execution by the server.

```
procedure <ER_HSER>is
begin
    [build MSG]                    -- build MSG structure
    OBCS.insrem(MSG);              -- insert execution request, and remove return message
    [Extract RTNMSG]               --extract return parameters
end<ER_HSER> ;
```

Figure 64 - Code structure for OPCS_ER with an HSER constraint

17.2.3.2.2 OPCS_ER for LSER or RLSER constraints.

The client process is suspended until begin of OPCS execution, thus OPCS_ER is the same as for HSER.

17.2.3.2.3 OPCS_ER for ASER , RASER constraints,

The client process is not blocked at all hence:

```
procedure <ER_ASER>is
begin
    [build MSG]                    -- build MSG structure
    OBCS.Insert(MSG);              -- insert OP execution request in sending queue
                                   -- no wait for return parameters
end <ER_ASER>
```

Figure 65 - Code structure for OPCS_ER with an HSER constraint

The FIFO queue based communication mechanism is moreover encapsulated within an OBCS object which is implemented for each active object. This allows to have OPCS_ER or OPCS_SER code that abstracts from the specificities of different target OS or communication software. Since an object may be both a server and a Client its OBCS may be partitioned in at least two child objects «ClientOBCS» and «ServerOBCS» as illustrated in *Figure 63 - Architecture Principle of the OBCS software* below.

The code associated to ClientObcs and ServerObcs parts is defined as a set of classes grouped into the HRTS_OBCS object. Illustration of client code (i.e. OPCS_ER and OPCS_SER code) is more detailed in section of Appendix *J.10"TARGET Code Illustrations FOR PROTOCOL CONSTRAINED OPERATIONS"*.

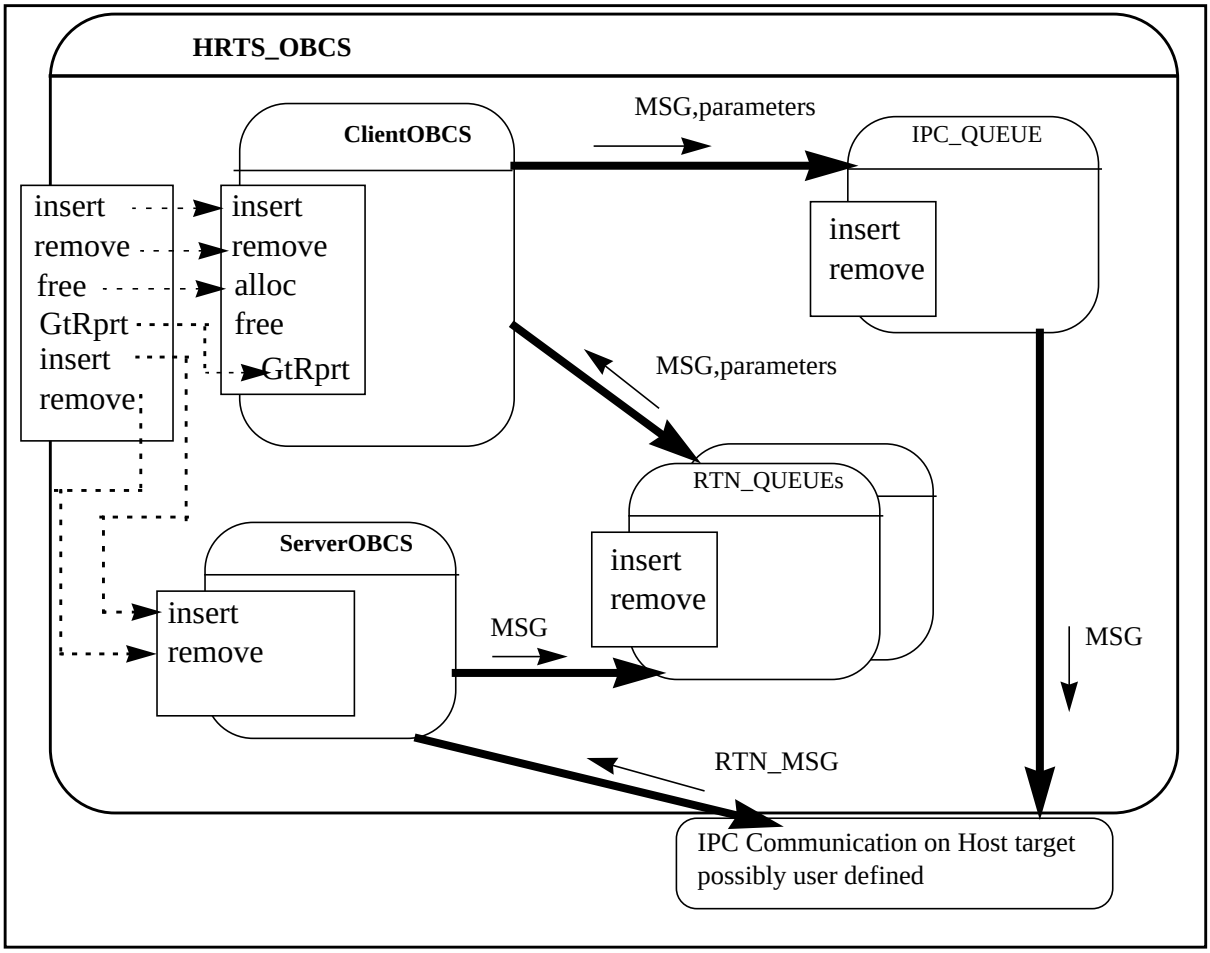

Figure 63 - Architecture Principle of the OBCS software

Moreover, in order to achieve more concise descriptions the following conventions shall be used:

17.2.3.2 OPCS_ER specification

The OPCS_ER code shall build up the IPC MSG data structure (especially it does the *marshalling*[9] converting the parameters structure into streams). The OBCS.Insert call allocates a RTNQUEUE and puts its identification in the MSG. The OBCS.remove call blocks the current thread on this RTNQUEUE.

Since several clients may call a same server at the same time, and still have to wait on «*end of operation execution or acknowledge*», there is one execution request queue (ERQ) and a number of return message queues[10] (RTNQ). *Figure 62* - below details the client-server implementation schema defined in section 3.2.4, whereas highlighting the client and server spaces.

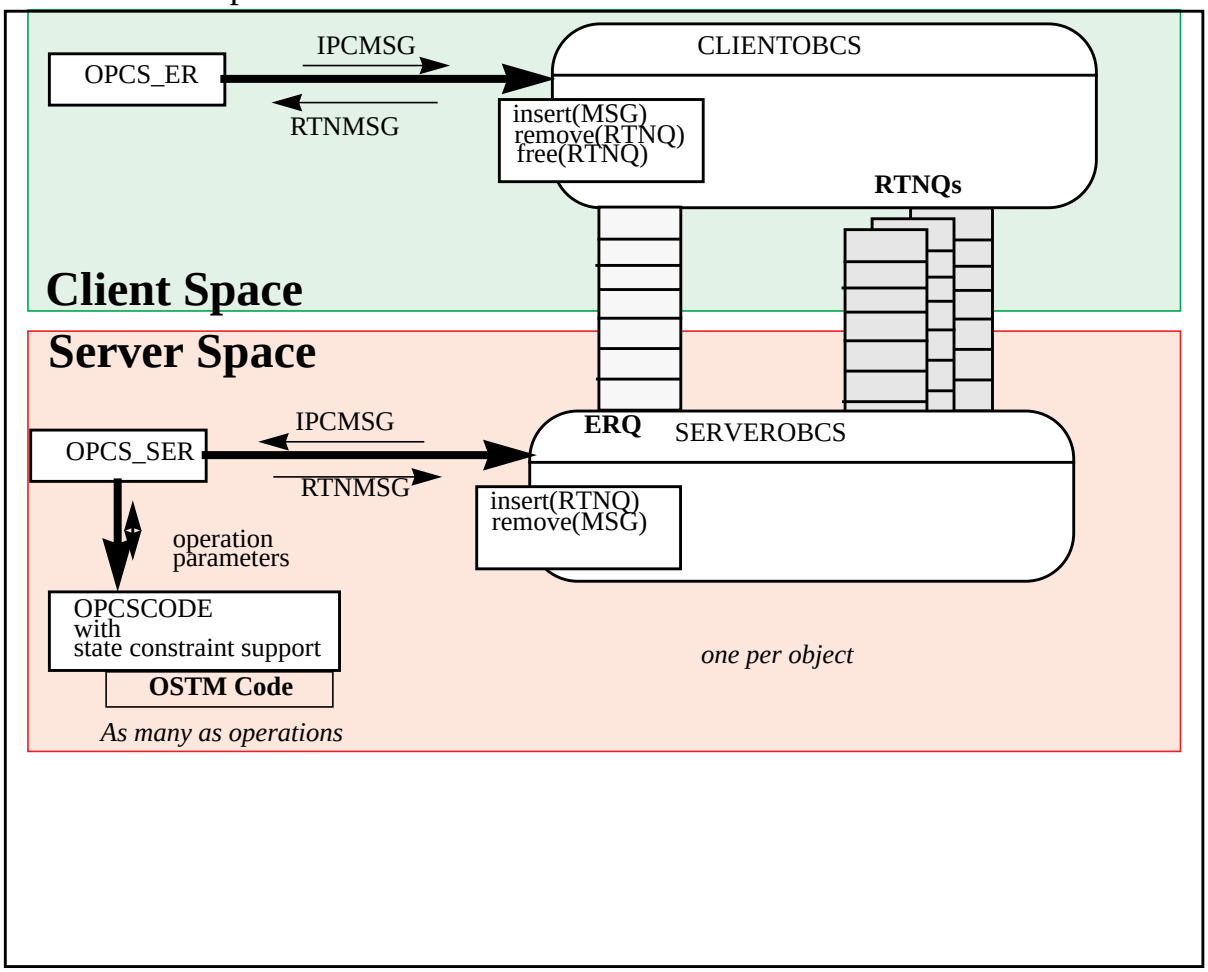

Figure 62 - Protocol constraints implementation principles

[9] marshalling is the action of transforming remote subprogram parameters used a stream-oriented representation which is suitable for transmission between remote processes.Unmarshalling is the reverse action.ADA9X languages libraries shall provide two supporting type attribute functions for a type S, S'write and S'read.In C++, classes to be exchanged should inherit from a virtual stream one, with two IO function >> and <<.

[10] as many as defined in the HRTS_PE.MAX_PENDING_ER constant. This solution allows a great flexibility for tuning in case of efficiency problems.Working with a pool of predefined queues gives improved efficiency after initialization and the size of pool can be set during integration testing. Efficient implementations of «one_element_queues» dedicated to handle RTNMSG should not be more costly than semaphore based synchronisation.

17.2.3.1.2 OPCS FOOTER specification

```
--- semaphore has already been released in OPCS_HEADER so we put just an exception handler
exception
   when others=>
   HRTS_EXCEPTIONS.LOG('from<objectName>).<OperationName>. OPCS_FOOTER','others');
   HRTS_EXCEPTIONS.raise; -- transmit to client level
end;
```

Figure 58 - Code structure for OPCS_FOOTER for state constraint

```
begin
   HRTS_SEMAPHORES.V(<objectName>)-- release exclusive access;
exception
   when others=>HRTS_EXCEPTIONS.LOG('from OPCS_FOOTER','others');
   HRTS_EXCEPTIONS.raise;  -- transmit to client level
end;
```

Figure 59 - Code structure for OPCS_FOOTER for MTEX state constraint or MTEX constraint

```
begin
   HRTS_RWMonitor.ROentry(<objectName>); -- signals one reader out
exception
   when others=>HRTS_EXCEPTIONS.LOG('from OPCS_FOOTER','others');
end;
```

Figure 60 - Code structure for OPCS_FOOTER for ROER constraint

```
begin
   HRTS_RWMonitor.RWOutry(<objectName>); -- signals writer out in
exception
   when others=>HRTS_EXCEPTIONS.LOG('from OPCS_FOOTER','others');
end;
```

Figure 61 - structure for OPCS_FOOTER for RWER constraint

17.2.3.1.1 OPCS HEADER specification

```
begin
   HRTS_SEMAPHORES.P(<objectName>); -- request exclusive access
   HRTS_FSMs.FIRE(<objectName>,<operationName>)--execution request for OP of object OBJ
   HRTS_SEMAPHORES.V(<objectName>)-- release exclusive access;
exception
   when X_Bad_Execution_Request=>
      HRTS_SEMAPHORES.V(<objectName>)-- release exclusive access;
      HRTS_EXCEPTIONS.LOG('from <objectName>.<operationName> in FSM.FIRE',
                          'X_Bad_Execution_Request');-
      HRTS_EXCEPTIONS.raise;              -- transmit to client level
   when others=>
      HRTS_EXCEPTIONS.LOG('<objectName,<operationName>. FSM.FIRE','others');
      HRTS_EXCEPTIONS.raise; -- transmit to client level
end
-- if we come here we are no more in mutual exclusion and state is OK
```

Figure 53 - Code structure for OPCS HEADER for a state constraint

```
begin
   HRTS_SEMAPHORES.P(<objectName>); -- request exclusive access
   HRTS_FSMs.FIRE(<objectName>,<operationName>)--execution request for  OP of object OBJ
exception
   when X_Bad_Execution_Request=>
      HRTS_EXCEPTIONS.LOG( ' from <objectName>.<operationName> in FSM.FIRE',
                          'X_Bad_Execution_Request');-
      HRTS_SEMAPHORES.V(<objectName>)-- release exclusive access;
      HRTS_EXCEPTIONS.raise;                    -- transmit to client level
   when others=>
      HRTS_EXCEPTIONS.LOG('from FSM.FIRE','others');
      HRTS_SEMAPHORES.V(<objectName>)-- release exclusive access;
      HRTS_EXCEPTIONS.raise;  -- transmit to client level
end;-- if we come here we are in mutual exclusion and state is OK
```

Figure 54 - Code structure for OPCS HEADER for MTEX state constraint

```
begin
   HRTS_SEMAPHORES.P(<objectName>); -- request exclusive access
end;-- if we come here we are in mutual exclusion and state is OK
```

Figure 55 - Code structure for OPCS HEADER for MTEX constraint

```
begin
   HRTS_RWMonitor.ROEntry(<objectName>); -- signals one reader in
end;-- if we come here we are in mutual exclusion and we are in RO mode
```

Figure 56 - Code structure for OPCS_HEADER for ROER constraint

```
begin
   HRTS_RWMonitor.RWEntry(<objectName>); -- signals one writer in
end;-- if we come here we are in mutual exclusion and we are in WRiter alone mode
```

Figure 57 - Code structure for OPCS_HEADER for RWER constraint

- Such code is standardised and associated code generation rules are defined, using the HRTS_FSMs module. Appendix *J"HOOD RUN TIME SUPPORT LIBRARY"* gives a full illustration of a FSM supporting code as implemented in the HRTS library
- .procedure HRTS_FSMS.FIRE is called from each OPCS HEADER code to check the state of the object by firing an operation/transition from current state or from the one given as input parameter.
- Accessing the state in the FSM in mutual exclusion is needed since several clients may try to access it concurrently[8]. As a result this semaphore may also be used to protect the access to the OPCS_BODY code when the operation is MUTEX constrained, in which case the semaphore is only released in the OPCS_FOOTER part.

```
procedure <State_Constrained_Operation>  is -- target code structure
begin
-- OPCS_HEADER part (automatically generated)-------------------------------
    HRTS_SEMAPHORE.P(<object Name>); -- seize MUTEX Semaphore
    HRTS_FSMs.FIRE(<object Name>, <operationName>);      -- try to fire operation transition
    -- if execution not allowed in current state, exception BAD_EXECUTION REQUEST is raised.
    HRTS_SEMAPHORE.V(<object Name>); -- only release MUTEX Semaphore if no MUTEX constraint
-- end OPCS_HEADER -------------------------------
--------------------------------------------------
    OPCS BODY CODE ; -- extracted directly from ODS fields
--------------------------------------------------
-- OPCS_FOOTER part (automatically generated)-------------------------------
    HRTS_SEMAPHORE.V(<object Name>); -- release MUTEX Semaphore only here if MUTEX constraint
-- end OPCS_FOOTER -------------------------------
```

Figure 52 - State Constraints Target Implementation Principle

The approach taken for enforcing concurency constraints is to call appropriate semaphore or monitor operations before the core code of the operation, and after respectively the OPCSHEADER and OPCS_FOOTER part,; *section 17.2.3.1.1* and *section 17.2.3.1.2* below illustrate a possible implementation based on services provided by the HRTS library.

[7] (see the paper *[SOUR95]*showing the difficulty of enforcing correct state (with respect to a state specification)).

[8] Optionally a pragma «NOMUTEX» may be used to avoid the associated overhead if the operation is always «thread safe» required .

and orders execution requests with a local temporal order, and that sometimes server may execute operation with different orders[4].

The interface to FIFO QUEUE implementation is defined within the HRTS modules (see Appendix *J"HOOD RUN TIME SUPPORT LIBRARY"*) and could be improved by a designer in order to meet special project efficiency requirements or target implementation constraints.

17.2.3 OPCS TARGET CODE PARTs Specifications

The conventions used in the description of OPCS target code parts are the following:

- the description uses an Ada like pseudo code
- it is supposed that a HRTS library of modules and/or classes is available, especially:
  - EXCEPTIONS for multi-target exceptions handling,
  - SEMAPHORES for multi-target mutual exclusion semaphore handling,
  - FSMS for multi-target finite state machine handling,
  - OBCS and VNCS using a **standard** (remote) fifo inter-process communication mechanism,
- all calls to HRTS modules and functions are prefixed by HRTS_[5]
- the expression <ObjectName> refers to the current HOOD object name, <ClassName> refers to the current HOOD CLASS name and <OperationName> to the current operation.

17.2.3.1 State and Concurency Constraint Support

The approach taken for enforcing the state of an object or a class is to raise an exception[6] if the state was not OK for the operation to execute according to the specification given in the OSTD. illustrates the target procedure code associated to a state constrained operation.

The semantics of *state constrained operations* is thus enforced, since the return to the client is made either with the OPCS executed (the state was OK at the operation request) or with OPCS non executed (the state was not OK at calling time). This is well founded and pragmatic, since automatically enforcing a state according to a specification given in an Object State Transition Diagram (OSTD) could lead to uncontrollable and less understandable[7] code:

[4]. Designers should check HRTS implementation and/or use appropriate HSER, LSER and ASER protocols combinations in order to execute operation in a proper global temporal order.

[5]. Thus the pseudo code can be directly translated in C++, whereas Ada code, could either use a rename clause.

[6]. (to abort the operation request)

17.2.2 Implementation of Protocol and Time constraints

Protocol constraint implementation leads to interprocess communication (pure synchronisation and/or data communication) between at least one client thread and one server thread which execute each in their own *memory partition*.[3]
Figure 51 - shows that OPCS_ER and ClientOBCS code are executed by the client thread, whereas all other parts are executed by server thread[s]. Note that ClientOBCS and ServerOBCS are common to all constrained operations of an object, whereas there is one OPCS_xx part for every constrained operation. A full specification of associated code is given in *section 17.2.3* for each kind of protocol constraint.

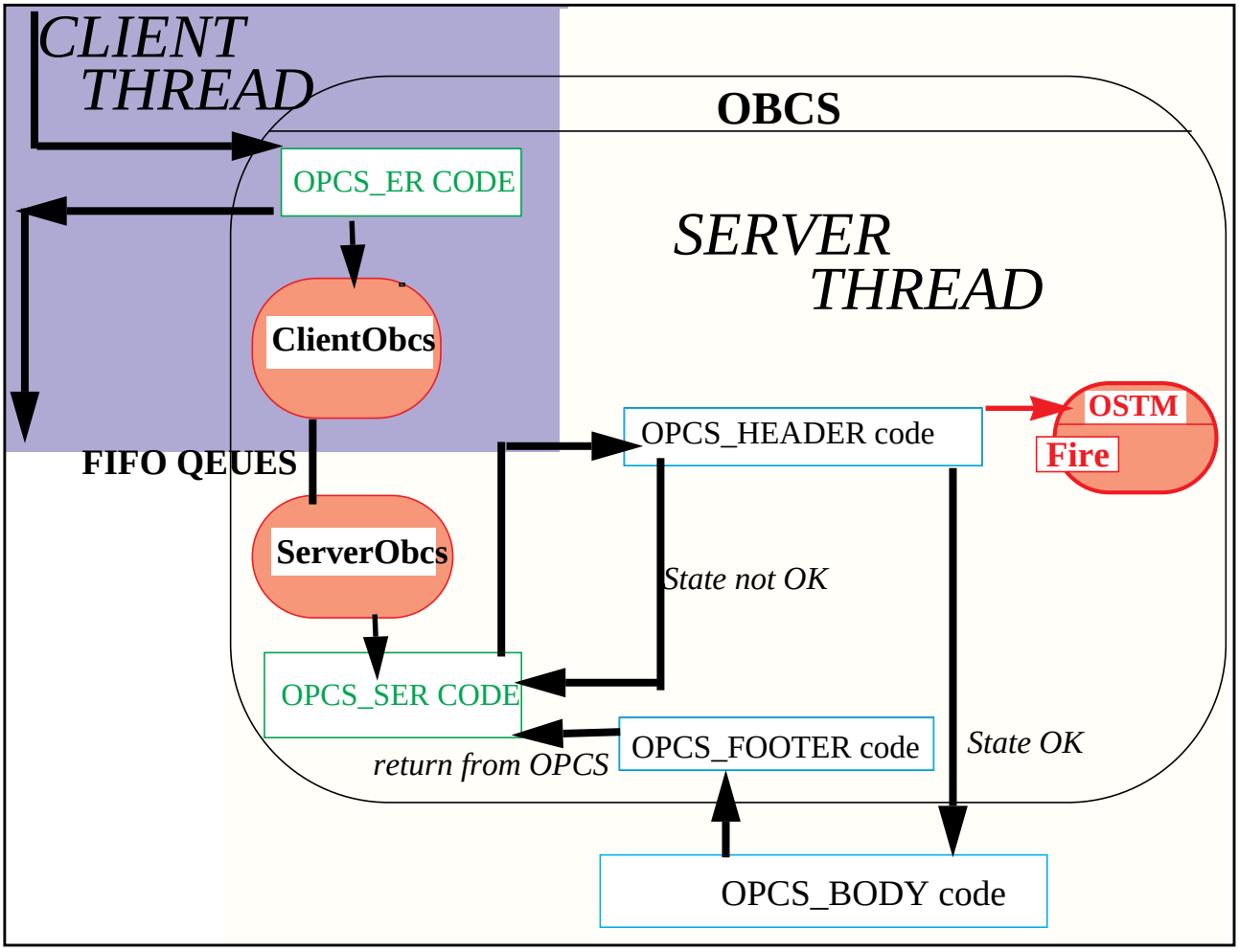

Figure 51 - Protocol constraint Implementation principles

Note that a standard inter communication process mechanism has been chosen which is the FIFO QUEUE and which may support distribution over a network of processors without affecting the above client code. Note that FIFO is related to local client time

[3] this memory partition may be the same for both threads (in which case threads are just execution abstractions like UNIX V threads or Ada tasks) or be distinct for each thread (in which case we have distinct, even remote processes)

protocol constraints defined so far, whereas a full description of the HRTS library is given in Appendix *J"HOOD RUN TIME SUPPORT LIBRARY"*.

17.2.1 Implementation of State and Concurency constrained operations

Figure 50 - illustrates the implementation principle consisting in generating “precondition code and post-condition target code” associated to an OPCS_BODY. Concurency constraints are mapped into calls[2] to semaphores and monitors hidden in the HRTS library. State constraints are mapped into calls to services provided by an HRTS_FSM module. *The object states handling and testing may thus be standardized.* The code of that OSTM could be generated automatically from the OSTD by a HOOD toolset, so that the designer has only to express the possible sequences of an object/class by drawing an OSTD. Appendix *I"Ostm and Ostd Descriptions and Syntax"* gives an illustration of such an OSTM implementation code.

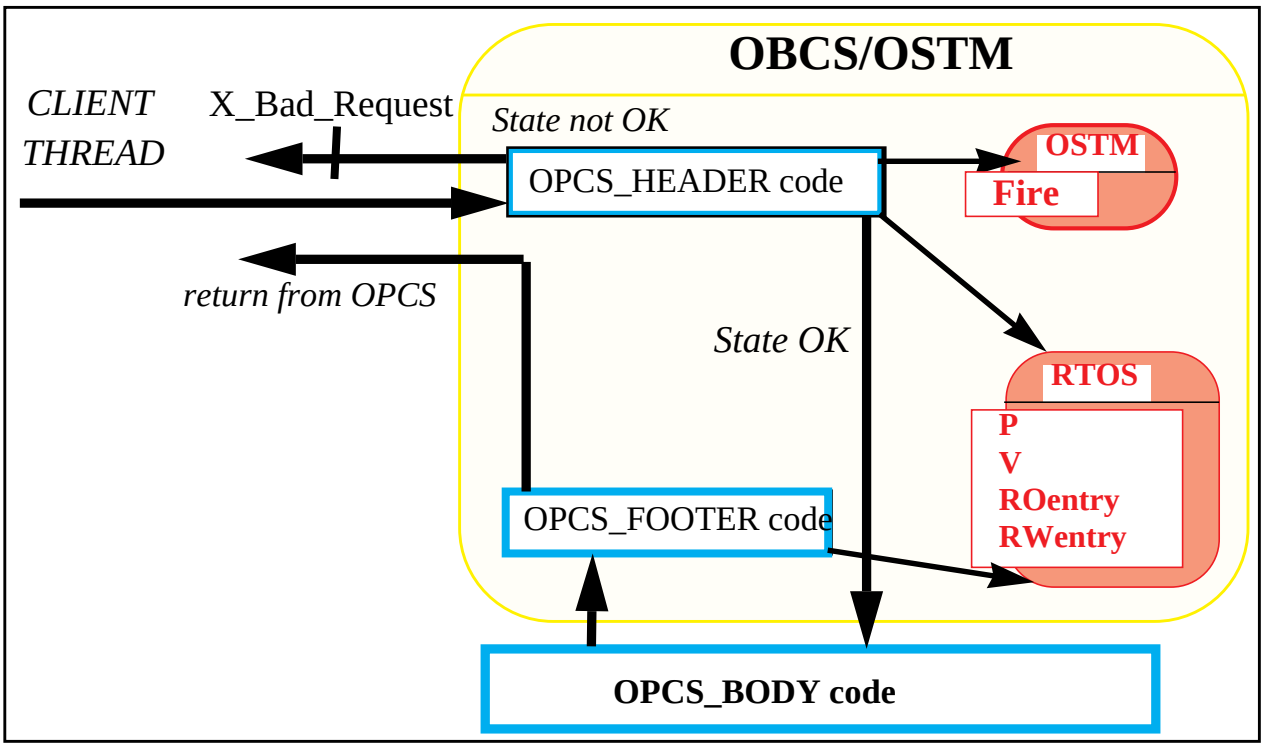

Figure 50 - State Constraints Implementation Principle

[2] Placed into OPCS_HEADER and FOOTER parts

cremental development, testing and integration. Such advantages can be gained from HOOD object with constrained operations and/or with virtual nodes, where an object named <*ObjectName*> is mapped into:

- <***ObjectName***>***_SERVER*** target module for the effective, functional code, integrated in server space
- <***ObjectName***> target module for the client code supporting execution requests, integrated in client space. Such code allows to keep client code unaffected when changing the implementation (e.g. on a client-server architecture) of the object.
- <***ObjectName***>***_RB*** target module for the execution Request Broker code part, integrated in server space.

- *implementation of object relationships.*
  INCLUDE relationships will be implemented through target encapsulation mechanisms if they exists, or by direct file inclusion at source code level. Visibility mechanism such as the use of Ada “with” clauses may also be used. USE relationships will be implemented trough target language mechanisms controlling intra and inter unit visibility, and possibly by using code inclusion. (as e.g. in C or C++).
- *preserving HOOD identifiers as much as possible*.
  Thus avoiding any additional need to change ODS user inputs. This may raise some difficulties especially when the target is not Ada; hence HOOD recommends to follow coding standards and naming conventions when documenting ODS target dependent fields.
- *direct inclusion of ODS code fields* in the associated target unit skeletons.

17.2 MULTI-TARGET CONSTRAINED OPERATION SUPPORT

Target independent constraint operation support is achieved by the definition of a set of HRTS objects implemented in the HRTS library and the splitting of constrained support code into both OBCS code and OPCS code parts that are put before and after the core OPCS body in the associated target language procedure.

In the following we recall the implementation schema for OBCS, and then give the specifications for these OPCS parts that implement the OBCS according to state and

17 IMPLEMENTATION FOR TARGET ENVIRONMENTS

As early as the prototyping stage or when the HOOD design is complete, a HOOD toolset should generate a corresponding set of target language code to implement the design. A set of HOOD pragmas allow a designer to control partially the different translation schemes.

17.1 CODE GENERATION PRINCIPLES

HOOD defines code generation rules allowing the definition of target units and modules by extraction of code fields from the textual ODS fields :

- **HOOD entities declaration** such as OBJECTS, PROVIDED and REQUIRED INTERFACE (TYPES, CLASSES, OPERATIONS and EXCEPTIONS) as well as OPERATION SIGNATURES are "**target language independent**" and can be used to build a target unit architecture using visibility and encapsulation mechanisms as available in the target language/system.
- The associated body fields are "target language dependent" and are directly included in the right place within the target skeleton architecture.
- The following code generation principles are enforced :
  - *mapping HOOD objects to target modules/units*.
    Encapsulation mechanisms provided by the language and target environment are used, e.g. an Ada package, or a C or C++ module is associated to each HOOD object defined by two files with extension .h and .c.
  - *mapping HOOD objects to client-server structures*.
    Structured target code according to the client-server architecture paradigm allows a given software code to benefit easily from powerful physical processing resources. However the structure of the target code units should allow phased, in-

- *the required interface of any child.*

R-6 "OPERATION_SETS" field shall list all operation_set whose one operation at least is used by a MODULE.

R-7 "OPERATIONS" field shall list all operations used by an MODULE.

R-8 "EXCEPTIONS" field shall list all exceptions that might be propagated by required operations.

16.9.2 GENERIC definition

R-9 The FORMAL_PARAMETERS of a GENERIC can only be types or classes, constants or operations.

R-10 "FORMAL_PARAMETERS" field in a GENERIC definition shall list all formal parameters i.e. types, main type of classes, constants and operations required by the class itself or any of its descendants.

R-11 "FORMAL_PARAMETERS" in the required interface of descendants of a class definition shall be consistent with FORMAL PARAMETERS declarations in the class definition.

16.9.3 GENERIC instance

R-12 The required items listed in a INSTANCE_OF a GENERIC shall be complete and compliant, with respect to parameter matching rules with the formal parameter declaration in the GENERIC definition.

R-13 A formal type shall be instanciated by an actual type or class.

R-14 A formal constant shall be instanciated by an actual constant or value.

R-15 A formal operation shall be instanciated by an actual operation.

R-16 The parameters of an INSTANCE_OF shall correspond to the formal parameters of the GENERIC definition.

C-26 The OSTD of a TERMINAL (not VN) MODULE shall describe the possible transitions associated only to provided OPERATIONs.

C-27 The OBCS of a TERMINAL PASSIVE (not VN) MODULE shall be reduced to the OSTD and OSTM fields, with each constrained operation OPCS comprising an associated OPCS_HEADER and OPCS_FOOTER field.

C-28 *The reference to an entity in the REQUIRED_INTERFACE of a client MODULE shall be consistent with the declaration of that entity in the PROVIDED_INTERFACE of the server MODULE.*

16.9 REQUIRED INTERFACE

16.9.1 General

R-1 "REQUIRED_MODULE" field shall list all of (and only) actual MODULEs (excluding environment) i.e. siblings, uncles required by the MODULE.

R-2 "REQUIRED_MODULE" field shall list all of environment MODULEs required by the MODULE itself and not those required by its descendants.

R-3 For each actual MODULE, the list of all required resources (types, classes, constants, operation_set, operations, exceptions) shall be given.

R-4 "TYPES" field shall list all of (and only) the following types: those required from sibling, environment or uncle MODULEs and referenced in:

- *the provided interface of this MODULE for type definitions, constant declarations or operation parameter types,*
- *the internals section of this MODULE for instantiations, types definitions, data definitions, constant definitions, operation parameter types,*
- *the required interface of any child.*

R-5 "CONSTANTS" field shall list all of (and only) the following constants: those required from siblings, environment or uncle MODULEs and referenced in:

- *the provided interface of this MODULE for type definitions or constant declarations,*
- *the internals section of this MODULE for instantiations, types definitions, data definitions or constant definitions,*

C-6 Information defined in the INHERITANCE field of the ODS of a CLASS shall be consistent with INHERITANCE relationships in HOOD diagrams.

C-7 Information defined in the USED_OPERATION field of the ODS of a MODULE shall be consistent with OP_USE relationships in HOOD diagrams.

C-8 Information defined in the PROPAGATED_EXCEPTION field of the ODS of a MODULE shall be consistent with PROVIDED_INTERFACE of server MODULEs.

C-9 Information defined in the DATAFLOW field of the ODS shall be consistent with relevant OP_USE and TYPE_USE relationships in HOOD diagrams

C-10 Information defined in the EXCEPTION_FLOW field of the ODS shall be consistent with relevant OP_USE relationships in HOOD diagrams.

C-11 Only EXCEPTIONS listed in the PROVIDED_INTERFACE EXCEPTIONS field of a MODULE shall be propagated by OPERATIONS provided by the MODULE.

C-12 Parameters of an INSTANCE_OF shall be consistent with formal parameters of relevant GENERIC.

C-13 CHILD OPERATION declaration shall match PARENT OPERATION declaration

C-14 CHILD OPERATION_SET contents shall match PARENT OPERATION _SET contents.

C-15 CHILD OPERATION constraints shall match PARENT OPERATION constraints.

C-16 An operation listed in the USED_OPERATION field of an OPCS shall be either a required operation, an internal or a provided one.

C-17 The PROPAGATED_EXCEPTION list in the OPCS shall be a subset of the provided exception list in the PROVIDED_INTERFACE.

C-18 Each provided and internal OPERATION of a terminal MODULE shall have an OPCS.

C-19 Each TERMINAL MODULE shall have one OBCS

C-20 Each OPERATION, MEMBER_OF and the relevant OPERATION_SET shall be declared in the same MODULE.

C-21 A PARENT has no internal operations.

C-22 A PARENT has no OPCS.

C-23 A PARENT has no DATA.

C-24 An OP_CONTROL has no PROVIDED_INTERFACE.

C-25 An OP_CONTROL has no DATA.

V-12 An entity declared in the REQUIRED_INTERFACE of a MODULE may be referenced (i.e. is visible) anywhere in this MODULE.

V-13 A CHILD has visibility on the REQUIRED_INTERFACE of its PARENT.

V-14 A CHILD has no visibility on the PROVIDED_INTERFACE of its PARENT.

V-15 A CHILD has visibility on the PROVIDED_INTERFACE of its SIBLINGS.

V-16 PROVIDED_INTERFACE of a PARENT has visibility on the PROVIDED INTERFACE of its CHILDren for implementation.

V-17 A MODULE has visibility on ENVIRONMENTs and other ROOTs declared in the SYSTEM CONFIGURATION.

16.8 CONSISTENCY AND COMPLETENESS

All following rules have to be enforced for and only for a complete design:

C-1 Information defined in HOOD diagrams shall be consistent with their textual representation in the relevant ODS fields:

- *one ODS for each MODULE*
- *provided types, operations and exceptions*
- *required MODULE for each uncle MODULE*
- *internal MODULE for each child MODULE*
- *dataflow for each dataflow drawn*
- *exception flow for each exception flow drawn*
- *constrained operation for each operation with a trigger arrow*
- *protocol constrained operation for each operation with a labelled trigger arrow*
- *OBCS reduced to OSTD for PASSIVE MODULEs*
- *Full OBCS for ACTIVE MODULEs*
- *OPCS for each provided operation of a terminal MODULE.*

C-2 If MODULE A requires an OPERATION of MODULE B, then MODULE A shall OP_USE B.

C-3 If MODULE A requires a TYPE of MODULE B, then A shall TYPE_USE B.

C-4 If MODULE A requires a TYPE of CHILD B, then A shall TYPE_USE or INHERIT from or ATTRIBUTE B.

C-5 Information defined in the ATTRIBUTES field of the ODS of a CLASS shall be consistent with ATTRIBUTE relationships in HOOD diagrams.

TYPE. The main TYPE of a class may have the same name as the class or may be named "Instance[1]".

P-2 A TYPE provided by a MODULE may be described by an ATTRIBUTES internal ODS field containing the list of its data structure elements.

P-3 If a CLASS is ABSTRACT, then its main TYPE cannot be instanciated.

P-4 The provided interface of a TERMINAL VN shall have as many «Message_in» OPERATIONs as communication protocols with remote client VNs.

16.7 SCOPING AND VISIBILITY

16.7.1 Naming

V-1 A ROOT name shall be unique within a SYSTEM CONFIGURATION.

V-2 A MODULE name shall be unique within a design tree

V-3 A TYPE name shall be unique within a given MODULE.

V-4 A CONSTANT name shall be unique within a given MODULE

V-5 An EXCEPTION name shall be unique within a given MODULE

V-6 A DATA name shall be unique within a given MODULE

V-7 An OPERATION may be overloaded within an MODULE.

V-8 An ATTRIBUTE name shall be unique within a CLASS.

V-9 *A name shall not be a target language or HOOD reserved word (see Appendix E"Reserved Word List" or your target language reference manual*).

16.7.2 Visibility

V-10 A MODULE has visibility on outside world only through its REQUIRED_INTERFACE.

V-11 An entity (operation, type, constant, exception) not declared in the PROVIDED_INTERFACE of a MODULE can not be referenced (i.e. is not visible) outside this MODULE.

[1] thus following in ADA the naming conventions suggested in *[ROSEN95]*

U-14 A CLASS X ATTRIBUTES CLASS Y if main TYPE of X is composed of at least one element defined as an instance of main TYPE of Y.

U-15 INHERIT relationships shall not be cyclic.

U-16 If a CLASS INHERITs from, or ATTRIBUTEs an UNCLE, the latter shall be TYPE_USED by its PARENT.

U-17 A main TYPE of a CLASS shall only INHERIT from another main TYPE required by the CLASS

16.5 OPERATIONS

O-1 If an operation is protocol constrained, then it shall be provided by an ACTIVE MODULE.

O-2 If an OPERATION_SET is provided by a MODULE, then all its MEMBER_OF elements shall be provided by the same MODULE.

O-3 An element, MEMBER_OF an OPERATION_SET shall not be MEMBER_OF another OPERATION_SET.

O-4 An OPERATION_SET of a PARENT can only have OPERATIONS and/or operation_set as MEMBER_OF elements.

O-5 If an OPERATION is provided by or is internal to a CLASS, then its declaration shall specify one parameter, called the receiver, of reserved name ME or MYCLASS, and being of the main TYPE of the CLASS.

O-6 If an OPERATION is provided by or is internal to a CLASS, and is INHERITED from an OPERATION provided by or internal to the superclass, their names shall be identical and their declaration shall match, except for the receiver.

O-7 An ABSTRACT OPERATION shall only be defined in an ABSTRACT CLASS

16.6 PROVIDED INTERFACE

P-1 A CLASS shall provide a TYPE with an INHERITANCE ODS field containing the list (possibly empty) of its superclasses. This TYPE is called the CLASS main

16.3 USE RELATIONSHIP

The use relationships describe either functional, structural or behavioural dependencies between modules. USE means either TYPE_USE and/or OP_USE.

U-1 A MODULE OP_USES another MODULE when it requires at least one of its provided OPERATIONs.

U-2 A MODULE TYPE_USES another MODULE when it requires at least one of its TYPES.

U-3 An UNCLE of a CHILD MODULE is a MODULE used by its PARENT.

U-4 *If a child MODULE OP_USES an UNCLE, then it shall also be OP_USED by its PARENT.* This rule is implicit for ENVIRONMENTS.

U-5 *If a PARENT OP_USES another MODULE, then at least one of its children shall also OP_USE it.*

U-6 *If a child MODULE TYPE_USES an UNCLE, then it shall also be TYPE_USED by its PARENT.* This rule is implicit for ENVIRONMENTS.

U-7 *If a PARENT TYPE_USES another MODULE, then at least one of its children shall also TYPE_USE it.*

U-8 An OP_CONTROL may use any OPERATION, constrained or not.

U-9 Passive MODULEs should not use each other in a cycle.

U-10 A VN shall only use other VNs.

U-11 A Passive (not OP_CONTROL) MODULE may OP_USE an ACTIVE MODULE, only if it requires non constrained or state constrained OPERATIONS

U-12 OP_USE relationships between children of a parent should not be cyclic.

16.4 INHERITANCE AND ATTRIBUTE RELATIONSHIPS

U-13 A CLASS X INHERITS from CLASS Y if main TYPE of X is a subclass of main TYPE of Y.

16.2 INCLUDE RELATIONSHIP

The include relationship describes modular breakdown of a MODULE into other MODULEs.

I-1 A PARENT is a MODULE that INCLUDEs at least one other MODULE.

I-2 A CHILD is a MODULE which is INCLUDEd by a PARENT.

I-3 When a PARENT is broken down into CHILDren, they shall provide together the same functionality.

I-4 *A CHILD shall not have more than one PARENT.*

I-5 *A MODULE shall not INCLUDE itself.*

I-6 Each OPERATION provided by a PARENT shall be IMPLEMENTED_BY one OPERATION provided by one of its CHILDren.

I-7 Each TYPE provided by a PARENT shall be IMPLEMENTED_BY one TYPE provided by one of its CHILDren.

I-8 Each CONSTANT provided by a PARENT shall be IMPLEMENTED_BY one CONSTANT provided by one of its CHILDren.

I-9 Each EXCEPTION provided by a PARENT shall be IMPLEMENTED_BY one EXCEPTION provided by one of its CHILDren.

I-10 Each non constrained operation provided by a PARENT shall be IMPLEMENTED_BY a non constrained operation of a CHILD or by an OP_CONTROL.

I-11 Each constrained operation provided by a PARENT shall be IMPLEMENTED_BY an operation of a CHILD with the same constraint or by an OP_CONTROL.

I-12 Each OPERATION_SET provided by a PARENT shall be IMPLEMENTED_BY an OPERATION_SET provided by one of its CHILDren.

I-13 The OBCS of a PARENT shall be implemented _by one or more OBCS of its CHILDren.

I-14 A CLASS shall be TERMINAL.

I-15 A VN shall only include other VNs.

G-6 A MODULE may require one or more ENVIRONMENTs, each of which defines the interfaces of a ROOT declared in the SYSTEM_CONFIGURATION.

G-7 An OP_CONTROL is a TERMINAL MODULE without internal DATA, dedicated to the implementation of one and only one OPERATION.

G-8 A CLASS is a TERMINAL MODULE which provides one and only one main TYPE and OPERATIONs whose receiver is of type TYPE.

G-9 A state constrained operation is a CONSTRAINED OPERATION with a CONSTRAINED_BY STATE clause

G-10 A Concurency constrained operation is a CONSTRAINED OPERATION with a CONSTRAINED_BY clause containing at least one of the constraint attribute MTEX, RWER or ROER

G-11 A protocol constrained operation is a CONSTRAINED OPERATION with a CONSTRAINED_BY clause containing at least one of the constraint attribute ASER, HSER, LSER, RASER or RLSER.

G-12 A time out constrained operation is a CONSTRAINED OPERATION with a CONSTRAINED_BY clause containing at least the constraint attribute TO

G-13 TO constraint attributes shall only be combined with protocol constraints.

G-14 Protocol constraint attributes may be combined with state and/or Concurency constraints.

G-15 Concurency constraint attributes may be combined with state constraints.

G-16 A PASSIVE MODULE is a MODULE that provides non constrained operation and/or state constraints.

G-17 An ACTIVE MODULE is a MODULE that is not PASSIVE.

G-18 *A TERMINAL MODULE built by instantiation of a GENERIC class (resp OBJECT) is a class (resp an OBJECT).*

G-19 A TERMINAL VN is defined by allocation of (non VN) MODULEs of the same SYSTEM_CONFIGURATION.

G-20 A VN shall provide at least one predefined OPERATION «Message_In». specifying the communication protocol with remote client VNs.

G-21 A PASSIVE (not VN) MODULE shall be allocated to the same VN as all its clients.

16 HOOD RULES

In the following we use the term MODULE in order to designate a HOOD object, class, generic or all possible specialization as detailed in *section 8* to *section 12*, unless otherwise specified. The following conventions are used in order to distinguish between definitions and rules:

- a HOOD definition shall written in normal text.
- *a HOOD rule shall be written in emphasized text.*

16.1 GENERAL DEFINITIONS

G-1 A MODULE is of one and only one of the following types:

OBJECT, ACTIVE or PASSIVE,
CLASS, ACTIVE or PASSIVE,
OP_CONTROL,
GENERIC,
INSTANCE_OF (GENERIC).
ENVIRONMENT, ACTIVE or PASSIVE,
CLASS ENVIRONMENT, ACTIVE or PASSIVE,
GENERIC ENVIRONMENT, ACTIVE or PASSIVE,
VN.

G-2 A MODULE which does not have a parent is called a ROOT.

G-3 A SYSTEM_CONFIGURATION is a set of ROOTs which define the scope and visibility of a given ROOT considered as the «SYSTEM_TO_DESIGN».

G-4 A ROOT may be[0]:

-the current «SYSTEM_TO_DESIGN»
-an ENVIRONMENT, application or library
-a GENERIC (OBJECT or CLASS)
--a VIRTUAL NODE

G-5 A TERMINAL MODULE is a MODULE which has no CHILD MODULES.

[0]. HOOD recommends that classes be child of library objects thus encapsulating inheritance hierarchies.