

- Include relationship rules define the way a given parent object is implemented by a set of child objects or classes according to their types and components (Operations, Types, Constants, Exceptions, OBCS and Dataflow).
- breakdown rules describe the implemented by relationship properties according to the type of the associated operations (constrained, unconstrained, provided, required, internal operations, operation implemented by an OP_Control, operation_set) and the status of the associated object (Terminal, Non-terminal).
- Operation rules give the basic definitions and properties of the HOOD operations.
- Visibility and naming rules give, in addition to the above definitions (*section 14.1*), naming and consistency checking rules :
  - roots shall have different names in a system configuration,
  - object names shall not be overloaded within design tree,
  - an operation may be overloaded within an object.
- Consistency Rules check consistency of child descriptions with respect to their parent descriptions :
  - parent REQUIRED_INTERFACE with respect to union of child REQUIRED_INTERFACES,
  - parent PROVIDED_INTERFACE with respect to union of child PROVIDED_INTERFACES.
- Internal consistency checks ensure consistency of the HOOD entities within the ODS, and between ODS and diagrammatic description.
- These checks may be performed recurrently on one or more objects in the Design Process Tree.

  - The REQUIRED_INTERFACE of a child object has visibility on the names of all brothers, uncles, classes and root objects.
  - The REQUIRED_INTERFACE of a child object can indirectly access all accessible entities declared in all brothers, uncles, classes and root objects.
- the scoping within a system configuration is summarized within figure below.

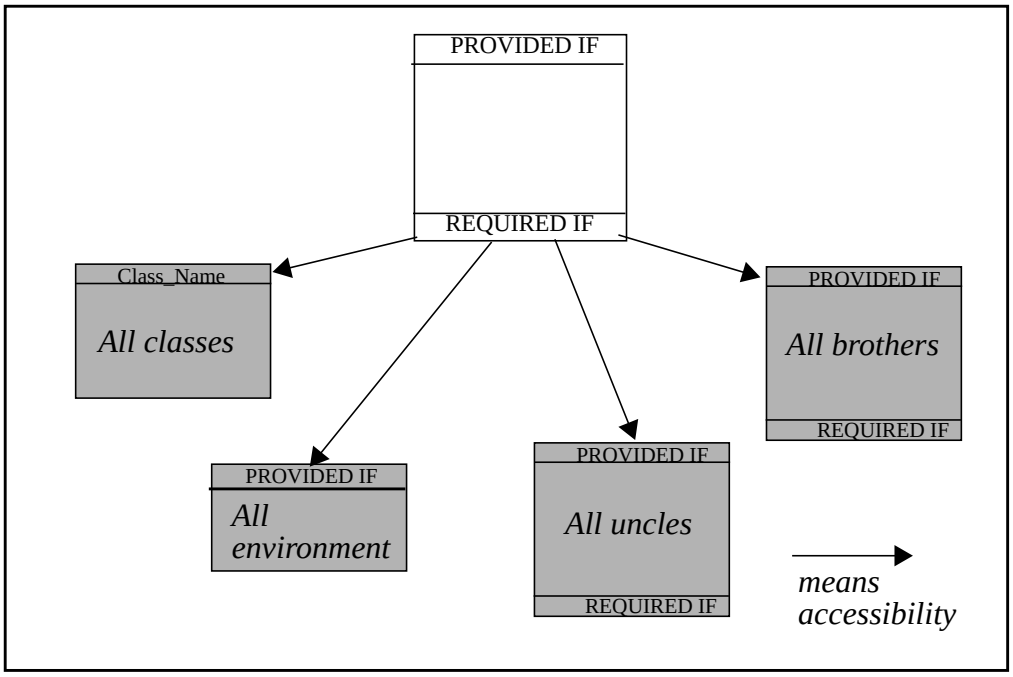

Figure 49 - Scoping within a system configuration

Homographs may appear in REQUIRED_INTERFACE. In that case, ambiguity must be solved by qualifying the entity reference[56].

15.2 HOOD RULES

Formal rules can be defined for checking consistency and completeness of HOOD design descriptions. In the following, an overview of several categories of rules are given; a detailed description of these rules is given in *section 16* of present document.

- General definitions rules give the basic definitions and properties of HOOD object types (Active, Passive, Environment , Class, Op_Control, Generic, VN) in the HDT.
- Use relationship rules define the way an object, class, generic or Vn can use each other within the respective scopes of the declaration of entities involved.

[56]. e.g. for overloaded operation, ambiguity is solved by replacing the operation name by its complete signature.

  - the PROVIDED_INTERFACE itself,
  - its OBCS (if any), or its VNCS (for VNs only),
  - its INTERNALS declarations (for terminal objects),
  - its IMPLEMENTED_BY part (for non terminal objects),
  - its OPCS parts(OPCS_HEADER,OPCS, OPCS_BODY).
- an entity declared in the INTERNALS of a terminal object is visible from :
  - the INTERNALS declarations themselves,
  - the OBCS parts(OSTM, CLIENT_OBCS, SERVER_OBCS if any),
  - the OPCS parts(OPCS_HEADER,OPCS, OPCS_BODY)
- an OPCS name is visible from :
  - the other OPCSs,
  - the OBCS parts.
- an entity listed in the REQUIRED_INTERFACE is indirectly accessible from the whole object[55].

• the scoping within an object, class, generic or VN is summarized in figure below.

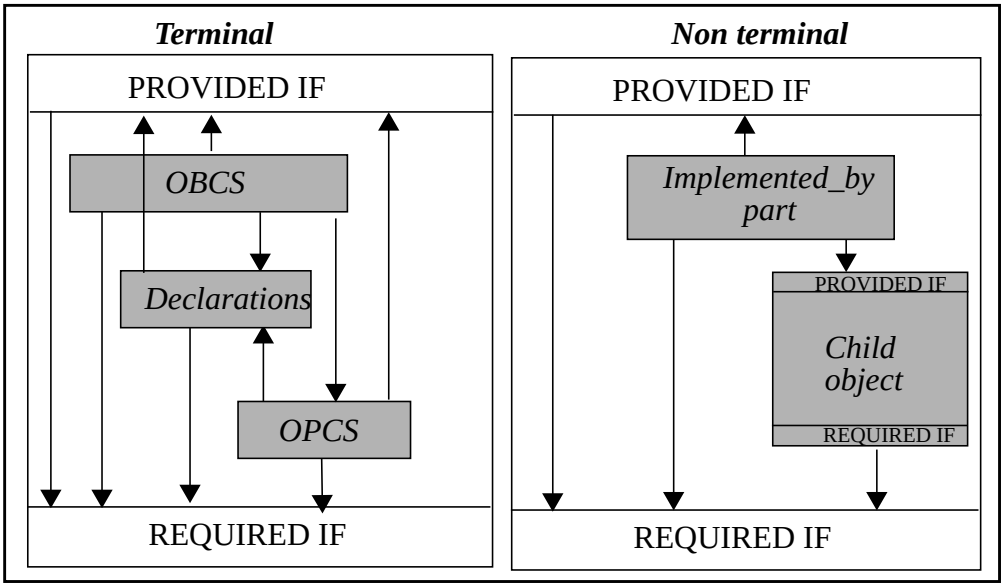

Figure 48 - Scoping inside an object

• Accessibility of an entity **within a system configuration** is defined as :
  - An entity (except object and generic) is accessible only if it is declared in the PROVIDED INTERFACE. Other entities are hidden.
  - The REQUIRED_INTERFACE of a root object has visibility on the names of all generics and other root objects.
  - The REQUIRED_INTERFACE of a root object can indirectly access all accessible entities declared in all generics and other root objects.

[55]. Reference to the entity shall only be done by selection (i.e. prefixing its name with the name of the object, using the dotted notation).

15 DESIGN CHECKING, SCOPING, VISIBILITY AND HOOD RULES

A HOOD model is a representation of a System to Design in terms of objects, classes, generics and relationships within a system configuration. In order to ensure consistency of these representations, HOOD defines visibility and scoping rules.

15.1 VISIBILITY AND SCOPING

In the following,

- entity stands for OBJECTS, CLASSES, GENERICS, VNs, OPERATIONS, TYPES, CONSTANTS, DATA, EXCEPTIONS, except stated otherwise;
- **visibility** of an entity is the property of that entity to be **directly accessible** by another one;
- **scope** of the declaration of an entity is defined as parts of the system configuration or ODS parts where the entity is accessible (directly or indirectly[54]).

In order to promote object orientation: encapsulation, information hiding, testability and reuse support principles, HOOD defines objects through well defined interfaces, the PROVIDED and REQUIRED ones, allowing context-independent definitions of objects.

As primary rules to enforce these principles, visibility of entities declared within an object is restricted to that object, and the scope of any declaration outside the object is defined with respect to the current system_configuration :

- An entity can only be visible if it was declared.
- The scope of an entity **within an Object, Class, Generic or VN** is defined as follows :
  - an entity declared in the PROVIDED_INTERFACE is visible from :

[54]. Indirect access means that the entity has to be referenced by selection i.e. by prefixing it with the name of the entity where it is declared (object or class).

turn parameters using services provided by a Client_Obcs instance. Such code is illustrated in *Appendix J.10.1.1* for ADA and in *Appendix J.10.3.1* for C++

- **The OPCS_SER** code part allows to describe for each protocol constrained operation, the processing of an effective call to the required operation upon a local copy object_SERVER and the definition and sending back of return parameters using services provided by a Server_Obcs instance.Such code is illustrated in *Appendix J.10.1.2* for ADA and in *Appendix J.10.3.2* for C++
- **The OPCS_HEADER** code part allows to implement for each concurrency constrained operation, the activation of a synchronization service (either call to a semaphore for MUTEX constraint, or call to HOARE monitor for ROER and RWER constraints), provided by a module of the HRTS library.
  Additionally, the object internal state is managed through a call to the FIRE service provided an FSM instance of class TFsm from the HRTS. If the state is OK, the execution resumes in the OPCS_body, if the state is not OK, the execution resumes into an exception of name X_BAD_REQUEST that gives control back to the client.
  The OPCS_FOOTER code part allows to implement for each concurrency constrained operation, the release of synchronization service possibly previously set in the OPCS_HEADER.
  OPCS_HEADER and FOOTER code is illustrated in *Appendix J.10.1.3* for ADA and in *section J.10.3.3* for C++.

• **Description of OPCS_BODY**

Description of OPeration Control Structures (OPCS) are produced in a twofold way :

- first as a short description in natural language sentences, aiming at describing how the operation works.
- second as a more formal description :
  - a pseudo code part allowing to use any PDL mixing keywords and parts of natural language sentences. As the design is refined, these natural language parts are gradually replaced by functions and expressions. This description may later be included as comments in the code part.
  - a code part refining the pseudocode description and implementing the operation in target language.

Some operation may have no OPeration Control Structures (OPCS) , especially class operations which are inherited or abstract :

- the OPCS of the inherited operations is the same as the one that is described in the ODS of the inherited class.
- the OPCS of an abstract operation shall be defined in the ODS of the subclass where the operation is redefined.

- ROER -- to specify that an operation shall be executed by multiple “reader threads” according to a “multiple readers-one writer” schema
- RWER -- to specify that an operation shall be executed by a single writer thread at a time, according to a “multiple readers-one writer” schema.

Note that the ROER and RWER constraints operation may be mapped intoAda protected record entries by using the pragma PROTECTED (see *section F.8*).

14.17.2 Description of OBCS Internal Parts

The INTERNALS part of the ODS allows the user to specify and edit, for each constrained operation, the target code associated to OBCS and OPCS units according to the type of operation constraints:

- **OBCS_CODE :**
  - **The OSTM** code field is the target language implementation of the OSTD, using a standard HRTS.FSM objects. Such code is illustrated in *Appendix I.2.1* for ADA and in *Appendix I.2.2* for C++. Such code may be automatically generated by a toolset from the OSTD description.
  - **The Client_Obcs** part allows to setup an interprocess communication facility for the object, (it possibly does marshalling of the parameter data structures into a stream of bytes) send it through a (possibly remote) FIFO queue to the Server_Obcs, waits on return message and process return parameters (possibly unmarshalling the return data). Such code is illustrated in *Appendix J.3.2.2* for ADA and in *section J.3.3.5* for C++.
  - **The Server_Obcs** part allows to describe for each protocol constrained operation, the polling and analysis of an interprocess communication message received through a FIFO queue, and its dispatching to Server_ER code.Such code is illustrated in *Appendix J.3.2.3* for ADA and in *Appendix J.3.3.5* for C++.

Client_OBCS part may be empty code especially when target system is full Ada. Server_OBCS part on the other hand is the intermediate part between the original server code and a remote client (see *section 17.2* for a detailed explanation on target code structure for client-server architecture). As such Server_Obcs part play the role of the Object Request Broker in the CORBA architecture*[OMG-CORBA]*. Moreover, when the target is supporting a CORBA implementation, the Server_Obcs[53] may directly acces the CORBA services for handling remote message exchanges.

- **OPCS related CODE:**
  - **The OPCS_ER** code part allows to describe for each protocol constrained operation, the definition of an interprocess communication message, its sending to a server process through a FIFO queue (remote or not) and the processing of re-

[53]. Such an implementation is under test in C++, and an illustration will be included in the final release of current document.

Exceptions propagated by child operations implementing parent operation, propagate also from parent to its clients. In the ODS, the exceptions which propagate from a server to a client must be stated in the provided interface of the server and in the required interface of the client.

14.17 DESCRIPTION OF OBCS

OBCS implementation description is splitted into several fields: OSTD-OSTM, Client_Obcs and Server_Obcs which are global to the object, and other code fields which are associated to each OPCS of constrained operations. This splitting has as goals to allow separated description and verification of pure state constraints from inter-process communication descriptions. The OBCS are moreover separated into visible and internal parts.

14.17.1 Description of OBCS Visible Part

The OBCS visible part is described through three fields:

- first as a short description in natural language sentences, aims at describing how the operation control flows interact and define the object or class behaviours :
- A description of the OSTD may then be established as a state transition diagram where transitions are only triggered by operation according to the semantic of the object or class.
- A description of the protocol constraints may then be given for the following protocol constraint alternatives :
    - **ASER** -- Software and message interrupt
    - **LSER** -- Acknowledge protocol
    - **HSER** -- RPC protocol
    - **RASER** -- Software interrupt with later asynchronous reporting
    - **RLSER** -- Acknowledge protocol with later asynchronous reporting
    - ASER_By_It <interrupt vector >-- to specify a hardware interrupt
    - <informal_text_label> giving provision for user defined protocols.
      Each protocol constrained alternative may be combined with **TO = duration** -- to specify a timeout constraint upon the operation
- Each above alternatives may moreover be combined with the following concurrency constraint, followed by <informal_text>:
    - **MTEX** -- to specify that a whole operation shall be executed in mutual exclusion

- primitive types of the target language
  or
- user defined types as provided or internal types of the current object
  or
- provided types of another HOOD object. This can be an object implementing an **Abstract Data Type** (ADT), or a HOOD class (see *section 9*). In that latter case the name of the class is implicitly provided as a HOOD type.

DATA instances of primitive, or user defined types may only be declared in the INTERNALS of terminal objects. DATA instances associated to implementations of ADTs may:

- be declared as data items in the INTERNALS of terminal objects.
- be dynamically created via execution of a specific constructor operation provided by that object or class. This dynamical creation takes place in the OPCS fields.

14.15 DESCRIPTION OF OPERATION_SET

The members of an operation_set are declared in the provided interface of the parent object with the keyword **MEMBER_OF** followed by the name of the operation_set (see *section 14.8* for an example). An operation can not be member of several operation_sets.

In the case the operation_set is implemented by a child operation_set, the associated **IMPLEMENTED_BY** link shall be defined in the INTERNALS of the parent object.

14.16 DESCRIPTION OF EXCEPTIONS

A HOOD exception is associated to an operation. A HOOD exception propagates along the use relationship from the operation where it is raised up to the operation of the client object which executes the associated recovering code. Exception handling is similar to Ada where an exception may be handled locally or propagated further.

Exceptions which are provided by a non terminal object shall be **implemented by** child objects. The internal exception field of the parent ODS is then filled with the keyword **IMPLEMENTED_BY** and the name of the child exception.

- either as the identifier of the HOOD module associated to the class, or
- specifically for Ada targets, as “Instance” (in which case the naming convention suggested in *[ROSEN95]* may be enforced.

A class is moreover specified through:

- an inheritance field, that defines all superclass from which the current class inherits, and shares properties (operations and public attributes). Inheritance is described using the target language syntax. A Class may have public inheritance (in which case the inheritance field is in the provided interface, or private inheritance (in which case the inheritance field is in the internals of the ODS).
- attributes field, that define data structures aggregated to the class. Attributes are described using the target language syntax. A class may have public attributes (in which case the attribute field is in the provided interface) and private attributes (in which case the attribute field is in the internals of the ODS). Public attributes are useful when describing static data models; however HOOD recommends to restrict the use of public attributes since associated instance data would be accessible with almost no control by clients.

section 14.7 gives an illustration of ODS and class description with C++ as target language, whereas *section 14.8* gives an illustration of class description for an Ada target.

14.14.3 Description of Constants

A provided constant is declared in the provided interface and only its name is accessible by clients. Its structure is hidden to clients and is described in the INTERNALS where two cases may occur :

- the constant structure is fully described when the object is terminal,
- the constant structure is IMPLEMENTED_BY a provided constant of a child object when the object is not terminal.

For internal constants (which can only appear in terminal objects), the constant is declared and fully defined in the INTERNALS.

14.14.4 Description of Data

HOOD objects exchange data via parameters of operations. Data can only be declared in terminal objects, and are defined in the INTERNALS of the object as data items in target language syntax.

Data are either instances of :

14.14 DESCRIPTION OF TYPES, CLASSES, CONSTANTS AND DATA

HOOD types are used to define formally parameters and data exchanged between objects through their operations. In HOOD, types define only syntactic entities i.e. type checking is limited to name consistency checks.

A type, constant or data is specified by a **name** (its **declaration**) and a **structure** (its **definition**).Types, constants and data structures are defined using the target language syntax.

14.14.1 Specification of Types

For provided types, two cases may occur :

- the type is declared in the provided interface and only its name is accessible by clients. Its structure is hidden to clients[52] and will be described in the INTERNALS where two cases may occur :
  - for a terminal object, the type structure is fully described.
  - for a non terminal object, the type structure is IMPLEMENTED_BY a provided type of a child object.
- the type is declared in the provided interface with name and structure, which are thus fully accessible to clients. Two cases may occur :
  - for a terminal object, the type structure is fully described in the provided interface and there is no any additional description field in the INTERNALS.
  - for a non terminal object, the type structure is IMPLEMENTED_BY a provided type of a child object.

For internal types (which can only appear in terminal objects), the type is declared and fully defined in the INTERNALS.

14.14.2 Specification of Classes

A Class is a HOOD provided type that defines the parameter *me* of the class operations. Since a HOOD CLASS may provide other types, the type associated to the class is called the *main type*. In order to improve readability, it is recommended to name the main type:

[52]. This type structure hiding is similar to the Ada Private and Limited Private type concepts. It enforces abstract data type and object structured orientation.

14.13 NON TERMINAL VN ODS

VIRTUAL_NODE *Virtual_Node_Name* **is**
DESCRIPTION
--| Informal text describing WHAT the VN is doing in a few lines.|--
IMPLEMENTATION_CONSTRAINTS
--| Informal text. Describes any implementation constraints. |--
[**PRAGMA** ALLOCATED_TO *Physical_Node*]
PROVIDED _INTERFACE
OPERATIONS
Message_In [list of (*Parameter* : *Mode Type* [*:= Default_Value*])]
added with a textual description of how the *operation is triggered (by active polling or by interrupts*)
List of (*VN_Other_Provided_Operation_Nam*e [list of (*Parameter* : *Mode Type* [*:= Default_Value*])]
[**RETURN** *Type*] added with a textual description of WHAT operation does)

VN_CONTROL_STRUCTURE
DESCRIPTION
--| Informal text describing the client-server and communication protocols. |--
CONSTRAINED_OPERATIONS
Message_IN [**CONSTRAINED_BY ASER** [*Label_text*]]

REQUIRED _INTERFACE
VN_Name --| For all required virtual nodes in order to reflect graphical description.|--
DATAFLOWS
--| Standard fields |--
EXCEPTION_FLOWS
--| Standard fields |--

INTERNALS
--| Formal description of interface implementation.|--
VNs
List of (*Child_VN_Name* added with textual description)
OPERATIONS
List of (*Provided_Operation_Name* **IMPLEMENTED_BY** *Child_VN_Name.Operation_Name*)
END *Virtual_Node_Name*

14.12 TERMINAL VN ODS

The ODS of a terminal Virtual Node is the following:

VIRTUAL_NODE *Virtual_Node_Name* **is**
DESCRIPTION
--| Informal text describing WHAT the VN is doing in a few lines.|--
IMPLEMENTATION_CONSTRAINTS
--| Informal text. Describes any implementation constraints. |--
[**PRAGMA** ALLOCATED_TO *Physical_Node*]

PROVIDED _INTERFACE
OPERATIONS
Message_In [list of (*Parameter* : *Mode Type* [*:= Default_Value*])]
added with a textual description of how the *operation is triggered (by active polling or by interrupts*)
List of (*VN_Other_Provided_Operation_Nam*e [list of (*Parameter* : *Mode Type* [*:= Default_Value*])]
[**RETURN** *Type*] added with a textual description of WHAT operation does)

VN_CONTROL_STRUCTURE
DESCRIPTION
--| Informal text describing the client-server and communication protocols. |--
CONSTRAINED_OPERATIONS
Message_IN [**CONSTRAINED_BY ASER** [*Label_text*]]

REQUIRED _INTERFACE
VN_Name --| For all required virtual nodes in order to reflect graphical description.|--
DATAFLOWS
--| Standard field |--
EXCEPTION_FLOWS
--| Standard field |--
INTERNALS --| Formal description of object allocation.|--
OBJECTS
List of (*Allocated_Object_Name* added with textual description)
OBJECT_CONTROL_STRUCTURE
DESCRIPTION
--| Informal text describing the client-server and communication protocols. |--
CONSTRAINED_OPERATIONS
Message_IN [**CONSTRAINED_BY ASER** [*Label_text*]]
CODE
ClientVNCS
{*Client code common to all remote operation*}
ServerVNCS
{*Server code common to all remote operation*}
OPERATIONS
List of (*Provided_Operation_Name*
IMPLEMENTED_BY *VNCS.Operation_Name*[51] |
END *Virtual_Node_Name*

[51] In case of overloaded operation the full signature is given.

14.11 GENERIC INSTANCE ODS

An **instance** of a generic is declared as follows :

OBJECT|CLASS ***G****eneric_Instance_Name* **is INSTANCE_OF** *Generic_Name*
[**INSTANCE_RANGE** *lower_bound.upper_bound*]
PARAMETERS
TYPES
List of ({ *Formal*[49]*_Type_Name* => *Object_Name.Actual_Type_Name* }
added with textual description)
CONSTANTS
List of ({*Formal_Constant_Name* => *Object_Name.Actual_Constant_Name* => *Value*[50] }
added with textual description)
OPERATIONS
List of ({*Formal_Operation_Name* => *Object_Name.Actual_Operation_Name* }
added with textual description)
DESCRIPTION
--| Standard fields |--
IMPLEMENTATION_CONSTRAINTS
--| Standard fields |--
PROVIDED _INTERFACE
--| Standard fields |--
REQUIRED _INTERFACE
--| for all required objects needed to provide the actual parameters |--
OBJECT *Object_Name*
TYPES
List of (*Actual_Type_Name*)
CONSTANTS
List of (*Actual_Constant_Name*)
OPERATIONS
List of (*Actual_Operation_Name*)
DATAFLOWS
--| Standard fields |--
EXCEPTION_FLOWS
--| Standard fields |--
--| No internals since same as in class ODS |--

END *Generic_Instance_Name*

[49] This is the name as it is declared in the parameter fields of the generic ODS.
[50] The value shall be consistent with the constant type.

```
OPERATION GetCurrentSize (Me: in TStack) return HRTS_PE.T_Integer is
   DESCRIPTION
   Function to get the number of current items in the Stack
   USED_OPERATIONS              --NONE;
   PROPAGATED_EXCEPTIONS        --NONE;
   HANDLED_EXCEPTIONS           --NONE;
   PSEUDO_CODE
        {return CurrentSize attribute}
   CODE
        return CurrentSize;
END_OPERATION GetCurrentSize;
```

14.10 GENERIC DESCENDANT ODS

The ODS of an object or class whose parent or root is a generic[48], and which require formal parameters is the following:

```
OBJECT|CLASS Name is PASSIVE | ACTIVE
   --| Standard fields |--
REQUIRED_INTERFACE
        FORMAL_PARAMETERS
            TYPES
                List of (Formal_Type_Name )
            CONSTANTS
                List of (Formal_Constant_Name)
            OPERATIONS
                List of (Formal_Operation_Name)
        --| Other required objects can only be root objects of the
            current system configuration |--
   --| Standard ODS for the remaining fields |--
END Name
```

If the descendant does not requires any formal parameter, its ODS is reduced to a standard one.

[48]. Such an object could be named a descendant of the root, and is not to be confused with a descendant defined as an instance of a generic.

GetStatus **return** TPE::OK_KO;
EXCEPTIONS
X_FULL **RAISED_BY** PUSH when no more space.
X_EMPTY **RAISED_BY** POP when no more items

OBJECT_CONTROL_STRUCTURE
DESCRIPTION
The semantic of STACK is warranted by our implementation only if operations are called according to the possible sequences as expressed in the OSTD. NOte that the non determinism existing in state N_E_N_F is resolved in the OPCS body code of POP and PUSH.
OSTD
Object State Transition Diagram as given in *Figure 5 -*
CONSTRAINED_OPERATIONS
reset, Push**;** Pop;

REQUIRED_INTERFACE
OBJECT *HRTS_PE*
TYPES TPE::OK_KO; TPE::integer;

DATAFLOWS NONE; *{since we are a terminal object}--*

EXCEPTION_FLOWS NONE; {*since we are a terminal object}--*

INTERNALS --| *no object field since a class is always terminal* |--
TYPES
TStack
INHERITANCE
NONE;--no private inheritance
ATTRIBUTES
Status --| TPE::OK_KO Status |--
StackBuffer --|TItem StackBuffer[STACK_SIZE]; |--
FSM --|Stack_OSTM FSM;|--
CONSTANTS
MaxSize--| **enum** STACK_SIZE = K_MAX_SIZE}**;** // trick to define the array size of StackBuffer|--

OBJECT_CONTROL_STRUCTURE
CODE
OSTM -- |as generated from the OSTD
code as illustrated in *Appendix I.2.2 - OSTM code illustration in C++-*

OPERATION_ CONTROL _STRUCTURES -
OPERATION Tstack (Me: **in out** TStack) **is**
DESCRIPTION
constructor to initialize the STACKSize and status attributes;
USED_OPERATIONS --NONE;
PROPAGATED_EXCEPTIONS --NONE;
HANDLED_EXCEPTIONS --NONE;
PSEUDO_CODE
{Initialize CurrentSize to 0}
CODE
{ CurrentSize = 0;
Status = TPE::OK;
}
Opcs_HEADER
FSM.Fire (Stack_OSTM::START);
Opcs_FOOTER
NONE; *--{since we have non concurency constraints}--*
END_OPERATION TStack*;*

14.9 GENERIC ODS

A Generic **module** is declared as follows :

OBJECT | CLASS *Generic_name* **is PASSIVE** | **ACTIVE**
FORMAL_PARAMETERS
TYPES
List of ({*Formal*[47]*_Type_ or_ Class_Declaration* }
added with textual description)
CONSTANTS
List of ({*Formal_Constant_Declaration* }
added with textual description)
OPERATIONS
List of ({*Formal_Operation_Declaration* }
added with textual description)
--| Standard ODS fields where required objects may be siblings or roots of the current system configuration |--
END *Generic_name*

Below is an illustration of a generic C++ Stack ODS, first defined in the ODS in *section 14.7*

CLASS G_TStack **is PASSIVE**
FORMAL_PARAMETERS
TYPES
TItem
CONSTANTS
K_STACK_SIZE;
OPERATIONS
operator=(TItem);
DESCRIPTION
Generic Abstract Data Type as illustrated in *Figure 3 - Graphical Representation of a HOOD Object*
IMPLEMENTATION_CONSTRAINTS
Use of C++ language
PROVIDED_INTERFACE \
TYPES
TStack --|the main type of the class has the same name|--
INHERITANCE NONE;
ATTRIBUTES --one for illustrations
CurrentSize;
--| **TPE::integer** CurrentSize |--
rTItem--|pointer onItem |--
--| **typedef** TItem* rTItem |--
rTStack --|reference on Stacks useful when in C++|--
--| **typedef** TStack& pTStack |--
CONSTANTS NONE;
OPERATION_SETS NONE;
OPERATIONS
TStack (**me: in out** TStack;); -- constructor
~TStack(**me: in** TStack;); -- destructor
Push(**me: in out** TStack; item**: in r**TItem);
Pop (**me: in out** TStack; item: **out r**TItem);
Reset();
GetMaxSize **return** TPE::integer;
GetCurrentSize **return** TPE::OK_KO;

[47] The formal parameter declarations are target language dependent. By default the Ada syntax for generic parameters is used [see also BNF description in appendix D and E].

```
            IPCProcess (Me);
         Opcs_SER
               Item: HRTS_PE.T_Integer;
               StringItem: String(1.. HRTS_PE.T_Integer'width);
               begin
                  TParams.Read(Params,StringItem);
                  item:=HRTS_PE.T_Integer'value(StringItem);
                  TStack_Server.Push (TheStack, Item);
   END_OPERATION Push;

   OPERATION Pop (Me: in out Instance) is
         --{Similar description as for Push}--
   END_OPERATION Pop;

   OPERATION GetMaxSize (Me: in Instance) return HRTS_PE.T_Integer is
         --{Similar description as for GetCurrentSize}--
   END_OPERATION GetMaxSize;

   OPERATION GetStatus (Me: in Instance) return HRTS_PE.T_OK_KO is
         --{Similar description as for GetCurrentSize}--
   END_OPERATION GetStatus;

 OPERATION Stop (Me: in out Instance) is
      --{Similar description as for Start}--
   END_OPERATION Stop;

END TStack;
```

```
OPERATION GetCurrentSize (Me: in Instance) return HRTS_PE.T_Integer is
   DESCRIPTION
   Function to get the number of current items in the Stack
   USED_OPERATIONS                --NONE;
   PROPAGATED_EXCEPTIONS      --NONE;
   HANDLED_EXCEPTIONS           --NONE;
   PSEUDO_CODE
        {return CurrentSize attribute}
   CODE
        return Me.CurrentSize;
      Opcs_HEADER
           TFsm.Fire (Me.FSM, Stack_OSTM.GetCurrentSize);
      Opcs_FOOTER
           NONE; --{since we have non concurrency constraints}--
      Opcs_ER
          Size: HRTS_PE.T_Integer;
          begin
              TMsg.Initialize     (Me         => Me.Message,     Sender     => Stack_PE.STACK,
                    Sendee     => Stack_PE.STACK_RB,     Operation   => Stack_PE.GETCURRENTSIZE,
                    Cnstrt     => HRTS_PE.NO_CONSTRAINT,     ParamSize   => 1);
                IPCProcess (Me);
                Params:=TMsg.GetParams(Me.Message);
                TParams.Read(Params,StringParam);
                Size:=HRTS_PE.T_Integer'value (StringParam);
                return Size;
      Opcs_SER
           CurrentSize: HRTS_PE.T_Integer;
            begin
             CurrentSize:= TStack_Server.GetCurrentSize (TheStack);
            TParams.Write(Params,HRTS_PE.T_Integer'image(CurrentSize);
END_OPERATION GetCurrentSize;

 OPERATION Push(Me: in out Instance) is
   DESCRIPTION
   Put the Item at top of the STACK
   USED_OPERATIONS                --NONE;
   PROPAGATED_EXCEPTIONS      --NONE;
   HANDLED_EXCEPTIONS           --NONE;
   PSEUDO_CODE
        {Put Item on top of STACK
        Increment CurrentSize
        If CurrentSize =StackSize then SET_SATE to FULL end if;}
   CODE
          Me.StackBuffer (Me.CurrentSize):= MyItem;
          Me.CurrentSize:= Me.CurrentSize + 1;
           if (Me.CurrentSize = StackSize) then
              TFsm.Set (Me.FSM, Stack_OSTM.FULL);
          end if;
      Opcs_HEADER
           TFsm.Fire (Me.FSM, Stack_OSTM.PUSH);
      Opcs_FOOTER
           NONE; --{since we have non concurrency constraints}--
      Opcs_ER
          TMsg.Initialize     (Me   => Me.Message,     Sender => Stack_PE.STACK,
           Sendee     => Stack_PE.STACK_RB,     Operation   => Stack_PE.PUSH,
           Cnstrt     => HRTS_PE.HSER, ParamSize   => 1);
          Params:=TMsg.GetParams(Me.Message);
          Tparams.write(Params,HRTS_PE.T_Integer'image(MyItem);
```

```
      when HRTS_PE.E_BadExecutionRequest => EXCEPTIONS.LOG(
           'TStack_RB.Dispatcher_HSER'& HRTS_PE.T_X_VALUE'image (HRTS_PE.X_BADREQUEST));
           TMsg.SetX (Message, HRTS_PE.X_BadExecutionRequest);
      when HRTS_PE.E_UNKNOWN_OPERATION => EXCEPTIONS.LOG(
           'TStack_RB.Dispatcher_HSER'& HRTS_PE.T_X_VALUE'image (X_UNKNOWN_OPERATION));
           TMsg.SetX (Message, HRTS_PE.X_UNKNOWN_OPERATION);
      when others =>   EXCEPTIONS.LOG(
       'TStack_RB.Dispatcher_HSER'»&HRTS_PE.T_X_VALUE'image(HRTS_PE.X_UNKNOWN_ERROR));
         TMsg.SetX (Message, HRTS_PE.X_UNKNOWN_ERROR);
       end;
       end RB_Dispatcher_HSER;

      procedure RB_Dispatcher_LSER (aMessage in out TMsg.TMsg) is
      begin
       case TMsg.GetOperation (aMessage) is
           --{similar code as for RB_Dispatcher_HSER
      end RB_Dispatcher_LSER;
      task body Server_OBCS  is
      begin   -- at package elaboration
       loop
           accept ExecuteRequest (Message: in out TMsg.TMsg) do--TServerObcs.Remove (OBCS, Message);
           if TMsg.getSender(Message) /=Stack_PE.STACK then
               raise HRTS_PE.E_UNKNOWN_SENDEE;
           end if;
           Params:=TMsg.GetParams(Message);
           Constraint:=TMsg.GetCnstrt(Message);
           Operation:=TMsg.getOperation(Message);
           case Constraint is
           whenHRTS_PE.HSER=> RB_Dispatcher_HSER(Message); IPCFormat(Message);
           whenHRTS_PE.LSER=> IPCFormat(Message);RB_Dispatcher_LSER(Message);
           whenHRTS_PE.RASER=>null; --TBS;
           whenHRTS_PE.RLSER=>null; --TBS;
            when others     =>   EXCEPTIONS.LOG(
           'TStack_RB.Dispatcher_ASER'& HRTS_PE.T_X_VALUE'image (X_UNKNOWN_CONSTRAINT));
           TMsg.SetX (Message, HRTS_PE.X_UNKNOWN_CONSTRAINT);
           end case;
           end ExecuteRequest;
      end loop;
    end Server_OBCS;

  OPERATION_ CONTROL _STRUCTURES -
     OPERATION Start (Me: in out Instance) is
       DESCRIPTION
       The first operation to call by a client to initialize the STACKSize;
       USED_OPERATIONS                --NONE;
       PROPAGATED_EXCEPTIONS      --NONE;
       HANDLED_EXCEPTIONS           --NONE;
       PSEUDO_CODE
            {Initialize CurrentSize to 0}
       CODE
            Me.CurrentSize:= 0;
           Opcs_HEADER
                TFsm.Fire (Me.FSM, Stack_OSTM.START);
           Opcs_FOOTER
                NONE; --{since we have non concurency constraints}--
           Opcs_ER
               TMsg.Initialize     (Me         => Me.Message,     Sender     => Stack_PE.STACK,
                Sendee     => Stack_PE.STACK_RB,     Operation   => Stack_PE.START,
                Cnstrt     => HRTS_PE.NO_CONSTRAINT,     ParamSize   => 0);
               IPCProcess (Me);
           Opcs_SER
                  TStack_Server.Start(TheStack);
    END_OPERATION Start;
```

```
INTERNALS                    --| no object field since a class is always terminal |--
   TYPES
   Instance
      INHERITANCE --no private inheritance

      ATTRIBUTES
         CurrentSize: HRTS_PE.T_Integer:= 0;
         StackBuffer: T_StackBuffer;
         FSM: Stack_OSTM.TFsm.Instance:= Stack_OSTM.Stack_FSM;
   CONSTANTS
      StackSize: constant HRTS_PE.T_Integer:= 20;
  OPERATIONS
      NONE;
  DATA
      Current_Instances : HRTS_PE.T_Integer:=0;
OBJECT_CONTROL_STRUCTURE
   CODE
      OSTM        -- |as generated from the OSTD
         same code as illustrated in Appendix I.2.1 - OSTM Code Illustration in Ada-

   Client_OBCS   --{code directly include in the body of package TSTack, on client side}--
         Params: TParams.PtrInstance;
         StringParam: String(1.. HRTS_PE.T_Integer'width);
      procedure IPCProcess (Me: in Instance) is
         Status: HRTS_PE.T_X_VALUE;
      begin
         TClientObcs.Insrem (Me.OBCS, Me.Message); -- insert message then remove returned message:
         Status:= TMsg.GetX (Me.Message);
         case Status is
         when HRTS_PE.X_KO =>                        raise HRTS_PE.E_KO;
         when HRTS_PE.X_UNKNOWN_SENDER =>        raise HRTS_PE.E_UNKNOWN_SENDER;
         when HRTS_PE.X_UNKNOWN_SENDEE =>        raise HRTS_PE.E_UNKNOWN_SENDEE;
         when HRTS_PE.X_UNKNOWN_OPERATION =>  raise HRTS_PE.E_UNKNOWN_OPERATION;
         when HRTS_PE.X_BADREQUEST =>               raise HRTS_PE.E_BADREQUEST;
         when HRTS_PE.X_FSMERROR =>                 raise HRTS_PE.E_FSMERROR;
         when HRTS_PE.X_OBCS_NOMOREQUEUES => raise HRTS_PE.E_OBCS_NOMOREQUEUES;
         when HRTS_PE.X_UNKNOWN_ERROR =>        raise HRTS_PE.E_UNKNOWN_ERROR;
         end case;
      end IPCProcess;

   Server_OBCS   -- {code to be directly included in package TStack_RB}--
      task Server_OBCS  is
         entry ExecuteRequest(message: in out TMsg.Msg);
      end Server_OBCS;

      procedure IPCMsgFormat(aMessage: in out TMsg.TMsg) is --common code to all operations
         PreviousSender: Stack_PE.T_HOODObject:= TMsg.GetSender (Message);
      begin
         TMsg.SetSender (aMessage, TMsg.GetSendee (Message));
         TMsg.SetSendee (aMessage, PreviousSender);
         PtrStream:= TMsg.GetParams (Message);
         TMsg.FlushParams (Message); -- enforce writing of parameters in the stream
      end IPCMsgFormat;
      procedure RB_Dispatcher_HSER (aMessage in out TMsg.TMsg) is
      begin
       case TMsg.GetOperation (aMessage) is
         when Stack_PE.START   =>        Start;
         when Stack_PE.STOP      =>       Stop;
         when Stack_PE.PUSH      =>       Push;
         when Stack_PE.POP       =>       Pop;
         when Stack_PE.GETMAXSIZE   =>   GetMaxSize;
         when Stack_PE.GETCURRENTSIZE   => GetCurrentSize;
         when Stack_PE.GETSTATUS   => GetStatus;
         when others         =>raise HRTS_PE.E_UNKNOWN_OPERATION;
       end case;
       exception
```

14.8 CLASS ODS ADA ILLUSTRATION

CLASS TStack **is ACTIVE**
DESCRIPTION
Stack Abstract Data Type as illustrated in *Figure 3 - Graphical Representation of a HOOD Object*
IMPLEMENTATION_CONSTRAINTS
Use of Ada classes with run time supporting tasking for protocol constrained operations
PROVIDED_INTERFACE \
TYPES
Instance --|the name of the class is Instance as we are used to the naming conventions suggested in*[ROSEN95]*|--
INHERITANCE --public inheritance
TStorage.Instance;
ATTRIBUTES --no public attributes
NONE;
CONSTANTS NONE;
OPERATION_SETS -- for illustration purposes
Start (Me: **in out** Instance) is **MEMBER_OF** TStack_Ops;
Stop (Me: **in out** Instance) is **MEMBER_OF** TStack_Ops;
Push (Me: **in out** Instance;MyItem: **in** HRTS_PE.T_Integer) is **MEMBER_OF** TStack_Ops;
Pop(Me: **in out** Instance;AnItem: **out** HRTS_PE.T_Integer) is **MEMBER_OF** TStack_Ops;
GetCurrentSize (Me: **in out** Instance) is **MEMBER_OF** TStack_Ops;
OPERATIONS
Start (Me: **in out** Instance);
Stop (Me: **in out** Instance);
Push (Me: **in out** Instance; MyItem: **in** HRTS_PE.T_Integer);
Pop (Me: **in out** Instance; AnItem: **out** HRTS_PE.T_Integer);
GetMaxSize (Me: **in** Instance) **return** HRTS_PE.T_Integer;
GetCurrentSize (Me: **in** Instance) **return** HRTS_PE.T_Integer;
GetStatus (Me: **in** Instance) **return** HRTS_PE.OK_KO;
EXCEPTIONS
X_FULL **RAISED_BY** PUSH when no more space.
X_EMPTY **RAISED_BY** POP when no more items
OBJECT_CONTROL_STRUCTURE
DESCRIPTION
The semantic of STACK is warranted by our implementation only if operations are called according to the possible sequences as expressed in the OSTD. NOte that the non determinism existing in state N_E_N_F is resolved in the OPCS body code of POP and PUSH.
OSTD
Object State Transition Diagram as given in *Figure 5* -
CONSTRAINED_OPERATIONS
Start,
Stop,
PUSH **CONSTRAINED_BY** HSER;
PUSH **CONSTRAINED_BY** LSER;
REQUIRED_INTERFACE
OBJECT *HRTS_PE*
TYPES List of (*Type_Name*)
CONSTANTS List of (*Constant_Name*)
OPERATIONS List of (*Operation_Name* [list of (*Parameter* :*Mode Type*)])

DATAFLOWS NONE; *{since we are a terminal object}--*

EXCEPTION_FLOWS NONE; {*since we are a terminal object}--*


```
   OPERATION Push(Me: in out Instance; item: in TString) is
     DESCRIPTION
     Put the Item at top of the STACK
     USED_OPERATIONS              --NONE;
     PROPAGATED_EXCEPTIONS        --NONE;
     HANDLED_EXCEPTIONS           --NONE;
     PSEUDO_CODE
          {Put Item on top of STACK
          Increment CurrentSize
          If CurrentSize =StackSize then SET_SATE to FULL end if;}
     CODE
   {if (CurrentSize < TStack::K_STACK_SIZE)  {
          StackBuffer[CurrentSize] = item;
          CurrentSize          = CurrentSize + 1;
          Status               = TPE::OK;
     }  else  {
          cerr << "STACK SIZE=" << K_STACK_SIZE  << endl << "CurrentSize="  << CurrentSize   << endl;
          Status = TPE::KO;
          EXCEPTIONS_RAISE(PROJECT_PE::X_FULL, "TStack_SERVER::Push",   "X_FULL");
     }
     }
          Opcs_HEADER
               FSM.Fire (Stack_OSTM::PUSH);
          Opcs_FOOTER
               NONE; --{since we have non concurency constraints}--
   END_OPERATION Push;

   OPERATION Pop (Me: in out TStack; item: out TString) is
          --{Similar description as for Push}--
   END_OPERATION Pop;

   OPERATION GetMaxSize (Me: in TStack)) return HRTS_PE.T_Integer is
          --{Similar description as for GetCurrentSize}--
   END_OPERATION GetMaxSize;

   OPERATION GetStatus (Me: in TStack)) return HRTS_PE.T_OK_KO is
          --{Similar description as for GetCurrentSize}--
   END_OPERATION GetStatus;

   OPERATION ~TStack (Me: in out TStack)) is
     --{Similar description as for Start}--
   END_OPERATION Stop;

END TStack;
```

INTERNALS --| *no object field since a class is always terminal* |--

TYPES
TStack
INHERITANCE
NONE;--no private inheritance
ATTRIBUTES
Status --| TPE::OK_KO Status |--
StackBuffer --|TString StackBuffer[K_STACK_SIZE]; |--
FSM --|Stack_OSTM FSM;|--

CONSTANTS
MaxSize--| **enum** **{K_STACK_SIZE = 512};** // trick to define the array size of StackBuffer|--

OBJECT_CONTROL_STRUCTURE
CODE
OSTM -- |as generated from the OSTD
code as illustrated in *Appendix I.2.2 - OSTM code illustration in C++-*

OPERATION_ CONTROL _STRUCTURES -

OPERATION Tstack (Me: **in out** TStack) **is**
DESCRIPTION
constructor to initialize the STACKSize and status attributes;
USED_OPERATIONS --NONE;
PROPAGATED_EXCEPTIONS --NONE;
HANDLED_EXCEPTIONS --NONE;
PSEUDO_CODE
{Initialize CurrentSize to 0}
CODE
{ CurrentSize = 0;
Status = TPE::OK;
}
Opcs_HEADER
FSM.Fire (Stack_OSTM::START);
Opcs_FOOTER
NONE; --*{since we have non concurency constraints}*--
END_OPERATION TStack*;*

OPERATION GetCurrentSize (Me: **in** TStack) **return** HRTS_PE.T_Integer **is**
DESCRIPTION
Function to get the number of current items in the Stack
USED_OPERATIONS --NONE;
PROPAGATED_EXCEPTIONS --NONE;
HANDLED_EXCEPTIONS --NONE;
PSEUDO_CODE
{return CurrentSize attribute}
CODE
return CurrentSize;
END_OPERATION GetCurrentSize*;*

14.7 CLASS ODS-C++ ILLUSTRATION

CLASS TStack **is PASSIVE**
DESCRIPTION
Stack Abstract Data Type as illustrated in *Figure 3 - Graphical Representation of a HOOD Object*
IMPLEMENTATION_CONSTRAINTS
Use of C++ language
PROVIDED_INTERFACE \
TYPES
TStack --|the main type of the class has the same name|--
INHERITANCE NONE;
ATTRIBUTES --one for illustrations
CurrentSize;
--| **TPE::integer** CurrentSize |--
pTStack --|pointer on Stacks |--
--| **typedef TStack* pTStack** |--
rTStack --|reference on Stacks useful when in C++|--
--| **typedef TStack& pTStack** |--
CONSTANTS NONE;
OPERATION_SETS NONE;
OPERATIONS
TStack (**me: in out** TStack;); -- constructor
~TStack(**me: in** TStack;); -- destructor
Push(**me: in out** TStack; item**: in r**Tstring);
Pop (**me: in out** TStack; item: **out r**TString);
Reset();
GetMaxSize **return** TPE::integer;
GetCurrentSize **return** TPE::OK_KO;
GetStatus **return** TPE::OK_KO;
EXCEPTIONS
X_FULL **RAISED_BY** PUSH when no more space.
X_EMPTY **RAISED_BY** POP when no more items
OBJECT_CONTROL_STRUCTURE
DESCRIPTION
The semantic of STACK is warranted by our implementation only if operations are called according to the possible sequences as expressed in the OSTD. NOte that the non determinism existing in state N_E_N_F is resolved in the OPCS body code of POP and PUSH.
OSTD
Object State Transition Diagram as given in *Figure 5* -
CONSTRAINED_OPERATIONS
reset, Push**;** Pop;
REQUIRED_INTERFACE
OBJECT *HRTS_PE*
TYPES TPE::OK_KO; TPE::OK_KO;

DATAFLOWS NONE; *{since we are a terminal object}*--

EXCEPTION_FLOWS NONE; {*since we are a terminal object}*--

14.3 NON TERMINAL ODS

OBJECT *Object_Name* **IS ACTIVE|PASSIVE**
--| Standard fields |--
INTERNALS -- describe the implemented_by links of PROVIDED INTERFACE Items
OBJECTS
List of (*Child_Object_Name*)
TYPES
List of (*Provided_Type_Name* **IMPLEMENTED_BY** *Child_Object_Name.Type_Name*)
CONSTANTS
List of (*Provided_Constant_Name* **IMPLEMENTED_BY** *Child_Object_Name.Constant_Name*)
OPERATION_SETS
List of (*Provided_Operation_Set_Name* **IMPLEMENTED_BY** *Child_Object_Name.Operation_Set*)
OPERATIONS
List of (*Provided_Operation_Name* **IMPLEMENTED_BY** *Child_Object_Name.Operation_Name*[46] |
IMPLEMENTED_BY *OP_Control_Name*)
EXCEPTIONS
List of (*Provided_Exception_Name* **IMPLEMENTED_BY** *Child_Object_Name.Exception_name*)
OBJECT_CONTROL_STRUCTURE **IMPLEMENTED_BY** List of (*Child_Object_Name*)
END *Object_Name*

14.4 ENVIRONMENT ODS

OBJECT | CLASS *Name* **is ENVIRONMENT PASSIVE| ACTIVE**
--| Standard fields |-- --{but NO INTERNALs PART}--
END name

14.5 ROOT ODS

A standard ODS is associated to a root, but required objects can only be roots within the current system configuration.

14.6 OP_CONTROL ODS

OP_CONTROL *OP_Control_Name* **is**
--| Standard fields |--
INTERNALS --{reduced to one OPCS}--
OPERATION_ CONTROL_STRUCTURES
--|operation description for the unique provided operation
END *OP_Control_Name.*

[46]. In case of overloaded operation the full signature is given.

```
REQUIRED_INTERFACE
   OBJECT HRTS_PE
         TYPES                    T_Integer, T_OK_KO;

DATAFLOWS      NONE; {since we are a terminal object}--

EXCEPTION_FLOWS      NONE; {since we are a terminal object}--
INTERNALS                  --| no object field since a class is always terminal |--
   TYPES
   T_StackBuffer; --|type T_StackBuffer is array (HRTS_PE.T_Integer range <>) of HRTS_PE.T_Integer;|--
   T_Stack; --|type T_Stack (Size: HRTS_PE.T_Integer) is record
                  Status: HRTS_PE.T_OK_KO:= HRTS_PE.OK
                  CurrentSize: HRTS_PE.T_Integer:= 0;
                  StackBuffer: T_StackBuffer (0.. Size);
               end record; |--
   CONSTANTS NONE
   OPERATIONS NONE;
   DATA   NONE;
   OBJECT_CONTROL_STRUCTURE NONE;
  OPERATION_ CONTROL _STRUCTURES -
   OPERATION GetCurrentSize (Me: in T_Stack) return HRTS_PE.T_Integer is
      DESCRIPTION
      Function to get the number of current items in the Stack
      USED_OPERATIONS              --NONE;
      PROPAGATED_EXCEPTIONS        --NONE;
      HANDLED_EXCEPTIONS           --NONE;
      PSEUDO_CODE
           {return CurrentSize attribute}
      CODE
           return Me.CurrentSize;
   END_OPERATION GetCurrentSize;

    OPERATION Push(Me: in out T_Stack) is
      DESCRIPTION
      Put the Item at top of the STACK
      USED_OPERATIONS              --NONE;
      PROPAGATED_EXCEPTIONS        --NONE;
      HANDLED_EXCEPTIONS           --NONE;
      PSEUDO_CODE
           {Put Item on top of STACK
           Increment CurrentSize
           If CurrentSize =StackSize then SET_SATE to FULL end if;}
      CODE
              Me.StackBuffer (Me.CurrentSize):= MyItem;
              Me.CurrentSize:= Me.CurrentSize + 1;
   END_OPERATION Push;

   OPERATION Pop (Me: in out T_Stack) is
              --{Similar description as for Push}--
   END_OPERATION Pop;
   OPERATION GetMaxSize (Me: in T_Stack) return HRTS_PE.T_Integer is
              --{Similar description as for GetCurrentSize}--
   END_OPERATION GetMaxSize;
   OPERATION GetStatus (Me: in T_Stack) return HRTS_PE.T_OK_KO is
              --{Similar description as for GetCurrentSize}--
   END_OPERATION GetStatus;
END TStack;
```

Object, operation, types and constants names shall be defined unambiguously, possibly using the dotted notation e.g. <object>.<operation>. In order to improve readability of ODSs :

- Informal text may be added where indicated, allowing to comment in natural language onto more formalized descriptions,
- Most of the fields in the ODS are optional, i.e. **if empty,** they are documented by the keyword **NONE**, or fully **removed**.

In the following, we shall give an illustration of the STACK (described in *Figure 3 - Graphical Representation of a HOOD Object*) and give different possible ODS configurations and **layouts**. We try to enforce the following naming conventions:

- a type identifier is always prefixed by “T_”
- a class identifier is always named “Instance” in Ada according to the naming and overloading schema suggested in *[ROSEN95]*. A class identifier is otherwise always the same as the class name and prefixed by “T”
- an exception identifier is always prefixed by “X_”
- a pointer identifier is always prefixed by “p”
- a reference identifier is always prefixed by “r”

14.2 TERMINAL ODS ILLUSTRATION

OBJECT TStack **is PASSIVE**
DESCRIPTION
Stack Abstract Data Type Manager as illustrated in *Figure 3 - Graphical Representation of a HOOD Object*
Not any constraints applied to provided operations
IMPLEMENTATION_CONSTRAINTS
Use of Ada supposing unlimited memory
PROVIDED_INTERFACE \
TYPES
T_Stack --|the name of the class is Instance as we are used to the naming conventions suggested in*[ROSEN95]*|--
--| **T_Stack (Size: HRTS_PE.T_Integer) is private**|--
CONSTANTS **NONE;**
OPERATION_SETS NONE;
OPERATIONS
Push (Me: **in out T_Stack**; MyItem: **in** HRTS_PE.T_Integer);
Pop (Me: **in out T_Stack**; AnItem: **out** HRTS_PE.T_Integer);
GetMaxSize (Me: **in T_Stack**) **return** HRTS_PE.T_Integer;
GetCurrentSize (Me: **in T_Stack**) **return** HRTS_PE.T_Integer;
GetStatus (Me: **in T_Stack**) **return** HRTS_PE.T_OK_KO;
EXCEPTIONS
X_FULL **RAISED_BY** PUSH when no more space.
X_EMPTY **RAISED_BY** POP when no more items
OBJECT_CONTROL_STRUCTURE
NONE; --{not any state or other constraints, for tutorial!}--

14 TEXTUAL FORMALISM

The use of a formal notation in the HOOD design process for capturing object properties aims at :

- allowing smooth successive transformations from a conceptual model of a solution down to the source code of the target implementation language,
- formalizing the design to avoid any imprecisions or ambiguities of the designer's natural language and to allow automated processing by tools for verification,
- providing a **Standard Interchange Format** (SIF) for exchanging design data between toolsets and platforms, as defined in *Appendix G - Standard Interchange Format- .*

The process of formalisation is a step-wise one, and should not hinder the creative process of the designer by enforcing too early implementation decisions. At the same time, it should allow to capture both structural and dynamic properties of the system.

The language used to describe a HOOD design is defined below. It enforces the following principles :

- formal definition of object interfaces and control structures so that verifications and checks can be supported by tools,
- expression of operations and their control description in pseudo code and in target language,
- use of informal text whenever the level of detail is not sufficient to justify the use of a formal notation.

14.1 DESCRIPTION

The description of objects and classes in HOOD is produced using an **Object Description Skeleton** (**ODS**) whose fields are updated as the design of the unit is more and more refined. The general structure of the ODS was already introduced in *section 3.3* above. Each field of an ODS is described by the **HOOD Design Language** (**HOOD DL**) according to a set of keywords and a grammar formally defined in BNF notation in *Appendix G - Standard Interchange Format- .*

13 CONCEPTS SUMMARY

Figure 47 - below summarizes the HOOD entities and relationships using an OMT like graphical representation *[OMT91]*

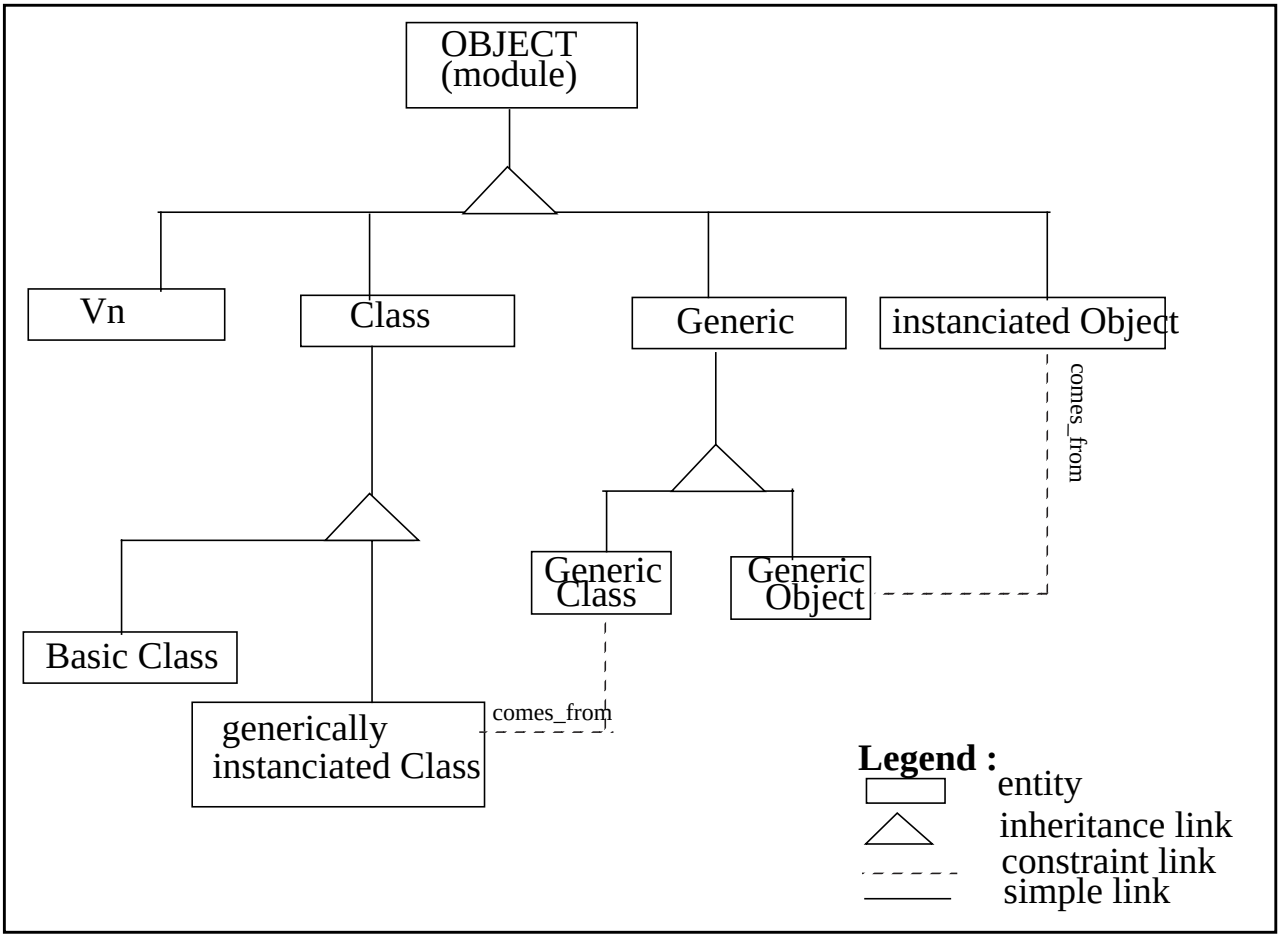

Figure 47 - Summary of HOOD Entities

One can see that the term object with the meaning of “software module”refers either to:

- a basic software module
- a VN
- a class (either basic or instantiated class)
- a generic (either generic class or generic object)
- an instanced object (from a generic object)

Configuration of the VNCS shall be done by defining such a configuration table in the textual formalism as more detailed in *section 14* below.

12.3.2 Partitioning Rules

Allocation of objects from a HDT among several VNs shall follow a number of rules that enforce the following principles:

- invariance of functional semantics
- efficiency consideration that limit communication bottlenecks
- Pure objects (objects having no internal data) may be replicated in several VNs
- Some restrictions may apply on operation parameters: especially on acces types and references. This is due to non direct acces of local (respectively remote) memory space by a remote (respectively local) processor.
- additional communication handlers for inter-VN communication support should be automatically generated from object allocation data

12.3.3 Configuration Rules

Configuration is the grouping of VNs into PNs according to:

- the number of processors and their capabilities
- the functional semantics
- each PN should at least include one active VN

It is possible to model the physical network by choosing a simple one-to-one mapping between terminal VNs and physical nodes. *Figure 46* - gives an example of a configuration table of VNs for a physical architecture of three PNs where VN4 is a common service and is duplicated[45] on PN1,PN2and PN33 for efficiency reasons.

Physical Nodes / Virtual Nodes	PN1	PN2	PN31I	PN32	PN33
VN4	X	X			X
VN1	X				
VN2			X		
VN3				X	

Figure 46 - Configuration of VNs to Physical Nodes

This allocation could now be redefined by defining two new PNs: P1_VN and P2_VN and allocating all objects of VNi nodes onto associated P1 or P2 VNs.

[45]. In such a case, care must be taken on data share between clients of VN4.

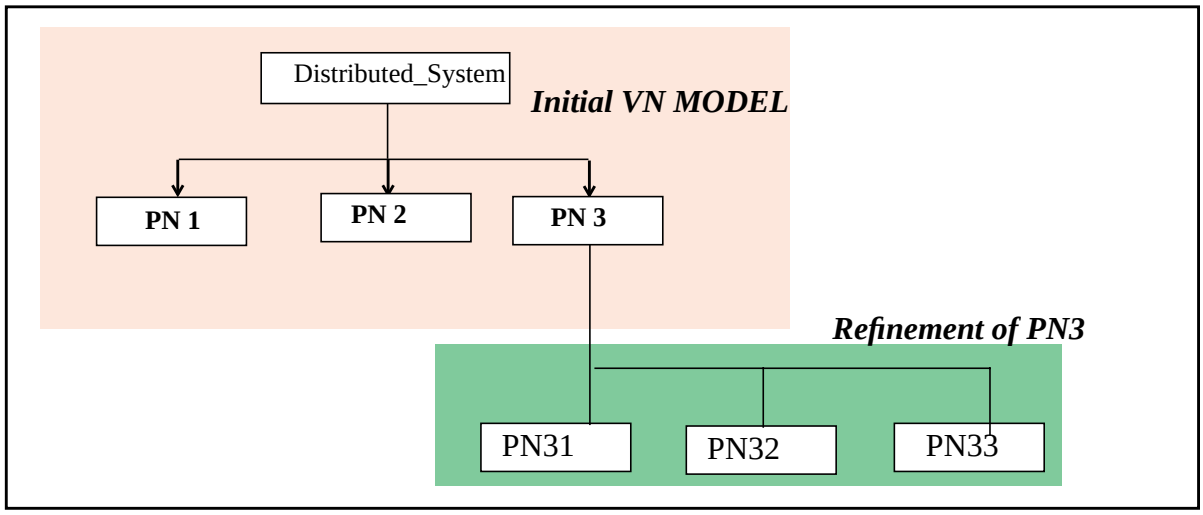

Figure 43 - PN tree modelling a Target Distributed System with three PN

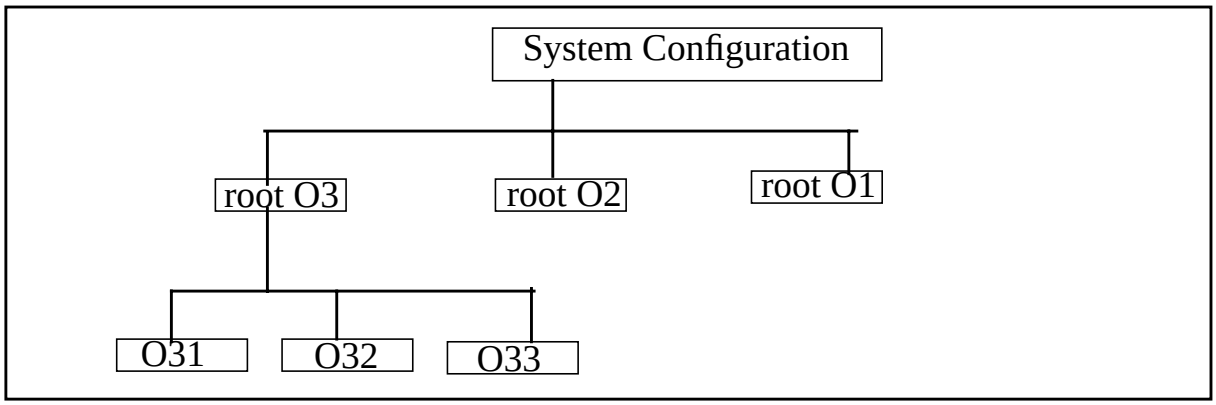

Figure 44 - Logical view associated to a system configuration

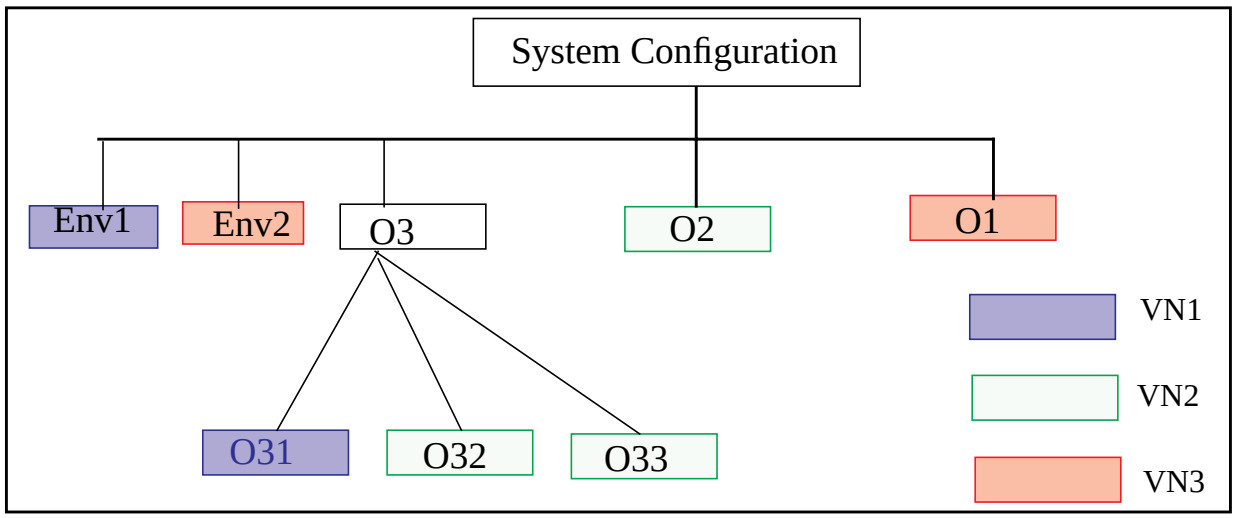

Figure 45 - Allocation view

12.3 DEFINING VN HIERARCHIES

In order to manage complexity, a distributed application is structured into a VN hierarchy by allocating HOOD objects and classes from the different HDTs defining the system configuration onto terminal VNs. *Thus merely terminal VNs will be the focus of object allocation and code implementation*, but parent VNs, defined as the composition of child VNs, may be used as an additional abstraction tool for representing large distributed systems.

12.3.1 Partitioning Process

Several terminal VNs may be allocated and configured onto one physical node, whereas a terminal VN may be “duplicated” (but not allocated) onto several processors. The allocation process is thus the following:

- A physical network may be first[44] modelled as a PN tree as in *Figure 43* -.Such a network may be later refined by replacing PN3 by three more processors PN31, PN32 and PN33.
- Second, objects and classes of a system configuration, as the one illustrated in *Figure 44* -, are allocated on a VN hierarchy. Such a hierarchy should ultimately be mapped on the PN one; if have more VN than PN, we may allocate several VNs on the same PN, if we less VNs than PNs, we may:
  - duplicate some VNs or
  - break down some terminal objects in the HDT again, and possibly coming to more VNs
- Third, optimization according to services (types, classes, operations, exceptions) and communication protocols shared by different VNs may lead to allocation/duplication and/or definition of additional VNs.

When a VN is to be distributed over several PN, the allocation of HOOD objects may have to be redefined (possibly after further breakdown of some objects in the HDT) into as much child VNs as necessary to allocate at least one VN to a PN.

Figure 44 - shows an example of a system configuration in a logical view, whereas *Figure 45* - illustrates a possible allocation schema.

[44]. The basic hypothesis would be to have a one-to-one mapping between a VN and a physical node.

objects or one for a full VN, and similarly in the Server VN. Such structures can be traded off according to resulting efficiency.

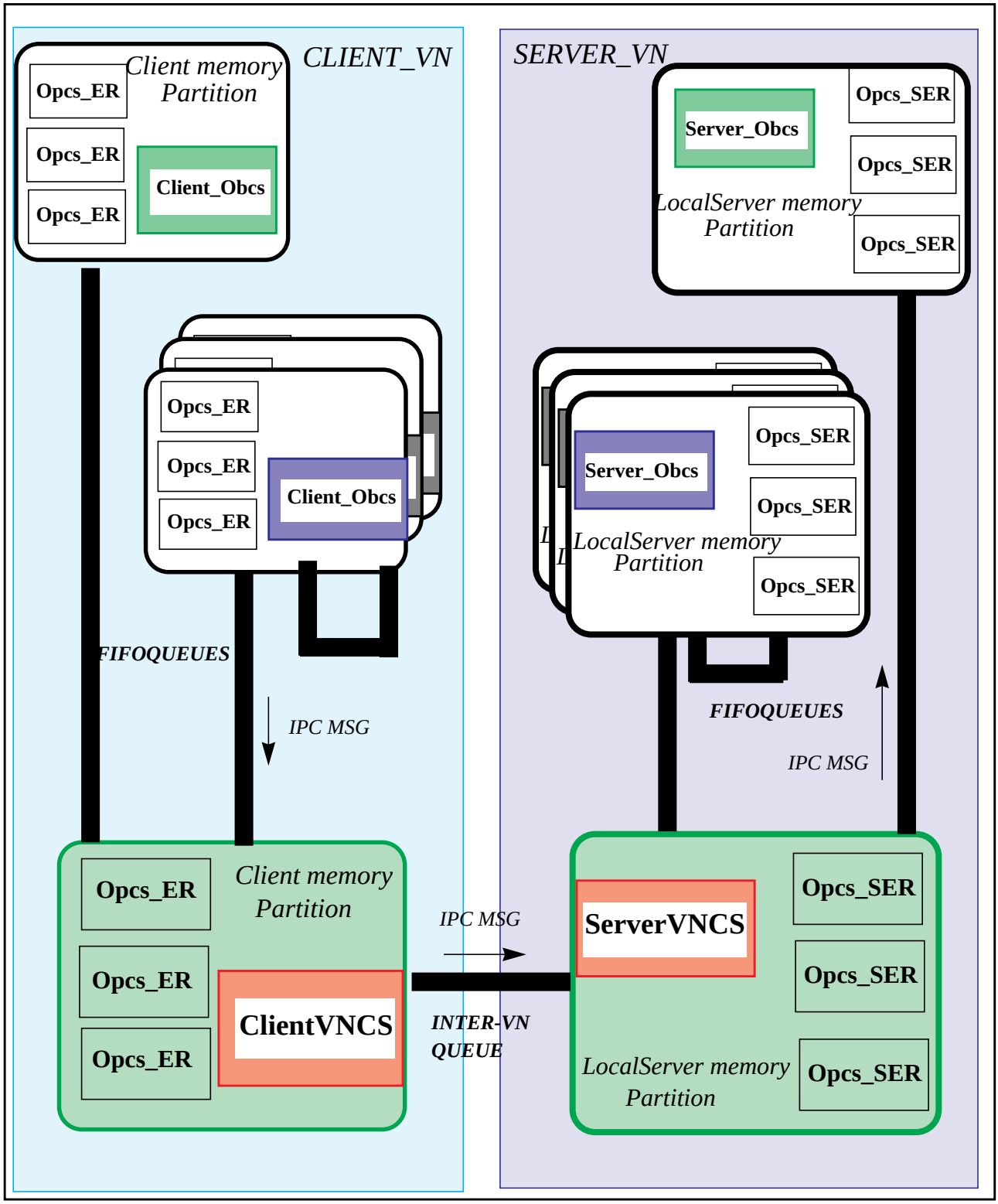

Figure 42 - Communication Optimization Principles between VNs

Figure 41 -illustrates the resulting target code structure for a VN, where:

- ClientVNCS and ServerVNCS units are automatically configured according to a VN_AllocationTable, directly computed from the VN ODS.
- one ClientVNCS and one ServerVNCS target units shall implement the VNCS software generated for each VN. These software shall be instances of class TVNCS as defined in the HRTS library.
- SurrogateServers code for remotely called objects is generated according to OPCS_ER definition (see section *17.2.4"Active Class Implementation Support "* for a detailed definition of OPCS_ER contents)
- SurrogateClient code is generated according to OPCS_SER definition (see section *17.2"Multi-Target Constrained Operation Support "* for a detailed definition of OPCS_SER contents)

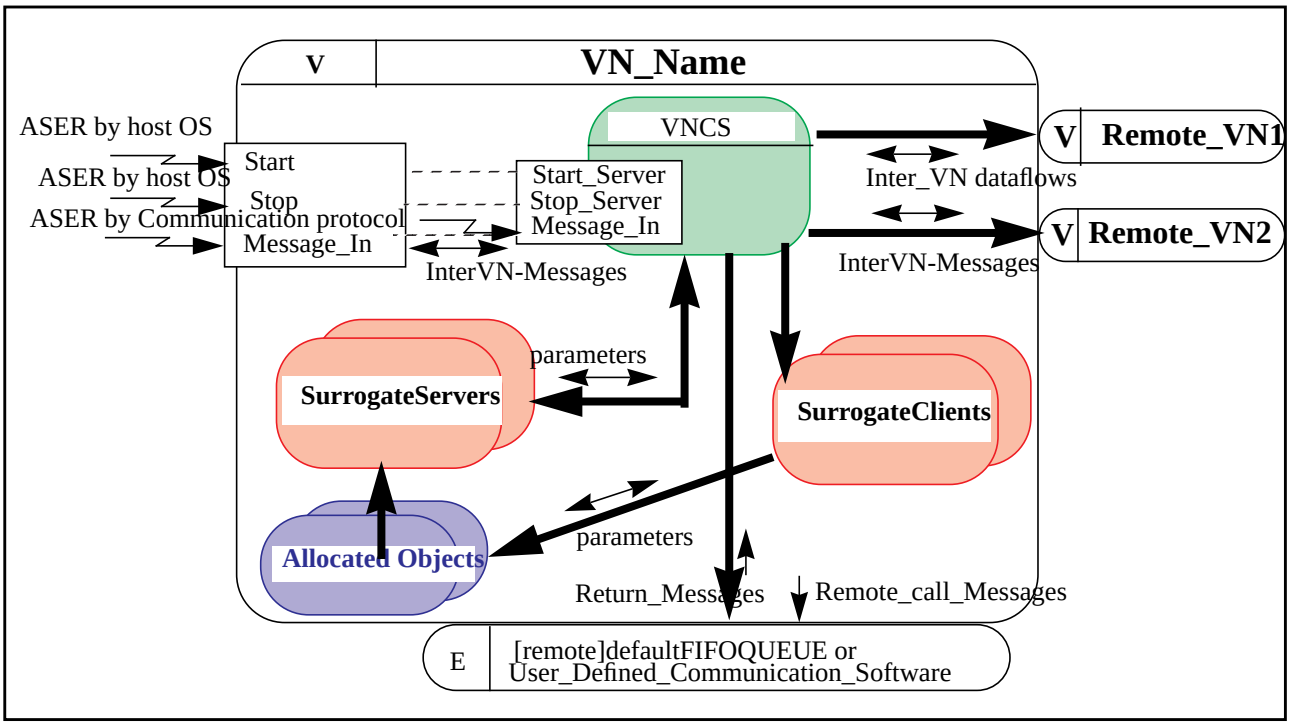

Figure 41 - Illustration of a terminal Virtual Node target code structure

12.2.2.3 Target Code Optimization Principles

When active objects are allocated, a number of Client_OBCS processes execute in parallel by default with the ClientVNCS processes. This solution works against contention of execution request when a server thread is looping within an OPCS operation, but may be relatively resource consuming in case there is no thread implementation available. Depending of the capability of parallel process management by the target, associated OBCS and VNCS processes may be grouped into one queuing system for several active

One solution achieved to meet those requirements is illustrated in *Figure 40 - Code Generation Principles for a terminal Virtual Node* and is based on standardized software specified in the HOOD RUN TIME SUPPORT library. Code generation for multiple targets is supported by the concept of a Virtual Node Control Structure (VNCS). This VNCS implementation isolates the core HOOD application code from target specific details and user-provided inter-process communication protocol. A given IPC protocols such as Message-Passing, Remote Procedure Call (RPC), transaction oriented Protocol (such as CCR[42]), or real-time oriented protocol (such as RTC[43]) may be specified via the HOOD **pragma** IPC (see *Appendix F - HOOD Pragmas-*).

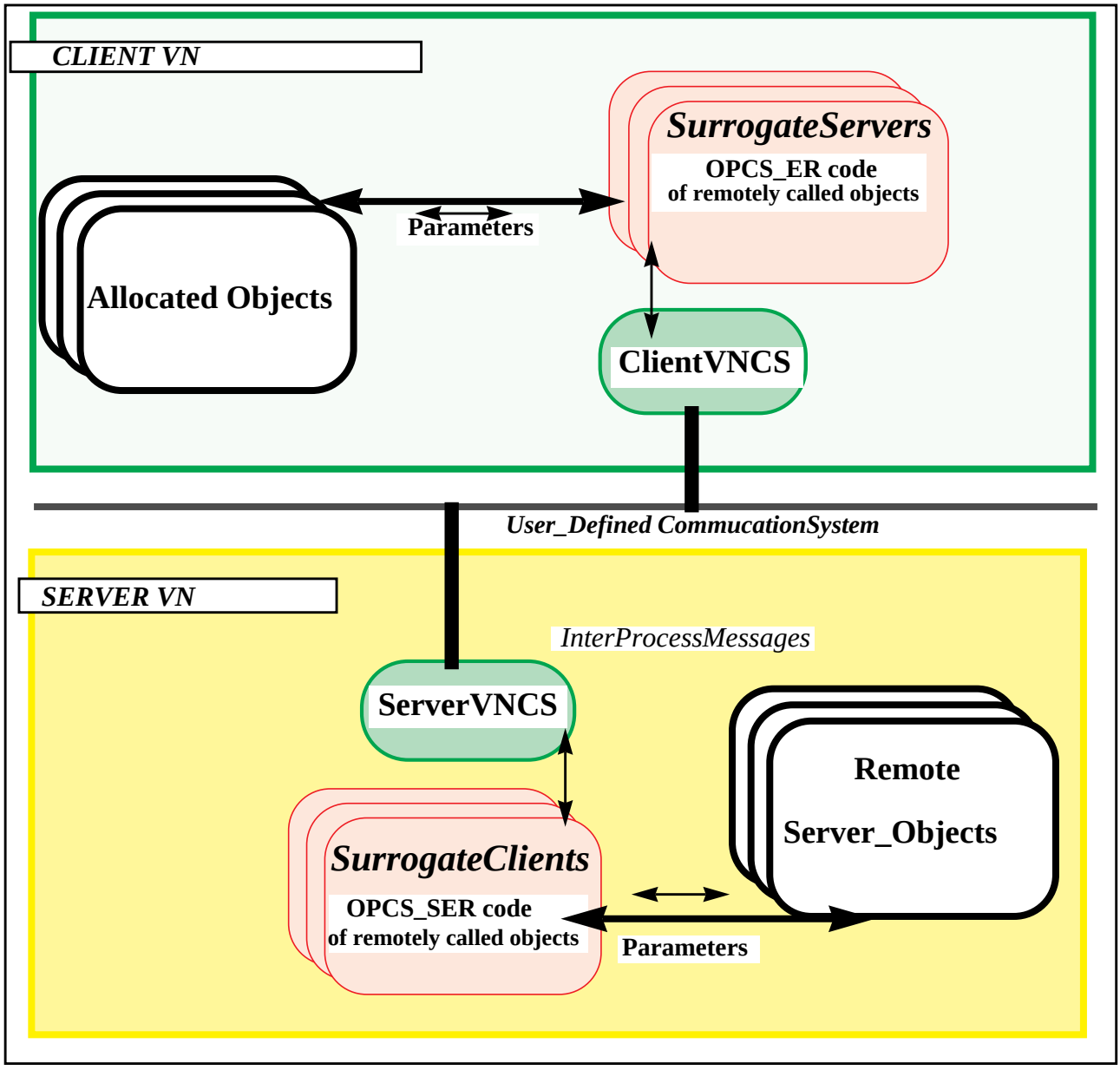

Figure 40 - Code Generation Principles for a terminal Virtual Node

[42] CCR = Commitment Concurrency and Recovery
[43] RTC= real Time Channel Protocol

- Afterwards, the surrogate server, using IPC services sends the IPCMSG to the surrogate client.
- the surrogate client, after the unmarshalling transform, sends the request to the original server
- the server reply, if any, is returned (after marshalling transform) to the surrogate server, and finally to the original client.

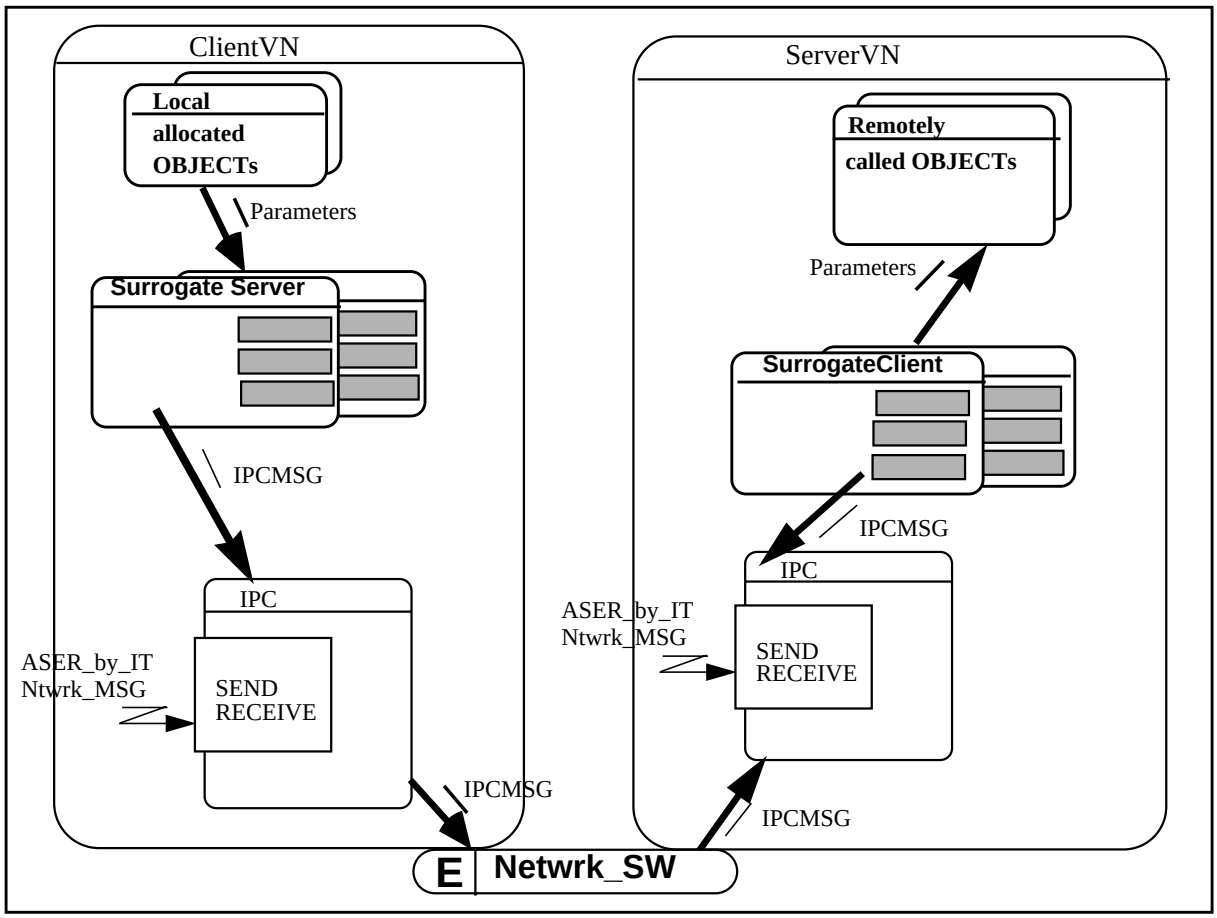

Figure 39 - Communication Model for VNs

12.2.2.2 Multi-Target Code Structure

The implementation of VNs target code shall:

- both enforce the above communication model and a CLIENT-SERVER architecture where clients are not known at server side, whereas still allowing full automated code generation of VNs from object allocation data.
- reuse as much as possible the target operation model as illustrated in *Figure 11 - Target Operation model*, thus allowing optimized implementation of inter-thread communication

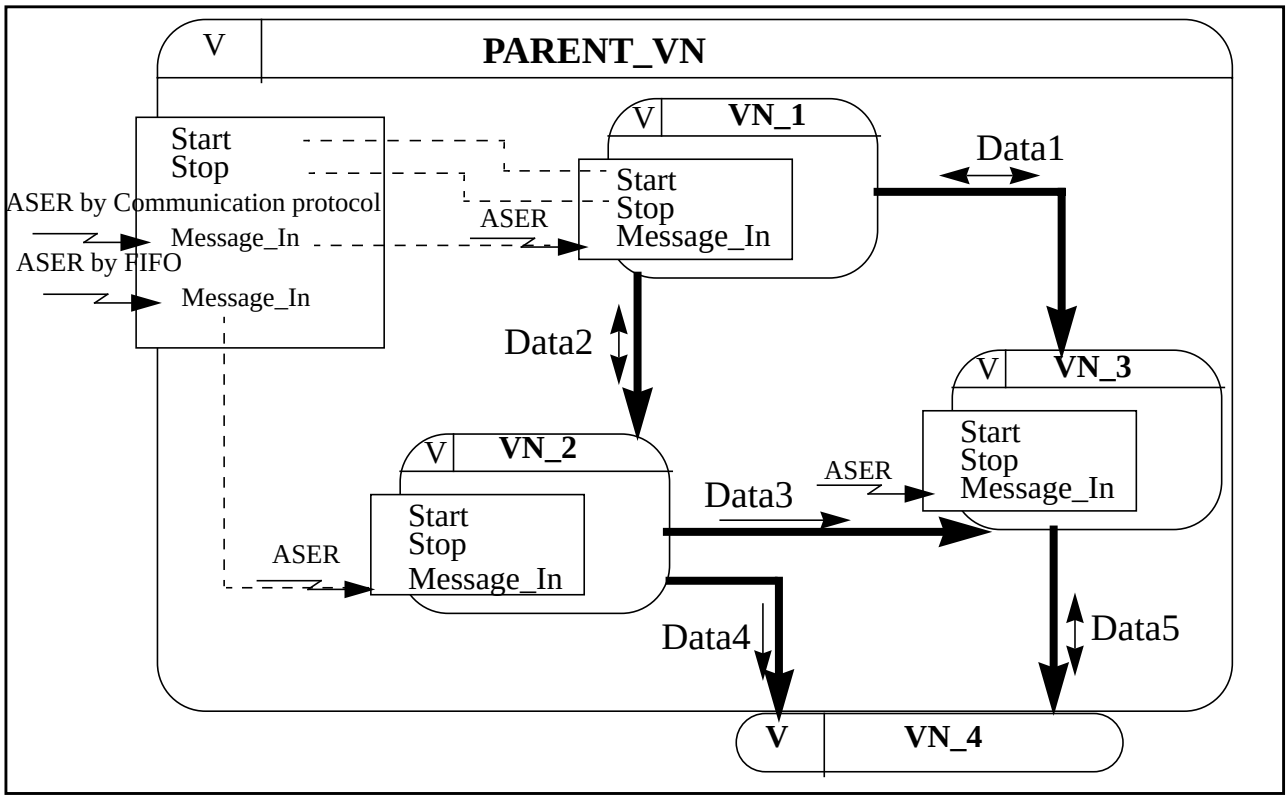

Figure 38 - Parent Virtual Node breakdown

12.2.2 VN Implementation Principles

The driving implementation principle is to allow efficient and automated support of the post-partitioning approach for distribution*[ATKINSON]*. That is client code of an object or class should not be affected when the latter is allocated on a VN and configured on a local or remote PN. This principle is best supported by the concept of surrogate objects having conforming interface, but different bodies than an original object.

12.2.2.1 Surrogate Objects

A surrogate object of a HOOD object allocated to a VN has the same provided interface and behaviour. The Internals part of a surrogate object replace the original code by body stubs. Such stubs include queues associated to provided operations and an interface to inter-processor communication package (IPC). Since a VN may both a client and server, it is useful to distinguish surrogate clients and surrogate servers:

- original clients send their requests to the (surrogate) server, which stores them in a IPCMSG in queue as a bit array (marshalling transform). The IPCMSG includes server name, operation name, parameters, request occurrence.

12.2.1 VN Definition

A **Virtual Node (VN)** is a **collection of HOOD objects** allocated from the logical design space into the “distribution space”. A VN may be summarized as an instance of an abstract activity type implementing a client-server communication between local and remotely allocated objects. *Figure 38* - illustrates such a terminal VN graphical description showing remote VN servers as uncle VNs.Virtual Node have the following properties :

- a terminal VN is defined as the encapsulation of HOOD objects, classes and generic instances **allocated** to that VN.
- a VN may have one or several predefined (possibly overloaded) provided operation “*Message_In*” allowing to specify the communication protocol with client VNs by means of ASER label.
- a terminal VN includes a predefined VNCS object and the allocated objects. The graphical representation of such a terminal VN is optionnal[41] since all graphical representation of terminal VN would be similar and only a textual description giving the allocated objects is relevant. See *section 14.12* below for a detailed description of a VN ODS.
- a VN can only use other VNs.
- **a VN may only be decomposed into other VNs thus defining a VN hierarchy**. This feature allows both abstraction and refinement of very large software programs into masterable and well defined VNs entities.*Figure 38* - gives an illustration of a parent VN breakdown. Note that Dataflow can be shown and that communication protocols are specified through the Message_In operation constraint.

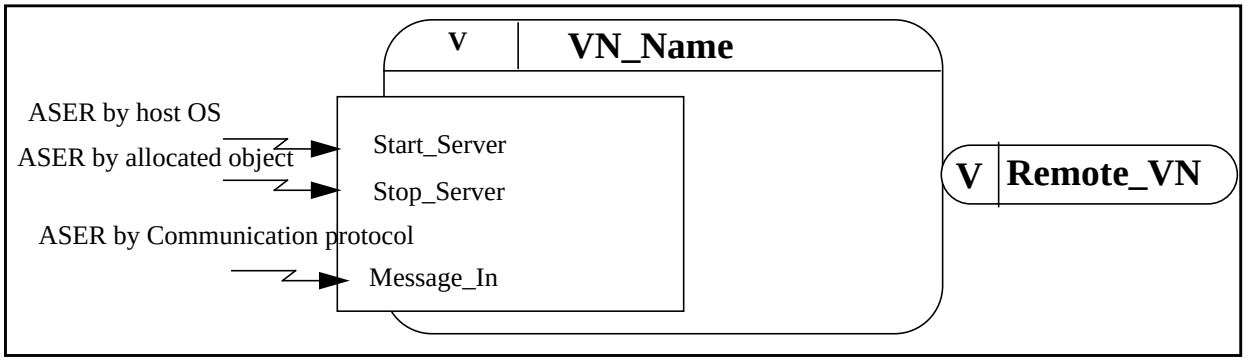

Figure 37 - Representation of a Virtual Node

[41]. The graphical representation of the breakdown of a terminal VN may be used for indicating special handling of VNCS operations, in particular, an allocated object may have to start or stop the VNCS activity at initialisation. *Figure 41* - illustrates such a Terminal VN content.

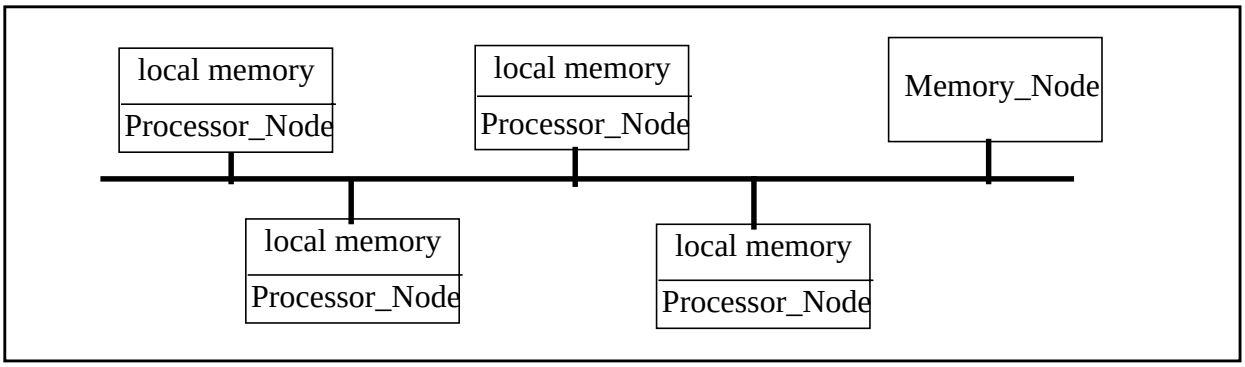

Figure 36 - Distributed System as a network of memory and processor nodes

With these considerations, distributing software is a process that should at least have two phases:

- **partitioning**, where software is broken into several software partitions, each being potentially allocated to a processor node. Partitioning is a design/programming process.
- **configuring**, where every partition of a distributed software is allocated to a processor of the distributed target hardware, for being processed (at least for a time slice of its life time) by that processor. Configuring may be either a design/programming process, a load-time process or a run-time process.

12.2 HOOD CONCEPTS

HOOD introduces three concepts to deal with distributed systems;

- The **Virtual Node** is a grouping of HOOD objects, which is allocatable to a processor and which is graphically represented as an object with a “V” in its upper left corner, as shown in *Figure 37 - Representation of a Virtual Node*.
- **Partitioning** is the process of splitting a HOOD Design tree among a set of Virtual Nodes. This partitioning does not necessarily match the HOOD logical breakdown, since two child objects of the same parent may be allocated to a same Virtual Node.
- The **Physical Node(PN)** is the HOOD design model of a runnable program, by the bind of one or several Virtual nodes. Such a binding includes:
  - at least one Virtual Node
  - representatives of objects located in remote physical nodes. Such representatives are named *surrogate* objects
  - a run-time environment (RTE) such as e.g. a micro kernel

12 VIRTUAL NODES

An important consideration is the tuning of software execution associated to a HOOD design, according to each project specific non functional constraints such as: available targets, distributed physical configurations, efficiency, reliability.... HOOD provides designers with the concept of Virtual Node (VN) as a mean of restructuring HOOD objects from a given HDT into software blocks able to execute in a given physical memory partition (local or remote).

12.1 DISTRIBUTED SYSTEMS

Distributed systems is a term used to define a wide range of computer systems from the weakly-coupled ones (such as WAN[36]s) to strongly coupled ones (such as multi-processors) including medium-coupled systems (such as LAN[37]s). Basically a distributed hardware may be modelled as a network of:

- processor nodes, having capabilities to process a program, able to address local memory and common memory space
- and memory nodes, having no capabilities to process programs this is the model of shared memory.

Software may be distributed on such hardware at three moments:

- at compile/link time[38], where every piece of software is definitively allocated to a dedicated processor.
- at load time[39], where a program is allocated to a processor for its full life time, depending on the workload of each processor of the network
- at run time[40], where a program is allocated to a processor for a slice of its life time, depending of the workload of each processor in the network.

[36]·WAN = wide area network systems
[37]·LANs = local area network systems
[38]·static binding
[39]·load sharing
[40]·Dynamic load sharing

11.2.2 Real Time Architecture and HOOD Design

As resources such as memory or processors are shared among software tasks of an RTSS, the real time architecture should be established, as much as possible in the early time of a HOOD design.

Such a real time architecture design includes:

- identification of required tasks
- definition of task interactions
- time budget allocation, for task execution and task interaction.

During the early basic design steps, the RT architecture is “nested” in the HOOD objects of the upper-level design trees, and task activation and interactions are modelled through constrained operations possibly with time budgets. Afterwards, this RT architecture is partitioned according to the HOOD object breakdown process, and time constraints may be refined into child operation time constraints.

Scheduling analysis may be performed as soon as a scheduling algorithm has been selected (such as Rate monotonic or Earliest Deadlines). Such an analysis states whether (or not) hard real time services can meet their deadlines. Such an approach is more detailed in *Appendix J.9 - OBJECT HRT_SCHEDs-* It allows to design and verify hard real time systems as suggested in the *[HRTOSK]* study.

- an active object may nest several execution threads
- a single execution thread may involve several other objects
- several tasks could share one object.

In order to provide a consistent mapping between object architecture and task architecture, HOOD recommends to associate to each task of a real time architecture, a triggering constrained operation provided by the object that includes the task

This operation implements the software reaction to a periodic or asynchronous event that activate the task. Such an operation (*as operation Start in Figure 34* -) is named activation operation and could support several other constraints. Special constraints may be used to specify periodic activation, or deadline.

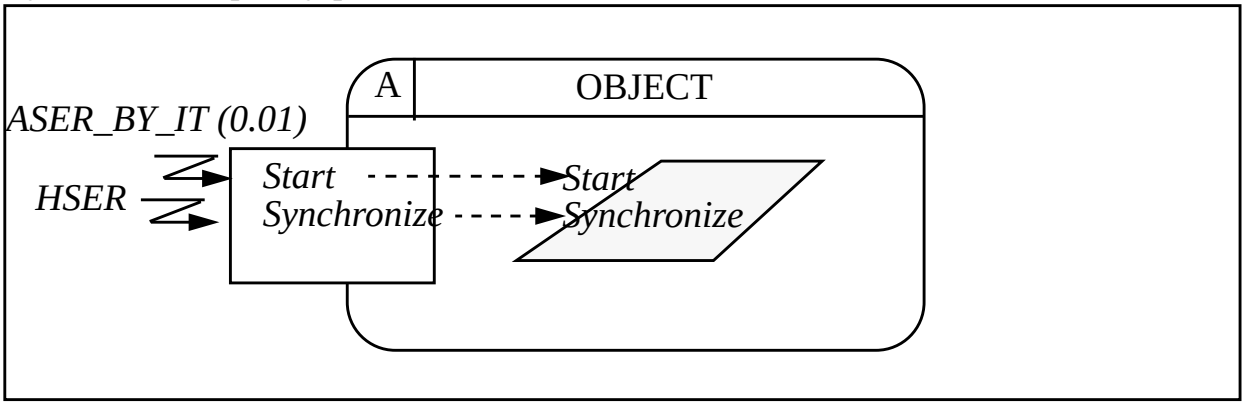

Figure 34 - Mapping Task and Objects

When a terminal object includes several tasks, it shall provides several associated activation operations.

11.2.1 Resources and Passive Objects

Since Resources are abstractions dedicated to be shared among tasks, they are conveniently modelled as passive objects providing Concurency constrained operations.

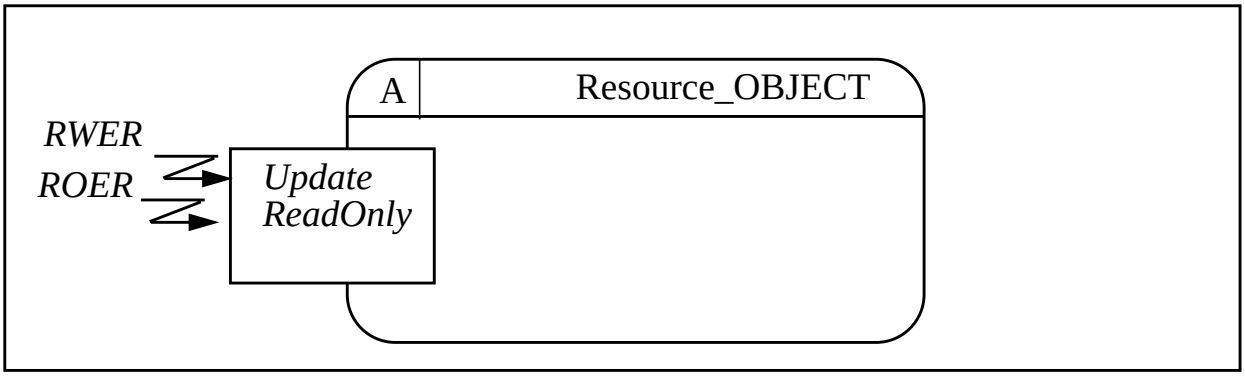

Figure 35 - Mapping Resources and Objects

- pieces of memory (stacks, heaps)
- synchronisation and communication objects (semaphores, monitors, mailboxes, queues, events)
- communication media (bus, network links)
- processsors

Tasks must share resources and have to compete to acquire them, while cooperating in the following two ways with other tasks:

- exchange data
- synchronize their execution flow

The means require to handle competition and cooperation are supported by the RTE which provides the following services to tasks:

- time management services
- memory management services
- data exchange management services
- synchronisation management services
- error management services
- processor allocation / deallocation management services
- I/O management services

Real Time Architecture design is a special activity in the development of RTSS, whose goal is to provide a description of all tasks and resources including:

- description of functions and activities allocated to the tasks defining a RTSS
- temporal characteristics of a task
- synchronisation protocols between tasks
- data communication between tasks
- global scheduling of tasks matching the real time requirements of the system.

11.2 REAL TIME SYSTEMS VS HOOD OBJECT MODEL

There is no mandatory one to one mapping between a task architecture and a HOOD object architecture since:

11 REAL TIME SYSTEMS

11.1 DEFINITIONS

Real Time Software Systems (RTSS) are generally considered as systems providing both general services and real time services :

- the correctness of general services provided by a RTSS is evaluated through the results of the logical computations which depend on input data values and system internal state.
- the correctness of real time services depends both on the results of the logical computations, and on the time at which these results are produced.

Classically RTSS are modelled as a collections of processes cooperating in order to provide services and competing to acquire computing resources. Cooperation and competition management are supported by the Run Time Executive (RTE) that implements the services at the intents of processes.

Processes or *tasks* are runnables / schedulables sequences of statements nested in a single execution thread and including requests to RTE services. Tasks are indivisible pieces of code (that cannot be distributed among several processors at the same time). Tasks are usually classified into:

- periodic tasks (with mandatory deadlines (hard tasks) or not (soft tasks)
- a-periodic tasks (with mandatory deadline or not)

Moreover it is possible to associate to a task an average and a worst case execution time.

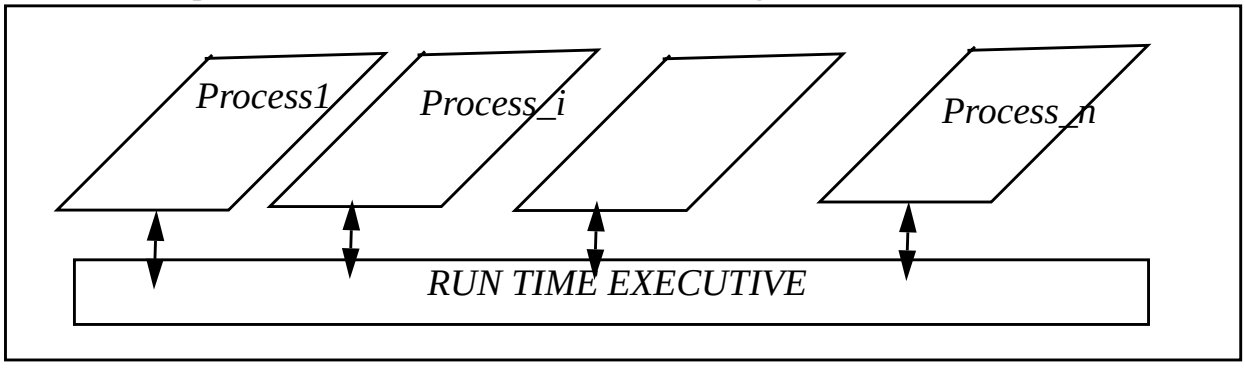

Figure 33 - Representation of RTE services

Resources are shared entities for which tasks compete in order to execute their program. Such resources may include:

The representation of several identical generic instances is a double shaped object or class with an indexed generic name “typed” with the name of the generic object. The names of the instances are generated as the generic name concatenated with the successive integer values of the index range.

In *Figure 32 - Representation of Instances*, Object Inst[2.. 10] : class_name is then equivalent to 9 objects e.g Inst2, Inst3..., Inst10.
Environment object Obj_Y and object Object_B provide the actual operations for Inst[2.10] :generic_name whereas Obj_X provides the actual operations for Inst1.

10.2 GENERIC INSTANCE DEFINITION

A generic instance will be created as an object or a class inside the current HDT (as child of an object or as child of a class) or as a root object. Each instance is identified by the instance name followed by the generic name (See *Figure 32 - Representation of Instances*).

A generic instance may be defined by giving values to its formal parameters . These parameters are called **actual parameters**; they may be supplied by environment objects, classes or siblings. Adequate matching checks shall be performed between the formal definition of generic parameters and actual ones; syntax and semantics rely fully on the target language ones.

In the client-server view of the graphical description, if an instance has an operation as actual parameter, then a use arrow shall identify the server (brother, uncle or environment object or class) which provides that operation.

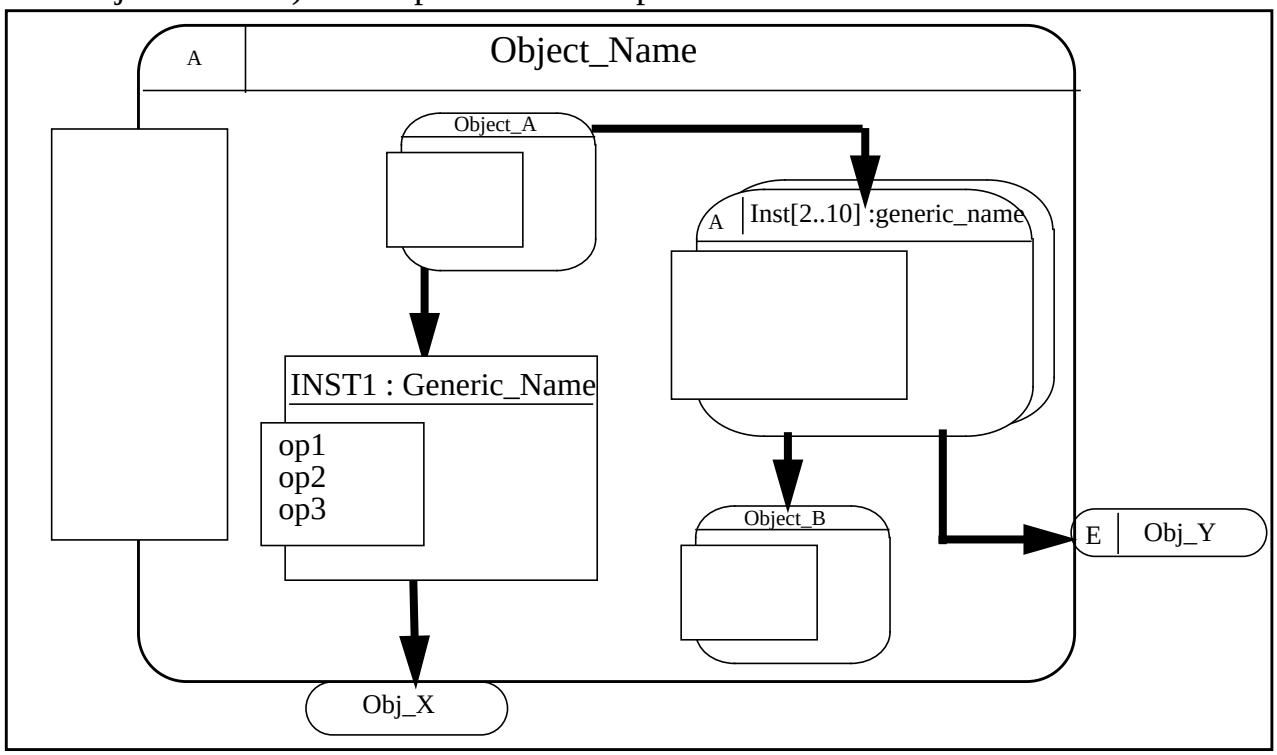

Figure 32 - Representation of Instances

A generic instance of an object inherits the full environment of the generic object which is not shown in the graphical representation.

The representation of a unique instance of a generic is an object or class whose name is “typed” with the name of the generic object.

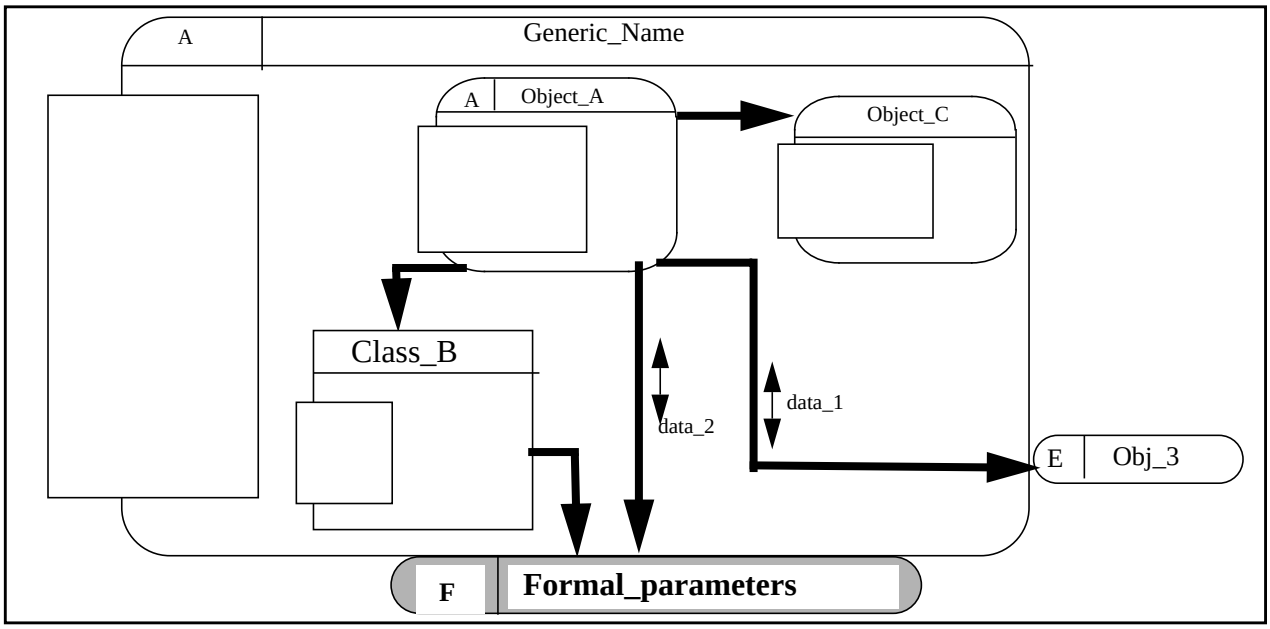

Figure 30 - Representation of a Client_server view of a Generic

In *Figure 30 - Representation of a Client_server view of a Generic*, the generic Generic_Name is composed of children A,B and C. The use arrow toward the formal parameters box shows that A and B require at least one operation from the formal parameters.

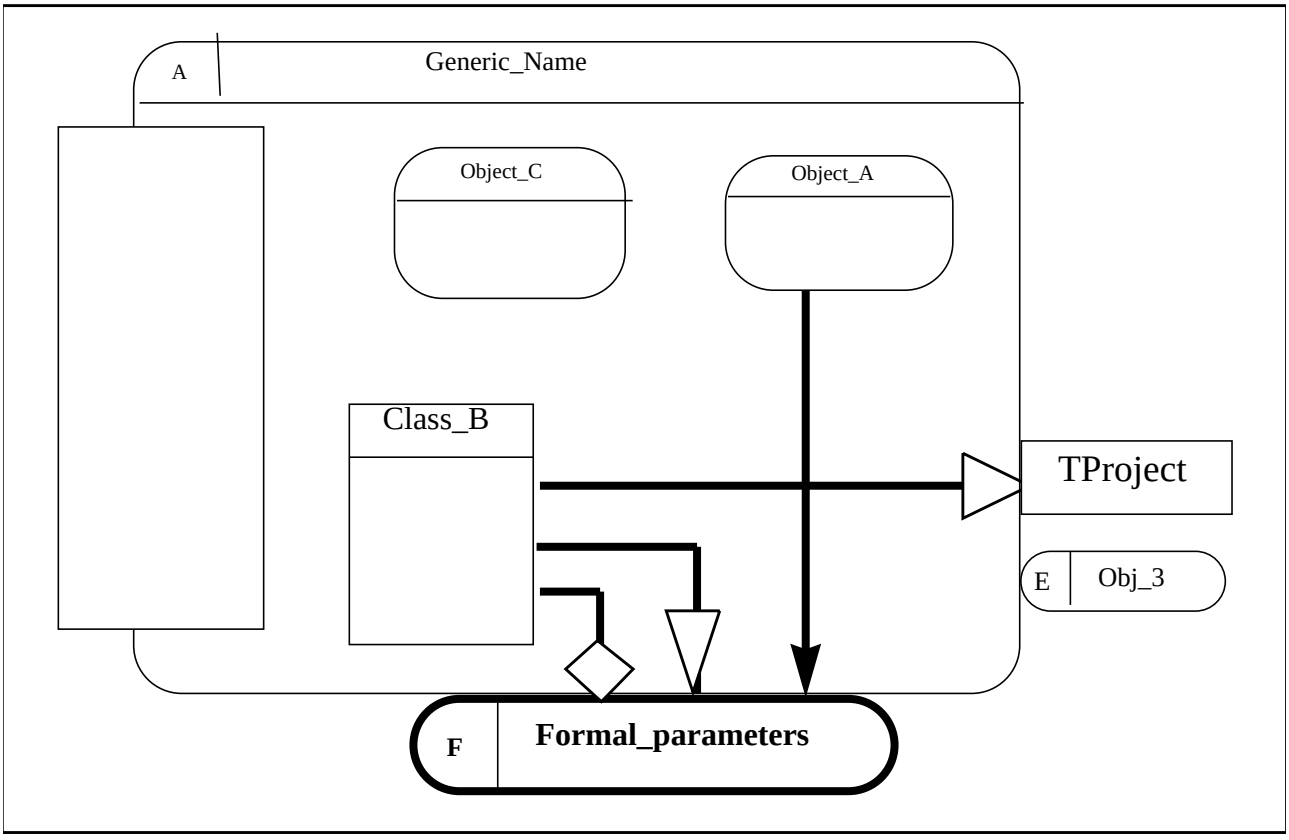

Figure 31 - Representation of a generic diagram in the structure view

10 GENERICS

A generic object is a representation of a pattern of object or class which can be **reused** and **parameterized** by types, classes, constants and operations. These parameters define the **formal parameters** of the generic object.

Generics are defined as root objects only[35], and may use siblings or environments.

An instance of a generic may be created within a parent by defining explicitly the different parameters (types, classes, constants, operations) thus creating an object or class within the design. These parameters must be provided by objects or classes that are directly visible from the instantiation location.

10.1 GENERIC DEFINITION

A generic unit may be designed specifically for the system under design when several similar objects or classes are needed. The generic unit may have as parameters types, classes, constants, operations and may be “Active” or “Passive”.

The declaration of the **formal parameters** of a generic shall be done in associated fields of the ODS in the target language syntax. A generic may :

- **only** be decomposed into objects or classes, **exclusive of any other generic.**
- include instances of other objects or Generics.

In the graphical views, a generic unit shall have a formal parameters box represented as an uncle named “Formal_parameters” and identified by a “F” in the left side.

- In the client server view a use relationship toward that object shall only be represented when one of its child requires execution of a formal operation provided by the object “Formal_parameters”.
- In the structure view, a type-use, an inheritance or an aggregate relationship toward the “Formal_parameters” shall only be represented if a child of the generic requires a type, inherits or aggregates one of the formal parameters.

35. The reason of this limitation is both one of limiting complexity of designs 9how shall we understand and check generic definition at multiple level in a hierarchy?), and the need of extended tool support for propagating formal parameters definition outside an object hierarchy

9.2 CLASS INSTANCE DEFINITION

A class instance is a HOOD DATA and has no graphical representation. In the textual formalism class instances are expressed in the target dependent DATA field of the INTERNALS in the ODS of a client of the class. See textual formalism for a detailed description of data instance syntax.

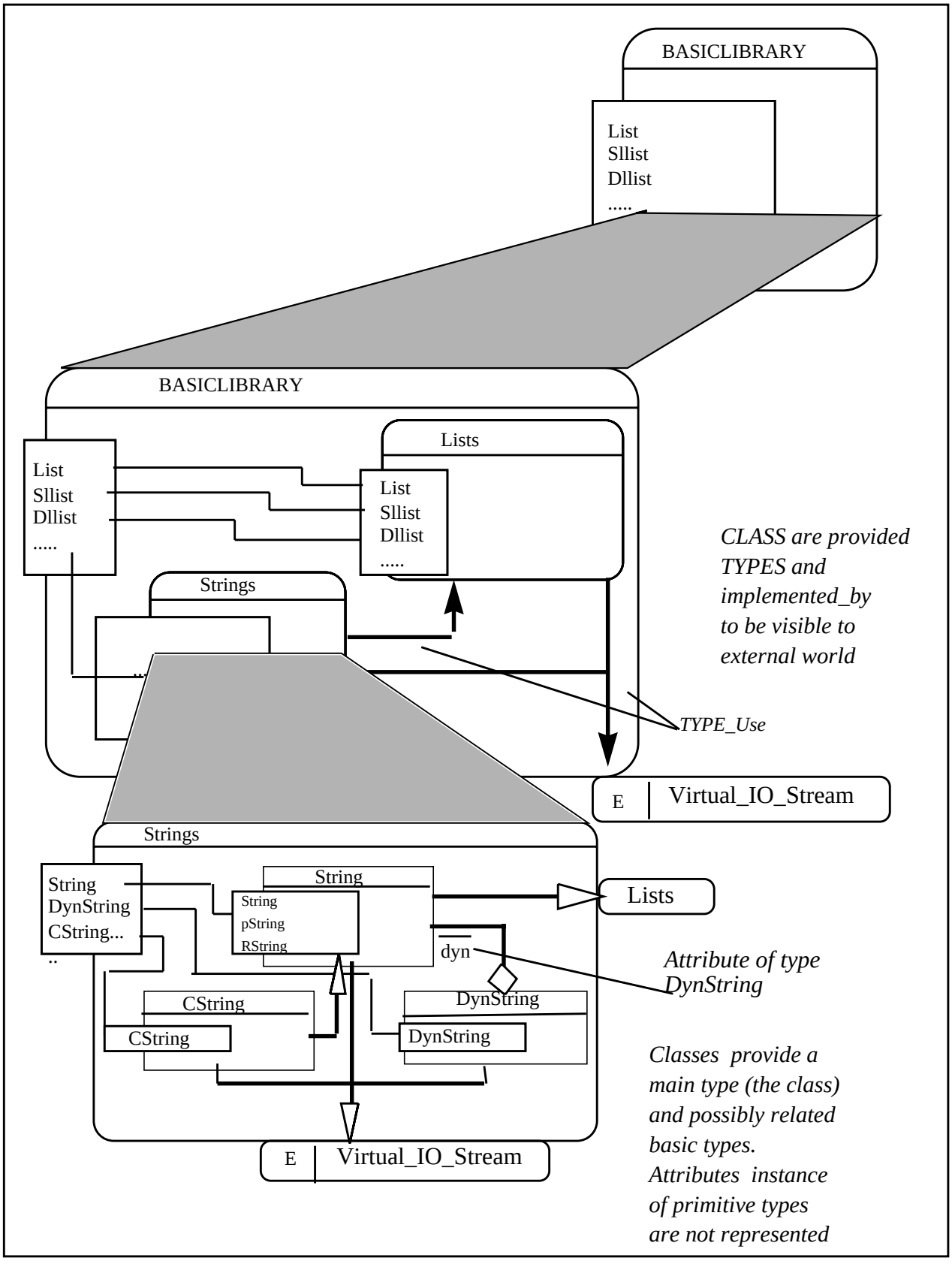

Figure 29 - Structure view of object LIBRARY

Figure 23 - and *Figure 27* - represent a library of basic C++ classes modelled as HOOD objects encapsulating classes and illustrating the representation of their relationships.

9.1.5 Graphical representation of Class libraries

In the following we give a summary of the graphical representations of a library of C++ classes for both client-server and structure views

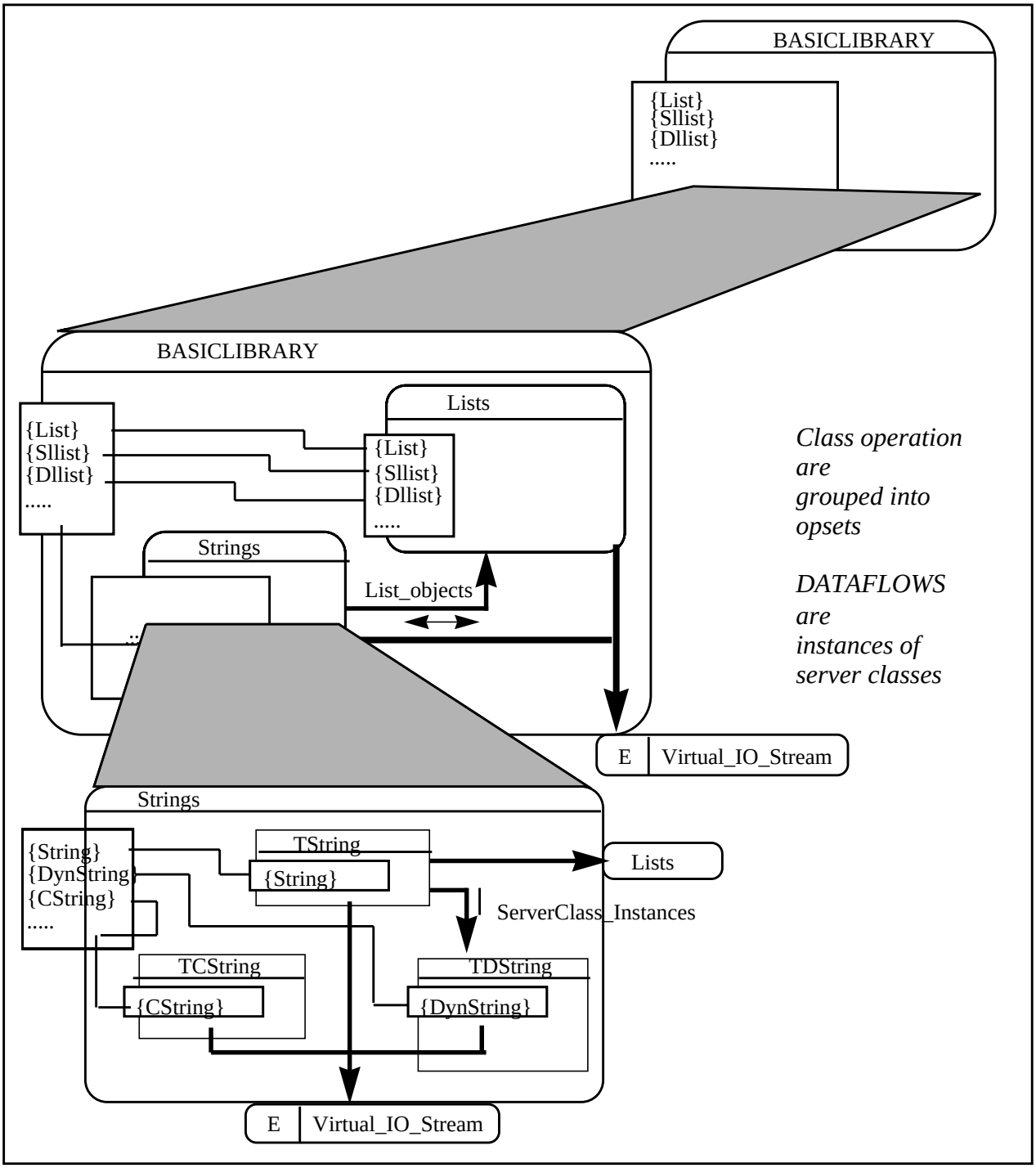

Figure 28 - Client-Server view of parent object LIBRARY

Attributes are typed entities, and are either of primitive type or of complex types provided by other HOOD objects, or classes. **Encapsulating attributes as instances of other classes allows complex data structures to be refined step-wise** by attribution relationships between the container class and additional “attributed” classes.

In the textual formalism, class attributes are expressed in the TYPE ATTRIBUTES field of the ODS. See textual formalism for a detailed description of classes instances syntax in *section 14*.

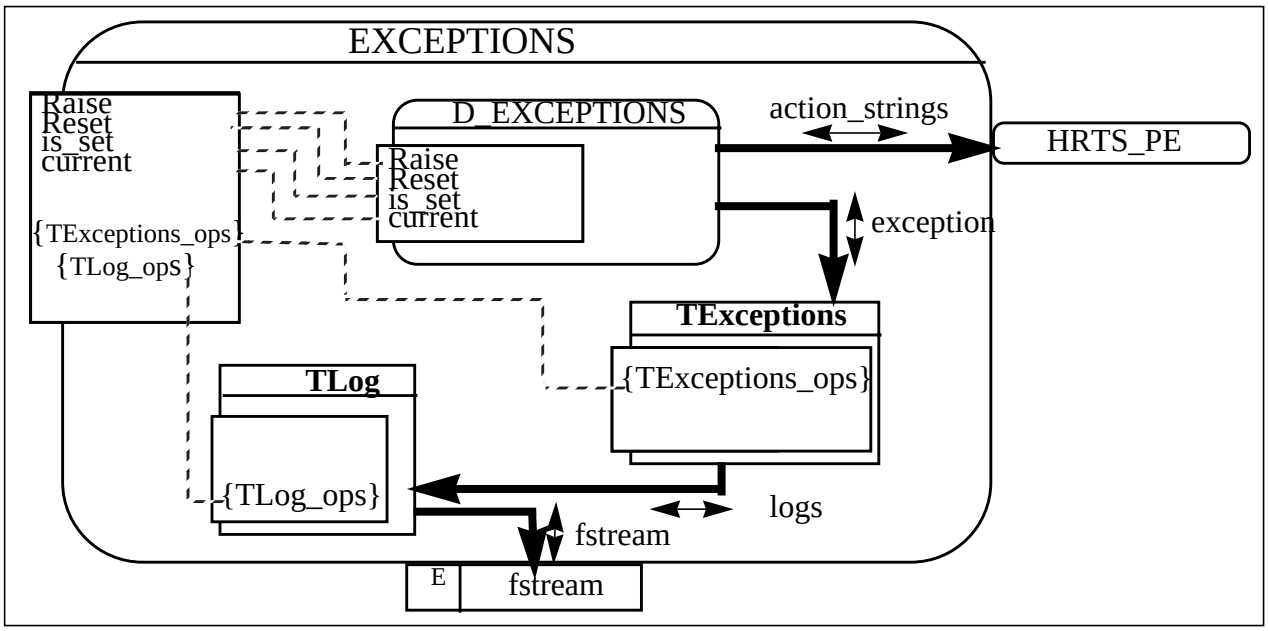

Figure 26 - Client-Server view of parent object HRTS.EXCEPTIONS

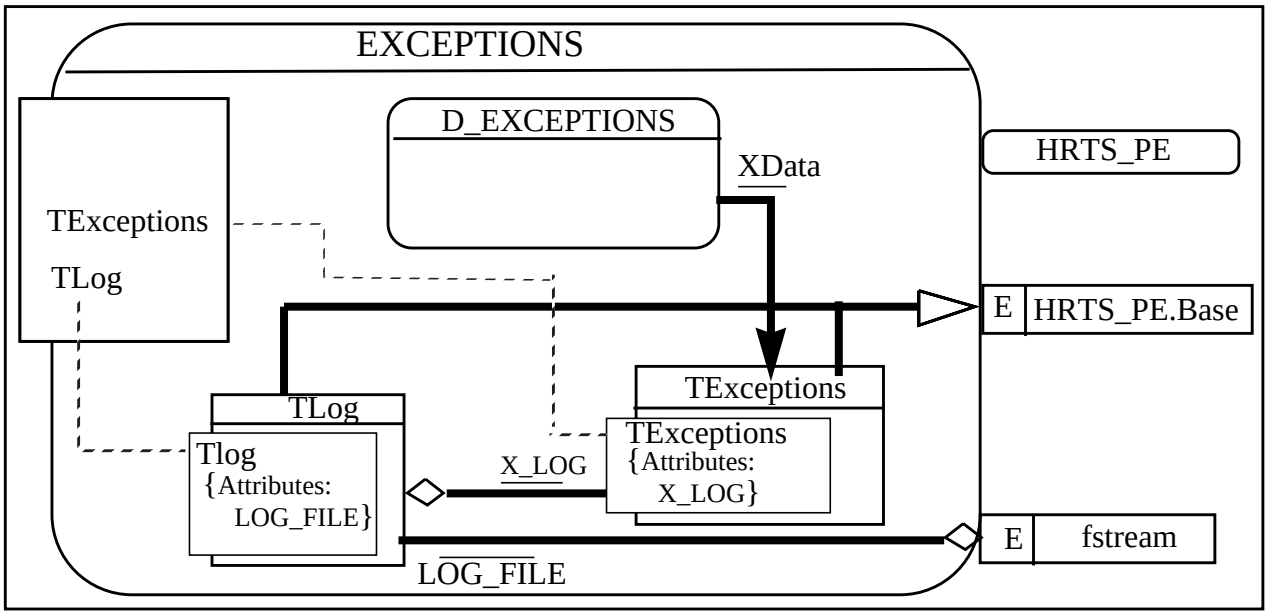

Figure 27 - structure view of object HRTS.EXCEPTIONS[34]

[34]. Note that in this view, objects provide types, and labels on arrows represent attributes or data instances

9.1.3 Class Inheritance Definition

Class inheritance is represented graphically, in the structure view, with an “inherit arrow“going from subclass to superclass. In the textual formalism, class inheritance is expressed in the TYPE INHERITANCE field of the CLASS ODS. See textual formalism for a detailed description of classes instances syntax in *section 14*.

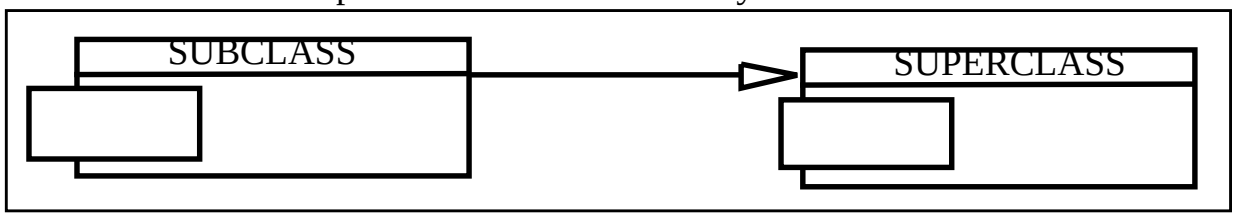

Figure 24 - Representation of inheritance relationships in the structure diagram

9.1.4 Attribution Definition

Since HOOD is dealing with design and implementation (rather than analysis), a class should be primarily thought as the physical composition[32] of its attributes, rather then the implementation of the representation of “has_a” or “part_of” relationships of a data model. HOOD enforces thus the meaning of “attribute” rather than the one of “aggregate” as *data field of a composite data structure*. Target implementations may thus vary depending of the available constructs: in C++ a data member of a class shall be a reference and not an aggregate if accessed by pointers, whereas in Smalltalk, only references exist...

Attribution relationships are represented with an “attribute arrow” going from container class to the attributed class. Class attributes are optionally represented, in the structure view, as:

- members of an “Attribute-set[33]” with reserved name “Attributes” (see *Figure 25* -) which contains the list of the most significant attributes for understanding the structure.
- labels on the attribute arrow (see *Figure 25* -) for attributes specifically defined instance of an attributed class.

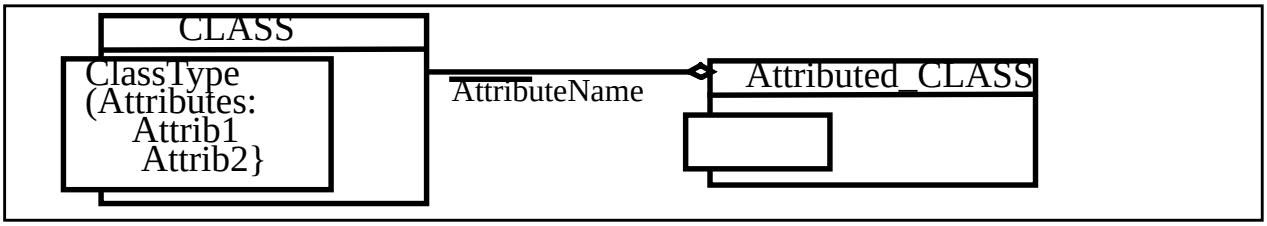

Figure 25 - Attribution relationships representations in the structure diagram

[32.] A full discussion of semantically differences between composition and attribution can be found in *[ODDEL94]*

[33.] An Attribute-set should only be seen as a graphical representation facility, allowing designs with numerous classes and attributes to be represented in an understandable way.

be applied to such attributes by declaring them as “private’ in the internals of the ODS in the textual formalism (see *section 14*).

- Classes may also encapsulate **internal data** (*class variables*) which are unique and shared by all instances of the class. Such data appear in the internal data field of the class ODS in the textual formalism.
- Classes may be ABSTRACT ones, which can only be inherited by other classes.A HOOD class is abstract as soon as it provides an ABSTRACT OPERATION.

9.1.1 Class Graphical Representations

A HOOD class may be a root or a child of another object. In order to favour better structure of numerous classes, HOOD recommends to define related classes (either a full inheritance tree or merely logically related ones) as children of a parent “library-object” (see *Figure 28* - and *Figure 29* -). This provides the OO designer with the necessary encapsulating facility which is lacking in popular methods such as *[OMT91]*and others.

In order to avoid overloaded diagrams when representing class relationships, two graphical views are defined:

- *the client-server view* shows the use/client-server relationships, data and exception flows between child objects and classes of a parent object. (see *Figure 26 - Client-Server view of parent object HRTS.EXCEPTIONS* below)
- *the structure view* shows the type-use relationships between objects, attribution and inheritance relationships between classes represented with their attributes.

Figure 23 - Representation of a class as a HOOD Server Object for Instance code and *Figure 27 - structure view of object HRTS.EXCEPTIONS* give an illustration of the graphical formalism for representing classes, use relationships, inheritance and attribution in the design of the HRTS library object for handling exceptions.

9.1.2 Type-use Relationships

An object or class is said to **type-use** another object or class if the former requires one type or class for type, class, parameter or data definition. The **type-use relationship** among objects and classes defines a require relationship that shall be documented in the required interface of the textual formalism (see detailed syntax in *section 14*) and is represented graphically only in the structure view. Note that a use relationship in the client-server view (require an operation) does not imply a type-use relationship in the structure view (since operation parameters may be defined in another object or class).

9.1 HOOD CLASS DEFINITION

When a class instance uses an operation of the class, client code associated with that instance "uses the class operation code" with a special parameter: the *receiver* data instance. A HOOD class is thus defined as:

- a special HOOD server object providing a type of same name to clients.
- a set of operations having each a parameter of reserved name *me* of type the type of the class that defines the receiver of the operation. Such operations may be constrained.
- a terminal object, encapsulating all provided operations that have the class as receiver.

The graphical representation is similar to a HOOD object but with squared instead of rounded corners.

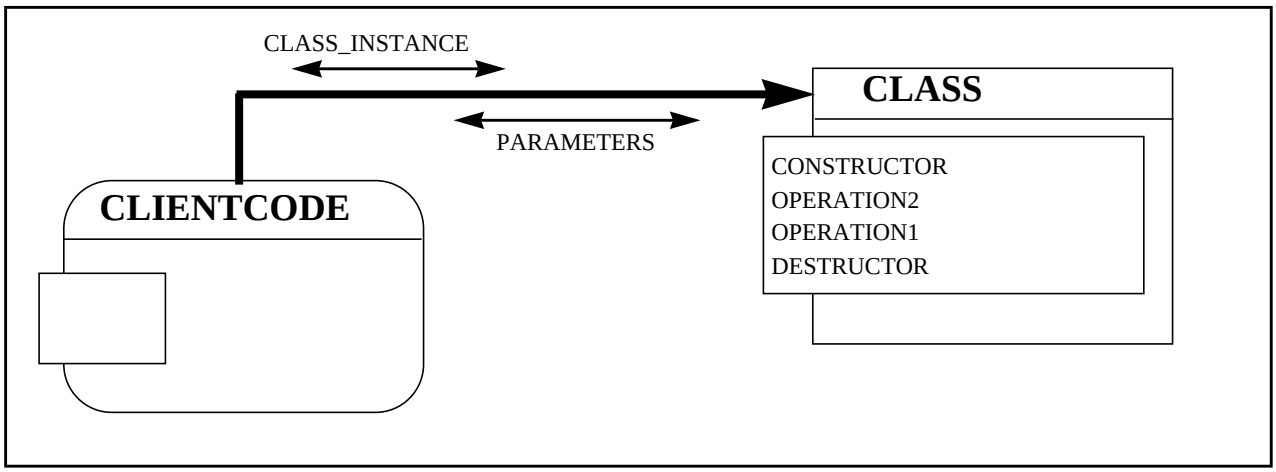

Figure 23 - Representation of a class as a HOOD Server Object for Instance code

- Operations of a class with receiver *me* implement the concept of "*instance method*s" since the receiver *"me"* of the operation is always a class instance. "*Class methods*" may also be defined by specifying the receiver as the reserved parameter name *myClass.* Such operation applies not on an instance but on the class itself. Such operations are useful to access values of *class variables*.
- Classes encapsulate **attributes** which are instantiated as internal data (also named *instance variables*) dedicated for each instance of a class. Since such variables are encapsulated in their specific instance, and can only be accessed by specifying the instance of the class where they belong, attributes may appear as public "data" and declared in the provided interface of a HOOD class. Restricted access can however

9 CLASSES

"Rather then describing individual objects, one concentrates on the patterns common to a whole class of objects. All such object will be called instances of the class " [MEYER88].

A HOOD class is an HOOD object implementing an abstract data type, and may thus be considered both as a type and a module. **A class however distinguishes from a type in that, it can extend its properties and operations by inheritance**, **and may itself be inherited from other classes**.

Classes are thus object oriented programming ones, that is *software modules*, that define shared code for all instances of the class. Such shared code can be extended through inheritance from other classes.

Classes can be instantiated into **instances** (i.e. data or OO objects) thus defining particular data and code sharing the common code of a class, including the one of inherited classes (see*Figure 22* -). Class target code is still made more reusable and flexible through **polymorphism** allowing parameters of the operations to denote objects of different types of an inheritance hierarchy and to select appropriate operation implementation.

Classes may be defined and ***refined by attribution***, describing how they are assembled from other components and/or classes. *Such components of a class are called* ***attributes,*** that shall be instanciated as **variables or data** in each instance of the class**.** Some attributes may be unique for all instances of a class and are implemented in target code as *class variables [WEGNER87].*

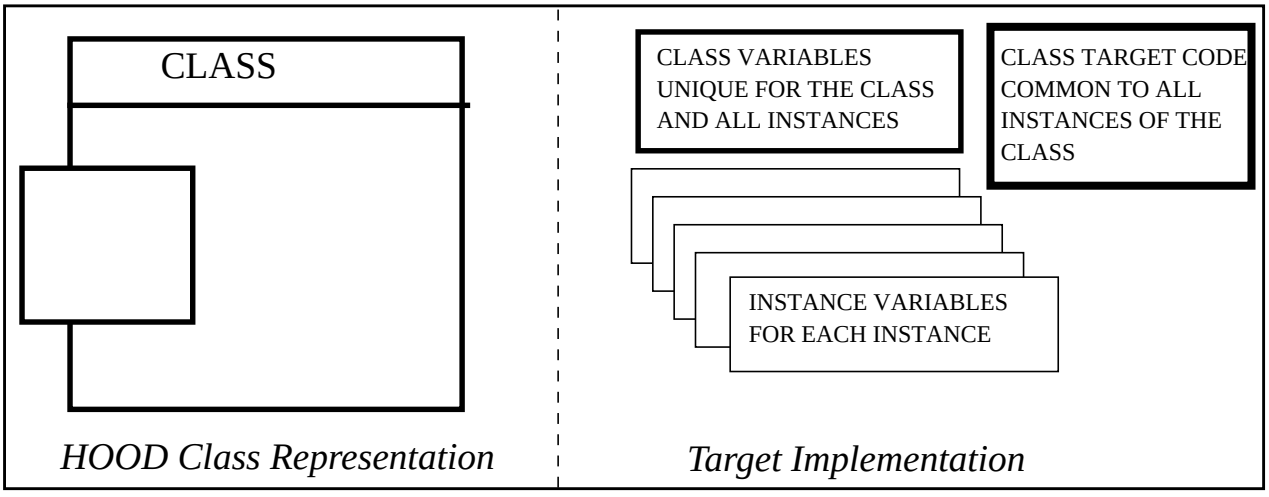

Figure 22 - Representation of a class and instances implementation

8 ENVIRONMENT OBJECT

Beside a given design tree, all design tree roots of the remaining logical space can be considered as context or environment of the design tree. An Environment Object is thus a view of a **root object** which is not part of the current HDT but, which is used by the system to design. Hence, environment objects give the flexibility to incorporate additional software into the HOOD design without impinging on the formal hierarchical decomposition principles in HOOD.

Environment objects partition the external world of a design tree and allow to check its interfaces with respect to its environment.

Within an HDT, an Environment Object is represented as an uncle object with the “E” letter on the left side, in the diagram of the object (see *Figure 21 - Environment Object Representation* where Obj_3 is an environment object represented as an uncle object). If the Environment Object has not yet been defined elsewhere, an associated root object may be created.[31]

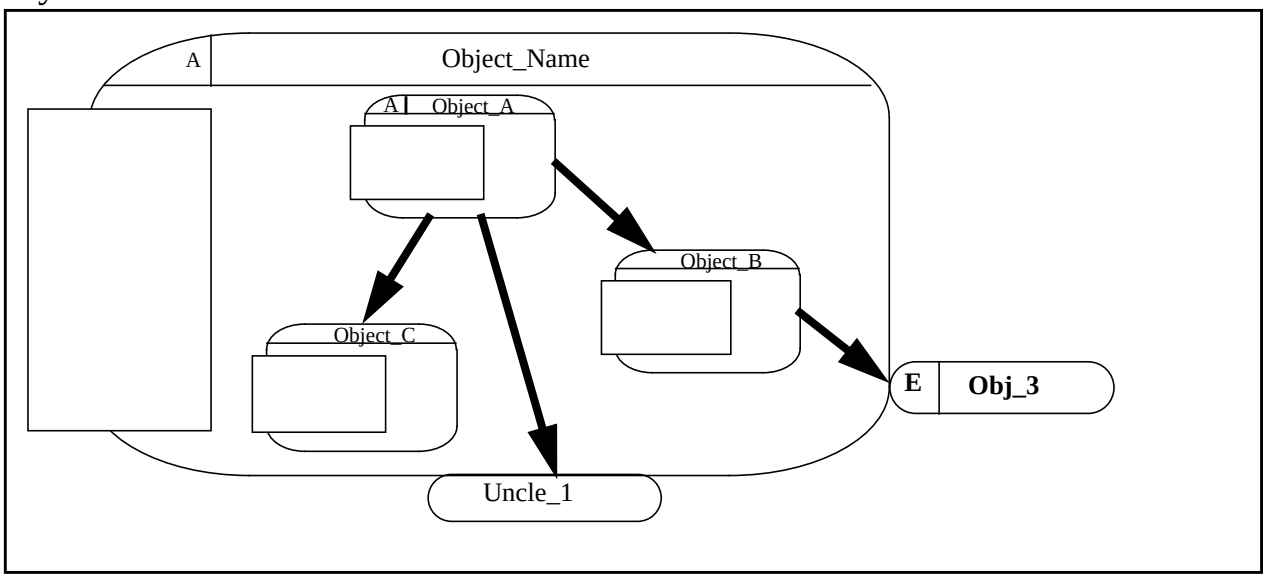

Figure 21 - Environment Object Representation

In the graphical representation, environment objects are only shown at the level where they add significant information. Once represented, they are propagated on all lower levels. This avoids adding too much complexity to the diagram, for those objects which are not part of the software to be developed explicitly. Thus Object_B in *Figure 21 - Environment Object Representation* may refer to environment Obj_3 even if its parent_object does not refer to it.

[31] The information to represent the associated interface may either be supplied from another existing HDT, or entered by the designer from a document describing this interface (e.g. Interface Control Document).

7 EXCEPTION FLOWS

An exception specifies the potentiality of abnormal return of control (to client) during execution of an operation. When such a situation occurs, control flows back to the client in order to notify it.The flow of exception is thus backwards the normal flow of control and this is shown by a line crossing the use or implemented_by relationships. This line is marked with the exception name(s).

The exception names are formal texts[29] which ensure consistency in the diagram with the exception propagation flows as specified in the provided interface of the server objects. In the Object Description Skeleton (ODS), a section named with the keyword **EXCEPTION_FLOWS** will be filled with those <exception_names><direction[30]><childOperation>OR<serverObject>as expressed both in the diagram and in the provided interface of the servers.

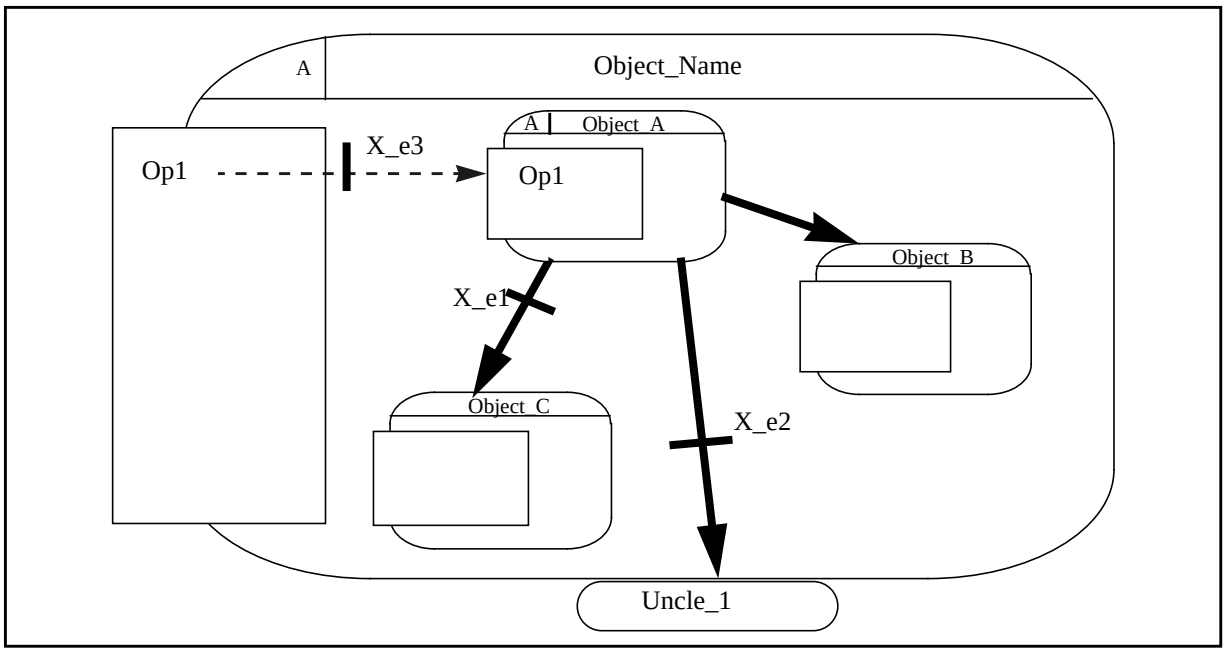

Figure 20 - Exception flow representation

As HOOD parent objects are “empty shells”, and since exceptions are linked to operations, a terminal HOOD object shall not contain any exception. Hence consistency rules can be established between Parent provided exception, child provided ones, and required ones. See *section 15.2*for more details on these consistency rules.

[29]. This text may be tailored according to project standards.

[30]. Direction shall be indicated by arrow signs and is always relative to the direction given by the implemented_by or use arrow towards a target which is respectively an operation or an object or uncle_object.

6 DATA FLOWS

In order to show the flow of a data, an arrow is used, with one or more data or class instance names alongside. The dataflow name is an **informal text**[27] which abstracts in the diagram the information exchanged between objects.

Data may flow :

- in the direction of the use or implemented_by relationship,
- in the opposite direction,
- or in both direction.

This representation is used to show the dataflow between objects (omitting error codes, etc...) in order to make the diagram more comprehensive. Dataflow represented along the implemented_by relationships, allow to illustrate the breakdown of object dataflow into operation dataflow. The latter should be consistent with the parameters modes of the operation.

In the Object Description Skeleton (ODS) of parent objects, a field defined by the keyword **DATAFLOWS** will be filled with <dataflow labels><direction[28]><childOperation>or<serverObject>as represented in the HOOD diagram.

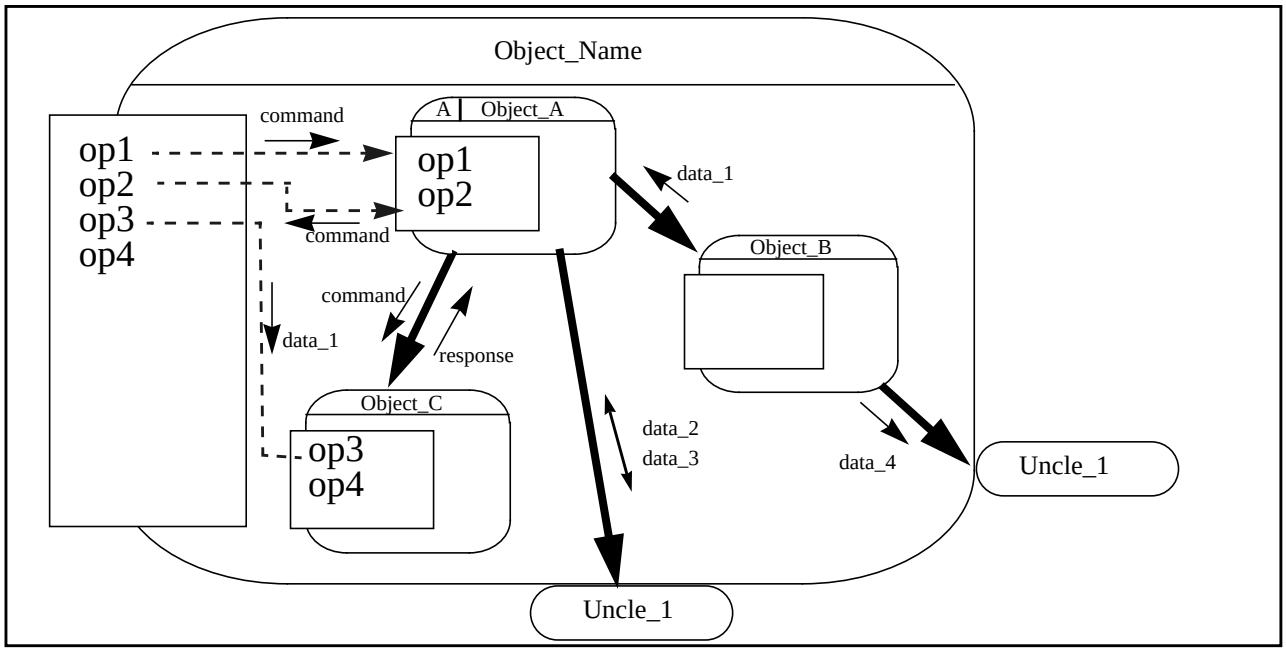

Figure 19 - Dataflow Representations

[27.] This text may be tailored according to project standards (e.g. referencing type identifiers).

[28.] Direction shall be indicated by arrow signs and is always relative to the direction given by the implemented_by or use arrow towards a target which is respectively an operation or an object or uncle_object.

- Object_B uses a passive Uncle_1 which Parent_Object also uses it at the upper level of decomposition.
- Parent_Object.{Child_C_ops} is an Operation_set implemented by Object_C.{opes_1}.
- Parent_Object.{Op_set} is broken down in two operation_sets i.e. {C_ops}, {D_ops} and into a single operation i.e. Operation3.
- {C_ops} and {D_ops} are respectively implemented by Object_C.{opes_2}, Object_D.{opes}.
- Operation3 is implemented by Object_D.Operation2.
- The provided operation Operation4 is implemented by Object_D.Operation3.

5.2.6 Internal Operations

For terminal objects, an OPeration Control Structure (OPCS) contains a full description including the pseudo-code and associated code. This OPCS may in turn require execution of **internal operations** and/or of operations provided by used objects.
The internal operations are not shown in the HOOD diagram but are declared in the OPERATION field of the INTERNALS of the ODS of a terminal object and have also an associated OPCS (see *section 14*).

5.3 TYPES AND CLASS REFINEMENT

In order to enforce object orientation (encapsulation and information hiding principles), types (whether true types or associated to a HOOD class - see *section 9*) may be declared in a non terminal object but their structure shall always be **implemented by** terminal child object or class. The internal type field of the parent ODS is thus filled with the keyword **IMPLEMENTED_BY** and the name of the child type. Hence :

- the provided and required interfaces of an object have no implicit dependencies on the parent,
- the principle of abstraction in successive refinements of design levels is enforced. The detailed implementation of a type is deferred to the same design level as the detailed implementation of the operations which apply to it.

Note that Parent objects may be considered as “empty code and data shells”, thus no DATA can be contained directly in a parent, and a parent ODS has no DATA field.

5.2.5 Operation_sets

Operations provided by a parent object may be numerous; in order to allow flexible grouping of operations, the concept of operation-set has been defined. An **operation_set** corresponds to a set of operations which are implemented at the next or further level of decomposition. An operation_set is intended as a **description facility** and is represented by curly brackets ({...}) in the diagram and by the keyword **OPERATION_SETS** in the Object Description Skeleton.

The operation_set can be decomposed into other operation_sets[26] and operations. A parent level operation_set which is not decomposed shall be **implemented by** a unique operation_set of a child object. An operation_set will be completely described in the ODS by attaching the keyword **MEMBER_OF** <set_name>, to each of its operation or operation_set member. In *Figure 18* -:

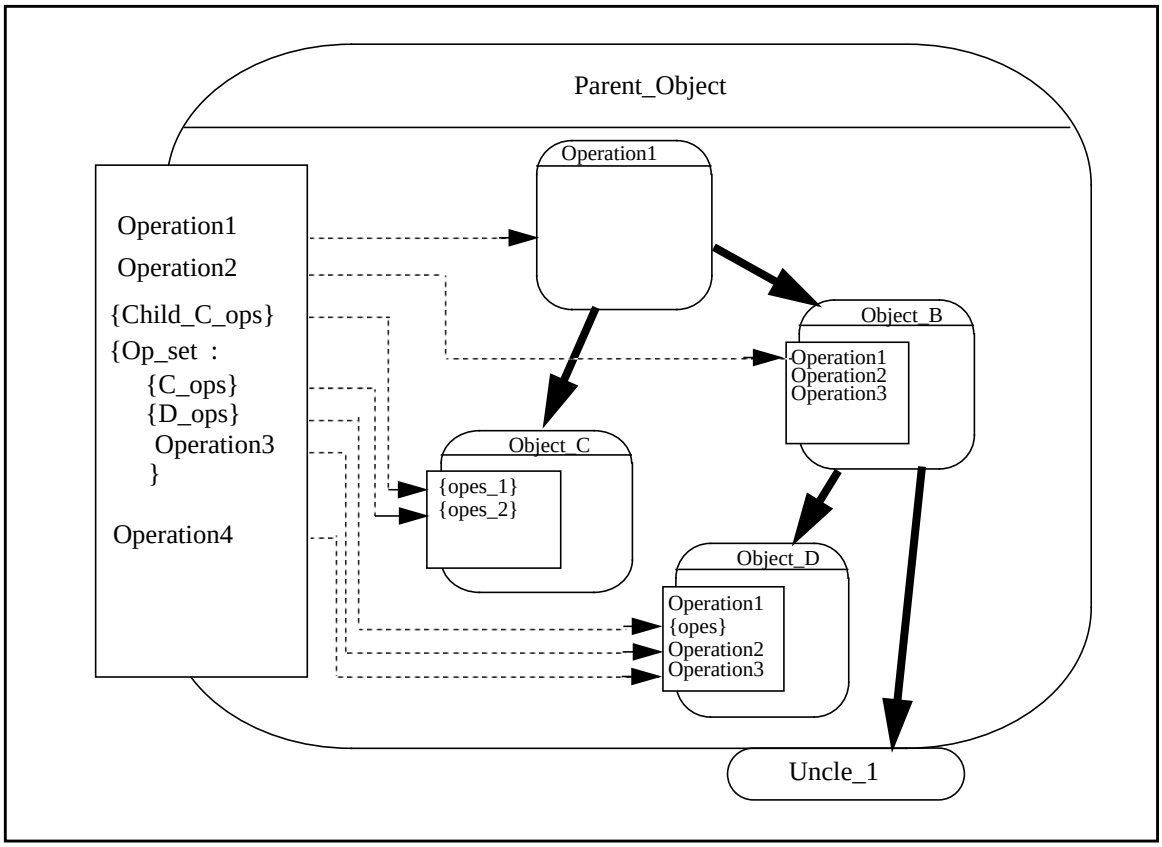

Figure 18 - Operation breakdown

- The parent provided operation Operation1 is implemented by several child operations trough an OP_CONTROL object named Operation1.
- When a client requests the provided operation Operation2, it effectively requests Object_B.Operation1.

[26]. which can only be defined at lower decomposition levels.

OP_CONTROL object which is represented as an object with no provided interface as shown in *Figure 18 - Operation breakdown*.

An OP_CONTROL object is always a terminal object, has no provided interface, is neither active, passive, constrained or unconstrained.

5.2.3 Overloaded Operations

Operations may be overloaded, i.e. they may have the same name but different signatures (parameters). In the graphical representation (see *Figure 17 - Overloaded Operation breakdown*), overloaded operations shall appear as many times as needed to represent one-to-one “implemented_by” links unambiguously.

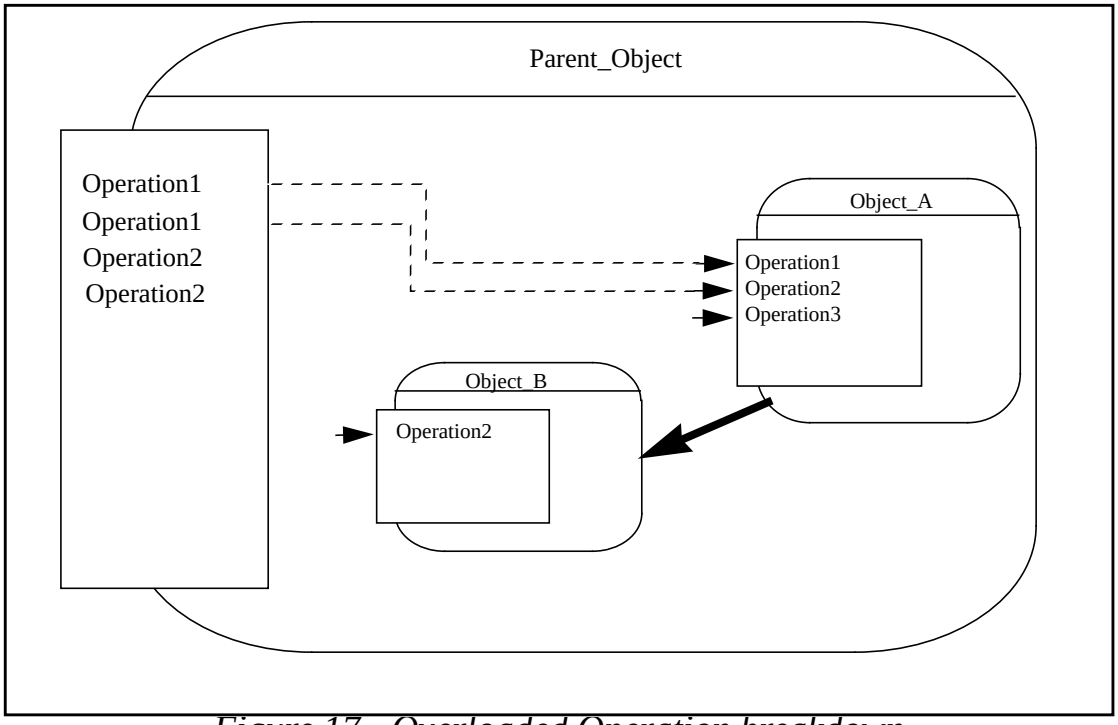

Figure 17 - Overloaded Operation breakdown

5.2.4 Unconstrained/Constrained Operation Mapping

An OP_CONTROL may implement a parent constrained operation by a non constrained child operation and vice versa. In that case, the OP_CONTROL object shall handle the control flow interactions[25] in order not to modify the dynamic properties of its parent object. Thus it is possible to build passive parent objects, which may be broken down into OP_CONTROL and **active** objects!

[25] An OP_CONTROL execution flow implementing an unconstrained operation into a constrained one may require OS services (“non-blocking” ones), or solely require ASER operation executions.

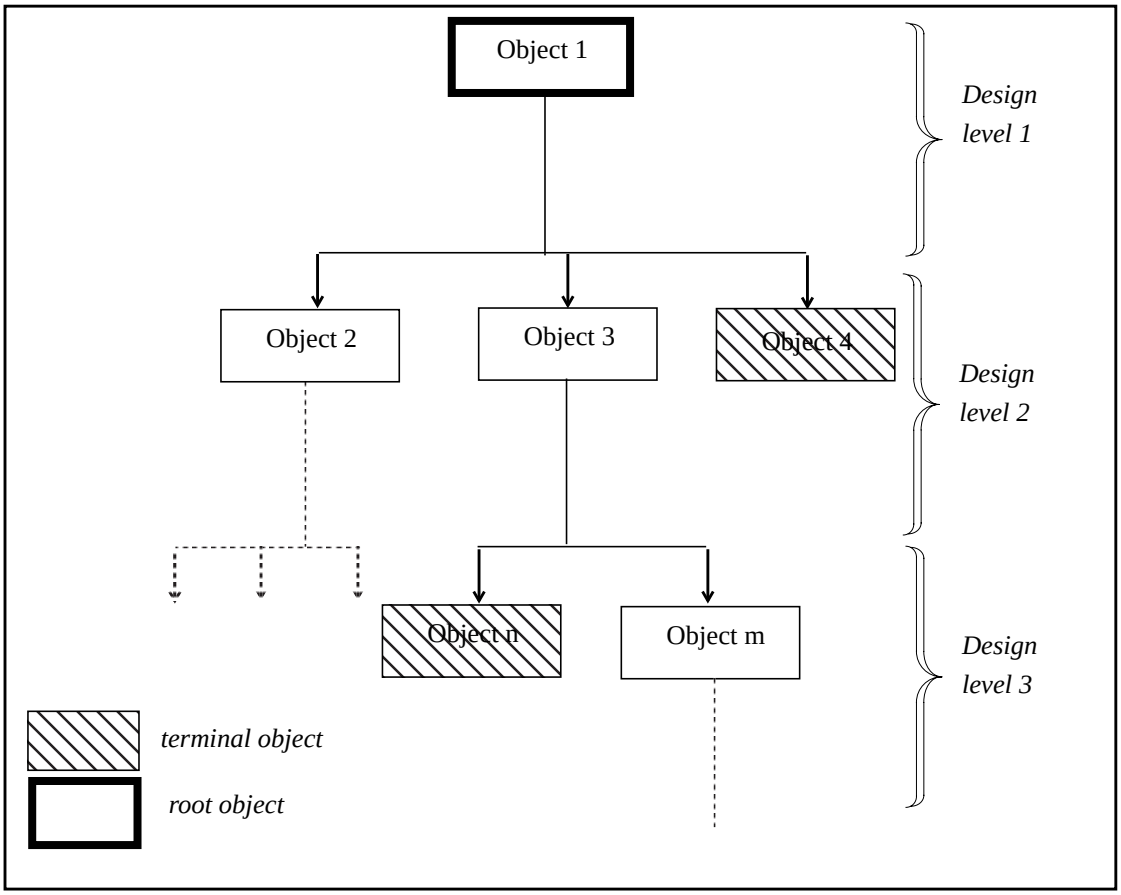

Figure 16 - Example of a HOOD Design Tree

5.2 OPERATION BREAKDOWN

5.2.1 One to one Mapping

Each parent operation shall be **implemented by** one child operation. The graphical representation is a dashed arrow from parent operation to child operation. In the internals of the ODS, the keyword **IMPLEMENTED_BY** and the identification of the implementing child operation is associated with the parent operation.

5.2.2 One to many Mapping

For a parent operation to be implemented through several child operations, a dedicated object of type **OP_CONTROL** may support the one to many mapping. The graphical representation is a dashed arrow going from the parent operation to the

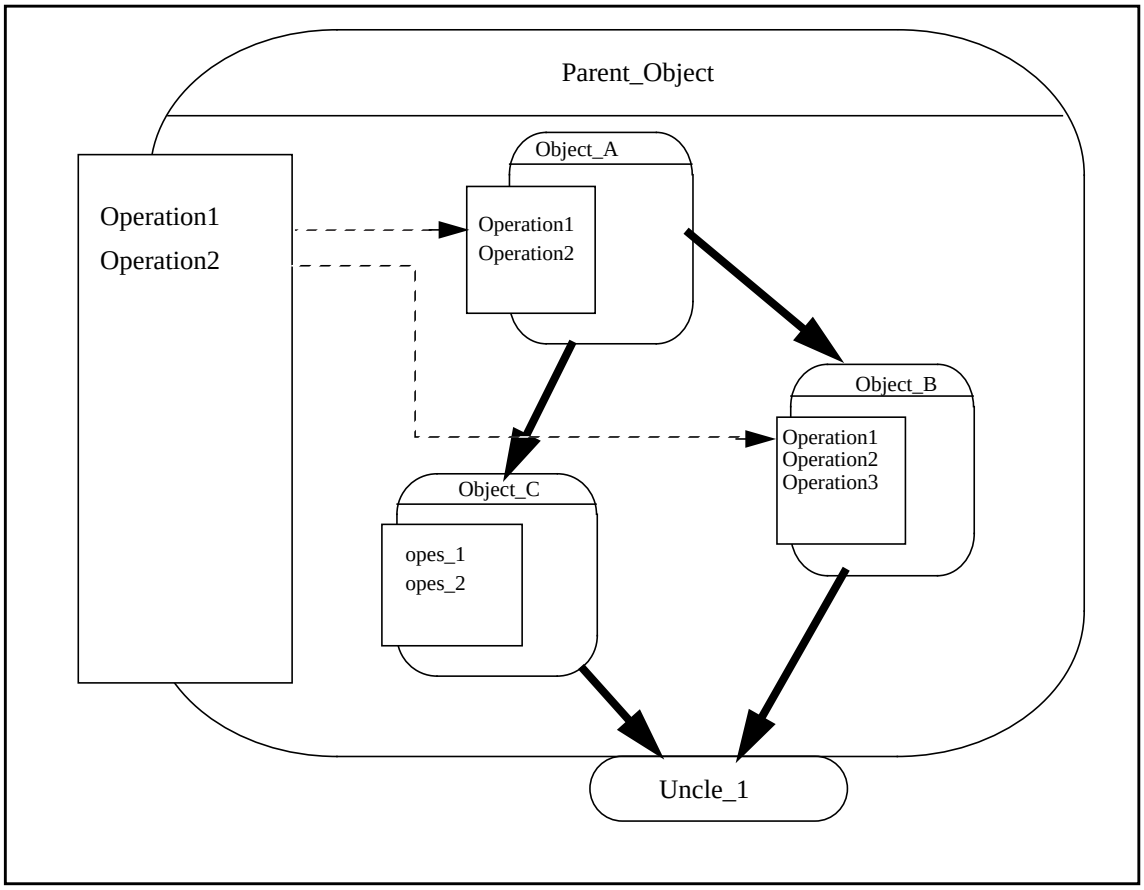

Figure 15 - Object breakdown

The **HOOD Design Tree** (**HDT**) is the tree of the system being designed. It consists of the root object and its successive breakdowns into child objects until terminal objects are reached.

Other root objects, seen as **environment objects** (see section 8), may be introduced to:

- allow consistent descriptions of the system to design with respect to its environment or,
- provide for **use** or **reuse of preexisting or externally designed software.**

Classes (see *section 9*) may also be introduced to factorize the description of object behaviours.

Figure 16 - gives an example of a HDT in which the breakdown has been made on three levels[24].

[24]. Two to six levels of breakdown are reasonable figures for a design

5 THE INCLUDE RELATIONSHIP

In order to provide a top-down breakdown of a system, a **parent object** is broken down into a set of **child objects** that collectively provide the same functionality as the parent. Each of these child objects may in turn be further broken down. This breakdown process is based on the **include relationship** between objects, where one object, the parent, includes other child objects.

A passive object may include active child objects as long as the parent passive properties are not violated[21] (see *section 16* for a detailed description of consistency rules).

The graphical representation (see *Figure 15 - Object breakdown*) is as follows :

- the include relation is represented by drawing the children inside the parent.
- the mapping between operations at parent level and operations at child level is represented by a dashed arrow called an **implemented_by** link.
- An object that is used by a parent object must also be used by at least one of the latter's child objects. This object, called an uncle object, is represented by a box, annotated with an active /passive indicator, attached at the border of the parent object. This box defines the connection port for child objects. Associated use arrows must be drawn from a child object to the uncle object and may have attached dataflows or exception flows already identified in parent diagram[22].

5.1 HOOD DESIGN TREE

A HOOD design is based on the principle of abstracting a system into an object architecture and refining it through successive breakdowns[23]. A given system can thus be described by a parent-child hierarchy with a root object representing primarily the system to design, and a number of objects at different lower decomposition levels. An object which is not decomposed into children is called a terminal object. Intermediate objects are called non-terminal objects.

[21] In that case the parent level decomposition previously validated would be inconsistent and would have to be redesigned, possibly leading to a full redesign of previous work!
[22] Consistency between parent level flow and child level flow should be ensured at design checking.
[23] As opposed to other design and refinement techniques, HOOD proposes a mechanism where the refinement of a given object (by definition of child objects) leaves upper level descriptions invariant.

4 THE USE RELATIONSHIP

An object is said to **use** another object if the former requires execution of at least one operation provided by the latter. The **use relationship** among objects defines **client-server** relationships allowing client objects to get services/operations executed by server objects according to a given communication protocols (and possibly expressed by applying an appropriate constraint on the server operation). The use relationship between objects is represented graphically by a bold arrow from the client to the server.

A system of objects using one another defines a use interconnection graph. HOOD recommends to avoid cycles in this graph according to guidelines given in *[PARNAS79]*. Such systems are much more difficult to test since they don't enforce an incremental development approach.

As an example, consider objects A, B, C and D (see *Figure 14 - Use Relationship Recommendations* below). If A uses B, B uses C, and C uses D, then HOOD recommends D not to use A.

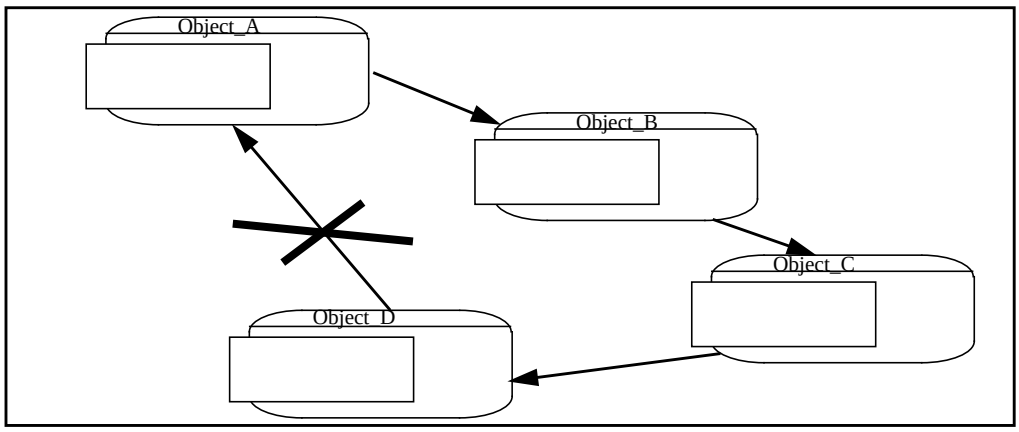

Figure 14 - Use Relationship Recommendations

In order to enforce software engineering principles and to improve software quality, HOOD encourages principles of low coupling and high cohesion between objects :

- The use interconnection graph of a set of objects should not be cyclic.
- The use interconnection graph should be of as low complexity as possible, i.e. objects should use as few other objects as possible but they should be used as much as possible.
- Passive objects should not use active objects (at least no concurency or protocol constrained operations), since their "system behaviour" could so be impacted. (they could so inherit active properties from their active servers)

INTERNALS --| *Formal Description of Interface Implementation* |--
OBJECTS
NONE -- since we are a terminal object
TYPES
List of (*Type_Name*[18] {*Complete_Type_Definition*}added with textual description)
CONSTANTS
List of (*Constant_Name*[1] {*Constant_Definition*} added with textual description)
OPERATIONS
List of (*Internal_Operation_Name* [list of (*Parameter : Mode Type*)] [**RETURN** *Type*]
added with textual description)
DATA
List of ({*Data_Name* : *Type_Declaration*} added with textual description)

OBJECT_CONTROL_STRUCTURE
CODE
OSTM
{*Object state Machine initialisation code in target language extracted from OSTD*}
Client_Obcs -- only if protocol constrained operation
{*Client code common to all protocol constrained operation-may be empty*}
Server_Obcs -- only if protocol constrained operation
{*Server code common to all protocol constrained operation-may be empty*}

OPERATION_ CONTROL _STRUCTURES
--| One operation description for each provided or internal operation |--
OPERATION *Name* [List of (*Parameter_Name : Mode Type*)] [**RETURN** *Type*]
DESCRIPTION
--| Textual description of HOW the operation works.|--
USED_OPERATIONS
List of ([*Object_Name*].*Operation_Name*)
PROPAGATED_EXCEPTIONS
List of (*Propagated_Exception_Name*)
HANDLED_EXCEPTIONS
List of (*Handled_Exception_Name*)
PSEUDO_CODE --| Pseudo code of the operation[19].|--
{*Associated pseudocode*}
CODE
{*OPCS_BODY code of the operation in target language*}[20]
--OBCS_CODE --| *Additional code for constrained operation support* |--
Opcs_HEADER ----| *OPCS HEADER code* |--
{*Associated code in target language*}
Opcs_FOOTER ----| *OPCS FOOTER code (may be empty)*|--
{*Associated code in target language*}
Opcs_ER ----| *Client ER code (may be empty)*|--
{*Associated code in target language*}
Opcs_SER ----| *Server SER code (may be empty)* |--
{*Associated code in target language*}
END_OPERATION *Name*

END_OBJECT *Object_Name*

[18]. Internal types and private part of provided types
[19]. Pseudo code field of the operation may contain any PDL.
[20]. This code field may contain internal declarations and the code sequence implementing the algorithms as well as the exception handlers in target language. It shall be taken as input for a code generator.

OBJECT[13] *Object_Name* **IS ACTIVE|PASSIVE**
DESCRIPTION
--| Informal text describing the functionalities (WHAT the object is doing) in a few lines. |--
IMPLEMENTATION_CONSTRAINTS
--| Describes informally any implementation and target constraints (memory, CPU, timing...) applicable to the object |--
PROVIDED_INTERFACE --| *Formal description of the services provided to other objects.* |--
TYPES
List of (*Provided_Type_Name* added with a textual description of WHAT the type is)
CONSTANTS
List of (*Provided_Constant_Name* added with a textual description of WHAT the constant is)
OPERATION_SETS
List of (*Provided_Operation_Set_Name* added with a textual description of the Set)
OPERATIONS
List of (*Provided_Operation_Nam*e [list of (*Parameter* : *Mode Type_Name* [*:= Default_Value*])[14]]
[**RETURN** *Type_Name*] [**MEMBER OF** *Operation_Set_Name*]
added with description of WHAT the operation does)
EXCEPTIONS
List of (*Provided_Exception_Nam*e **RAISED_BY**[15] list of (*Operation_Name*)
added with textual description of WHEN the exception is raised)
OBJECT_CONTROL_STRUCTURE
DESCRIPTION
--| Informal text describing the behaviour of the object in a few lines. |--
OSTD --| only if there are state constraints |--
--| Description of object behaviour model as a state transition diagram. |--
CONSTRAINED_OPERATIONS
List of (*Operation_Name* [**CONSTRAINED_BY** [*Label_text*]]
REQUIRED_INTERFACE --| *Formal description of the services required from other objects or classes*[16].|--
OBJECT *Required_Object_Name*

TYPES	List of (*Type_Name*)
CONSTANTS	List of (*Constant_Name*)
OPERATION_SETS	List of (*Operation_Set_Name*)
OPERATIONS	List of (*Operation_Name* [list of (*Parameter* : *Mode Type*)][17])
EXCEPTIONS	List of (*Received_Exception_Name*)

DATAFLOWS --| *Description of the Dataflow represented in the diagram* |--
List of (*Data_Name Direction Used_Object_Name* -- added with an informal textual description
List of (*Data_Name Direction OPERATION Op_Name* -- added with an informal textual description)
EXCEPTION_FLOWS --| *Description of the exception flows represented in the diagram* |--
List of (*Exception_Name Direction Used_Object_Name*) added with an informal textual description)
List of (*Exception_Name Direction OPERATION Op_Name*) added with an informal textual description)
--|-------------------------------------| *END OF VISIBLE PART OF THE OBJECT* |----------------------|--

[13] The following conventions are used to describe the ODS:
- keywords are in upper case,
- code entities names are marked in italic case,
- comments to the ODS are marked with the notation : --| text |--
- the other fields are to be filled in by the designer,
- optional entries are enclosed in square brackets : [...],
- target language dependent fields of the ODS are enclosed in braces {},
- “List of (*Items*)“means “item1, item2,..., itemN”,
- “Standard ODS” means all fields of active object ODS,
- “Standard field” means the field of the ODS of active objects related to the current topic.

[14] The syntax of parameter definitions is Ada whatever the target language.
[15] This allows to associate an exception to one or several operations.
[16] The fields of the Required Interface may be computed and documented automatically by toolsets, based on the analysis of other ODS fields (annotations, declarations of types, constants, data, operation parameters and OPCSs)
[17] The list of parameters is required to discriminate between operations of the same name (overloading).

3.3 OBJECT DESCRIPTION SKELETON

All properties of an object are described textually and whenever possible, formally, in a structured text framework called the **Object Description Skeleton** (**ODS**). This document is defined as a set of structured fields, separated by keywords, that may contain specific notations, languages or target language descriptions. An ODS may therefore be easily processed and exchanged for extracting relevant information for a given development activities (design checking, code generation, tracability, performance evaluation or metrication.) ODS fields are updated as the design of the unit is more and more refined. *Figure 13 - ODS Structure* outlines the main fields corresponding to the HOOD concepts described above: Object, operation, types, constants, data, OSTD, OSTM, OPCS and OBCS control structures.

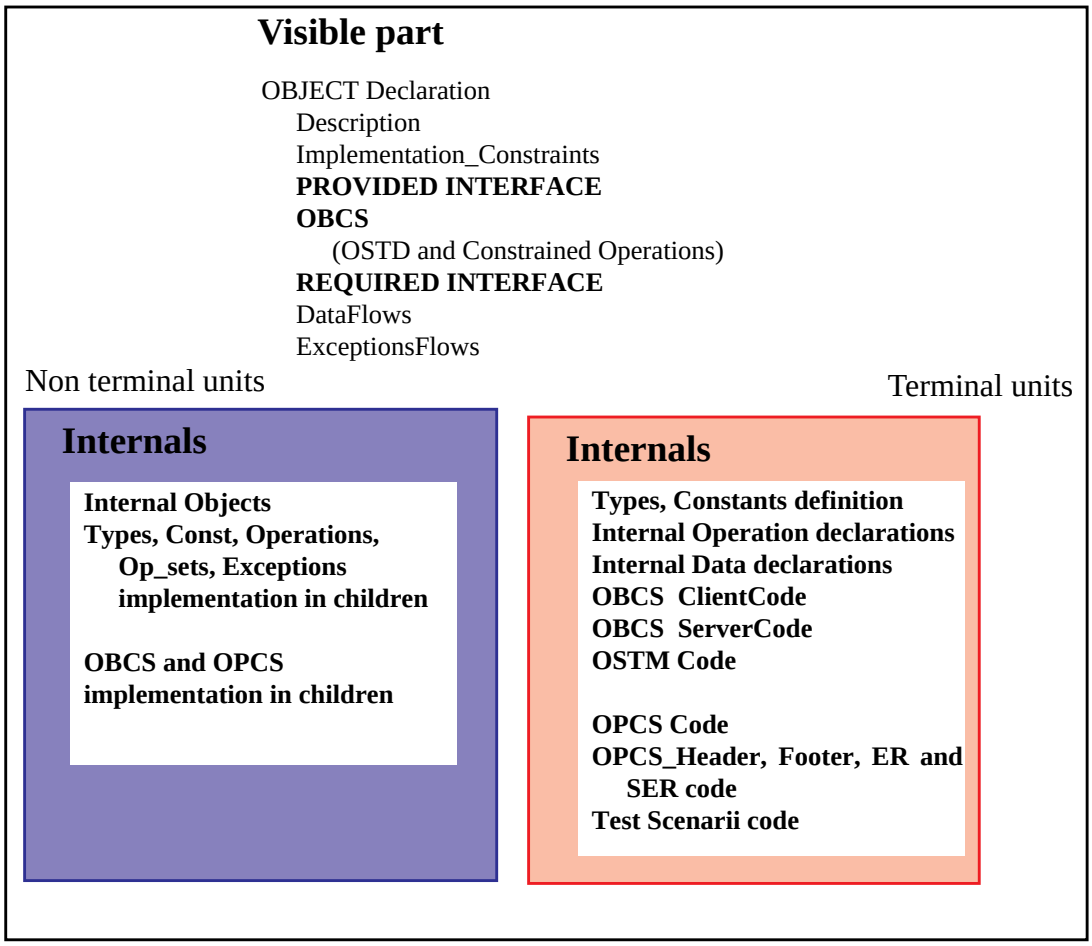

Figure 13 - ODS Structure

In the following we outline most of the fields of an ODS, whereas a detailed description of ODS variant and guidelines for filling the ODS fields are given in *section 14*.

3.2.5 ACTIVE and PASSIVE Objects

In order favour reuse and parallel development, HOOD partitions software as contributing or not to dynamic behaviour (which is most difficult to develop reliably), by defining two kinds of objects :

- **Passive Objects**: such objects shall not have any contribution to the dynamic behaviour of a system. Passive objects can thus be seen as “thread empty passive code” and may be developed with more common standards than other objects.
  Passive objects may thus have only state constrained operations, and their OBCS shall be reduced to an OSTD, and associated target code description parts (that is OSTM, OPCS_HEADER and OPCS_FOOTER fields see *section 3.3* below).
- **Active Objects**: these are all objects which contribute to the overall dynamic behaviour of a system. An active object can thus be seen as encapsulating one or several server execution threads, executing concurrently. **As a result, an object that is not passive is active.**

Figure 12 - displays the graphical representations associated to active and passive objects. Active objects have an “A” in the upper-left corner and have at least one protocol constrained operation.

Note that the property of “active” is really meaningful **at the level of terminal objects**(see *section 5* **below)** when reusing components; a designer has thus a direct mean to know whether its system behaviour may be affected when he decides to reuse a component in its design.

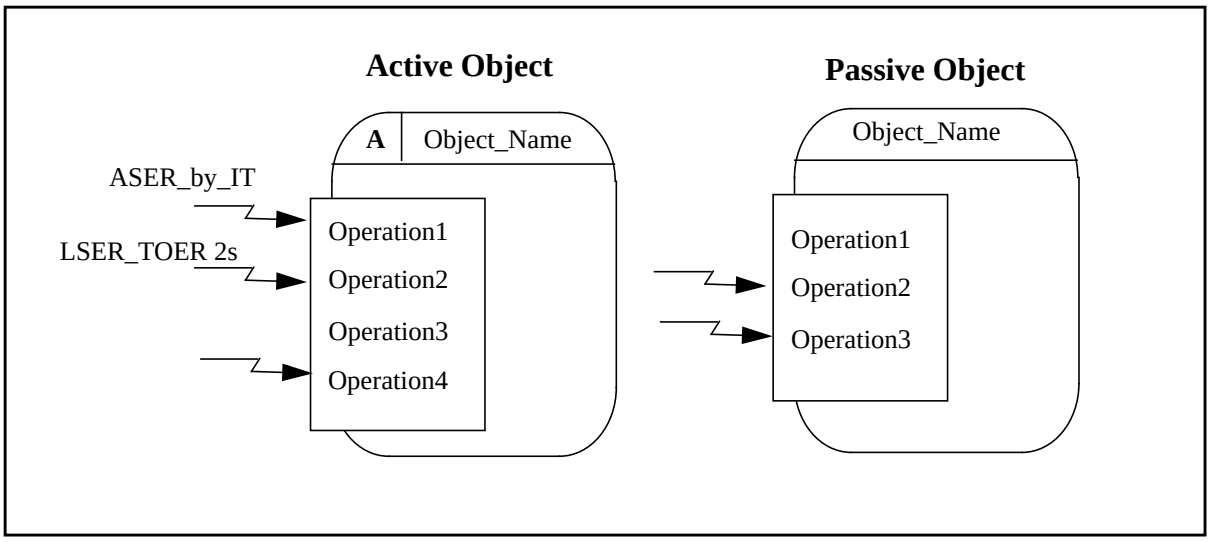

Figure 12 - Graphical Representation of Passive and Active Objects

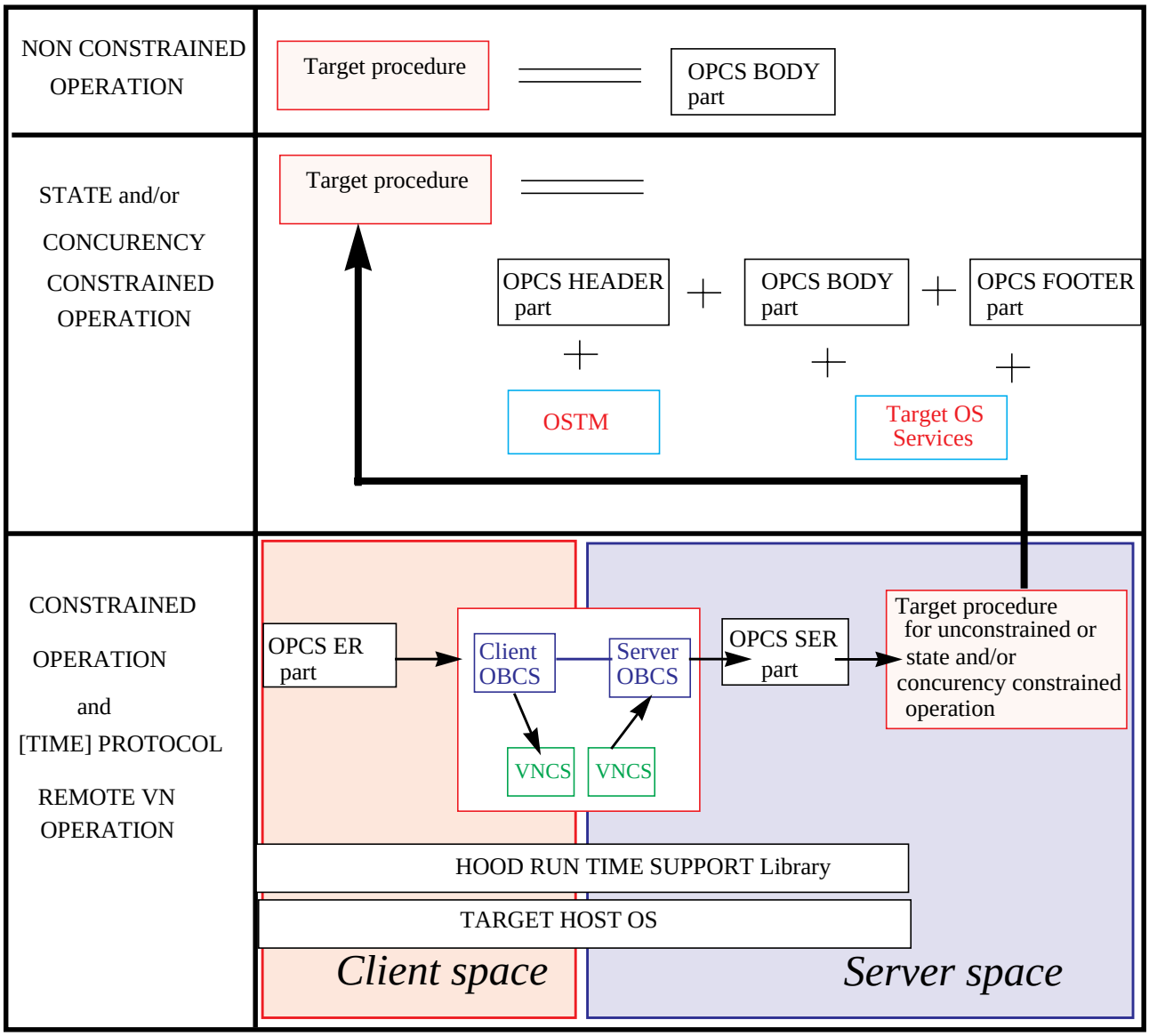

Figure 11 - Target Operation model

Note that Client_Obcs and Server_Obcs target units are common to all protocol constrained operation, whereas Opcs_ER, Opcs_SER, Opcs_HEADER, Opcs_BODY and Opcs_FOOTER are associated to each constrained operation.

Code generation rules can now be derived by defining target source code fields (which may be empty) associated to these logical parts according to the type of constraints attached to an operation. A full specification such associated code is given in *section 17* for each kind of constraint. **The code associated to constrained operations may thus be automatically generated and can be defined once for all**, so that the designer or programmer can concentrate on the OPCS body part.

HOOD Design implementation and code generation for multiple targets is discussed in detailed in *section 17*.

Implementation on target languages and systems shall enforce these principles of separating pure sequential, thread-safe software from concurrent inter-process software associated to HOOD protocol constrained operations :

- the sequential code is defined in the OPCS associated to an operation,
- the concurrent code is defined as additional code and is as much as possible encapsulated in one target unit.

This execution model, illustrated in *Figure 10 - HOOD Object execution model*, has several advantages for developing more reliable and reusable software. Especially it allows:

- separate and parallel development of reactive and transformational parts, possibly by specialized development teams,
- easy dynamical behaviour prototyping, before or independently of sequential code availability,
- expression of dynamical behaviours using high level or formal notations in order to perform verification and possibly formal checks on some behaviour parts,
- the definition of a client-server architecture implementation, independently of any ultimate target

An associated supporting client-server architecture is best achieved by splitting the OBCS supporting code into target units across client and server execution spaces with respect to operation constraints. An associated *target constrained operation model* illustrated in *Figure 11* - allows the specification of automated multi-target code generation whatever constraints and targets.

The logical code parts associated to the OBCS concepts have been defined as:

for state constrained operations:

OPCS_HEADER	code associated to a protected[12] and/or state constrained operation
OPCS_BODY	sequential code of the operation body
OSTM	Object state transition machine target code associated to the OSTD
OPCS_FOOTER	code associated to a protected constrained operation

for protocol constrained operations:

OPCS_ER	client code associated to a protocol and time constrained operation
CLIENT_OBCS	code to queue execution request and waiting return parameters
OPCS_SER	server code for executing the effective operation
SERVER_OBCS	code to dequeue Op requests, process them and return back effective operation parameters

[12]. protected means here ‘execution in mutual exclusion’ or in shared Read only

3.2.4 Control Structures and Target Models

3.2.4.1 Definitions

The description of the dynamical behaviour attached to the execution of operations (with or without attached constraints) deals with the description of *thread interaction and execution* that are ultimately implemented on target machine processors. The behaviour of such threads is captured through two orthogonal concepts :

- **the OPeration Control Structure** (*OPCS, one by operation*) which describes how a sequential thread executes within an operation. *The **OPCS** defines the way the execution flows sequentially within the operation.*
- **the OBject Control Structure** (*OBCS, one per object)* which controls the activation condition and the triggering of execution request onto server threads a**s defined by operation constraints**. *The **OBCS** defines the way client or server threads may execute provided operations according to their constraint specification.*

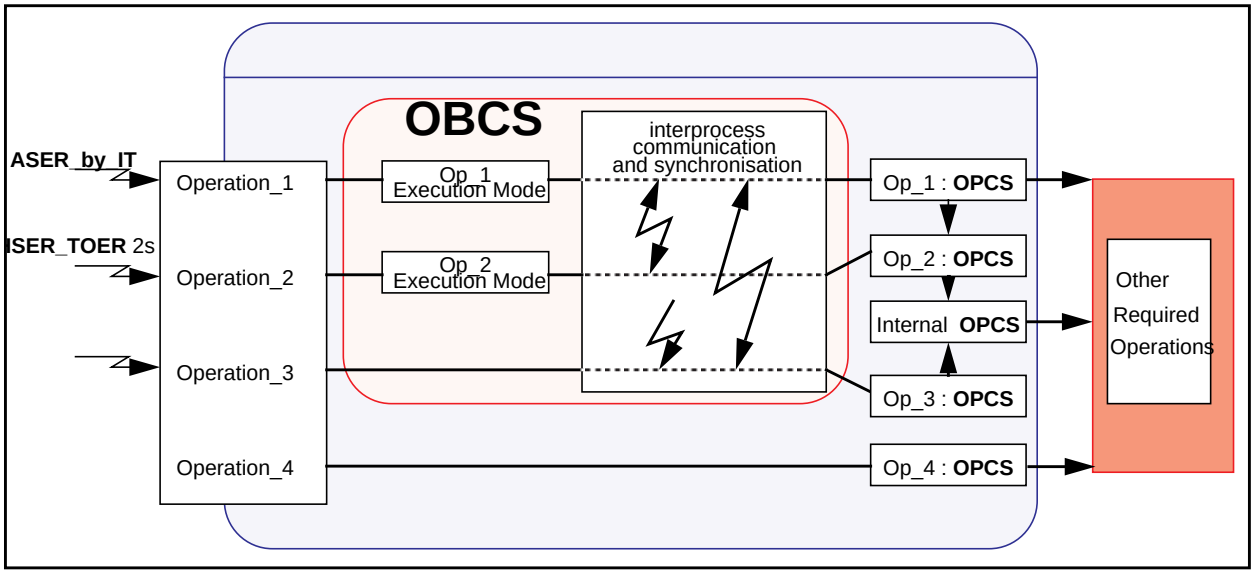

Figure 10 - HOOD Object execution model

3.2.4.2 Implementation Principles

HOOD OPCS and OBCS Control Structures concepts allow to **specify and implement** a client-server communication model **enforcing separation of sequential code from concurrent one**. This separation is intended to support feasibility analysis or prototyping of real time behaviours (by implementing all OBCSs) completely in parallel with the development of the functional/sequential code (defined in OPCSs) of a system.

3.2.3.4 Time constraints descriptions

The TimeOut constraint (TO) allows **a** client to request the operation to be executed within a given period of time. The timeout is the maximum elapsed time (within the client time reference) before the request is taken into account or executed by the server. TO can be combined with above protocol [10] constraints only. Client control flow is interrupted and resumes (see *Figure 9 - Timed-out Execution Requests (HSER)*) either when :

- the time-out has occurred or
- the service has been acknowledged or completed.

By default the server flow continues always its execution (until the nominal end of the operation or until an exception occurs), whatever the time-out has occurred[11].

For timed-out execution requests, a timeout default value is attached to the operation, but the client may override it at request time. The time is always measured in client's space, and no global time is supported.

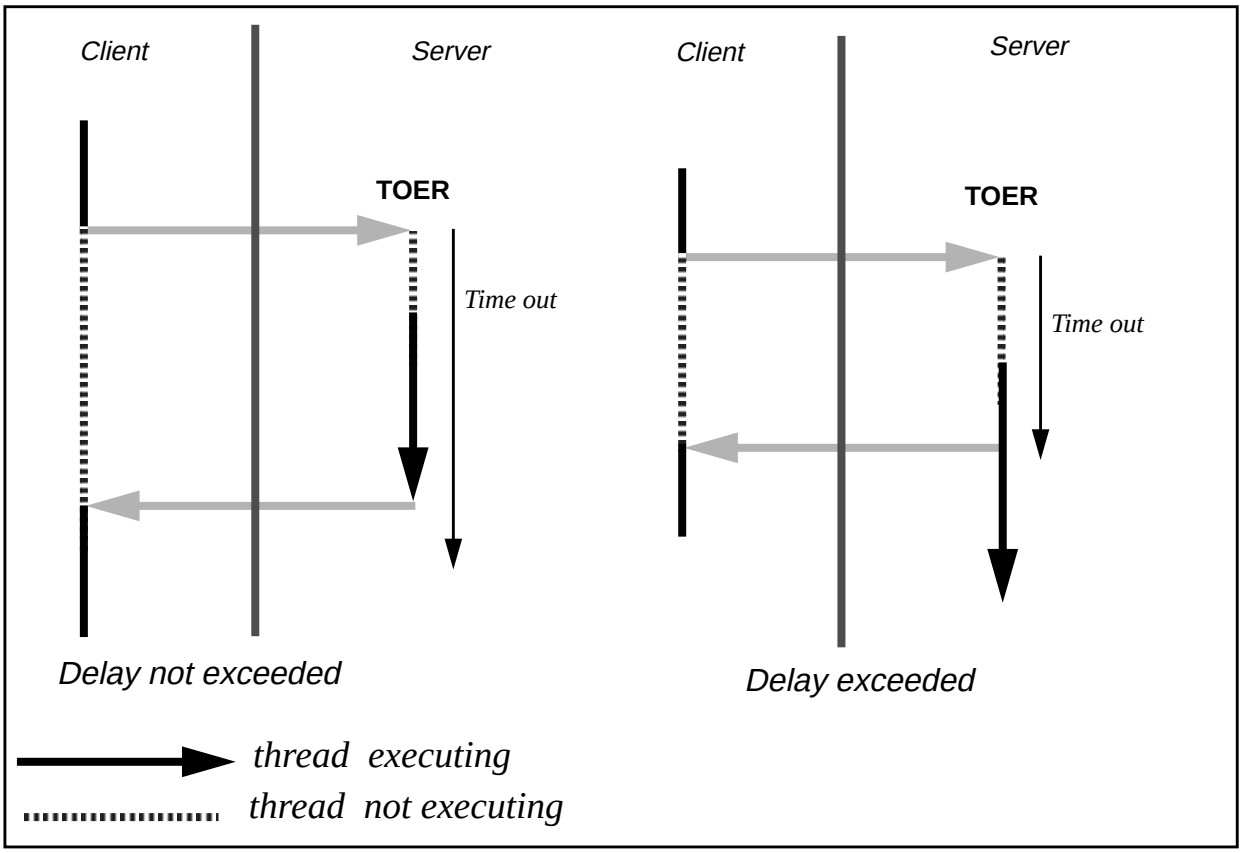

Figure 9 - Timed-out Execution Requests (HSER)

[10] Time out applied to a single execution flow is not sensible since in case of timeout, no other flow can perform an action.
[11] this is because it would be extremely difficult to implement a safe "abort" operation for all possible cases; thus this is left to the designer.

- **Reporting Loosely Synchronous Execution Request (RLSER) :** the client control flow is interrupted until the server acknowledges the request. At a later time the client shall await results from the server. (see *Figure 8 - RASER and RLSER Execution Requests*).
- **Reporting Asynchronous Execution Request (RASER) :** the client control flow is not interrupted, but the requested operation is triggered within the server. At a later time the client shall await results from the server[9]. (see *Figure 8 - RASER and RLSER Execution Requests*).

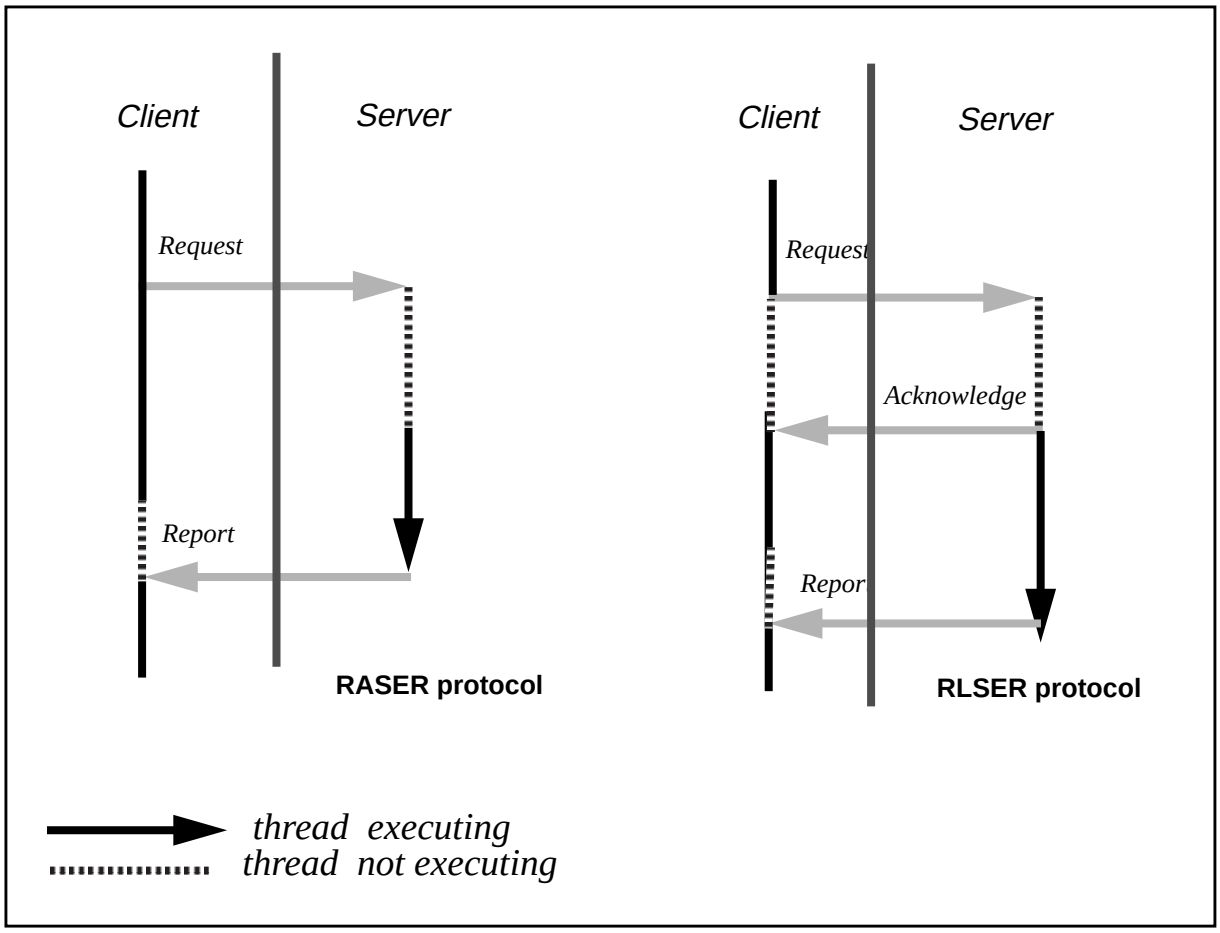

Figure 8 - RASER and RLSER Execution Requests

[9]. RLSER and RASER Operations could be implemented by normal ASER and LSER operations plus specific HSER operations to get associated results. The combination achieved however by RASER and LSER constraints allows the design of parallel programs to be achieved in a much more concise, readable and verifiable way.

3.2.3.3 Protocol constraints descriptions

Protocol constraints are described through predefined texts of trigger labels. It is assumed that *the underlying implementation communication model is asynchronous*, that is: when a client issues a request to a server, it may continue its execution even if the server is not ready to serve its request. A protocol constraint is defined with a text starting with a keyword that defines one of the following protocol:

- **Highly Synchronous Execution Request (HSER) :** the client control flow execution is interrupted by a target thread or operating system mechanism. Control is given back to the client when the requested operation has been performed (this corresponds to a WAIT_REPLY communication protocol, see *Figure 7 - Unconstrained and Protocol constrained Execution Requests*).
- **Loosely Synchronous Execution Request (LSER) :** the client control flow is interrupted. Control is given back to the client when the server acknowledges the request (this corresponds to an ACKNOWLEDGE communication protocol, see *Figure 7 - Unconstrained and Protocol constrained Execution Requests*).

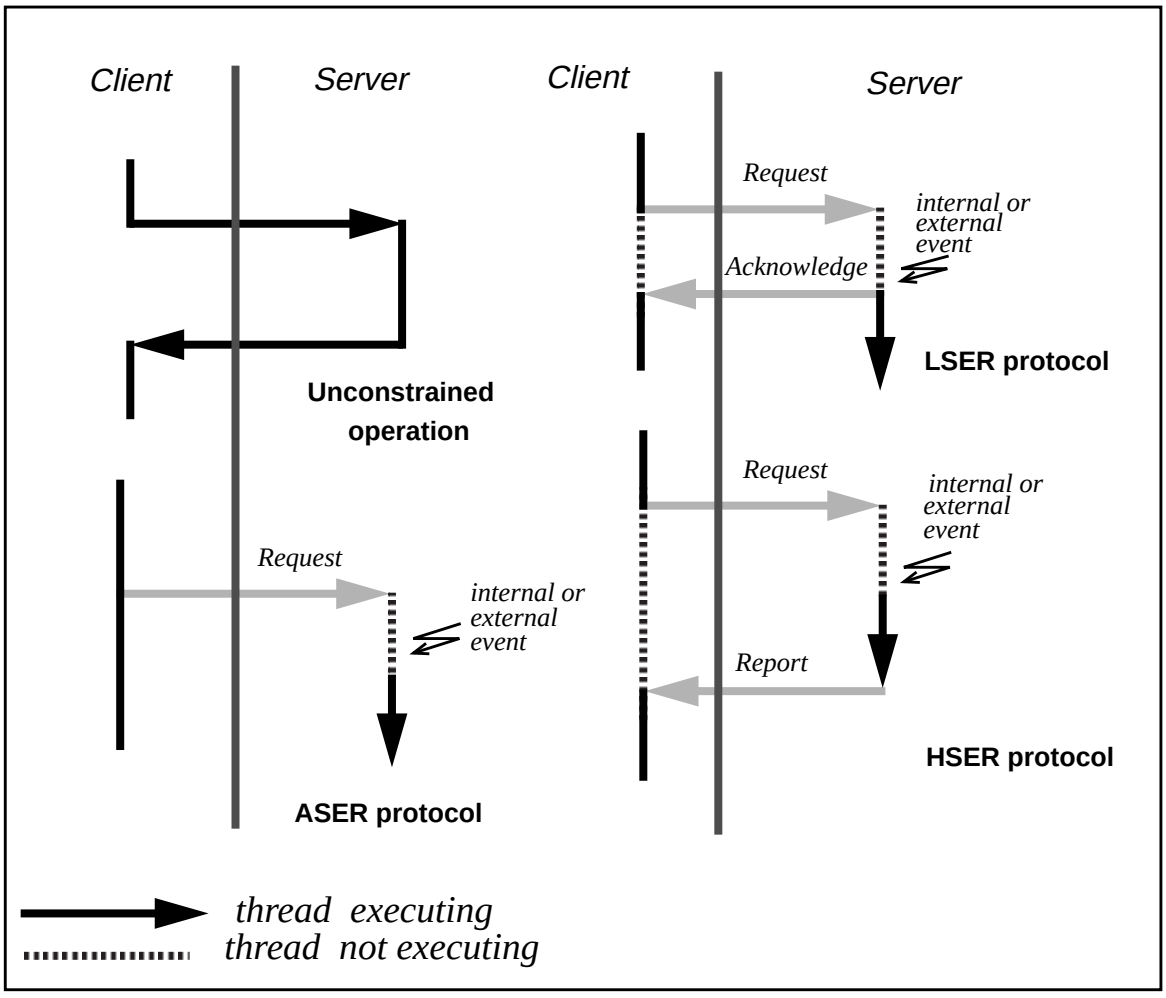

Figure 7 - Unconstrained and Protocol constrained Execution Requests

3.2.3.2 Concurency constraints descriptions

Concurency constraints are described through predefined texts of trigger labels:

- **Mutual EXclusion Execution Request (MTEX) :**
  The operation[7]is executed in mutual exclusion within the server's concurrent memory space. Such an execution request can be combined with above state constraints. *Figure 6* - illustrates the execution flows with MUTEX constraints.
- **Read Only Execution request (ROER)**:
  The operation may be executed together with different concurrent flows. Such constraint is used to declare this operation as a "reader" controlled by "reader-writer monitor"*[BURNS90]* and allowing several execution threads to operate simultaneously and get consistent results[8].
- **Read Write Execution request (RWER**):
  This constraint defines the operation as a "writer", controlled by "reader-writer monitoring *[BURNS90]* allows operation execution in mutual exclusion of all other concurrent reader threads.

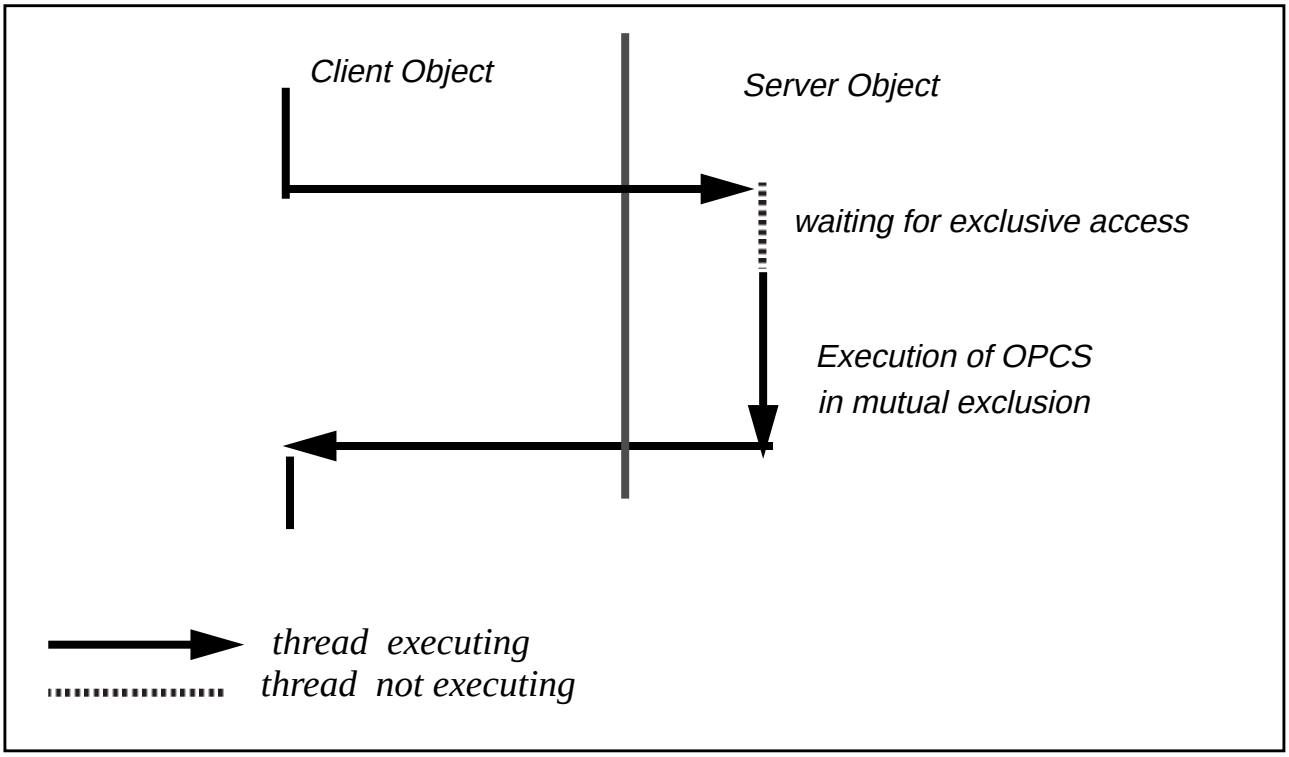

Figure 6 - MTEX Execution Requests representation

[7]. in fact the OPCS body

[8]. Note that when the target language is Ada, such operations may be mapped into access operation onto protected records

In order to support refinement (i.e. addition of more states leaving existing one unchanged) of such states, OSTD diagrams may be described as nested OSTD. Thus a transition may:

- “exit” from the OSTD, or
- “call” another OSTD, or
- “return” to the calling OSTD, in a given “return state”

Figure 5 - Example of an Object State Transition Diagram illustrates such an OSTD, for the STACK object represented in *Figure 4 -*. A full description of the OSTD textual formalism is given in *section 14*, while *Appendix B - HOOD GRAPHICAL SYMBOLS-* gives a full description of OSTD graphical syntax.

.

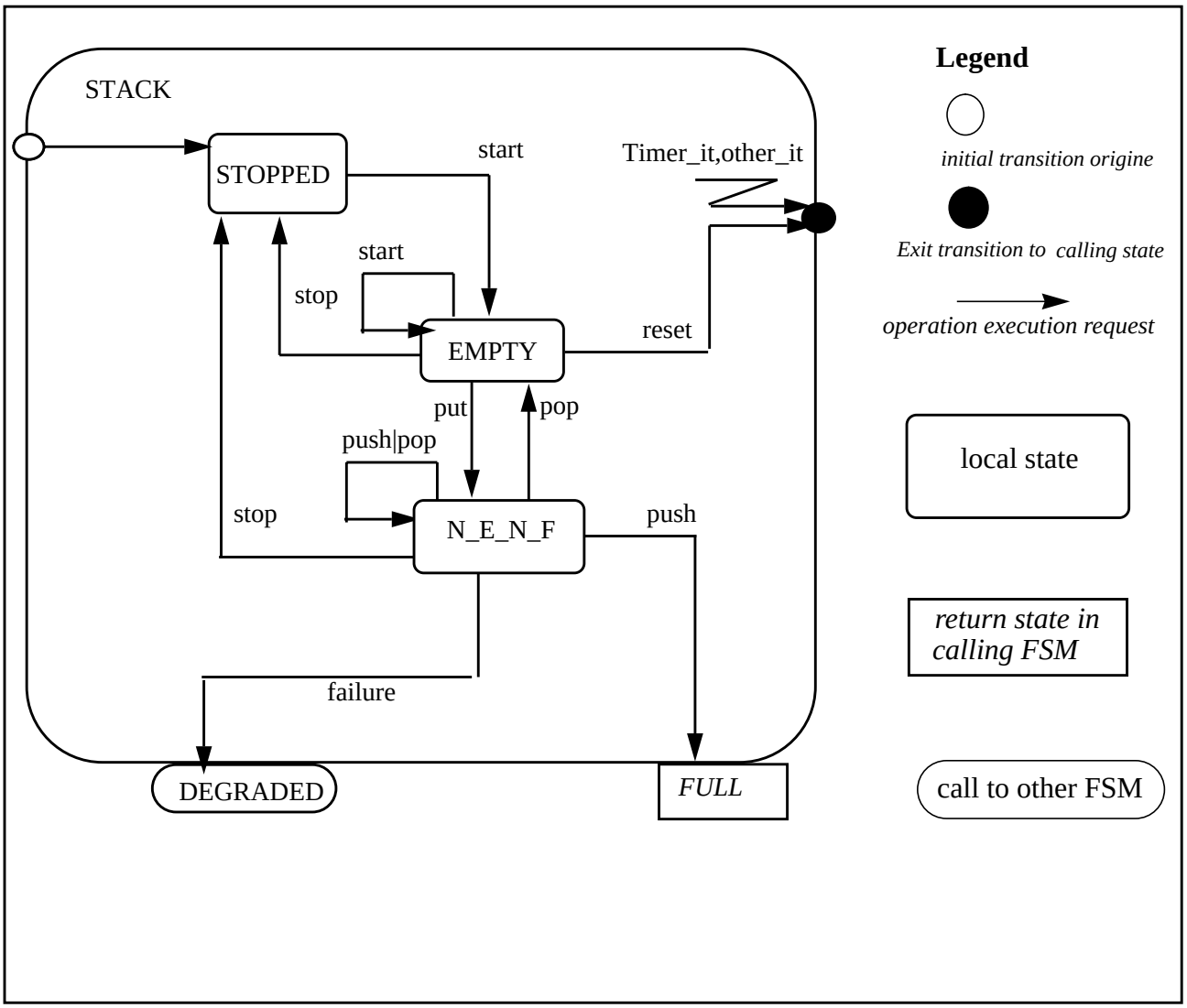

Figure 5 - Example of an Object State Transition Diagram

State, Concurency, and Protocol constraints are orthogonal concepts that may simultaneously apply to a same operation. Time constraints may only be combined with protocol constraints. Operation constraints are graphically represented by attaching a “trigger arrow” to an operation:

- arrows without label indicate that an operation is solely state constraint (see Start and Stop operations in *Figure 4* -)
- arrows with a label indicate that an operation has a combination of State, Concurency or Protocol constraints (see operation PUSH and POP in *Figure 4* -). This label holds a textual description of the constraint and may be used later for automated code generation. In that case, a precise syntax (as defined in *section 14*) shall be used.

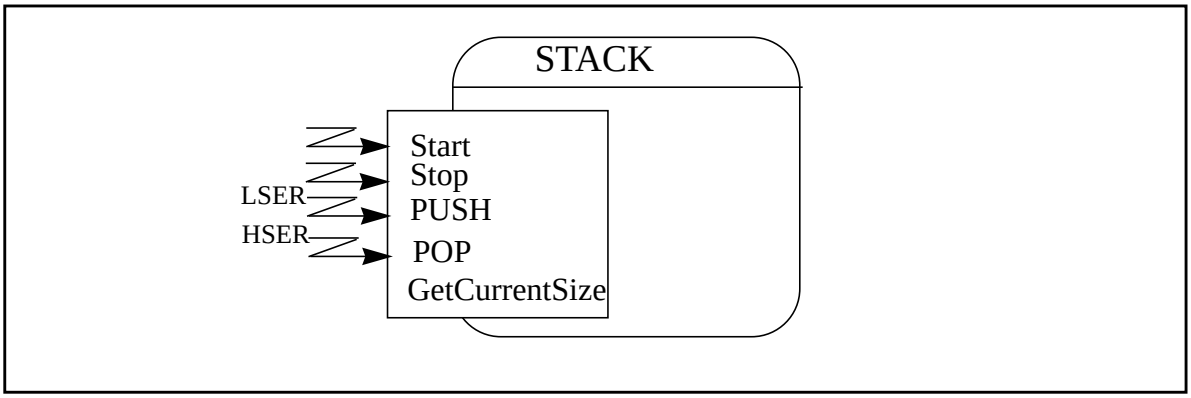

Figure 4 - Graphical Representation of Constrained Operations

3.2.3.1 State Constraints Description

State constraints may be described as a state transition system where transitions are triggered solely by provided constrained operation execution requests. This model implements the *precondition and postcondition programming approach [MEYER88], [MOTTET95]*for the specification of an operation and shall be described, in the textual formalism, within an *Object State Transition Diagram* (OSTD). As a result:

- a state in an OSTD is a subset of the global state of the object, where one or several provided constrained operations may be triggered. An OSTD state is represented graphically as a square box.
- a transition of an OSTD can only be labelled with a constrained operation name. A transition is represented graphically as a labelled arrow. Several transitions with same label are allowed from one state towards another[6] (including cycles).

[6]. non determinism of such descriptions shall be solved from the client code (OPCS) which shall be able to set the OSTD states according to locally known conditions.

and **constraining the execution** associated to operations, thus warranting a given semantic and state.

Some operation executions (such as iterators, or state copies) may never enforce a semantic change in the object state, whenever time they are triggered. However others must *enforce a* given *protocol, synchronise, enforce pre-conditions* and *post-conditions* to ensure a correct semantic for the object.

Let us take a STACK object represented in *Figure 4* -as an example. Such an object should never execute a PUSH operation when the STACK is FULL.Hence the PUSH operation shall be “state constrained”.

The following kinds of constraints are defined and may be combined orthogonally together:

- **State constraints**
  The execution of an operation within the server object depends on its internal state possibly resulting from previous executions. In order to avoid any behavioural impact, HOOD recommends a non-blocking implementation implying only the current (client)execution thread[4].
- **Concurency constraints**
  The execution of an operation may take place within a server thread in concurency with multiple other threads. The MUTEX constraint shall be used to specify that the target operation code is to be executed in mutual exclusion*[BURNS90]*.
  The ROER constraints shall be used to specify multiple read only access for the target operation code according to a reader-writer schema*[BURNS90]*
  The RWER constraints shall be used to specify exclusive write access within the target operation code according to a reader-writer schema.
- **Protocol constraints**
  A server receives an operation **execution request** and reacts in order to implement the associated communication protocol. Note also that the implementation of such constraints imply at least one client and one server execution threads to fulfil the operation.
- **Time constraints**
  The execution of an operation or a request may be limited in time. Therefore time constraints shall allow to specify how long an execution request can last in the worst case. The behaviour of the client execution thread when such deadlines are met, is as follows: when it gets control back[5], it may check that the deadline was overrun and execute an appropriate action.

[4] This results from pragmatic experience of client using possibly blocking code. The best working strategy is in case of automated code generation generally the non-blocking one, where the client is just to be warned tat the service he wanted was not available at the time he requested it.

[5] The exact behaviour is not specified and left to implementation, e.g. an exception Time_Constraint_Error could be raised, or asynchronous transfer of control could happen, or a status could be updated in the client thread memory space.

- the **asynchronous model** where the client resumes its execution just after issuing the request to the server, whatever the state in the server[2].

3.2.2 Control Flow and Dynamical Behaviour

The *dynamical behaviour* of a system is associated to the temporal ordering of execution threads and is a key issue in the design of complex systems.At the opposite of popular OO methods such as*[OMT91]*, *[COAD91]*, [*[SH&M92]*, *[BOOCH86]* and others, HOOD addresses concurrent, real time and distributed processing. HOOD provides a consistent framework to master the design of a wide range of operational systems such as embedded and control process ones. **A key feature of this framework, regarding system behaviour, is the concept of constrained operation.** This concept (see *section 3.2.3*) allows execution threads behaviours to be specified one by one, in an implementation independent way.

System behaviour is achieved as the composition of thread behaviours resulting from their execution within operations and their interactions as objects communicate. **A control flow behaviour is associated with an execution thread**[3] **and may be** :

- *Sequential:* in this case the same execution thread flows from a client operation into a server one and **no interaction with another execution thread can take place**. The flow of control is executing fully *sequentially* within the internals of the operation i.e. in the **OPeration Control Structure (OPCS)**. After completion, the control returns back to the client.
- *Concurrent:* in this case interactions between the client execution thread and other threads may take place as the control “flows in” the server. The **OBject Control Structure (OBCS)** receives execution requests from client threads. Reaction to such requests will depend on the server internal state as well as on the communication protocol constraint.

3.2.3 Constrained Operations

In order to model the interactions of sequential and concurrent behaviours of threads in real time software systems, HOOD provides a consistent set of *operation constraints* allowing various aspects of real time software design to be taken into account, including support for schedulability analysis. *Full communication and synchronisation protocols* between a client and a server may be specified by attaching execution requests

[2]. This requires that the implementation provides an intermediate resource (such as a mailboxes) to hold/store the request until the server is able to process it.

[3]. Such a thread has first to be seen as a logical one, and may be implemented in the target system as a scheduled entity such as simple Ada task or physical thread, or even as a physical heavyweight process*[MULLER89]*.

- The interface part defines the services (**types**, **classes, constants**, **operations** [with associated **parameters**] and **exceptions**) provided by the object, as well as the services required from other objects.
- The internals part defines the implementation of the provided interface, by internal types, constants, operations and exceptions, **or through internal objects and/or classes**. This property allows a set of object specifications to be defined in term of child object descriptions. This is a **key feature for refinement** that distinguishes HOOD from other design methods.

3.2 DYNAMIC PROPERTIES

3.2.1 Communication between Objects

Communication between objects is only possible by service requests, which are similar to Smalltalk method calls or Ada procedure / task entry calls. The communication is then achieved by execution of operations. An object requesting execution of an operation is called a **client object**. An object performing execution of an operation is called a **server object**. The following conditions must be satisfied :

- the provider of the service must be named by the client.
- the name of the client is not known by the server.
- a client object requesting a service from a server object is said to **use**[1] **the server** and this is represented graphically by a bold use arrow. (see *Figure 14 - Use Relationship Recommendations*).
- there is a many-to-one connection pattern, reflected by the naming scheme.
- information and **data items can be exchanged** between communicating objects and the associated information flow may be bidirectional and is represented by arrows along the use relationship one (see *Figure 19 - Dataflow Representations* in section 6).

HOOD supports two models of communication:

- the **synchronous model** where an operation execution request defines a synchronization point between the client and the server; if the server is busy when the client send the request, the latter is suspended until the server is able to service the request.

[1.] This communication model (which should be enough detailed at the design level) abstracts from any implementation model which may be synchronous (i.e. when the client send a request to a server, it is suspended if the server is not ready to process it) or asynchronous (i.e. when the client issues a request, it is not impacted by the state of the server). Implementation issues are further discussed in *section 3.2.4* below.

3 OBJECT AND OPERATION CONCEPTS

HOOD provides mechanisms to overcome the limitations of flat design descriptions in the field of large and complex systems by defining an object model and architecture through a globally top-down stepped design process allowing validation and verification procedures to be improved.

The formalisation of the object concept is based on both graphical and textual notations and supported by the **Object Description Skeleton** (**ODS**). These formalisms are used in a complementary way in order to capture both static and dynamic properties of an object.

3.1 STATIC PROPERTIES

An object has a visible part (the **interface**), and a hidden part (the **internals**) which cannot be accessed directly by external objects. An object is only accessible from the external world by its name. The associated graphical representation is shown in *Figure 3 -*

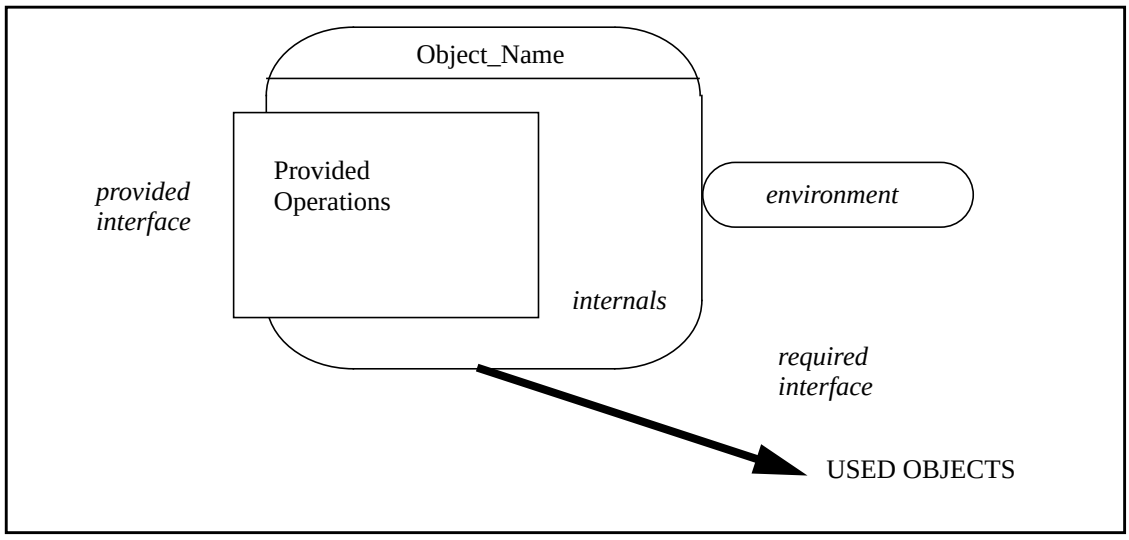

Figure 3 - Graphical Representation of a HOOD Object

2.2 SYSTEM DESIGN

A system to be designed may be modelled according to three views corresponding to different abstraction levels :

- the **logical space** view, consisting of a **design space** structured as a set of design trees and a set of ENVIRONMENT trees that define formally the environmental and reusable software. In order to factorize descriptions of similar and context-free reusable objects, GENERIC objects or classes may be defined either within the design space or in a dedicated generic space.
- a **distribution space** view, which deals with the definition of indivisible units of distribution and **physical processes** by **allocation** of objects.
- a **physical space** view, which deals with the definition of physical nodes by **configuration** of distribution units.

HOOD deals with logical and distribution views as illustrated in *Figure 2 - System Design Views*.

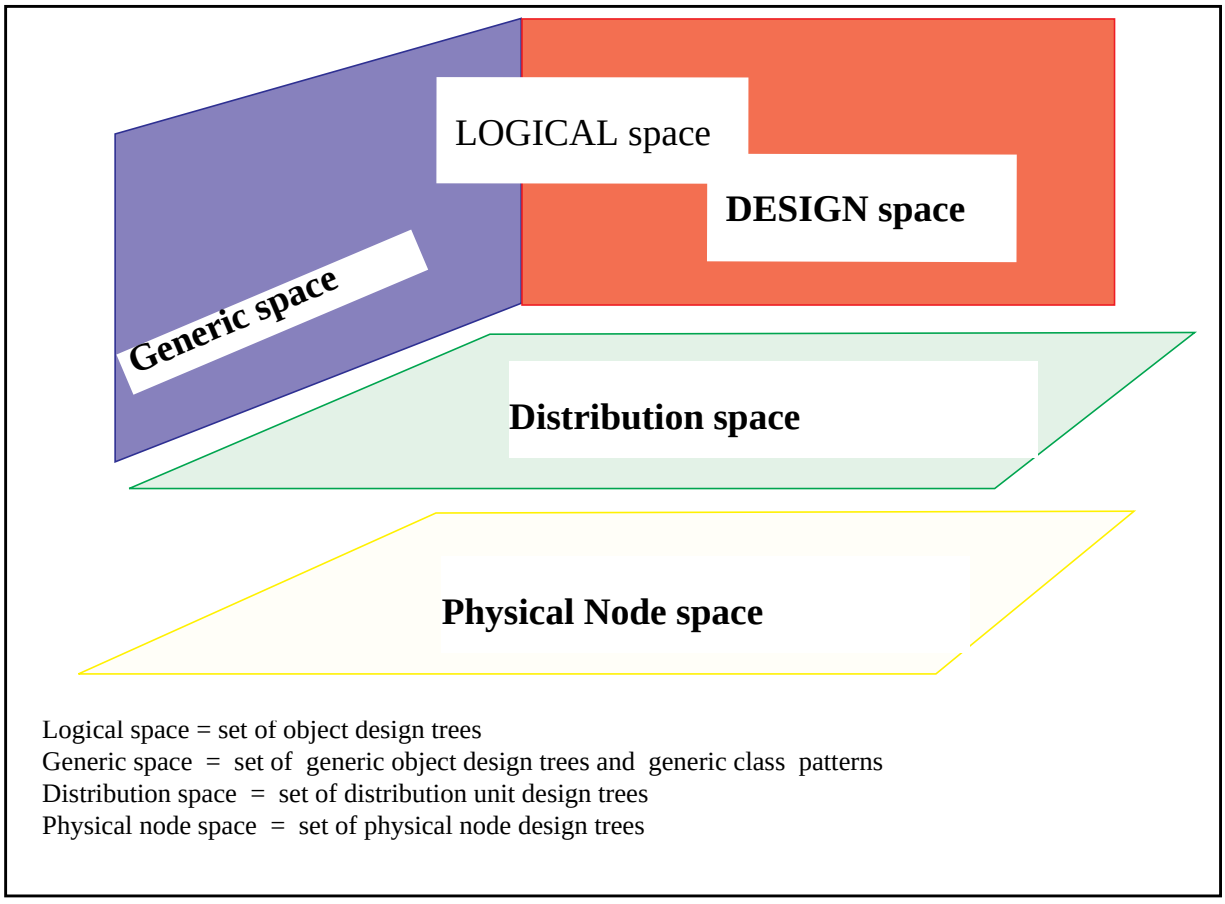

Figure 2 - System Design Views

HOOD defines a design technique which allows structuring of a software solution in modules both associated to the above design entities and enforcing the following principles of:

- *Abstraction, information hiding and encapsulation :*
  An object is defined by the **services** it provides to other objects, the services it requires from other objects and its **behaviour**, whereas its **internal structure is hidden to clients**. An object is thus defined through its **interfaces** with respect to its environment. Hence a system can be built by interconnecting objects and verified according solely to the properties provided by the interface descriptions.
- *Hierarchy and Refinement :*
  - Objects may **require execution of operations** provided by other objects, so that a system can be represented as a **use interconnection graph**. This graph should be structured into a senior-junior hierarchy as suggested in *[GOOD86]*
  - Objects may **be decomposed into other objects and classes**, so that a system can be represented as an **include** graph or **parent-child** hierarchy of objects *[GOOD86]*, *[MOTTET91]*.
  - **Successive breakdowns** and **refinements** of parent objects into child objects allow complexity to be mastered and a **HOOD design tree** (HDT) (see *section 5.1* below) to be defined.
  - **Classes** are defined as terminal objects that can be **refined by aattribution and inheritance**, allowing their implementation to be factorized by sharing common structures and properties.
- *Separation of operation and object dynamical behaviours according to concurrence and client -server spaces:*
  Operations of objects are **activated** and executed along **control flows** corresponding to the execution of **logical threads which may be:**
  - encapsulated in objects or in virtual nodes.
  - implemented on an underlying target machine, or on several physical computing resources.
  - There may be several control flows operating simultaneously in an object. Control flows appear through asynchronous events in real time systems and furthermore through implementation of parallelism on sequential machines.

  Control structures are used to describe such thread execution behaviour or **control flows**:
  - sequential operation executions are described within an **OPeration Control Structure** (see *section 3.2*)
  - temporal ordering of operation executions and/or concurrent behaviour of objects are described within an **OBject Control Structure** (see *section 3.2*), which may be implemented over a client and server memory partition.

2 HOOD OVERVIEW

2.1 BASIC CONCEPTS

Historically OOD[*[BOOCH86]*] was first a technique using **objects as the basic unit of modularity** in system design. The earlier definition of HOOD object relied fully on this terminology. Later, as work in Object Oriented (OO) area and terminology matured [*[MEYER88]*], the concept of class emerged as the unit of modularity in OO, and the term object was used with the meaning “instance of a class”.

This did not preclude modifying existing HOOD concepts and instead, *a HOOD object remains the unit of modularity, but is now defined as an encapsulation of services containing OO classes, OO objects* (as OO class instances), *data* (as primitive type instances) *and associated operations***.** Moreover a class (see *section 9*) is a module shared by its instances, and is represented itself as a special HOOD object.

In order to avoid any misunderstanding, the following terminology rules have been enforced throughout this book :

- the term **“object”** is always used to mean a **module of software** (possibly referring to a simple module of software, an OO class, a generic unit, or a virtual node), unless specified otherwise;
- The term **“class instance**” is always used instead **“OO object”** to designate the entity associated to an OO class instance.

A HOOD object is thus **a software module specification**, **being primarily an encapsulation of services *provided* to other client software,** and which contains types, classes, data and operations working on that data. A HOOD object may be mapped e.g. into an Ada package, or a C++*[STROU91]* module, whereas an elementary operation may be implemented as an Ada procedure or C++ function.

- **the include hierarchy**, allowing an object to be a composition of other objects, and which is necessary to properly subcontract pieces of software.
- **the inheritance hierarchy**, allowing definition of objects/classes by increments over shared code with other objects/classes.

These considerations led to the merging of Abstract Machines and Object Oriented Design into a common direction, thus defining the Hierarchical Object Oriented Design methodology. The HOOD method has been adapted by CISI, CRI and MATRA-MARCONI-SPACE, in order to meet the needs of the EUROPEAN SPACE AGENCY.

Since, the HOOD method has been enhanced by the HOOD TECHNICAL GROUP (HTG) under control of the HOOD User's Group (HUG) comprising major aerospace and industry companies in Europe.

The major enhancements of the present release named $HOOD_4$ over previous official release*[HRM3.1]* have been experimented and validated by the HTG from 1994 on and are:

- **OO classes and inheritance support** allowing HOOD to be used as the unique design representation for components developed either with classical languages or with object oriented languages.
- **enforcement of separation principles from design through target code** into:
  - *pure sequential code* through the concept of operation control structure
  - *state integrity code* through the concept of state constraints and concurency constraints
  - *inter-process communication code* through the concept of protocol constraints

  This separation allows the overall complexity to be broken down in sub-areas, thus:
  - dedicated techniques, development standards, target code, and people may be used for developing and verifying in parallel specific, separated properties of the software,
  - the test and verification activities are more efficient.

- **multi-target code generation support** for several target platforms:
  - HOOD4 provides the designer with an Object Oriented Framework by means of the HOOD RUN TIME SUPPORT library together with automated code generation.
  - HOOD4 thus supports the development of highly reusable components by shielding HOOD object designs and associated target code from complex semantic differences between multiple OS platforms.

1 INTRODUCTION

HOOD (Hierarchical Object Oriented Design) is a design method which is fully compliant with Ada*[ADA]*, other Object Oriented languages such as C++*[STROU91]* and more classical languages program developments. HOOD supports identification of an object architecture and leads naturally into detailed design where operations and objects or classes are further refined and coded. HOOD also supports testing activities by structuring the object properties and providing a unified documentation framework for both the target code and testing code. Figure 1 indicates HOOD usability within the life-cycle *[BSSC91]*.

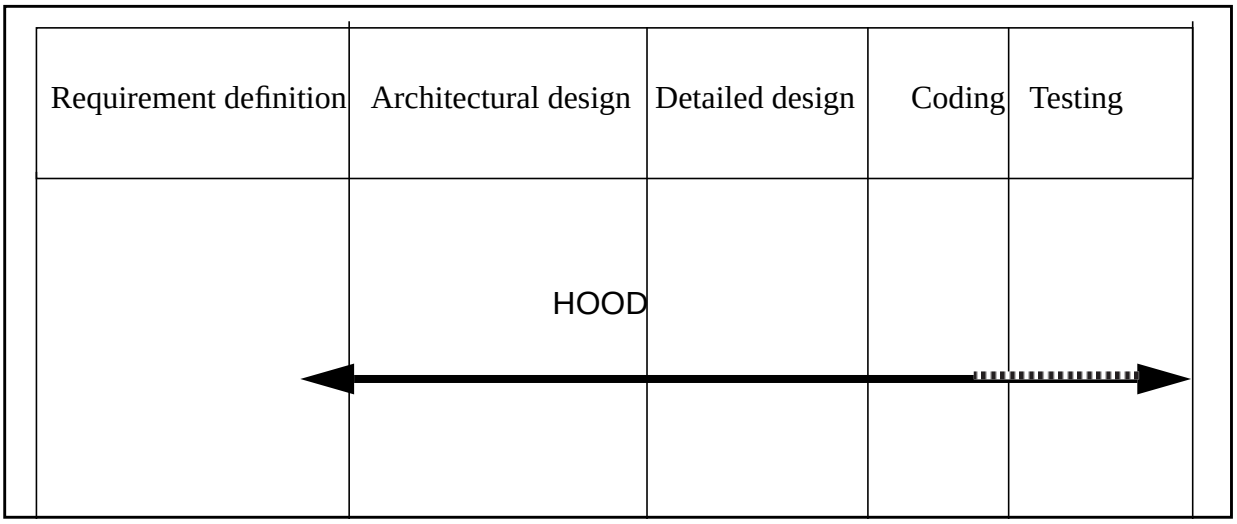

Figure 1 - HOOD in the Software Life-Cycle

HOOD has resulted from merging MATRA-MARCONI-SPACE's experience on Abstract Machines (AM) [*[MACH85]*] and CISI's experience on Object Oriented Design (OOD) [*[BOOCH86]*]. The concepts of object and machine are quite similar, and even complementary: AM enforces a hierarchical structure of objects which is lacking in OOD and is recognized as a key issue for large projects, whereas OOD enforces the design of more coherent objects.

However for very large projects, and to cope with the problem of distributing the development among several companies, the hierarchical features had to be enhanced, and therefore three types of hierarchies had to be defined :

- **the use hierarchy**, already present in the AM concept, in which objects on top of the hierarchy control and use objects underneath. This concept is necessary for structuring numerous objects into virtual layers of high cohesion and low coupling;